Pattern Classification and Machine Learning

Matthias Seeger
Probabilistic Machine Learning Laboratory
Ecole Polytechnique Fédérale de Lausanne
INR 112, Station 14, CH-1015 Lausanne
matthias.seeger@epfl.ch

May 4, 2019

Contents

Chapter 1

Introduction

In this chapter, some general comments are given on the subject of these notes, how to read them, what kind of mathematical preliminaries there are, as well as some recommendations for further reading.

1.1 Learning Outcomes

We will understand what machine learning is all about as we move through the course. Suffice to say, it is of growing importance in most fields which grow, and it is an incredible exciting field to learn about and to work in, whether in research or in one of those many companies which embrace the concept.

These notes complement a course on machine learning at a fairly introductory level. However, the major aim is to train the reader as an *expert* in the development and understanding of machine learning concepts and techniques, not simply as a *user*. At the end of this course, you will understand

- What the basic machine learning methods and techniques are, at least when it comes to supervised machine learning (pattern classification, curve fitting).
- How to apply these methods to real-world problems and datasets.
- Why/when these methods work, and why/when they do not.
- Relationships between methods, opportunities for recombinations. We will learn about the basic "red threads" through this rapidly growing field.
- How to derive and safely implement machine learning methodology by building on underlying standard technology.

1.2 How To Read These Notes

It is highly recommended to read these notes in an interleaved fashion with the classroom lectures. The reader should allocate a substantial amount of time on working through the notes, in particular as a postprocessing to classroom lectures.

Two points are special about these course notes. First, there is substantial material on *Techniques*. Mastering machine learning, whether in research or for working in one of those cool companies, is mastering a bag of tricks. Fortunately, a rather limited number of mathematical tools come up over and over again, and the reader will find many of these in the *Techniques* sections. Each of them is directly motivated by an application in the main text. In order to benefit from this material, the reader has to actively work through it, jotting down notes on a piece of paper.

Second, these notes contain more material than is covered in the classroom lectures. Slightly more advanced material is presented in sections labelled by "(*)". Notice that sections labelled as "(*)" may still feature in classroom lectures and the exam (sections labelled as "(*)", which are not discussed during classroom lectures, will not feature in the exam, unless explicitly said otherwise).

These course notes are brand new. The author very much hopes for feedback from readers for how to make them more useful, more readable, more fun to learn from, and for fixing mistakes.

1.3 Mathematical Preliminaries

Given that machine learning may be considered a discipline of computer science, it places somewhat different demands on students compared to fields which have traditionally been considered core computer science (such as logics, complexity, verification, databases, *etc.*). Briefly, we need continuous mathematics: *linear algebra*, *differential calculus* and *continuous optimization*. And we need *probability* and *computational statistics*. Here the good news: machine learning offers you an amazing route to refresh your skills in these disciplines, to learn them from a different and immediately practically rewarding perspective, and to recognize their purpose even if you do not care about physics.

In these notes, required basic mathematical skills are reviewed on the fly, a sort of "maths as we go" approach. Instead of tucking it away in some Appendix, the maths is presented where needed, and a lot of "hyperlinks" are used to get the reader to visit these pages. However, these parts can be no more than a reminder. They are neither complete, nor very didactic, and certainly they are not rigorous. Here are some recommended sources for further study, which meet these criteria:

- For linear algebra, the author's favourite is the book by Strang [45]. Make sure to check out the accompanying videos.
- For calculus, the book by Simmons [42] seems to be widely acclaimed for its intuitive approach and focus on applications to real-world problems.
- An acclaimed introductory text on probability is by Grinstead and Snell [20], this book can be downloaded free of charge from the web. The author can also recommmend the book by Grimmett and Stirzaker [19].

1.4 Recommended Machine Learning Textbooks

There are many books on machine learning. Some are more basic than these course notes, and they contain many examples, code and "tricks of the trade". While these are certainly helpful as an addition to the material here, the author is not convinced enough by any of them for a recommendation. One intermediate level book is by Duda, Hart and Stork [13]. While this is a useful book to read, it is the author's opinion that the second edition is rather a step backwards from the first one, which is excellent but out of date. It also comes with an excessive price tag, so think twice before you buy it. Then, there is a host of books written by leading machine learning researchers today, each of which contains substantially *more* (and somewhat more advanced) material than these notes. The book by Bishop [6] may be the best to read in addition to these notes, several of the excellent figures are used here. His older textbook [5] is less up to date, but still an excellent source, in particular for nonlinear optimization and multi-layer perceptrons. The book by MacKay [30] is one of the author's absolute favourites, and full of great exercises to train your understanding (check it out; it is freely available from the author's homepage). A good book on statistical data analysis and kernel methods is that by Hastie *et.al.* [21]. The book by Koller and Friedmann [27] is encyclopaedic and deals exclusively with graphical models and Bayesian inference. Finally, two new textbooks came out in 2012, by David Barber (UC London) [1] and by Kevin Murphy (Google Research) [31]. Both are excellent books which go far beyond these notes and come with a host of real-world examples and publicly available `Matlab` code. Between the two, Barber's book is somewhat more accessible, while Murphy's book has a broader scope. Both books emphasize the importance of probabilistic modelling and Bayesian inference.

1.5 Thanks

The author would like to thank Hesam Setareh for creating many of the figures.

Chapter 2

Linear Classification

In this chapter, we will study classifiers based on linear discriminant functions in a fixed, finite-dimensional feature space. A running example will be the classification of hand-written digits. We will learn to know a first algorithm to train a linear classifier on data. By studying properties and convergence of this perceptron algorithm, we develop geometric intuition which will be important in subsequent chapters. More generally, we will see that many machine learning problems can be phrased in terms of mathematical optimization, as minimization of an error function. We will get a first idea about the squared error function, learn to know batch and online gradient descent optimization, and finally develop a second viewpoint on the perceptron learning algorithm.

2.1 A First Example

Machine learning is a big field, but let's not be timid and jump right into the middle. Consider the problem of *classifying hand-written digits* (Figure 2.1). The US postal service (USPS) decides to commission a system to automatize the presorting of envelopes according to ZIP codes, and your company wins the contract. How does such a system look like? Let us ignore hardware issues: a scanned image of the envelope comes in, a ZIP code prediction comes out. The first step is *preprocessing*: detecting the writing on the envelope, detecting the ZIP code, segmenting out the separate digits, normalizing them by correcting for simple transformations such as translation, rotation and scale, and quantizing them onto a standard grid of equal size. Preprocessing is of central importance for any real-world machine learning system. However, since preprocessing tends to be domain-dependent, we will not discuss it any further during this course, whose objective is to cover machine learning principles of wide or general applicability.

Datasets of preprocessed handwritten digits are publicly available, the most well known are MNIST (`http://yann.lecun.com/exdb/mnist/`) and USPS (`http://www.gaussianprocess.org/gpml/data/`). In the middle ages of machine learning, these datasets have played a very important role, providing a

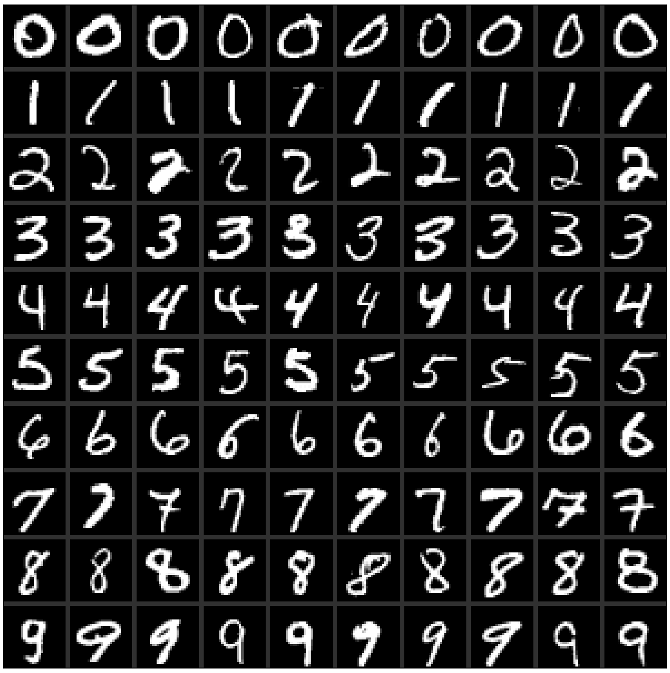


Figure 2.1: Patterns from the training dataset of MNIST handwritten digits database. Shown are ten patterns per class, drawn at random.

unified testbed for all sorts of innovative ideas and algorithms, *publicly available to anyone.*

How does the MNIST dataset look like? For much of this course, a *dataset* is an unordered set of *instances* or *cases*. Each case decomposes into attributes in the same way. MNIST is a *classification* dataset (more specifically, multi-way classification): each case consists of an *input point* (or pattern) and a *target* (or class label). We can formalize this by introducing variables $\boldsymbol{x}$ for the input point attribute, t for the target, and $(\boldsymbol{x}, t)$ for the case. A dataset like MNIST is simply an set of variable instances:

$$\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}.$$

In other words, the dataset has n instances (or data points), and the i-th instance is $(\boldsymbol{x}_i, t_i)$. Each attribute has a data type, comes from a defined value range. For MNIST, $t \in \{0, \dots, 9\}$, the ten digits. The input point $\boldsymbol{x}$ is a 28×28 bitmap, each pixel quantized to $\{0, \dots, 255\}$ (0 for no, 255 for full intensity). It is common practice to represent bitmaps as *vectors*, stacking columns on top of each other (from left to right):

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \quad \boldsymbol{x}_i = \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,d} \end{bmatrix}$$

Refresh your memory on vectors in Section 2.1.1. Given the MNIST specification, the attribute space for $\boldsymbol{x}$ should be $\{0, \dots, 255\}^d$, where $d = 28 \cdot 28 = 784$.

However, we tend to use the weaker specification $\boldsymbol{x} \in \mathbb{R}^d$. d is known as *input space dimensionality.*

The MNIST *multi-way classification* problem is as follows. The goal is to learn a *classifier* $f(\boldsymbol{x})$, mapping $\boldsymbol{x} \in \mathbb{R}^d$ to target predictions $f(\boldsymbol{x}) \in \{0, \dots, 9\}$. How to do that? We could hire some people who spend the rest of their life fine-tuning some large, elaborate set of rules and cryptic computer code, driven by their "intuition". This is *not* learning in any sense, whether human or machine. We could also use a large set of examples and choose a good classifier by statistically fitting it to this data. In essence, this is what machine learning is about. Don't bother with constructing every detail of an architecture, supposed to solve a problem you do not fully understand anyway. Instead, collect big data and induce predictive knowledge *directly* from there. For the rest of this chapter, we restrict ourselves to binary classification, where the target can take two different values only.

Binary Classification

A binary classifier $f(\boldsymbol{x})$ maps input points $\boldsymbol{x}$ (often, $\boldsymbol{x} \in \mathbb{R}^d$) to binary targets $t \in \mathcal{T}$. The value space $\mathcal{T}$ for t is of size two. In principle, any binary set can be used. We will mainly use $\mathcal{T} = \{0, 1\}$ or $\mathcal{T} = \{-1, +1\}$ in this course.

A running example for binary classification during the rest of this section is discriminating 8s from 9s among the MNIST digits. We will use the label space $\mathcal{T} = \{-1, 1\}$, mapping 8 to -1, 9 to 1. Any good ideas? As a computer scientist, your first attempt may be to setup a database, feeding it with your data $\{(\boldsymbol{x}_i, t_i) \mid t_i \in \{-1, 1\}\}$. For this purpose, we would stick with the finite domain $\{0, \dots, 255\}^d$ for $\boldsymbol{x}$. Given some pattern $\boldsymbol{x}_*$, we query the database. In case we find it, say $\boldsymbol{x}_* = \boldsymbol{x}_i$ for some $i \in \{1, \dots, n\}$, we output t_i. Otherwise, we output "don't know". This method is also known as *lookup table*:

$$f_{\text{lut}}(\boldsymbol{x}) = \left\{ \begin{array}{ll} t_i & \mid \boldsymbol{x} = \boldsymbol{x}_i, i \in \{1, \dots, n\} \\ \text{Don't know} & \mid \boldsymbol{x} \neq \boldsymbol{x}_i, i = 1, \dots, n \end{array} \right\}.$$

Unfortunately, the lookup table approach will never work. While the input domain is finite, it is extremely large (exponential in the dimensionality d), and any conceivable dataset would only ever cover a vanishing fraction. Essentially, $f_{\text{lut}}(\boldsymbol{x}_*) =$ "don't know" all the time. For people who like lookup tables and similar things, this is known as the "curse of dimensionality".

Learning starts with the insight that most attributes of the real world we care about are *robust* to some changes. If $\boldsymbol{x}$ represents an 8 written by hand, neither of the following modifications will in general turn it into a 9: modifying a few components x_j by some small amount; translating the bitmap slightly; rotating the bitmap a bit. It is therefore a reasonable *assumption* on how the world works that with respect to a sensible distance function between bitmaps, if $\boldsymbol{x}$ belongs to a class (say, 8), any $\boldsymbol{x}'$ close to $\boldsymbol{x}$ in this distance is highly likely to belong to the same class. As $\boldsymbol{x}$, $\boldsymbol{x}'$ are vectors, let us use the standard *Euclidean distance*:

$$\|\boldsymbol{x} - \boldsymbol{x}'\| = \sqrt{(\boldsymbol{x} - \boldsymbol{x}')^T(\boldsymbol{x} - \boldsymbol{x}')} = \sqrt{\sum_{j=1}^{d} (x_j - x_j')^2}.$$

Refresh your memory on Euclidean distance in Section 2.1.1. Here is a vastly better idea than f_{lut}. We still use the whole dataset $\{(\boldsymbol{x}_i, t_i)\}$ for classification. For a pattern $\boldsymbol{x}_*$, we output the label t_i of the nearest[1] pattern $\boldsymbol{x}_i$ in our database:

$$f_{\text{NN}}(\boldsymbol{x}) = t_i \quad \Leftrightarrow \quad \|\boldsymbol{x} - \boldsymbol{x}_i\| \le \|\boldsymbol{x} - \boldsymbol{x}_j\|, \;\; j = 1, \dots, n.$$

This is the *nearest neighbour* (NN) classification rule. It is one of the oldest machine learning algorithms, and variants of it are widely used in practice. Moreover, the theoretical analysis of this rule, initiated by Cover and Hart, is a cornerstone of learning theory. Some details about nearest neighbour methods can be found in [6, Sect. 2.5.2], its theoretical analysis (with original citations) is detailed in [12]. Despite the fact that variants of NN work well on our digits problem (whether binary or multi-way), there are some substantial drawbacks:

- The whole dataset has to be kept around, as each single case $(\boldsymbol{x}_i, t_i)$ potentially takes part in a classification decision. For large datasets, this is a costly liability.
- Research in NN tends to be mainly algorithm-driven. After all, clever data structures and search techniques are needed in order to find nearest neighbours rapidly. Most of these techniques have a hard time if the dimensionality d is larger than 20 or so.
- There is not much to be learned about the data from NN. In fact, some machine learning work aims to replace the Euclidean by a distance adapted to the job, but compared to model-based mechanisms for specifying domain knowledge, these possibilities are limited.

In order to avoid having to store our whole MNIST dataset, we have to represent the information relevant for classification in a far smaller number of parameters. For example, we could try to find $m \ll n$ (m "much smaller than" n) prototype vectors $\boldsymbol{w}_c$, $c = 1, \dots, m$, each associated with a target value t_c^w, then to do NN classification w.r.t. the prototype set $\{(\boldsymbol{w}_c, t_c^w) \mid c = 1, \dots, m\}$ rather than the full dataset:

$$f^{(m)}(\boldsymbol{x}) = f_{\text{NN}}(\boldsymbol{x}; \{(\boldsymbol{w}_c, t_c^w)\}).$$

Here, *learning* refers to the automatic choice of the parameters $\{(\boldsymbol{w}_c, t_c^w)\}$ from the data $\{(\boldsymbol{x}_i, t_i)\}$, or to the modification of the parameters as new data comes in. Having learned our classifier (i.e., fixed its parameters), we *predict* the label of a pattern $\boldsymbol{x}_*$ by NN on the prototype vectors. How can we learn automatically? How to choose m (number of prototypes) given n (size of dataset)? If n increases, as new data comes in, do we need to increase m or can we keep it fixed? These are typical machine learning questions.

Taking this idea to the limit, let us pick exactly $m = 2$ prototype vectors $\boldsymbol{w}_{-1}$, $\boldsymbol{w}_{+1}$, one for each class, with $t_c^w = c$, $c = -1, +1$. The classification rule $f^{(2)}(\boldsymbol{x}_*)$ for a pattern $\boldsymbol{x}_*$ is simple: if $\|\boldsymbol{x}_* - \boldsymbol{w}_{+1}\| < \|\boldsymbol{x}_* - \boldsymbol{w}_{-1}\|$ output $+1$, otherwise

[1] If there are ties, we pick one of the nearest neighbour labels at random.

output -1. Let us simplify this even more:

$$
\begin{aligned}
& \|\boldsymbol{x}_* - \boldsymbol{w}_{+1}\| < \|\boldsymbol{x}_* - \boldsymbol{w}_{-1}\| \quad \Leftrightarrow \quad \|\boldsymbol{x}_* - \boldsymbol{w}_{+1}\|^2 < \|\boldsymbol{x}_* - \boldsymbol{w}_{-1}\|^2 \\
\Leftrightarrow\ & \|\boldsymbol{x}_*\|^2 - 2\boldsymbol{w}_{+1}^T\boldsymbol{x}_* + \|\boldsymbol{w}_{+1}\|^2 < \|\boldsymbol{x}_*\|^2 - 2\boldsymbol{w}_{-1}^T\boldsymbol{x}_* + \|\boldsymbol{w}_{-1}\|^2 \\
\Leftrightarrow\ & \boldsymbol{w}_{+1}^T\boldsymbol{x}_* - \|\boldsymbol{w}_{+1}\|^2/2 > \boldsymbol{w}_{-1}^T\boldsymbol{x}_* - \|\boldsymbol{w}_{-1}\|^2/2 \\
\Leftrightarrow\ & (\boldsymbol{w}_{+1} - \boldsymbol{w}_{-1})^T\boldsymbol{x}_* + \frac{1}{2}\left(\|\boldsymbol{w}_{-1}\|^2 - \|\boldsymbol{w}_{+1}\|^2\right) > 0.
\end{aligned}
$$

Problems with expanding $\|\boldsymbol{x}_* - \boldsymbol{w}_{+1}\|^2$? Have a look at Section 2.1.1. If we set $\boldsymbol{w} = \boldsymbol{w}_{+1} - \boldsymbol{w}_{-1}$, $b = (\|\boldsymbol{w}_{-1}\|^2 - \|\boldsymbol{w}_{+1}\|^2)/2$, this is simply a *linear* inequality:

$$
f^{(2)}(\boldsymbol{x}_*) = \left\{ \begin{array}{ll} +1 & \mid\ \boldsymbol{w}^T\boldsymbol{x}_* + b > 0 \\ -1 & \mid\ \text{otherwise} \end{array} \right\} = \operatorname{sgn}\left(\boldsymbol{w}^T\boldsymbol{x}_* + b\right).
$$

The *sign function* $\operatorname{sgn}(a)$ takes the value $+1$ for $a > 0$, -1 for $a < 0$. The definition[2] of $\operatorname{sgn}(0)$ typically does not matter, let us define $\operatorname{sgn}(0) = 0$. We came some way: from lookup tables over nearest neighbours all the way to *linear classifiers.*

2.1.1 Techniques: Vectors, Inner Products, Norms

In order to understand linear classification or anything else built on top of it, we need to be familiar with vector spaces. Start with some scalar space $\mathcal{A}$ for the coefficients and consider columns of d entries a_j from $\mathcal{A}$:

$$
\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \end{bmatrix}.
$$

Note that we use columns, not rows. The set of all such columns $\boldsymbol{a}$ is denoted by $\mathcal{A}^d$: the d-fold direct product. This is a d-dimensional *vector space* if $\mathcal{A}$ is a field. In this course, $\mathcal{A} = \mathbb{R}$, the real numbers. We will denote vectors $\boldsymbol{a}$ in bold face, to distiguish them from scalars. In other books, you might encounter notations like $\vec{a}$ or $\underline{a}$, or simply a (no distinction between scalars and vectors). The dimension of the space, d, is the maximum number of linearly independent vectors it contains (read up on “linearly independent”, “basis” in [45]). You can add vectors, multiply with scalars, it all works by doing the operations on the coefficients separately. In fact, the general definition of *vector space* is that of a set which is closed under addition and multiplication with scalars (being “closed” means that if you perform any such operation on vectors, you end up with another vector, not with something outside of your vector space). The transpose operator converts a column into a row vector (and vice versa): $\boldsymbol{a}^T = [a_1, \dots, a_d] \in \mathbb{R}^{1\times d}$. We can identify $\mathbb{R}^d$ with $\mathbb{R}^{d\times 1}$: column and row vectors are special cases of matrices (Section 2.4.3). Some texts use $\boldsymbol{a}'$ to denote transposition instead of $\boldsymbol{a}^T$.

[2] If $\operatorname{sgn}(\boldsymbol{w}^T\boldsymbol{x} + b)$ is a classification rule supposed to output 1 or -1 for every input $\boldsymbol{x}$, we should sample the output at random if $\boldsymbol{w}^T\boldsymbol{x} + b = 0$.

We will often define vectors by giving an expression for its coefficients. For example, $\boldsymbol{a} = [f(\boldsymbol{x}_i)]_i$ (or $\boldsymbol{a} = [f(\boldsymbol{x}_i)]$ if the index i is obvious from the context) means that the coefficients are defined by $a_i = f(\boldsymbol{x}_i)$. A number of special vectors will be used frequently in these notes. $\mathbf{0} = [0, \dots, 0]^T$ is the vector of all zeros. $\mathbf{1} = [1, \dots, 1]^T$ is the vector of all ones. Their dimensionality is typically clear from the context. We also need delta vectors $\boldsymbol{\delta}_k \in \mathbb{R}^d$, defined by $\boldsymbol{\delta}_k = [\mathrm{I}_{\{j=k\}}]_j$ (recall that $\mathrm{I}_{\{j=k\}} = 1$ if $j = k$, 0 otherwise): the k-th coefficient of $\boldsymbol{\delta}_k$ is 1, all others are 0. Again, the dimensionality is typically clear from the context.

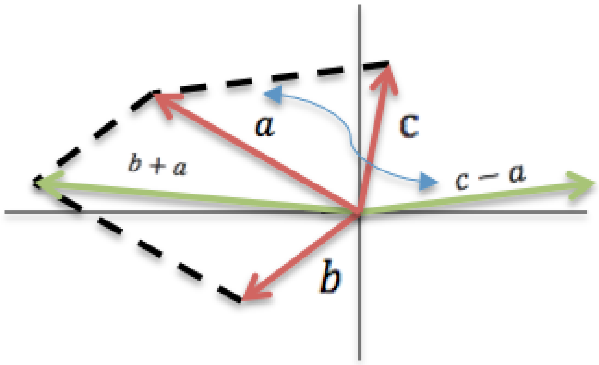


Figure 2.2: A vector $\boldsymbol{a} \in \mathbb{R}^2$ can be visualized as arrow from the origin to $\boldsymbol{a}$. The order in which vectors are added does not matter (parallelogram identity).

Vectors $\boldsymbol{a}$ in $\mathbb{R}^2$ or $\mathbb{R}^3$ can be visualized as arrows in a Cartesian coordinate system, pointing from the origin $\mathbf{0}$ to the position $\boldsymbol{a} = [a_1, a_2]^T$ or $[a_1, a_2, a_3]^T$. You can visualize $\boldsymbol{a} + \boldsymbol{b}$ by translating the $\boldsymbol{b}$ arrow so that it starts from the endpoint of $\boldsymbol{a}$: it will then point to $\boldsymbol{a} + \boldsymbol{b}$. Flip $\boldsymbol{a}$ and $\boldsymbol{b}$, and you get to the same point (what you have now is a parallelogram, see Figure 2.2). $-\boldsymbol{a}$ just mirrors the $\boldsymbol{a}$ arrow about the origin. Add $-\boldsymbol{a}$ to $\boldsymbol{a}$, and you are back where you started: great fun (do it!). In our notation, the vectors defining the Cartesian coordinate axes are $\boldsymbol{\delta}_1$, $\boldsymbol{\delta}_2$ (and $\boldsymbol{\delta}_3$), but be aware that in some physics-based texts, they may be called $\boldsymbol{i}$, $\boldsymbol{j}$ (and $\boldsymbol{k}$).

If $\mathcal{V}$ is a vector space, then $\mathcal{U} \subset \mathcal{V}$ is a (linear) *subspace* if $\mathcal{U}$ itself is a vector space: closed under addition and multiplication with scalars $\alpha \in \mathbb{R}$. Note that each vector space always contains $\mathbf{0}$, since that is zero times any other vector. An *affine subspace* of $\mathcal{V}$ has the form

$$\boldsymbol{v} + \mathcal{U} = \left\{ \boldsymbol{v} + \boldsymbol{u} \;\middle|\; \boldsymbol{u} \in \mathcal{U} \right\} \subset \mathcal{V},$$

where $\mathcal{U}$ is a (linear) subspace of $\mathcal{V}$. An affine subspace need not contain $\mathbf{0}$. In fact, it is a linear subspace if and only if it contains $\mathbf{0}$ (Figure 2.3).

We can also combine two vectors. The most basic operation is the *inner product*

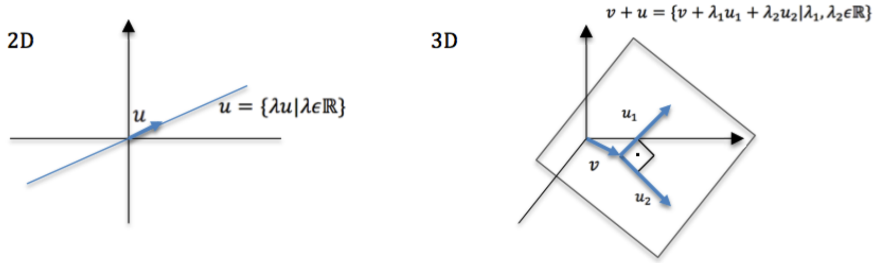

2D
u
$u = \{\lambda u | \lambda \epsilon \mathbb{R}\}$
3D
$v + u = \{v + \lambda_1 u_1 + \lambda_2 u_2 | \lambda_1, \lambda_2 \epsilon \mathbb{R}\}$
u_1
v
u_2

Figure 2.3: Left: One-dimensional linear subspace in $\mathbb{R}^2$, spanned by $\boldsymbol{u}$. Right: Two-dimensional affine subspace in $\mathbb{R}^3$, spanned by the basis $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$.

(or scalar product), resulting in a scalar:

$$\boldsymbol{a}^T\boldsymbol{b} = [a_1, \dots, a_d] \begin{bmatrix} b_1 \\ \vdots \\ b_d \end{bmatrix} = \sum_{j=1}^{d} a_j b_j.$$

Other notations you might encounter are $\boldsymbol{a} \cdot \boldsymbol{b}$ or $(\boldsymbol{a}, \boldsymbol{b})$ (or $\langle \boldsymbol{a} | \boldsymbol{b} \rangle$ if you want to be a really cool quantum physicist), but not in these notes. There are obvious properties, such as $\boldsymbol{a}^T\boldsymbol{b} = \boldsymbol{b}^T\boldsymbol{a}$ (symmetry) or $\boldsymbol{a}^T(\boldsymbol{b} + \boldsymbol{c}) = \boldsymbol{a}^T\boldsymbol{b} + \boldsymbol{a}^T\boldsymbol{c}$ (linearity). By the way, what are $\boldsymbol{a}\boldsymbol{b}$ or $\boldsymbol{a}^T\boldsymbol{b}^T$? Nothing, such operations are not defined (unless you are in $\mathbb{R}^1$). What is $\boldsymbol{a}\boldsymbol{b}^T$? That works. More about these in Section 2.4.3

The geometrical meaning of the inner product will be discussed in Section 2.3. Here only two points. First, the square root of the inner product of $\boldsymbol{a}$ with itself is the standard (or *Euclidean*) *distance* of $\boldsymbol{a}$ from the origin, its *length* or its (Euclidean) *norm*:

$$\|\boldsymbol{a}\| = \sqrt{\boldsymbol{a}^T\boldsymbol{a}} = \sqrt{\sum_{j=1}^{d} a_j^2}.$$

We will often use the squared norm $\|\boldsymbol{a}\|^2$ to get rid of the square root. The (Euclidean) distance between $\boldsymbol{a}$ and $\boldsymbol{b}$ is the norm of $\boldsymbol{b} - \boldsymbol{a}$ (or $\boldsymbol{a} - \boldsymbol{b}$, since $\|-\boldsymbol{c}\| = \|\boldsymbol{c}\|$), something that is obvious once you draw it. Convince yourself that if $\|\boldsymbol{a}\| = 0$, then $\boldsymbol{a}$ must be the zero vector $\mathbf{0}$. If $\|\boldsymbol{a}\| \neq 0$, we can *normalize* the vector: $\boldsymbol{a} \to \boldsymbol{a}/\|\boldsymbol{a}\|$. The outcome is a *unit vector* (vector of length 1). We will use unit vectors if we really only want to represent a direction. In fact, any non-zero vector $\boldsymbol{a}$ can be represented by its length $\|\boldsymbol{a}\|$ and its direction $\boldsymbol{a}/\|\boldsymbol{a}\|$.

Second, sometimes it happens that $\boldsymbol{a}^T\boldsymbol{b} = 0$: the inner product between $\boldsymbol{a}$ and $\boldsymbol{b}$ is zero, such vectors are called *orthogonal* (or perpendicular). The angle between orthogonal vectors is a right one (90 degrees). Note that a set of mutually orthogonal vectors ("mutually" applied to a set means: "for each pair") is also linearly independent, so there can be no more than d such vectors (an example of such a set is $\{\boldsymbol{\delta}_1, \dots, \boldsymbol{\delta}_d\}$).

Moreover, a vector $\boldsymbol{v}$ is orthogonal to a subspace $\mathcal{U} \subset \mathbb{R}^d$ if and only if $\boldsymbol{v}^T\boldsymbol{u} = 0$ for all $\boldsymbol{u} \in \mathcal{U}$. Two subspaces $\mathcal{U}_1$, $\mathcal{U}_2$ of $\mathbb{R}^d$ are called *orthogonal* if for any $\boldsymbol{u}_1 \in \mathcal{U}_1$, $\boldsymbol{u}_2 \in \mathcal{U}_2$: $\boldsymbol{u}_1^T\boldsymbol{u}_2 = 0$. In other words, each $\boldsymbol{u}_1 \in \mathcal{U}_1$ is orthogonal to $\mathcal{U}_2$. Given a subspace $\mathcal{U} \subset \mathbb{R}^d$, its *orthogonal complement* $\mathcal{U}^\perp$ is the subspace of all $\boldsymbol{v} \in \mathbb{R}^d$ orthogonal to $\mathcal{U}$. We will take up these definitions in Section 4.2.2, where we discuss orthogonal projections.

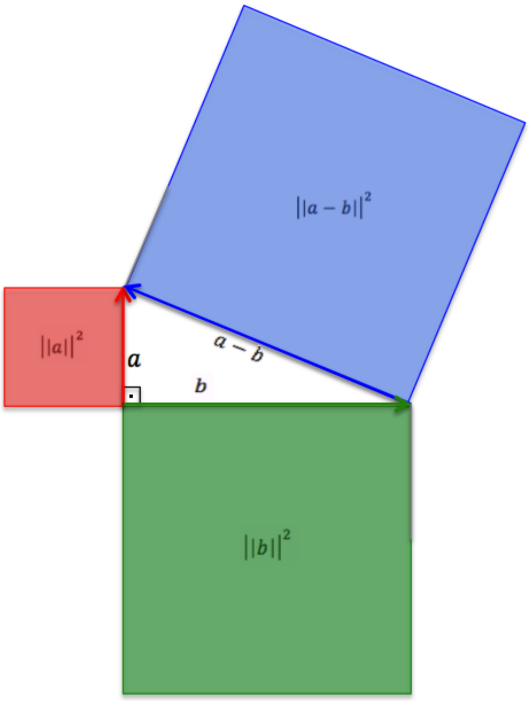


Figure 2.4: Pythagorean theorem: $\|\boldsymbol{a}-\boldsymbol{b}\|^2 = \|\boldsymbol{a}\|^2 + \|\boldsymbol{b}\|^2$ if and only if $\boldsymbol{a}$, $\boldsymbol{b}$ are orthogonal ($\boldsymbol{a}^T\boldsymbol{b} = 0$).

We will frequently manipulate squared distances between vectors, an example has been given in Section 2.1. This is a computation you should know by heart, in either direction:

$$\|\boldsymbol{a}-\boldsymbol{b}\|^2 = (\boldsymbol{a}-\boldsymbol{b})^T(\boldsymbol{a}-\boldsymbol{b}) = \boldsymbol{a}^T\boldsymbol{a} - \boldsymbol{a}^T\boldsymbol{b} - \boldsymbol{b}^T\boldsymbol{a} + \boldsymbol{b}^T\boldsymbol{b} = \|\boldsymbol{a}\|^2 - 2\boldsymbol{a}^T\boldsymbol{b} + \|\boldsymbol{b}\|^2.$$

First, we use the linearity, then the symmetry of the inner product. This equation relates the squared distance between $\boldsymbol{a}$ and $\boldsymbol{b}$ to the squared distance of $\boldsymbol{a}$ and $\boldsymbol{b}$ from the origin respectively. The term $-2\boldsymbol{a}^T\boldsymbol{b}$ is sometimes called "cross-talk". It vanishes if $\boldsymbol{a}$ and $\boldsymbol{b}$ are orthogonal, so that $\|\boldsymbol{a}-\boldsymbol{b}\|^2 = \|\boldsymbol{a}\|^2+\|\boldsymbol{b}\|^2$. This is simply the *Pythagorean theorem* from high school (Figure 2.4). I bet it took more pains back then to understand it! As we move on, you will note that this innocuous observation is ultimately the basis for orthogonal projection, conditional expectation, least squares estimation, and bias-variance decompositions. Test your understanding by working out

$$\|\boldsymbol{a}-\boldsymbol{b}\|^2 + \|\boldsymbol{a}+\boldsymbol{b}\|^2 \quad \text{and} \quad \|\boldsymbol{a}\|^2 + 4\boldsymbol{a}^T\boldsymbol{c}.$$

The first is called the parallelogram identity (draw it!). Anything still unclear? Then you should pick your favourite source from Section 1.3.

2.2 Hyperplanes and Feature Spaces

In Section 2.1 we have seen one motivation for linear classifiers, discriminating handwritten 8s from 9s. There are many other motivations for this class of discriminants, and we will explore a number of them as we proceed. Linear classifiers (or, more generally, linear functions) are the single most important building block of statistics and machine learning, and it is essential to understand their properties. While our goal is of course to find algorithms for learning such classifiers from data (such as the MNIST database), let us first build some geometrical intuition about them.

One way to define a binary classifier $f : \mathcal{X} \to \{-1, +1\}$, mapping input points $\boldsymbol{x} \in \mathcal{X}$ (for example, $\mathcal{X} = \mathbb{R}^d$) to target predictions $f(\boldsymbol{x}) \in \{0, 1\}$, is via a *discriminant function* $y : \mathcal{X} \to \mathbb{R}$:

$$f(\boldsymbol{x}) = \mathrm{sgn}(y(\boldsymbol{x})) = \left\{ \begin{array}{ll} +1 & \mid y(\boldsymbol{x}) > 0 \\ -1 & \mid y(\boldsymbol{x}) < 0 \end{array} \right\}.$$

In other words, we threshold $y(\boldsymbol{x})$ at zero. The value of $f(\boldsymbol{x})$ for $y(\boldsymbol{x}) = 0$ is often left unspecified, a good practice is to draw it uniformly at random. The set

$$\left\{ \boldsymbol{x} \;\middle|\; y(\boldsymbol{x}) = 0 \right\}$$

is called *decision boundary*, literally the region where the decisions made by $f(\boldsymbol{x})$ are switching between the classes. Classifier and discriminant function are equivalent concepts. However, it is usually much simpler to define and work with functions mapping to $\mathbb{R}$ than to $\{-1, +1\}$.

For the moment, let us accept the following definition, which we will slightly generalize towards the end of this section. A binary classifier $f(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^d$, is called *linear* if it is defined in terms of a linear discriminant function $y(\boldsymbol{x})$:

$$f(\boldsymbol{x}) = \mathrm{sgn}(y(\boldsymbol{x})), \quad y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b, \quad \boldsymbol{w} \in \mathbb{R}^d \setminus \{\boldsymbol{0}\}, b \in \mathbb{R}.$$

Here, $\boldsymbol{w}$ is called *weight vector* (or also *normal vector*), and b is called *bias parameter.*

Geometry of Linear Discriminants

How can we picture a linear discriminant function? Let us start with a linear equation $x_2 = ax_1 + b$ of the high school type. While this form suggests that x_1 is the input (or free), x_2 the output (or dependent) variable, we can simply treat (x_1, x_2) as points in the Euclidean $\mathbb{R}^2$ coordinate system. Obviously, this equation describes a line. a is the slope: for $a = 0$, the line is horizontal, for $a > 0$ it is increasing. b is the offset: we cross the x_2 axis at $x_2 = b$, and the line runs through the origin if and only if $b = 0$. Let us rewrite this equation:

$$x_2 = ax_1 + b \quad \Leftrightarrow \quad [a, -1] \left[\begin{array}{c} x_1 \\ x_2 \end{array} \right] + b = 0.$$

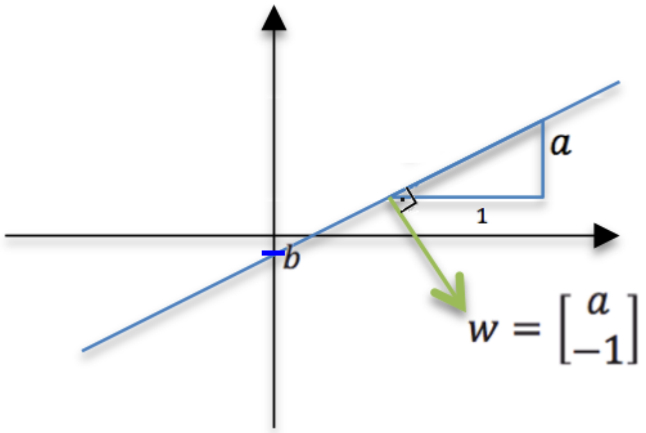


Figure 2.5: Line $x_2 = ax_1 + b$ in $\mathbb{R}^2$ (one-dimensional affine subspace). The parameterization $\boldsymbol{w}^T\boldsymbol{x} + b = 0$, $\boldsymbol{w}$ the normal vector, generalizes to hyperplanes ($(d-1)$-dimensional affine subspaces) in $\mathbb{R}^d$.

Our equation is equivalent to $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b = 0$, where $\boldsymbol{w} = [a, -1]^T$. We can also describe our line as the set of points of the form $[0, b]^T + \lambda \boldsymbol{d}$, $\lambda \in \mathbb{R}$, where $\boldsymbol{d} = [1, a]^T$ is a direction along the line. Importantly, $\boldsymbol{w}^T\boldsymbol{d} = 0$: the weight vector of $y(\boldsymbol{x})$ is orthogonal to any direction along the line (Figure 2.5).

How about $\mathbb{R}^3$ or general $\mathbb{R}^d$? We simply generalize from $\mathbb{R}^2$. A line is a one-dimensional affine subspace ("affine what?" Section 2.1.1). In $\mathbb{R}^d$, $\boldsymbol{w}^T\boldsymbol{x} + b = 0$ for $\boldsymbol{w} \neq \boldsymbol{0}$ defines a $(d-1)$-dimensional affine subspace of $\mathbb{R}^d$. In $\mathbb{R}^3$, this is a plane, and in the "hyperspace" $\mathbb{R}^d$, it is called a *hyperplane*. It corresponds to a set of points of the form $\boldsymbol{v}_0 + \mathcal{U}$, where $\boldsymbol{v}_0 \in \mathbb{R}^d$ and $\mathcal{U}$ is a (linear) subspace of $\mathbb{R}^d$, of dimension $d-1$. $\boldsymbol{w}$ is a *normal vector* of this hyperplane: it is orthogonal to any vector $\boldsymbol{u} \in \mathcal{U}$. $\boldsymbol{w}$ is uniquely determined, in that any normal vector of the hyperplane is a nonzero scalar multiple of $\boldsymbol{w}$. Just as in $\mathbb{R}^2$, the hyperplane contains the origin (i.e., is a linear subspace) if and only if $b = 0$. Finally, we can easily construct an offset point $\boldsymbol{v}_0$ as follows: if $\boldsymbol{v}_0 = -(b/\|\boldsymbol{w}\|^2)\boldsymbol{w}$, then

$$\boldsymbol{w}^T\boldsymbol{v}_0 + b = -(b/\|\boldsymbol{w}\|^2)\boldsymbol{w}^T\boldsymbol{w} + b = 0,$$

so that $\boldsymbol{v}_0$ lies on the hyperplane. This choice of $\boldsymbol{v}_0$ is special: it is the point on the hyperplane closest to the origin, which you can see by noting that it is proportional to the normal vector $\boldsymbol{w}$ (it is the orthogonal projection of $\boldsymbol{0}$ onto the hyperplane, see Section 4.2.2).

The geometry of a linear discriminant function $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$ is clear now. The decision boundary, defined by $y(\boldsymbol{x}) = 0$, is a hyperplane with normal vector $\boldsymbol{w}$. This hyperplane separates the full space $\mathbb{R}^d$ into two halfspaces, defined by

$$\mathcal{H}_{+1} = \left\{ \boldsymbol{x} \;\middle|\; y(\boldsymbol{x}) > 0 \right\} \quad \text{and} \quad \mathcal{H}_{-1} = \left\{ \boldsymbol{x} \;\middle|\; y(\boldsymbol{x}) < 0 \right\}$$

respectively. $f(\boldsymbol{x}) = \operatorname{sgn}(y(\boldsymbol{x}))$ outputs 1 for $\boldsymbol{x} \in \mathcal{H}_{+1}$, -1 for $\boldsymbol{x} \in \mathcal{H}_{-1}$, so the decision regions for each target value are these halfspaces. The normal vector $\boldsymbol{w}$, started from anywhere on the hyperplane, points into $\mathcal{H}_{+1}$.

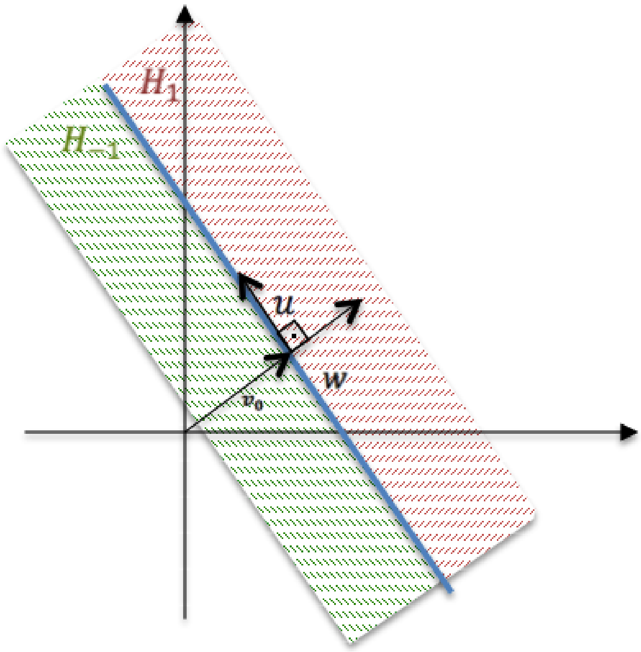


Figure 2.6: Separating hyperplane in $\mathbb{R}^2$. The decision boundary (blue) is defined by the normal vector $\boldsymbol{w}$ and an offset $b \in \mathbb{R}$. $\boldsymbol{v}_0$ is a point on the hyperplane, obtained by orthogonal projection of the origin. The plane separates $\mathbb{R}^2$ into two halfspaces $\mathcal{H}_{+1}$ $(\boldsymbol{w}^T\boldsymbol{x} + b > 0)$ and $\mathcal{H}_{-1}$ $(\boldsymbol{w}^T\boldsymbol{x} + b < 0)$, the decision regions of the corresponding linear discriminant.

There is some redundancy in our definition. If we replace $y(\boldsymbol{x})$ by $\alpha y(\boldsymbol{x})$, $\alpha > 0$, the linear classifier $f(\boldsymbol{x})$ does not change at all. If $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$ is a linear discriminant, we can always normalize $\boldsymbol{w}$ to become a unit vector: $\boldsymbol{w} \to \boldsymbol{w}_0 = \boldsymbol{w}/\|\boldsymbol{w}\|$, $b \to b_0 = b/\|\boldsymbol{w}\|$, and $y_0(\boldsymbol{x}) = \boldsymbol{w}_0^T\boldsymbol{x} + b_0$ induces the same classifier. It is therefore without loss of generality to restrict the weight vector $\boldsymbol{w}$ to be of unit norm, and this is often done in papers. The set

$$\mathcal{S}_d = \left\{ \boldsymbol{w} \;\middle|\; \|\boldsymbol{w}\| = 1 \right\} \subset \mathbb{R}^d$$

is called *unit hypersphere*, the set of all unit vectors in $\mathbb{R}^d$. Another "hyperconcept", in $\mathbb{R}^2$ this is the unit circle (draw it!), and in $\mathbb{R}^3$ the unit sphere.

How do we *learn* a classifier? Given a dataset of examples from the problem of interest, $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid \boldsymbol{x}_i \in \mathcal{X}, t_i \in \mathcal{T},\ i = 1, \dots, n\}$, we should pick a classifier which does well on $\mathcal{D}$. For example, if $f(\boldsymbol{x})$ is a linear classifier with weight vector $\boldsymbol{w}$ and bias parameter b, we should aim to adjust these parameters so to achieve few errors on $\mathcal{D}$. This process of fitting f to $\mathcal{D}$ is called *training.* A dataset used to train a classifier is called *training dataset.*

Suppose we are to train a linear classifier $f(\boldsymbol{x}) = \operatorname{sgn}(\boldsymbol{w}^T\boldsymbol{x} + b)$ on a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$. At least within this chapter, our goal will be to classify all training cases correctly, to find a hyperplane for which $\boldsymbol{x}_i \in \mathcal{H}_{t_i}$,

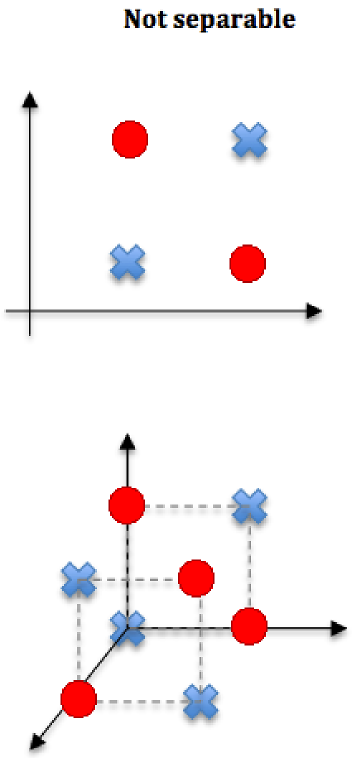


Figure 2.7: Left column: Datasets which are not linearly separable in $\mathbb{R}^2$ and $\mathbb{R}^3$ respectively. Right column: Linearly separable datasets.

or $f(\boldsymbol{x}_i) = t_i$, for all $i = 1, \dots, n$. A hyperplane with this property is called *separating hyperplane* for $\mathcal{D}$. Not all datasets have a separating hyperplane. A dataset $\mathcal{D}$ for which there is at least one separating hyperplane is called *linearly separable*. If no mistakes on the training set are permitted, the best we can hope for is to find a training algorithm which outputs a separating hyperplane for every linearly separable training set. In Figure 2.7, we depict non-separable datasets in the left, linearly separable datasets in the right column.

Feature Maps

Until now, we concentrated on classifiers whose discriminant functions are linear in the input point $\boldsymbol{x}$: $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$. However, this restriction seems somewhat artificial. We should allow ourselves some freedom to preprocess $\boldsymbol{x}$ before doing the linear combination. A *feature function* (or just *feature*) is a real-valued function $\mathcal{X} \to \mathbb{R}$. A *feature map* $\boldsymbol{\phi}(\boldsymbol{x})$ is a collection of p features $\phi_j(\boldsymbol{x})$, mapping $\mathcal{X}$ to $\mathbb{R}^p$ as $\boldsymbol{\phi}(\boldsymbol{x}) = [\phi_j(\boldsymbol{x})] \in \mathbb{R}^p$. In this context, $\mathcal{X}$ is called *input space*, while $\mathbb{R}^p$ is called *feature space* (a vector space containing the image $\boldsymbol{\phi}(\mathcal{X})$).
If nothing else is said, a feature map $\boldsymbol{\phi}(\boldsymbol{x})$ is fixed up front and does not have

to be adjusted to data at the time of learning a classifier.

Given a fixed feature map $\boldsymbol{\phi}(\boldsymbol{x})$, we generalize our definition of linear classifiers. A *linear discriminant function* (or just linear discriminant) is a real-valued function of $\boldsymbol{x}$ which is linear in the *feature vector* $\boldsymbol{\phi}(\boldsymbol{x})$: $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + b$. Here, $\boldsymbol{w} \in \mathbb{R}^p$ is called *weight vector*, $b \in \mathbb{R}$ *bias parameter*. A *linear classifier* is a classifier of the form $f(\boldsymbol{x}) = \operatorname{sgn}(y(\boldsymbol{x}))$, where $y(\boldsymbol{x})$ is a linear discriminant.

Here is an example. Maybe we can boost the performance of a digits classifier by allowing it to take dependencies between pixels in $\boldsymbol{x}$ into account, such as correlations of the form $x_j x_k$ for $j \neq k$. If $\boldsymbol{x} \in \mathbb{R}^d$ ($d = 28 \cdot 28$ for MNIST digits), consider the map of all linear and quadratic features:

$$\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ x_1 x_1 \\ \vdots \\ x_1 x_d \\ x_2 x_2 \\ \vdots \\ x_d x_d \end{bmatrix} = \begin{bmatrix} \boldsymbol{x} \\ [x_j x_k]_{j \leq k} \end{bmatrix} \in \mathbb{R}^{d(d+3)/2}.$$

You should work out as an exercise how the dimensionality $d(d+3)/2$ comes about. How does a linear discriminant for this feature map look like? The weight vector $\boldsymbol{w}$ lives in $\mathbb{R}^{d(d+3)/2}$. If we index its components by j for the linear, then jk ($j \leq k$) for the quadratic features, then

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + b = \sum_j w_j x_j + \sum_{j \leq k} w_{jk} x_j x_k + b.$$

This means that $y(\boldsymbol{x})$ is a *quadratic* function of $\boldsymbol{x}$. The decision boundary $\{y(\boldsymbol{x}) = 0\}$, a hyperplane in the feature space $\mathbb{R}^{d(d+3)/2}$, can take more complex forms in the input space $\mathbb{R}^d$ (namely, solutions to quadratic equations). In the same way, we could take into account higher-order dependencies between more than two components of $\boldsymbol{x}$, leading to discriminants $y(\boldsymbol{x})$ which are multivariate polynomials. Moreover, we do not have to include all interactions, but can restrict correlation terms to pixels spatially close in the bitmap, thus reducing the feature space dimensionality. To conclude, choosing appropriate feature maps can greatly enhance the flexibility of linear classifiers, resulting in decision boundaries which can be arbitrarily far from linear hyperplanes.

Given that we step from $\boldsymbol{x}$ to $\boldsymbol{\phi}(\boldsymbol{x})$, we can just as well incorporate the bias parameter b into the weight vector, by appending a constant 1 to the feature map. If $\tilde{\boldsymbol{w}} = [\boldsymbol{w}^T, b]^T$, $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}) = [\boldsymbol{x}^T, 1]^T$, then $\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + b = \tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{\phi}}(\boldsymbol{x})$. In this new feature space, all our hyperplanes contain $\mathbf{0}$, therefore are subspaces. Whether or not the bias parameter is kept separate, is often up to the taste of the community. We will frequently use the bias-free parameterization for simplicity of notation (and of geometry), but will keep b explicit in cases where incorporating it into $\boldsymbol{\phi}(\boldsymbol{x})$ would change the underlying method.

2.3 Perceptron Classifiers

The perceptron has been an early cornerstone of brain-inspired machine learning. Its history is detailed in several books. I recommend you have a look at Figure 4.8 in [6, Sect. 4.1.7]. For the purpose of this course, the perceptron is simply a binary linear classifier $f(\boldsymbol{x}) = \mathrm{sgn}(y(\boldsymbol{x}))$, $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$, which is trained on data in a particular way. In this section, we will absorb a bias parameter into $\boldsymbol{w}$, so that decision hyperplanes in feature space always contain the origin $\mathbf{0}$.

Suppose we are given a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$, where $t_i \in \{-1, +1\}$. We will use the simplifying notation $\boldsymbol{\phi}_i = \boldsymbol{\phi}(\boldsymbol{x}_i) \in \mathbb{R}^p$. Let us assume for now that $\mathcal{D}$ is linearly separable in our feature space: there exists a separating linear classifier. The perceptron algorithm is a simple method for finding such a classifier. It has the following properties:

- Iterative: The algorithm cycles over the data points in some random ordering. Visiting $(\boldsymbol{\phi}_i, t_i)$, it extracts some information, based on which $\boldsymbol{w}$ may be updated.
- Local updates: Each update depends only on the pattern currently visited, not on what has been encountered previously.
- Updates only for misclassified patterns: When we visit a pattern $(\boldsymbol{\phi}_i, t_i)$ which is correctly classified by the current discriminant,
$$\mathrm{sgn}(\boldsymbol{w}^T\boldsymbol{\phi}_i) = t_i, \quad \text{or simpler} : t_i\boldsymbol{w}^T\boldsymbol{\phi}_i > 0,$$
then $\boldsymbol{w}$ is not changed.

Recall our geometric picture. $\boldsymbol{w}$ defines a hyperplane, separating $\mathbb{R}^p$ into two halfspaces, $\mathcal{H}_{+1}$ and $\mathcal{H}_{-1}$, with $\boldsymbol{w}$ pointing into the positive halfspace $\mathcal{H}_{+1}$. It makes a mistake on the i-th pattern if and only if $\boldsymbol{\phi}_i$ does not lie in its halfspace $\mathcal{H}_{t_i}$, or simpler if the *signed pattern* $t_i\boldsymbol{\phi}_i$ lies in the negative $\mathcal{H}_{-1}$. Intuitively, the *angle* between $t_i\boldsymbol{\phi}_i$ and $\boldsymbol{w}$ is more than a right one, larger than 90°.

Here is what the inner product is really doing: if $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d$, the *angle* between them is θ, where
$$\cos\theta = \frac{\boldsymbol{a}^T\boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|} \in [-1, 1].$$
If we consider normalized vectors only, the inner product is simply a way to measure the angle. The *Cauchy-Schwarz inequality* is related to this definition:
$$\left|\boldsymbol{a}^T\boldsymbol{b}\right| \le \|\boldsymbol{a}\|\|\boldsymbol{b}\|.$$

In our linear classification problem, we can aim to correct mistakes by decreasing the angle between $\boldsymbol{w}$ and signed patterns $t_i\boldsymbol{\phi}_i$. In other words, we should increase $t_i\boldsymbol{w}^T\boldsymbol{\phi}_i$ on patterns for which it is negative. A simple way to do so is to add the signed pattern $t_i\boldsymbol{\phi}_i$ to $\boldsymbol{w}$:
$$t_i\boldsymbol{\phi}_i^T(\boldsymbol{w} + t_i\boldsymbol{\phi}_i) = t_i\boldsymbol{\phi}_i^T\boldsymbol{w} + \|\boldsymbol{\phi}_i\|^2 > t_i\boldsymbol{\phi}_i^T\boldsymbol{w}.$$

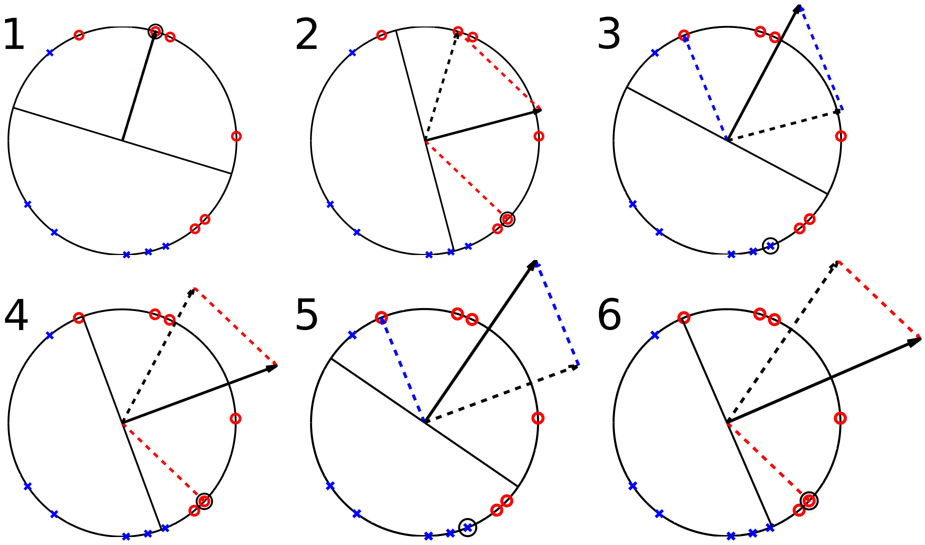


Figure 2.8: Perceptron algorithm on $\mathbb{R}^2$ vectors. Blue crosses represent -1, red circles $+1$ patterns. For each iteration, the update pattern is marked by black circle. Shown is signed pattern as well as old and new weight vector $\boldsymbol{w}$.

Algorithm 1 Perceptron algorithm for training a linear classifier.

```
Discriminant is $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$. Use normalized patterns:
$\tilde{\boldsymbol{\phi}}_i = \boldsymbol{\phi}(\boldsymbol{x}_i)/\|\boldsymbol{\phi}(\boldsymbol{x}_i)\|$. Initialize $\boldsymbol{w} \leftarrow \mathbf{0}$.
repeat
  for $i \in \{1,\dots,n\}$ (random ordering) do
    if $t_i\boldsymbol{w}^T\tilde{\boldsymbol{\phi}}_i \le 0$ (pattern misclassified) then
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + t_i\tilde{\boldsymbol{\phi}}_i$
    end if
  end for
until no update on any $i \in \{1,\dots,n\}$
```

In fact, this is all we need for the perceptron algorithm, which is given in Algorithm 1.

In practice, we run random sweeps over all data points, terminating if there is no more mistake during a complete sweep. One subtle point is that we normalize all feature vectors $\boldsymbol{\phi}(\boldsymbol{x}_i)$ before starting the algorithm. Convince yourself that this cannot do any harm: $\boldsymbol{w}$ is a separating hyperplane for $\mathcal{D}$ if and only if it is one for the set obtained by normalizing each $\boldsymbol{\phi}(\boldsymbol{x}_i)$. We will see in Section 2.3.2 that normalization does not only simplify the analysis, but also leads to faster convergence of the algorithm. A run of the algorithm is shown in Figure 2.8.

2.3.1 The Perceptron Convergence Theorem

Convergence? There is no a priori reason to believe that the perceptron algorithm will work. While an update decreases the angle to a misclassified (signed) pattern, these angles can at the same time increase for other patterns not currently visited (see Figure 2.8).

Theorem 2.1 (Perceptron Convergence) *Let $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$ be a binary classification dataset, $t_i \in \{-1, 1\}$, which is linearly separable in a feature space given by $\boldsymbol{\phi}(\boldsymbol{x})$. Then, the perceptron algorithm terminates after finitely many updates and outputs a separating discriminant. More precisely, let $\boldsymbol{w}_*$ be the unit norm weight vector for some separating hyperplane: $t_i \boldsymbol{w}_*^T \tilde{\boldsymbol{\phi}}_i > 0$ for all $i = 1, \dots, n$, where $\tilde{\boldsymbol{\phi}}_i = \boldsymbol{\phi}(\boldsymbol{x}_i)/\|\boldsymbol{\phi}(\boldsymbol{x}_i)\|$. The perceptron algorithm terminates after no more than $1/\gamma^2$ updates, where*

$$\gamma = \min_{i=1,\dots,n} t_i \boldsymbol{w}_*^T \tilde{\boldsymbol{\phi}}_i. \tag{2.1}$$

Beware that $\boldsymbol{w}_*$ determines just *some* separating hyperplane, not necessarily the one which is output by the algorithm. $\boldsymbol{w}_*$ has to exist because $\mathcal{D}$ is separable, it is necessarily only in order to define γ, thus to bound the number of updates. γ is a magic number, we come back to it shortly.

Recall that $\boldsymbol{w}_*$ and all $\tilde{\boldsymbol{\phi}}_i$ are unit vectors. Denote by $\boldsymbol{w}_k$ the weight vector after the k-th update of the algorithm, and $\boldsymbol{w}_0 = \boldsymbol{0}$. $\boldsymbol{w}_k$ is not a unit vector, its norm will grow during the course of the algorithm. The proof is geometrically intuitive. We will monitor two numbers, the inner product $\boldsymbol{w}_*^T \boldsymbol{w}_k$ and the length $\|\boldsymbol{w}_k\|$. Why these two? The larger $\boldsymbol{w}_*^T \boldsymbol{w}_k$ the better, because we want to decrease the angle between $\boldsymbol{w}_*$ and $\boldsymbol{w}_k$. However, the inner product could simply grow by $\boldsymbol{w}_k$ becoming longer, so we need to take care of its length $\|\boldsymbol{w}_k\|$ (while $\|\boldsymbol{w}_*\| = 1$). Now, both numbers may increase with k, but the first does so much faster than the second. By Cauchy-Schwarz (or the definition of angle), $\boldsymbol{w}_*^T \boldsymbol{w}_k \le \|\boldsymbol{w}_k\|$, so this process cannot go on forever. Let us zoom into the update $\boldsymbol{w}_k \to \boldsymbol{w}_{k+1}$, which happens (say) on pattern $(\tilde{\boldsymbol{\phi}}_i, t_i)$. First,

$$\boldsymbol{w}_*^T \boldsymbol{w}_{k+1} = \boldsymbol{w}_*^T (\boldsymbol{w}_k + t_i \tilde{\boldsymbol{\phi}}_i) = \boldsymbol{w}_*^T \boldsymbol{w}_k + t_i \boldsymbol{w}_*^T \tilde{\boldsymbol{\phi}}_i \ge \boldsymbol{w}_*^T \boldsymbol{w}_k + \gamma$$

by definition of γ. At each update, $\boldsymbol{w}_*^T \boldsymbol{w}_k$ increases by at least γ, so after M updates: $\boldsymbol{w}_*^T \boldsymbol{w}_k \ge M\gamma$. Second, let us expand a squared norm (Section 2.1.1):

$$\|\boldsymbol{w}_{k+1}\|^2 = \|\boldsymbol{w}_k + t_i \tilde{\boldsymbol{\phi}}_i\|^2 = \|\boldsymbol{w}_k\|^2 + \underbrace{2 t_i \boldsymbol{w}_k^T \tilde{\boldsymbol{\phi}}_i}_{\le 0 \text{ (mistake!)}} + \underbrace{\|t_i \tilde{\boldsymbol{\phi}}_i\|^2}_{=1} \le \|\boldsymbol{w}_k\|^2 + 1.$$

Crucially, the "cross-talk" $t_i \boldsymbol{w}_k^T \tilde{\boldsymbol{\phi}}_i$ is nonpositive, because $\boldsymbol{w}_k$ errs on the i-th pattern (otherwise, there would be no update). Therefore, at each update, $\|\boldsymbol{w}_k\|^2$ increases by at most 1, so after M updates: $\|\boldsymbol{w}_k\| \le \sqrt{M}$. Now,

$$\boldsymbol{w}_*^T \boldsymbol{w}_k \le \|\boldsymbol{w}_k\| \quad \Rightarrow \quad M\gamma \le \sqrt{M} \quad \Leftrightarrow \quad M \le 1/\gamma^2.$$

What does this mean? γ is a fixed number, and we must have $M \le 1/\gamma^2$, where M is the number of updates to the weight vector. Since the perceptron algorithm updates on misclassified patterns only, this situation arises at most $1/\gamma^2$ times. After that, all patterns are classified correctly, and the algorithm stops. This concludes the proof.

The perceptron algorithm is probably the simplest procedure for linear classification one can think of. Nevertheless, it provably works on every single linearly separable dataset. However, there are always routes for improvement. What happens if $\mathcal{D}$ is not linearly separable? In such a case, the perceptron algorithm runs forever. Moreover, if γ is very small but positive, it may run for many iterations. It is also difficult to generalize the perceptron algorithm to multi-way classification problems. Nevertheless, its simplicity and computational efficiency render the perceptron algorithm an attractive choice in practice.

2.3.2 Normalization of Feature Vectors

Why do we normalize the feature vectors $\boldsymbol{\phi}_i = \boldsymbol{\phi}(\boldsymbol{x}_i)$ in the perceptron algorithm, using $\tilde{\boldsymbol{\phi}}_i = \boldsymbol{\phi}_i/\|\boldsymbol{\phi}_i\|$ instead of $\boldsymbol{\phi}_i$? We already convinced ourselves that doing so does not change the set of all separating hyperplanes, so normalization should not make things worse. In fact, a closer look at the proof of Theorem 2.1 suggests that normalization should be beneficial, making the perceptron algorithm converge faster in general. To this end, let us try to replicate the proof with general unnormalized $\boldsymbol{\phi}_i$. $\boldsymbol{w}_*$ will still be a unit vector. While in the normalized case, we needed one number:

$$\gamma = \min_{i=1,\dots,n} \frac{t_i \boldsymbol{w}_*^T \boldsymbol{\phi}_i}{\|\boldsymbol{\phi}_i\|},$$

now we need two of them:

$$\tilde{\gamma} = \min_{i=1,\dots,n} t_i \boldsymbol{w}_*^T \boldsymbol{\phi}_i, \quad P = \max_{i=1,\dots,n} \|\boldsymbol{\phi}_i\|.$$

The first step remains unchanged: $\boldsymbol{w}_*^T \boldsymbol{w}_{k+1} \ge \boldsymbol{w}_*^T \boldsymbol{w}_k + \tilde{\gamma}$. For the second step:

$$\|\boldsymbol{w}_{k+1}\|^2 = \|\boldsymbol{w}_k\|^2 + \underbrace{2t_i \boldsymbol{w}_k^T \boldsymbol{\phi}_i}_{\le 0 \text{ (mistake!)}} + \|t_i \boldsymbol{\phi}_i\|^2 \le \|\boldsymbol{w}_k\|^2 + P^2.$$

Proceeding as above:

$$\boldsymbol{w}_*^T \boldsymbol{w}_k \le \|\boldsymbol{w}_k\| \quad \Rightarrow \quad M\tilde{\gamma} \le \sqrt{MP^2} \quad \Leftrightarrow \quad M \le (P/\tilde{\gamma})^2.$$

We need to show that

$$(P/\tilde{\gamma})^2 \ge (1/\gamma)^2 \quad \Leftrightarrow \quad \tilde{\gamma}/P \le \gamma.$$

Now,

$$\tilde{\gamma}/P = \min_{i=1,\dots,n} t_i \boldsymbol{w}_*^T (\boldsymbol{\phi}_i/P) \le \min_{i=1,\dots,n} t_i \boldsymbol{w}_*^T (\boldsymbol{\phi}_i/\|\boldsymbol{\phi}_i\|) = \gamma,$$

using that $P \ge \|\boldsymbol{\phi}_i\|$ and $t_i \boldsymbol{w}_*^T \boldsymbol{\phi}_i > 0$ for all i. This means that the upper bound on the number of updates in the perceptron algorithm never increases (and typically decreases) if we normalize the feature vectors. The message is: *normalize your feature vectors before running the perceptron algorithm.* You should try it out for yourself, seeing is believing. Run the algorithm on normalized vectors, then rescale some of them and compare the number of updates the algorithm needs until convergence.

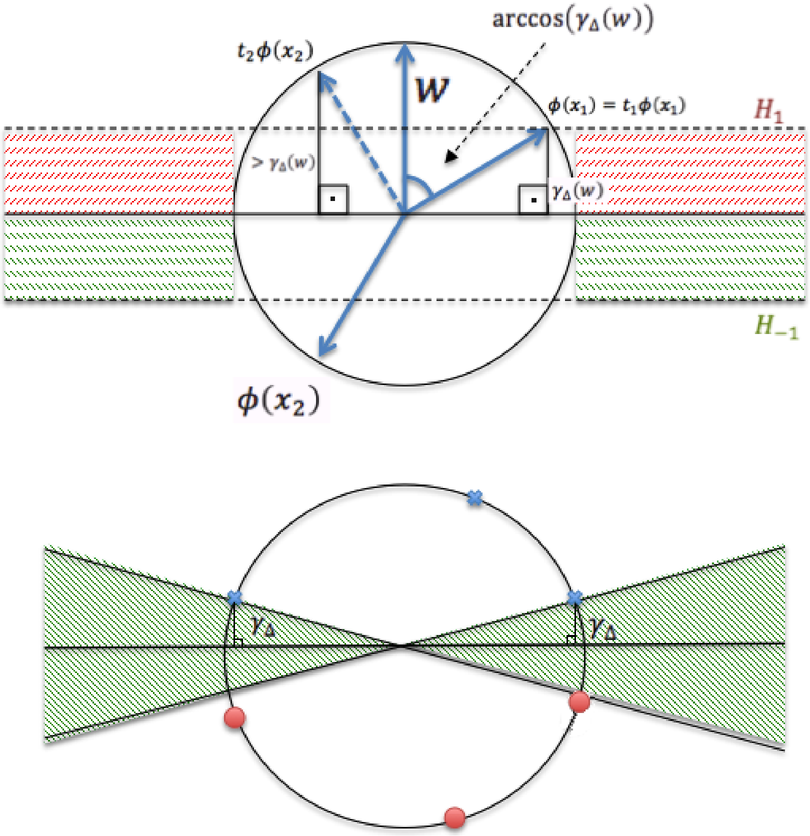


Figure 2.9: Illustration of margin as maximum distance to closest patterns (top) or as angle swept out by all possible separating hyperplanes (bottom). Details given in the text.

2.3.3 The Margin of a Dataset (*)

This section can be skipped at a first reading. We will require the margin concept again in Chapter 9, when we introduce large margin classification and support vector machines.

Recall the magic number γ from Theorem 2.1. Intuitively, γ quantifies the difficulty of finding a separating hyperplane (the smaller, the more difficult). For general vectors, we have

$$\gamma_{\mathcal{D}}(\boldsymbol{w}) = \min_{i=1,\dots,n} \frac{t_i \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i)}{\|\boldsymbol{w}\| \|\boldsymbol{\phi}(\boldsymbol{x}_i)\|}.$$

Note that $\boldsymbol{w}$ is a separating hyperplane for $\mathcal{D}$ if and only if $\gamma_{\mathcal{D}}(\boldsymbol{w}) > 0$. Aiming for the best bound in Theorem 2.1, we choose[3]

$$\gamma_{\mathcal{D}} = \max_{\boldsymbol{w}\in\mathbb{R}^p\setminus\{\boldsymbol{0}\}} \gamma_{\mathcal{D}}(\boldsymbol{w}) = \max_{\boldsymbol{w}\in\mathbb{R}^p\setminus\{\boldsymbol{0}\}} \min_{i=1,\dots,n} \frac{t_i \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i)}{\|\boldsymbol{w}\|\|\boldsymbol{\phi}(\boldsymbol{x}_i)\|}.$$

$\gamma_{\mathcal{D}}$ is called the *margin of* $\mathcal{D}$ for normalized patterns. A dataset $\mathcal{D}$ is linearly separable if and only if $\gamma_{\mathcal{D}} > 0$. In that case, the margin determines the best upper bound on number of perceptron algorithm updates we can obtain by the proof of Theorem 2.1.

What is the geometrical meaning of the margin? Pick a unit vector $\boldsymbol{w}$ so that $\gamma_{\mathcal{D}}(\boldsymbol{w}) > 0$. Then, $\arccos(\gamma_{\mathcal{D}}(\boldsymbol{w}))$ is the largest angle between $\boldsymbol{w}$ and any signed normalized pattern $t_i\tilde{\boldsymbol{\phi}}(\boldsymbol{x}_i)$. Since $\boldsymbol{w}$ is orthogonal to the separating hyperplane, $\pi/2 - \arccos(\gamma_{\mathcal{D}}(\boldsymbol{w}))$ is the *smallest angle between any pattern* $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}_i)$ *and the hyperplane* (Figure 2.9, bottom). For unit vectors, angle and distance are closely related. The distance between $\boldsymbol{a} \in \mathcal{S}_p$ (the unit hypersphere, Section 2.2) and the hyperplane is $\delta = \|\boldsymbol{a} - \hat{\boldsymbol{a}}\|$, where $\hat{\boldsymbol{a}}$ is the orthogonal projection of $\boldsymbol{a}$ onto the hyperplane (more about orthogonal projection in Section 4.2.2). What do we know about $\hat{\boldsymbol{a}}$? First, $\boldsymbol{w}^T\hat{\boldsymbol{a}} = 0$, as it lies on the hyperplane. Second, $\boldsymbol{a} = \hat{\boldsymbol{a}} + \delta\boldsymbol{w}$, since $\boldsymbol{a} - \hat{\boldsymbol{a}}$ is orthogonal to the hyperplane, therefore pointing along $\boldsymbol{w}$. Therefore, $\boldsymbol{w}^T\boldsymbol{a} = \boldsymbol{w}^T\hat{\boldsymbol{a}} + \delta\|\boldsymbol{w}\|^2 = \delta$. If we apply this derivation to $\boldsymbol{a} = t_i\tilde{\boldsymbol{\phi}}(\boldsymbol{x}_i)$, we see that $\gamma_{\mathcal{D}}(\boldsymbol{w})$ is the *smallest distance between any pattern* $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}_i)$ *and the hyperplane.* $\gamma_{\mathcal{D}}(\boldsymbol{w})$ quantifies the space between the separating hyperplane and the closest pattern (Figure 2.9, top).

The margin $\gamma_{\mathcal{D}} = \max_{\boldsymbol{w}\in\mathbb{R}^p\setminus\{\boldsymbol{0}\}} \gamma_{\mathcal{D}}(\boldsymbol{w})$ quantifies the separation between the sets of patterns for each class $-1, +1$ (assuming that $\gamma_{\mathcal{D}} > 0$). It is the largest room to move for any separating hyperplane. Imagine you have to separate the training set not by a thin hyperplane, but by a *slab* with as large a width as possible. The maximum width you can attain is $2\gamma_{\mathcal{D}}$ (Figure 2.9, top). Or imagine all possible separating hyperplanes for $\mathcal{D}$. Then, $\pi - 2\arccos(\gamma_{\mathcal{D}})$ is the largest signed angle[4] between any two of them (Figure 2.9, bottom). The larger $\gamma_{\mathcal{D}}$, the easier it is to linearly separate $\mathcal{D}$.

2.4 Error Function Minimization. Gradient Descent

We have worked out the simple perceptron algorithm (Algorithm 1) for finding a separating hyperplane if there exists one. Moreover, we obtained a precise geometrical understanding of this algorithm and analyzed its convergence behaviour. There are open issues. Most troubling perhaps, if our data $\mathcal{D}$ is not linearly separable, then the algorithm never terminates. How can we fix that? What about more general situations, such as multi-way classification or nonlinear discrimination? We just have an algorithm, but do not really know what it is doing. Which mathematical problem does it solve?

[3] Why max over $\boldsymbol{w} \in \mathbb{R}^p \setminus \{\boldsymbol{0}\}$ and not sup (supremum)? The supremum will always be attained for some $\boldsymbol{w}$. To see this, note that we restrict it to the unit hypersphere $\boldsymbol{w} \in \mathcal{S}_d$, which is compact, and the argument is continuous in $\boldsymbol{w}$.

[4] In the context of this statement, the signed angle between two hyperplanes is the angle between their normal vectors, which can lie in $[0, \pi]$.

As computer scientists, we love *algorithms*. Yet, the modern approach to science and engineering is to think about *problems*[5] instead. Once these are well characterized and understood, we may search for algorithms to solve them well. In this context, a “problem” is an abstract formulation of what we really want to do (for example, classifying those MNIST digits), which is amenable to computationally tractable mathematical treatment. In the context of much of machine learning, this translates to *mathematical optimization problem*.

Error Functions

Recall that our problem is to find a linear classifier $f(\boldsymbol{x}) = \mathrm{sgn}(y(\boldsymbol{x}))$, $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$, which separates handwritten 8s from 9s. To this end, we use the MNIST subset $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$ as training dataset: if $f(\boldsymbol{x})$ commits few errors on $\mathcal{D}$, it should work well for other patterns. Let us formalize this idea in an optimization problem, by setting up an error function $E(\boldsymbol{w})$ of the weight vector, so that small $E(\boldsymbol{w})$ translates to good performance on the dataset. We could minimize the number of errors:

$$E_{0/1}(\boldsymbol{w}) = \sum_{i=1}^{n} \mathrm{I}_{\{f(\boldsymbol{x}_i) \neq t_i\}} = \sum_{i=1}^{n} \mathrm{I}_{\{t_i \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) \leq 0\}}.$$

Recall that $\mathrm{I}_{\{\mathcal{A}\}} = 1$ if $\mathcal{A}$ is true, $\mathrm{I}_{\{\mathcal{A}\}} = 0$ otherwise. While this error function formalizes our objective well, its minimization is computationally[6] difficult. $E_{0/1}(\boldsymbol{w})$ is piecewise constant in $\boldsymbol{w}$, so that local information such as the gradient cannot be used. For a better choice, we turn to the oldest and most profound concept of scientific computing, the one with which Gauss and Legendre started it all: *least squares estimation*. Consider the *squared error* function

$$\begin{aligned} E(\boldsymbol{w}) &= \frac{1}{2} \sum_{i=1}^{n} (y(\boldsymbol{x}_i) - t_i)^2 = \frac{1}{2} \sum_{i=1}^{n} \left(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) - t_i \right)^2 \\ &= \frac{1}{2} \sum_{i=1}^{n} \left(\sum\nolimits_{j=1}^{d} w_j \phi_j(\boldsymbol{x}_i) - t_i \right)^2. \end{aligned}$$

The significance of the prefactor 1/2 will become clear below. This error function is minimized by achieving $t_i \approx \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i)$ across all training patterns. Whenever $(\boldsymbol{x}_i, t_i)$ is misclassified, i.e. $t_i y(\boldsymbol{x}_i) < 0$, then $(y(\boldsymbol{x}_i) - t_i)^2 > 1$: errors are penalized even stronger than in $E_{0/1}(\boldsymbol{w})$. By pushing $y(\boldsymbol{x}_i)$ closer to t_i, the mistake is corrected. All in all, $E(\boldsymbol{w})$ seems a reasonable criterion to minimize in order to learn a classifier on $\mathcal{D}$.

The major advantage of $E(\boldsymbol{w})$ is that its minimization is computationally tractable and well understood in terms of properties and algorithms. Once more, geometry helps our understanding. Build the vector of targets $\boldsymbol{t} = [t_i] \in$

[5] As we proceed with this course, we will see that it is even more useful to think about *models* of the domain of interest. These give rise, in an almost automatical fashion, to the problems we should really solve, and then we can be clever about algorithms. But for now, let us stick with the *problems*.

[6] The problem $\min_{\boldsymbol{w}} E_{0/1}(\boldsymbol{w})$ is NP-hard to solve in general.

$\{-1,+1\}^n$ and the so-called *design matrix*

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^T \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_n)^T \end{bmatrix} = \begin{bmatrix} \phi_1(\boldsymbol{x}_1) & \dots & \phi_p(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ \phi_1(\boldsymbol{x}_n) & \dots & \phi_p(\boldsymbol{x}_n) \end{bmatrix} \in \mathbb{R}^{n \times p}.$$

Feeling rusty about matrices? Visit Section 2.4.3. Now, $\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{\phi}(\boldsymbol{x}_i)^T\boldsymbol{w}$ is simply the i-th element of $\boldsymbol{\Phi}\boldsymbol{w} \in \mathbb{R}^n$. In other words, if we define $\boldsymbol{y} = [y(\boldsymbol{x}_i)] \in \mathbb{R}^n$, meaning that $y_i = y(\boldsymbol{x}_i) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)$, then $\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{w}$, therefore

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y(\boldsymbol{x}_i) - t_i)^2 = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{t}\|^2 = \frac{1}{2}\|\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\|^2.$$

The squared error is simply half of the squared Euclidean distance between the vector of targets $\boldsymbol{t}$ and the vector of predictions $\boldsymbol{y}$, which in itself is a linear transform of $\boldsymbol{w}$, given by the design matrix $\boldsymbol{\Phi}$. We will frequently use this representation of $E(\boldsymbol{w})$ in the sequel. Converting expressions from familiar sums over i and j into vectors, matrices and squared norms may seem unfamiliar, but once you get used to it, you will appreciate the immense value of "vectorization". First, no more opportunities to get the indices wrong. Debugging is built in: if you make a mistake, you usually end up with matrix multiplications of incompatible dimensions. More important, we can use our geometrical intuition about lengths, orthogonality and projections. Finally, writing $E(\boldsymbol{w})$ as squared norm immediately leads to powerful methods to solve the minimization problem $\min_{\boldsymbol{w}} E(\boldsymbol{w})$ with tools from (numerical) linear algebra, as we will see in Chapter 4.

2.4.1 Gradient Descent

How can we solve the optimization problem $\min_{\boldsymbol{w}} E(\boldsymbol{w})$, more specifically find a minimizer $\boldsymbol{w}_*$ so that $E(\boldsymbol{w}_*) = \min_{\boldsymbol{w}} E(\boldsymbol{w})$? In this section, we will specify a simple method which can be applied to many other error functions as well. A few points about the least squares problem. First, a minimizer exists. This is because $E(\boldsymbol{w})$ is lower bounded (by zero), continuous, and the domain $\boldsymbol{w} \in \mathbb{R}^p$ is closed[7]. Second, the error function $E(\boldsymbol{w})$ is continuously differentiable everywhere. The mental picture you should develop is that of an error function landspace, where $\boldsymbol{w}$ is your location and $E(\boldsymbol{w})$ is height above sea level. To go down, a good idea is to take a small step in the *direction of steepest descent*. Standing at $\boldsymbol{w}$, for a tiny $\varepsilon > 0$ you step to $\boldsymbol{w} + \varepsilon\boldsymbol{d}$, where $\|\boldsymbol{d}\| = 1$ (unit vector). The steepest descent direction is that $\boldsymbol{d}$ for which $E(\boldsymbol{w}) - E(\boldsymbol{w} + \varepsilon\boldsymbol{d})$ is largest.

Let us start with a one-dimensional example: $E(w)$ for $w \in \mathbb{R}$. There are not many directions in $\mathbb{R}$, in fact only $d = -1, +1$. By Taylor's theorem, $E(w \pm \varepsilon) = E(w) \pm E'(w)\varepsilon + O(\varepsilon^2)$, where $E'(w) = dE(w)/dw$ is the derivative. This is simply a way of saying that $\lim_{\varepsilon \to 0}(E(w + \varepsilon) - E(w))/\varepsilon = E'(w)$, the definition of the derivative. Suppose that $E'(w) \neq 0$. For very small $\varepsilon > 0$, $E(w) - E(w + d\varepsilon) = \varepsilon(-E'(w)d + O(\varepsilon))$, which is positive for the direction

[7] We also need the domain to be bounded. It is easy to see that $E(\boldsymbol{w}) \to \infty$ for $\|\boldsymbol{w}\| \to \infty$, so we can restrict ourselves to some large enough hyperball.

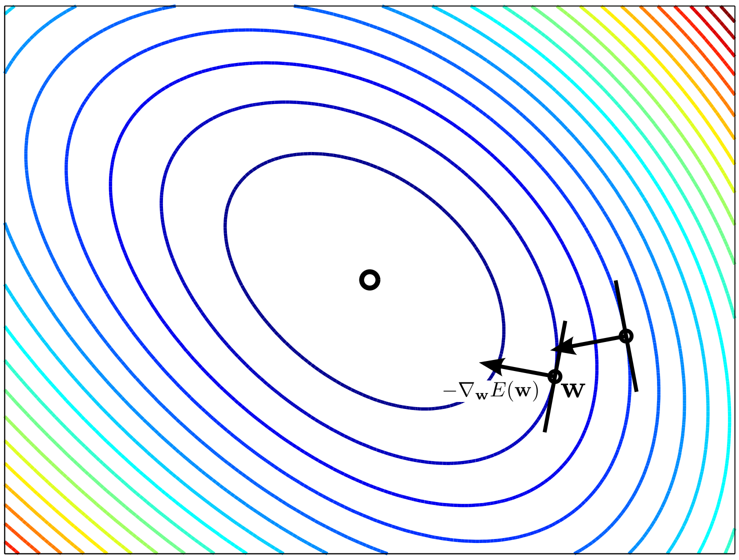


Figure 2.10: Optimization by steepest descent, following the negative gradient direction.

$d = \mathrm{sgn}(-E'(w))$ only: the sign of the negative derivative. For $\boldsymbol{w} \in \mathbb{R}^d$, we use the multivariate Taylor's theorem:

$$\begin{aligned} &E(\boldsymbol{w} + \varepsilon \boldsymbol{d}) = E(\boldsymbol{w}) + \varepsilon (\nabla_{\boldsymbol{w}} E(\boldsymbol{w}))^T \boldsymbol{d} + O(\varepsilon^2), \\ &\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = [\partial E(\boldsymbol{w}) / \partial w_j] \in \mathbb{R}^p. \end{aligned}$$

Here, $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$ is the *gradient* of $E(\boldsymbol{w})$ at $\boldsymbol{w}$. As in the univariate case, we find the direction $\boldsymbol{d}$ of steepest descent as the maximizer of $E(\boldsymbol{w}) - E(\boldsymbol{w} + \varepsilon \boldsymbol{d}) = \varepsilon(-\nabla_{\boldsymbol{w}} E(\boldsymbol{w}))^T \boldsymbol{d} + O(\varepsilon^2)$ as $\varepsilon \to 0$. By the Cauchy-Schwarz inequality, $(-\nabla_{\boldsymbol{w}} E(\boldsymbol{w}))^T \boldsymbol{d}$ is maximized by $\boldsymbol{d}$ being parallel to $-\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$: the direction of steepest descent is

$$\boldsymbol{d} = \frac{-\nabla_{\boldsymbol{w}} E(\boldsymbol{w})}{\|\nabla_{\boldsymbol{w}} E(\boldsymbol{w})\|},$$

the *direction of the negative gradient.* To descend most steeply from $E(\boldsymbol{w})$, we compute the gradient $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$ and then march precisely in the opposite direction. This is the basis for *gradient descent optimization.* In practice, an iteration works as follows:

- Compute the gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} E(\boldsymbol{w})$.
- Pick a step size $\eta > 0$. Update $\boldsymbol{w}' = \boldsymbol{w} - \eta \boldsymbol{g}$.

The step size η is typically kept constant for many iterations, which corresponds to scaling the steepest descent direction by the gradient size $\|\nabla_{\boldsymbol{w}} E(\boldsymbol{w})\|$. We take large[8] steps as long as much progress can be made, but slow down when the terrain gets flatter. If $\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = \mathbf{0}$, we have reached a stationary point of $E(\boldsymbol{w})$, and gradient descent terminates. Whether or not this means that we

[8] This does not always work well, and we will discuss improvements shortly.

have found a minimum point of $E(\boldsymbol{w})$ (a local or even a global one), depends on characteristics of $E(\boldsymbol{w})$ [3]. Only so much: for the squared error $E(\boldsymbol{w})$ over linear functions, we will have found a global minimum point in this case (see Section 4.2).

Will this method converge (and how fast)? How do I choose the step size η? Shall I modify η as I go? Answers to these questions depend ultimately on properties of $E(\boldsymbol{w})$. For the squared error, they are well understood. Many results have been obtained in mathematical optimization and machine learning. A clear account is given in [3].

Gradient for Squared Error

What is the gradient for the squared error $E(\boldsymbol{w})$? Let us move in steps. First,

$$\frac{\partial E}{\partial y_i} = \frac{1}{2}\frac{\partial}{\partial y_i}(y_i - t_i)^2 = y_i - t_i \quad \Rightarrow \quad \nabla_{\boldsymbol{y}} E = \boldsymbol{y} - \boldsymbol{t} = \boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}.$$

This is why we use the prefactor 1/2. Second, $y_i = \boldsymbol{\phi}(\boldsymbol{x}_i)^T\boldsymbol{w}$, so that $\nabla_{\boldsymbol{w}} y_i = \boldsymbol{\phi}(\boldsymbol{x}_i)$. Using the chain rule of differential calculus:

$$\nabla_{\boldsymbol{w}} E = \sum_{i=1}^{n} \frac{\partial E}{\partial y_i}\nabla_{\boldsymbol{w}} y_i = \sum_{i=1}^{n}(y_i - t_i)\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{\Phi}^T(\boldsymbol{y} - \boldsymbol{t}) = \boldsymbol{\Phi}^T\left(\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\right).$$

As more complicated settings lie ahead, you should train yourself to do derivatives in little chunks, exploiting this fabulous divide and conquer rule. The expression for $\nabla_{\boldsymbol{w}} E$ is something we will see over and over again. $\boldsymbol{y} - \boldsymbol{t} = \boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}$ is called *residual vector*, it is the difference between prediction and true target for each data case. The gradient is obtained by mapping the residuals back to the weights, multiplying by $\boldsymbol{\Phi}^T$. In a concrete sense developed in Section 4.2, we project that part of the residual error we can do anything about by changing $\boldsymbol{w}$.

Problems, Error Functions and Algorithms

We have reformulated linear binary classification as the optimization problem of minimizing the squared error function $E(\boldsymbol{w})$, which we can do in practice by gradient descent optimization. The importance of this step will become apparent as we move to more challenging problems. The perceptron algorithm is nice, but it solves precisely one problem. Modify the latter a bit, and you are stuck. For your next machine learning problem to solve, it is easier to design an appropriate error function and optimization problem, then to pick a solver for it "off the shelf", or at least start from 200 years of prior experience in this domain. Our guiding principles will be:

- For the most part, machine learning is about mapping real-world classification, estimation or decision-making problems to *mathematical optimization problems*, which can be solved by robust, well understood and efficient algorithms.

- We separate three things: (a) the real-world problem motivating our efforts, (b) the optimization problem we try to map it to, and (c) the method for solving the optimization problem. In this section, (a) we would like to classify handwritten 8s versus 9s. To this end, (b) we minimize the squared error function $E(\boldsymbol{w})$, and (c) we do so by gradient descent optimization. The rationale should be obvious: the mapping from one to the other is not one to one. If we mix up the problem and an algorithm to solve it, we will miss out on other algorithms solving the same problem in a better way.

2.4.2 Online and Batch Learning. Perceptron Algorithm as Gradient Descent

Note that the squared error function $E(\boldsymbol{w})$ is a sum of terms $(y_i - t_i)^2/2$, one for each data point $(\boldsymbol{x}_i, t_i)$. This additive structure occurs very frequently in machine learning:

$$E(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} E_i(\boldsymbol{w}).$$

Note that in this section, we consider error functions to be normalized by the number n of training points. The structure implies that in order to compute the gradient $\nabla_{\boldsymbol{w}} E$, we have to run an accumulation loop over the whole dataset. If the number of data points n is very large, gradient descent looks a whole lot less attractive. To ensure proper convergence, the step size η must be rather small, yet every update requires a run over all training points. Such optimization algorithms are called *batch methods*: for every update of $\boldsymbol{w}$, the whole dataset (or at least large batches thereof) has to be processed. In contrast, *online methods* update the weight vector based on $\nabla_{\boldsymbol{w}} E_i(\boldsymbol{w})$ computed on a single case. An example is *stochastic gradient descent*, whose k-th iteration is:

- Pick training case at random (say, $i(k)$). Compute $\boldsymbol{g}_k = \nabla_{\boldsymbol{w}_k} E_{i(k)}$.
- Update $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \eta_k \boldsymbol{g}_k$

The goal remains unchanged: minimize $E(\boldsymbol{w})$, a sum over all data points. Compared to gradient descent, each update is about n times faster to compute. On extremely large training sets, running an online algorithm may be the sole viable option. On the other hand, stochastic gradient descent does not update along the direction of steepest descent for $E(\boldsymbol{w})$. An update may even increase the error function value, and this will go unnoticed in general. Why would such an algorithm ever work?

Some rough intuition goes as follows. The usual gradient is

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{w}} E_i(\boldsymbol{w}),$$

the empirical average of the stochastic gradient terms $\nabla_{\boldsymbol{w}} E_i(\boldsymbol{w})$. Each $\nabla_{\boldsymbol{w}} E_i(\boldsymbol{w})$ is like a random sample with mean $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$, a random perturbation of the true gradient. As η_k decreases to zero, the fluctuations average out over successive steps. On the other hand, η_k must decrease sufficiently slowly in order to avoid premature convergence (a rate of $\eta_k = 1/k$ is typically used). Under certain conditions on $E_i(\boldsymbol{w})$, convergence can be established [7].

Many machine learning algorithms are of the online, stochastic gradient descent type. Let us close this section be showing that the perceptron algorithm (Algorithm 1) is among them. To this end, we need to determine an error function $E_{\text{perc}}(\boldsymbol{w}) = n^{-1} \sum_{i=1}^n E_{\text{perc},i}(\boldsymbol{w})$, so that the update of the perceptron algorithm on case $(\boldsymbol{x}_i, t_i)$ corresponds to a stochastic gradient step. As the perceptron algorithm does not feature a step size η_k and is finitely convergent, we can assume[9] that $\eta_k = 1$ here. Then,

$$\nabla_{\boldsymbol{w}} E_{\text{perc},i}(\boldsymbol{w}) = \left\{ \begin{array}{ll} -t_i \tilde{\boldsymbol{\phi}}_i & \mid \ t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i \le 0 \\ \boldsymbol{0} & \mid \ t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i > 0 \end{array} \right\},$$

since the right hand side is what is subtracted from $\boldsymbol{w}$ when visiting $(\tilde{\boldsymbol{\phi}}_i, t_i)$. This means that $E_{\text{perc},i}(\boldsymbol{w})$ should be zero for $t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i > 0$, $E_{\text{perc},i}(\boldsymbol{w}) = -t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i$ for $t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i \le 0$. Concisely,

$$E_{\text{perc},i}(\boldsymbol{w}) = -t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i \mathrm{I}_{\{t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i \le 0\}} = g\left(-t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i\right), \quad g(z) = z \mathrm{I}_{\{z \ge 0\}}.$$

To conclude, the perceptron algorithm can be understood as a variant of stochastic gradient descent, minimizing the perceptron error function

$$E_{\text{perc}}(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^n g\left(-t_i \boldsymbol{w}^T \tilde{\boldsymbol{\phi}}_i\right), \quad g(z) = z \mathrm{I}_{\{z \ge 0\}}.$$

This function is continuous, but not everywhere differentiable and piecewise linear. In general, stochastic gradient descent and even gradient descent can fail to converge on nondifferentiable criteria. However, in the case of the perceptron algorithm, convergence is established independently by Theorem 2.1.

2.4.3 Techniques: Matrices and Vectors. Outer Product

At first sight, a matrix is a two-dimensional array of numbers:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \dots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \dots & a_{pq} \end{bmatrix} \in \mathbb{R}^{p \times q}.$$

The space of all real-valued matrices of p rows, q columns is denoted $\mathbb{R}^{p \times q}$. It is itself a vector space. Addition and scalar-multiplication of matrices work component-wise as for vectors. In fact, our column vectors are special matrices: $\mathbb{R}^p = \mathbb{R}^{p \times 1}$. We use similar notation than with vectors. For example, the design matrix of Section 2.4 can be defined as $\boldsymbol{\Phi} = [\phi_j(\boldsymbol{x}_i)]_{ij} \in \mathbb{R}^{n \times p}$. Special matrices are $\boldsymbol{0} = [0]$ (matrix of all zeros) and the identity

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \in \mathbb{R}^{p \times p}.$$

[9] More precisely, the perceptron algorithms runs independently of the size of $\eta_k > 0$, since we can rescale $\boldsymbol{w}$ by any positive constant. Setting $\eta_k = 1$ leads to the simplest notation.

A matrix is called *square* if $p = q$ (same number of rows and columns).

A more useful way to understand matrices is as linear transforms of finite-dimensional vector spaces. A matrix $\boldsymbol{A} \in \mathbb{R}^{p\times q}$ acts on a vector $\boldsymbol{x} \in \mathbb{R}^q$, mapping it to $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \in \mathbb{R}^p$:

$$y_i = \sum_{j=1}^{q} a_{ij} x_j \quad \Rightarrow \quad \boldsymbol{y} = \sum_{j=1}^{q} x_j \boldsymbol{a}_j, \quad \boldsymbol{A} = [\boldsymbol{a}_1 \dots \boldsymbol{a}_p].$$

In particular, $\boldsymbol{A}\boldsymbol{\delta}_j = \boldsymbol{a}_j$ (recall $\boldsymbol{\delta}_j$ from Section 2.1.1). A good way to think about a matrix is in terms of its columns. It is the linear transformation which maps the standard Euclidean coordinate system $[\boldsymbol{\delta}_j]$, $j = 1, \dots, q$, to $[\boldsymbol{a}_j]$. For example, the identity matrix has columns $\boldsymbol{\delta}_j$: $\boldsymbol{I} = [\boldsymbol{\delta}_1 \dots \boldsymbol{\delta}_p]$. It maps each vector onto itself. The transpose of a matrix $\boldsymbol{A} \in \mathbb{R}^{p\times q}$, denoted by $\boldsymbol{A}^T$, is obtained by interchanging row and column indices: $\boldsymbol{A}^T = [a_{ji}]_{ij} \in \mathbb{R}^{p\times q}$. The columns of $\boldsymbol{A}$ become the rows of $\boldsymbol{A}^T$, and vice versa.

If $\boldsymbol{A} \in \mathbb{R}^{p\times q}$ and $\boldsymbol{B} \in \mathbb{R}^{r\times p}$, we can concatenate the corresponding linear transformations. If $\boldsymbol{x} \in \mathbb{R}^q$, then $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \in \mathbb{R}^p$ and $\boldsymbol{z} = \boldsymbol{B}\boldsymbol{y} \in \mathbb{R}^r$:

$$z_i = \sum_{j=1}^{p} b_{ij} y_j = \sum_{j=1}^{p} b_{ij} \sum_{k=1}^{q} a_{jk} x_k = \sum_{k=1}^{q} \left(\sum_{j=1}^{p} b_{ij} a_{jk} \right) x_k = \sum_{k=1}^{q} c_{ik} x_k.$$

The new matrix $\boldsymbol{C} = \boldsymbol{B}\boldsymbol{A} \in \mathbb{R}^{r\times q}$ is the *matrix-matrix product* (or just matrix product) of $\boldsymbol{B}$ and $\boldsymbol{A}$. The matrix product is associative and distributive, but *not commutative*: $\boldsymbol{A}\boldsymbol{B} \neq \boldsymbol{B}\boldsymbol{A}$. If $\boldsymbol{A}$ and $\boldsymbol{B}$ have no special structure, computing $\boldsymbol{C}$ costs $O(r\,p\,q)$. Compare this to $O(p\,q + r\,p)$ for computing $\boldsymbol{z}$ from $\boldsymbol{x}$ by two *matrix-vector products*. These differences become even more pronounced if $\boldsymbol{A}$ and $\boldsymbol{B}$ have useful structure, which is lost in $\boldsymbol{C}$.

A special matrix product is that between a column and a row vector:

$$\boldsymbol{x}\boldsymbol{y}^T = [y_1\boldsymbol{x} \dots y_q\boldsymbol{x}] = \begin{bmatrix} x_1y_1 & \dots & x_1y_q \\ \vdots & \ddots & \vdots \\ x_py_1 & \dots & x_py_q \end{bmatrix} \in \mathbb{R}^{p\times q}, \quad \boldsymbol{x} \in \mathbb{R}^p,\ \boldsymbol{y} \in \mathbb{R}^q.$$

This is called the *outer product* between $\boldsymbol{x}$ and $\boldsymbol{y}$. Compare this to the inner product (Section 2.1.1) $\boldsymbol{x}^T\boldsymbol{y} \in \mathbb{R}$, which works for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$ only (same dimensionality). The linear transformation of the outer product $\boldsymbol{x}\boldsymbol{y}^T$ maps any vector $\boldsymbol{z} \in \mathbb{R}^q$ to a scalar multiple of $\boldsymbol{x}$:

$$\left(\boldsymbol{x}\boldsymbol{y}^T\right)\boldsymbol{z} = \boldsymbol{x}\left(\boldsymbol{y}^T\boldsymbol{z}\right) = \alpha\boldsymbol{x}, \quad \alpha = \boldsymbol{y}^T\boldsymbol{z}. \tag{2.2}$$

It is an extreme case which shows you that even if a linear transform maps *into* an output vector space $\mathbb{R}^p$ of p dimensions, it may not reach every vector in that space. Test your understanding by the following column representations of a matrix in terms of outer products:

$$\boldsymbol{A} = \sum_{j=1}^{q} \boldsymbol{a}_j \boldsymbol{\delta}_j^T, \qquad \boldsymbol{I} = \sum_{j=1}^{q} \boldsymbol{\delta}_j \boldsymbol{\delta}_j^T.$$

Moreover, how does the matrix $\mathbf{1}\mathbf{1}^T$ look like ($\mathbf{1} \in \mathbb{R}^p$, $\mathbf{1}^T \in \mathbb{R}^{1\times q}$)?

When learning about a subject like machine learning or data analysis, there are a *few very big steps* you can take, which will simplify your working life enormously. One of them is getting used to writing tedious, complicated expressions involving sums and indexing in terms of matrices and vectors. Doing so avoids indexing bugs, exposes the bigger picture about an equation, and directly leads to expressions which are most efficient to compute. We have already started doing that for linear least squares estimation in Section 2.4. Here some examples. To extract columns and rows of a matrix, use the delta vectors:

$$\boldsymbol{A}\boldsymbol{\delta}_j = \boldsymbol{a}_j = [a_{ij}]_i, \quad \boldsymbol{\delta}_i^T\boldsymbol{A} = [a_{i1}, \dots, a_{iq}].$$

To sum up columns or rows of a matrix, use the vector $\mathbf{1}$ of all ones:

$$\boldsymbol{A}\mathbf{1} = \sum_{j=1}^{q} \boldsymbol{a}_j = \left[\sum\nolimits_{j=1}^{q} a_{ij}\right], \quad \mathbf{1}^T\boldsymbol{A} = \sum_{i=1}^{p}[a_{i1}, \dots, a_{iq}].$$

For example, $q^{-1}\boldsymbol{A}\mathbf{1}$ is the arithmetic mean of the columns of $\boldsymbol{A}$. We will use tricks like these repeatedly during the course, and I strongly encourage you to adopt vectorization for yourself.

Vectors are matrices: column vectors in $\mathbb{R}^{p\times 1}$, row vectors in $\mathbb{R}^{1\times q}$. Another important class of square matrices determined by vectors are *diagonal matrices*:

$$\operatorname{diag}\boldsymbol{a} = \begin{bmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_p \end{bmatrix} \in \mathbb{R}^{p\times p}, \quad \boldsymbol{a} \in \mathbb{R}^p.$$

The matrix-vector multiplication with a diagonal matrix is simply the component-wise product: $(\operatorname{diag}\boldsymbol{a})\boldsymbol{x} = [a_i x_i]$. This is also called the Schur or Hadamard product, denoted $\boldsymbol{a} \circ \boldsymbol{x}$. The operator diag maps vectors in $\mathbb{R}^p$ to diagonal matrices in $\mathbb{R}^{p\times p}$. We often have to access the diagonal of a square matrix:

$$\operatorname{diag}(\boldsymbol{A}) = [a_{ii}] \in \mathbb{R}^p, \quad \boldsymbol{A} \in \mathbb{R}^{p\times p}.$$

Note that we use the diag operator in two different ways, which can be distiguished by checking whether its argument is a vector or a square matrix. Test: What is $\operatorname{diag}\operatorname{diag}(\boldsymbol{A})$? What is $\operatorname{diag}(\operatorname{diag}\boldsymbol{a})$?

Finally, $\boldsymbol{A} \in \mathbb{R}^{p\times q}$ linearly maps vectors from $\mathbb{R}^q$ to $\mathbb{R}^p$, but the best way to think about $\boldsymbol{A}$ is that it linearly maps the vector space $\mathbb{R}^q$ onto a linear subspace of $\mathbb{R}^p$, called the *range* and denoted by $\boldsymbol{A}\mathbb{R}^q$. This is a linear subspace precisely because $\boldsymbol{A}$ is a linear transform. The range is equal to the span of $\{\boldsymbol{a}_j\}$, the columns of $\boldsymbol{A}$:

$$\boldsymbol{A}\mathbb{R}^q = \operatorname{span}(\{\boldsymbol{a}_j\}) = \left\{\sum\nolimits_{j=1}^{q} x_j\boldsymbol{a}_j \;\middle|\; \boldsymbol{x} \in \mathbb{R}^q\right\}.$$

Picture it for yourself. A line in $\mathbb{R}^2$ is the range of a vector $\boldsymbol{d} \in \mathbb{R}^2\setminus\{\mathbf{0}\}$, pointing along the line. The *rank* of a matrix $\boldsymbol{A}$, $\operatorname{rk}(\boldsymbol{A})$, is the dimensionality of its range. We have that $\operatorname{rk}(\boldsymbol{A}) \le \min\{p, q\}$. Namely, $\boldsymbol{A}\mathbb{R}^q$ is a subspace of $\mathbb{R}^p$ ($\le p$), and it is generated by the q columns of $\boldsymbol{A}$ $(\leq q)$. If $\mathrm{rk}(\boldsymbol{A}) = \min\{p, q\}$, we say that $\boldsymbol{A}$ has *full rank*. An important result is that $\mathrm{rk}\,\boldsymbol{A} = \mathrm{rk}\,\boldsymbol{A}^T$: the ranges $\boldsymbol{A}\mathbb{R}^q$ and $\boldsymbol{A}^T\mathbb{R}^p$ have the same dimensionality.

Some examples. The identity $\boldsymbol{I} \in \mathbb{R}^{p\times p}$ always has full rank p. On the opposite end, $\mathrm{rk}\,\boldsymbol{0} = 0$ (zero matrices are the only matrices of rank zero). What is the rank of the outer product $\boldsymbol{x}\boldsymbol{y}^T$? Recall (2.2): its range is the span of the single vector $\boldsymbol{x}$, so $\mathrm{rk}(\boldsymbol{x}\boldsymbol{y}^T) = 1$ (unless $\boldsymbol{x} = \boldsymbol{0}$ or $\boldsymbol{y} = 0$, then the rank is zero).

While $\boldsymbol{A}\mathbb{R}^q$ is a subspace of $\mathbb{R}^p$, the second important player is a subspace of $\mathbb{R}^q$, the *null space* (or *kernel*) of $\boldsymbol{A}$:

$$\ker \boldsymbol{A} = \left\{ \boldsymbol{x} \in \mathbb{R}^q \;\middle|\; \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0} \right\}.$$

Please verify for yourself that $\ker \boldsymbol{A}$ is indeed a linear subspace. The role of the null space is as follows. Suppose that $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}_0$. If you add any $\boldsymbol{v} \in \ker \boldsymbol{A}$ to $\boldsymbol{x}_0$, then $\boldsymbol{x}_0 + \boldsymbol{v}$ is still mapped to $\boldsymbol{y}$. In fact, the set of *all* solutions $\boldsymbol{x}$ to the *linear system* $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$ is precisely the affine subspace

$$\boldsymbol{x}_0 + \ker \boldsymbol{A} = \left\{ \boldsymbol{x}_0 + \boldsymbol{v} \;\middle|\; \boldsymbol{v} \in \ker \boldsymbol{A} \right\}.$$

Test yourself: what is the kernel of the matrix $\boldsymbol{w}^T \in \mathbb{R}^{1\times q}$? Does this remind you of something in Section 2.2? It is argued in [45] that you only really understand matrices if you understand the four subspaces: $\boldsymbol{A}\mathbb{R}^q$, $\boldsymbol{A}^T\mathbb{R}^p$, $\ker \boldsymbol{A}$, and $\ker \boldsymbol{A}^T$. I highly recommend you study chapters 3 and 4 of [45] to refresh your memory and gain a better perspective. One important result is that the range $\boldsymbol{A}\mathbb{R}^q$ and the null space $\ker \boldsymbol{A}^T$ are orthogonal subspaces (Section 2.1.1). Namely, if $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{v}$ is from $\boldsymbol{A}\mathbb{R}^q$, $\boldsymbol{y} \in \ker \boldsymbol{A}^T$, then

$$\boldsymbol{y}^T\boldsymbol{x} = \boldsymbol{y}^T\boldsymbol{A}\boldsymbol{v} = \left(\boldsymbol{A}^T\boldsymbol{y}\right)^T \boldsymbol{v} = \boldsymbol{0}^T\boldsymbol{v} = 0.$$

To test your understanding, combine this fact with $\mathrm{rk}\,\boldsymbol{A} = \mathrm{rk}\,\boldsymbol{A}^T$ to show that for $\boldsymbol{A} \in \mathbb{R}^{p\times q}$, the sum of $\mathrm{rk}\,\boldsymbol{A}$ and the dimensionality of $\ker \boldsymbol{A}$ is q.

Finally, a square matrix $\boldsymbol{A} \in \mathbb{R}^{p\times p}$ is *invertible* if and only if it is full rank: $\mathrm{rk}\,\boldsymbol{A} = p$, $\boldsymbol{A}\mathbb{R}^p = \mathbb{R}^p$. Equivalently, the square matrix $\boldsymbol{A}$ is invertible if and only if $\ker \boldsymbol{A} = \{\boldsymbol{0}\}$: $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{0}$ implies that $\boldsymbol{v} = \boldsymbol{0}$. It is only in this case that we can go back and invert a system for every pair of vectors: $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ to $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{y}$. The *inverse* is defined by

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I} = \boldsymbol{A}^{-1}\boldsymbol{A}.$$

As we will see later during the course, it is not in general a good idea to compute the inverse in practice, where other techniques are faster and more reliable.

Chapter 3

The Multi-Layer Perceptron

In this chapter, we discuss multi-layer perceptrons (MLPs), a layered nonlinear extension of linear techniques treated in the previous chapter. MLPs are among the most widely used so-called neural network models. We will motivate why they can improve upon linear classifiers and detail how they can be trained efficiently through error backpropagation. We discuss some aspects of MLPs in practice and give links to modern nonlinear optimization methods which play a central role in applied machine learning.

While many books[1] and very many papers have been written about MLPs, they are mathematically speaking not very deep concepts. They are nonlinear function classes, whose convenient layered structure leads to efficient computation of gradients (and even Hessians). The celebrated backpropagation technique is simply the chain rule of differential calculus. In particular, MLPs are not good models for parts of the brain. Real neurons exhibit stochastic (noise, leaking of current) and dynamical behaviour (rhythms, synchronization), MLPs simply represent fixed deterministic functions. In this course, we will treat MLPs as a convenient tool to approach machine learning problems.

3.1 Why Nonlinear Classification?

In the previous chapter, we studied linear classification. We worked out a simple algorithm to learn such perceptrons and understood a good deal about its geometric properties. Using feature spaces, linear classifiers can be configured by complex decision boundaries, and they can be trained very efficiently. Why would we need anything else?

The simplicity of linear discriminants come at a price: they lack in flexibility. One way to assess the flexibility of a class of classifiers is to ask how many different discrimination problems can be solved perfectly, versus how many

[1] In the author's opinion, based on limited exposure, there are few *good* books about MLPs, or more general neural networks. The author's favourite is [5]. Then there is [23] and [38].

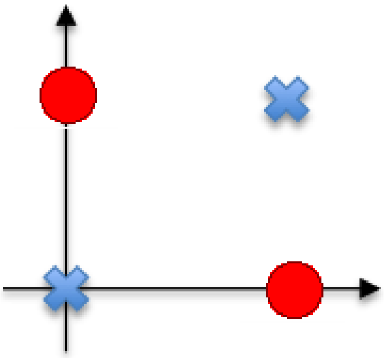


Figure 3.1: The XOR problem is not linearly separable in $\mathbb{R}^2$. In order to solve it without error, we need discriminant functions which are nonlinear in the input space $\mathbb{R}^2$.

parameters have to be adjusted during training. Consider a linear classifier $f(\boldsymbol{x}) = \operatorname{sgn}(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}))$, where $\boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^p$. No matter what features you pick, it is not hard to show that there is always a dataset $\{(\boldsymbol{x}_i, t_i)\}$ of size $\leq p+1$ which is not linearly separable. The ratio between size of (always) separable datasets and number of free parameters is one. One example is the XOR problem (Figure 3.1): four patterns in $\mathbb{R}^2$ which are not linearly separable (note that $p = 3$, since $\boldsymbol{w}$ includes a bias parameter). If the input points are $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$, the function sought after is the exclusive-or. In higher dimensions, these non-separable datasets are not rare worst-case examples, but they are in fact the typical case. With nonlinear classifiers, we expect to do much better in that respect.

There are many ways to step from linear to nonlinear. Two powerful ideas will be studied in this course:

- Make use of feature maps $\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})$ which are themselves parameterized by $\boldsymbol{\theta}$, and learn $\boldsymbol{\theta}$ during training as well. Since we know about linear mappings already, a most sensible approach would be to make use of them in the construction of $\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})$. This idea leads to multi-layer perceptrons, the topic of this chapter.

- Use a feature map $\boldsymbol{\phi}(\boldsymbol{x})$ and weight vector $\boldsymbol{w}$ in a very high-dimensional space, p exponential in d (input space dimensionality), maybe even $p = \infty$. Find training and prediction algorithms which do not represent $\boldsymbol{w}$ directly, and whose scaling is independent of p. We follow up on this idea in Chapter 9.

3.2 Multi-Layer Perceptrons

We are computer scientists. If a technique works and we understand it well, but it just does not do the next job properly, we glue it together in a composite architecture and check whether it works. A first attempt would be to concatenate linear discriminants. For example, for the linear discriminant $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$, we could use feature functions $\phi_j(\boldsymbol{x}) = \boldsymbol{\theta}_j^T\boldsymbol{x}$, coming with free parameters $\boldsymbol{\theta}_j \in \mathbb{R}^d$. However, this does not lead us outside of the linear world. The concatenation of linear maps is a linear map again (Section 2.4.3), so our combined discriminant is a linear function and nothing is gained. How about combining linear classifiers (as opposed to discriminants) in this way? In the example above, we could use $\phi_j(\boldsymbol{x}) = \operatorname{sgn}(\boldsymbol{\theta}_j^T\boldsymbol{x} + b_j)$. Since $\operatorname{sgn}(\cdot)$ is a highly nonlinear function, the corresponding discriminant $y(\boldsymbol{x})$ is nonlinear. A major problem with this class in practice is that, due to the discontinuous $\operatorname{sgn}(\cdot)$ functions, it is very hard to train such a model on data. If we used a continuously differentiable, yet nonlinear transfer function in place of $\operatorname{sgn}(\cdot)$, we could compute the gradient w.r.t. parameters and run gradient descent. This would be a first example of a *multi-layer perceptron* (MLP).

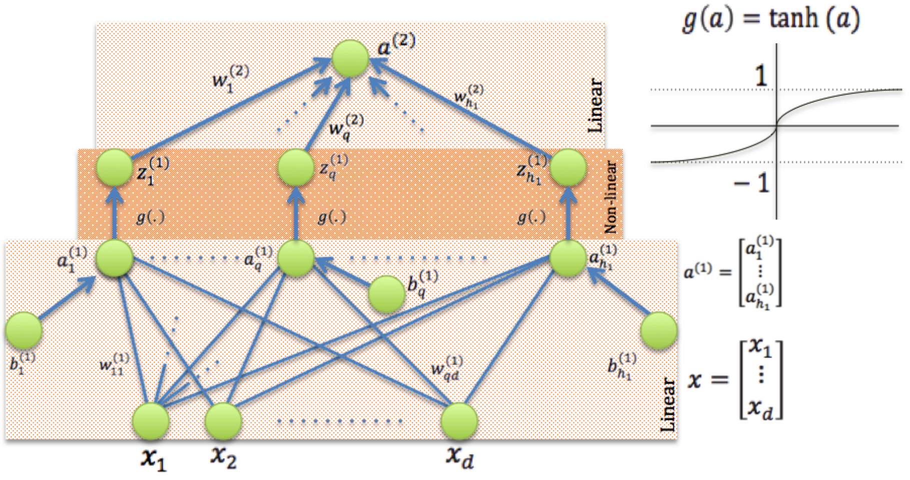


Figure 3.2: Multi-layer perceptron with two layers and $\tanh(\cdot)$ transfer function. See text for explanations.

Let us work out an example and thereby fix concepts. Given inputs $\boldsymbol{x} \in \mathbb{R}^d$, define *activations*

$$a_q^{(1)} = (\boldsymbol{w}_q^{(1)})^T\boldsymbol{x} + b_q^{(1)} = \sum_{j=1}^{d} w_{qj}^{(1)} x_j + b_q^{(1)},\ q = 1, \dots, h_1.$$

We can collect the activations in a vector $\boldsymbol{a}^{(1)} = [a_q^{(1)}] \in \mathbb{R}^{h_1}$. Note that this is really a mapping of $\boldsymbol{x}$. Next, we pass each activation through a *transfer function* $g(\cdot)$: $z_q^{(1)} = g(a_q^{(1)})$. h_1 is the size of the first layer. We say that the first layer has h_1 *units*, more specifically *hidden units* (“hidden” refers to the fact that activation values are not given as part of a training dataset). $z_q^{(1)}$ is the output of the q-th unit in the first layer.

The transfer function is chosen as part of the architecture. It must be defined on all of $\mathbb{R}$, nonlinear and continuously differentiable. A simple generalization is to use different transfer functions for each layer, or even for each unit. We use a single $g(\cdot)$ in this chapter for notational simplicity only. A frequently used transfer function is the *tangent hyperbole*:

$$g(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}.$$

This is an odd function, $g(-a) = -g(a)$, $g(0) = 0$. Its asymptotes are $\lim_{a\to\pm\infty} g(a) = \pm 1$ (Figure 3.2, right). Moreover, $g'(a) = 1 - g(a)^2$, in particular $g'(0) = 1$, so that $g(\varepsilon) \approx \varepsilon$ for $\varepsilon \approx 0$: for small arguments, $g(a)$ behaves like a linear function, but it saturates for larger ones. Notice that $a \mapsto g(a/\varepsilon)$ tends to the step function $\operatorname{sgn}(a)$ as $\varepsilon \to 0$, so that $\tanh(\cdot)$ can be seen as smooth approximation to the sign function.

We now use $\boldsymbol{z}^{(1)} = [z_q^{(1)}] \in \mathbb{R}^{h_1}$ as input to the next layer:

$$a_q^{(2)} = (\boldsymbol{w}_q^{(2)})^T \boldsymbol{z}^{(1)} + b_q^{(2)}, \quad z_q^{(2)} = g(a_q^{(2)}), \quad q = 1, \dots, h_2.$$

Note that this layer looks the same as the first, only that inputs are hidden $\boldsymbol{z}^{(1)}$ instead of observed $\boldsymbol{x}$. We can use as many layers as we like, but the simplest choice above the linear class is to use two layers. The activations $a_q^{(2)}$ of the uppermost layer are called *outputs*. They are taken as they are, no further transfer functions and $z_q^{(2)}$ are used. For our binary classification example, we would have $h_2 = 1$ (just one unit in the second layer), and $a^{(2)}(\boldsymbol{x})$ corresponds to the nonlinear discriminant function $y(\boldsymbol{x})$ (we drop the subscript "1" here, as there is a single activation only). The resulting classifier is $f(\boldsymbol{x}) = \operatorname{sgn}(a^{(2)}(\boldsymbol{x}))$. An example of a two-layer MLP is given in Figure 3.2. Its parameters are $\{(\boldsymbol{w}_q^{(1)}, b_q^{(1)}) \mid q = 1, \dots, h_1\}$ for the first layer, and the usual $\boldsymbol{w}^{(2)}$, $b^{(2)}$ for the second, linear layer. Altogether,

$$f(\boldsymbol{x}) = \operatorname{sgn}(a^{(2)}(\boldsymbol{x})), \quad a^{(2)}(\boldsymbol{x}) = \sum_{q=1}^{h_1} w_q^{(2)} g\Bigg(\underbrace{\sum_{j=1}^{d} w_{qj}^{(1)} x_j + b_q^{(1)}}_{=a_q^{(1)}}\Bigg) + b^{(2)}.$$

Our comment at the beginning of section is clear now. A two-layer MLP has the form of a linear model $a^{(2)}(\boldsymbol{x}) = (\boldsymbol{w}^{(2)})^T \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta}) + b^{(2)}$, where the features

$$\phi_q(\boldsymbol{x}; \boldsymbol{\theta}) = g\left((\boldsymbol{w}_q^{(1)})^T \boldsymbol{x} + b_q^{(1)}\right)$$

are nonlinear functions of $\boldsymbol{x}$, configured by additional parameters $\boldsymbol{\theta} = \{\boldsymbol{w}_q^{(1)}, b_q^{(1)}\}$. The fact that $\boldsymbol{\theta}$ has to be learned alongside $\boldsymbol{w}^{(2)}$ and $b^{(2)}$ is what makes this setup nonlinear. Note that the number of layers is determined by the number of linear activation maps. Beware that other books may count the number of variable layers, so call our example "three-layer network" ($\boldsymbol{x}$ would be the "input layer" for them), while others count the number of hidden layers and would call this a "single hidden layer network".

Multi-layer perceptrons (MLPs) are learned from training data in the same general way as linear classifiers. Recall Section 2.4. We pick an error function

$E(\boldsymbol{w})$, quantifying the mismatch between predictions (outputs) and training set targets. Here, we collect all parameters of the network model in a single large vector $\boldsymbol{w}$. Test your understanding by writing down how $\boldsymbol{w}$ would look like for the two-layer example above. You have to end up with $(d+1)h_1 + h_1 + 1$ parameters. Let us use the squared error function:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(a^{(2)}(\boldsymbol{x}_i) - t_i\right)^2.$$

Next, we learn weights $\boldsymbol{w}$ by minimizing $E(\boldsymbol{w})$, for example by gradient descent. From Section 2.4.1, we know that the gradient is

$$\nabla_{\boldsymbol{w}} E = \sum_{i=1}^{n}\left(a^{(2)}(\boldsymbol{x}_i) - t_i\right)\nabla_{\boldsymbol{w}} a^{(2)}(\boldsymbol{x}_i).$$

This has the same flavour than in the linear case. First, we compute residuals $a^{(2)}(\boldsymbol{x}_i) - t_i$ for every training case. Then, we combine these with gradients of the outputs in order to accumulate the gradient. For online learning (stochastic gradient descent), we compute the gradient on a single case only. Then, we make a short step along the negative gradient direction. There is of course one key difference to the linear case: $\nabla_{\boldsymbol{w}} a^{(2)}(\boldsymbol{x}_i)$ is not just some fixed $\boldsymbol{\phi}(\boldsymbol{x}_i)$ anymore, but looks daunting to compute. We will meet this challenge in Section 3.3.

A final “historical” note before we move on. A surprisingly large amount of work has been spent on figuring out which functions can or cannot be represented by an MLP with two layers (one hidden, one output). If you really want to know about this, [5, ch. 4] gives a brief overview. For example, a two-layer network with tanh transfer functions can approximate any continuous function on a compact set arbitrarily closely, if only the number h_1 of hidden units is large enough. These results are of close to no practical relevance, not only because the number h_1 has to be very large, but also because what really counts is whether we can *learn* complex functions from training data in a reliable and efficient way.

3.2.1 Vectorization of MLP Formalism

Recall from Section 2.4.3 that part of our mission is to vectorize our procedures. Vectorization does not only expose important structure most clearly and helps with “debugging” expressions, it is an essential step towards elegant and efficient implementation. Most machine learners use `Matlab` or `Python` for development and prototyping, interpreted languages in which vectorization is a must. Let us vectorize the MLP function evaluation. We already have vectors $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{a}^{(1)} \in \mathbb{R}^{h_1}$. The weight matrix $\boldsymbol{W}^{(1)} \in \mathbb{R}^{h_1 \times d}$ consists of rows $(\boldsymbol{w}_q^{(1)})^T$, the bias vector $\boldsymbol{b}^{(1)} = [b_q^{(1)}] \in \mathbb{R}^{h_1}$. Then, $\boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$. Moreover, we extend scalar functions to vectors by simply applying them component-wise. For example, $f(\boldsymbol{v}) = [f(v_j)]$, where $f : \mathbb{R} \to \mathbb{R}$. Then, the vector of first layer unit outputs is $\boldsymbol{z}^{(1)} = g(\boldsymbol{a}^{(1)})$. All in all, the first layer is vectorized as:

$$\boldsymbol{z}^{(1)} = g(\boldsymbol{a}^{(1)}), \quad \boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}.$$

Our second layer is linear with a single activation $a^{(2)}$:

$$a^{(2)} = (\boldsymbol{w}^{(2)})^T \boldsymbol{z}^{(1)} + b^{(2)}.$$

If instead, we used a second hidden layer, it would be given by

$$\boldsymbol{z}^{(2)} = g(\boldsymbol{a}^{(2)}), \quad \boldsymbol{a}^{(2)} = \boldsymbol{W}^{(2)} \boldsymbol{z}^{(1)} + \boldsymbol{b}^{(2)}, \quad \boldsymbol{W}^{(2)} \in \mathbb{R}^{h_2 \times h_1}. \tag{3.1}$$

Note the clear separation between linear activation maps and nonlinear transfers.

3.3 Error Backpropagation

The content of this section can be summarized in two sentences. Error backpropagation is nothing but the chain rule of differential calculus. Due to the multi-layered feed-forward architecture of an MLP, this rule allows us to compute the gradient $\nabla_{\boldsymbol{w}} E$ of an error function $E(\boldsymbol{w})$ in time linear in the number of training points and the number of weights. This computational edge is what makes MLPs so attractive in practice.

One point up front. Recall our mantra from Section 2.4. We will carefully separate our real-world problem, its abstraction as optimization problem, and the algorithm for solving the latter. Backpropagation is even lower down the hierarchy, it is simply a deterministic computational technique. You can call it an algorithm, but *not* one which solves our optimization problem $\min_{\boldsymbol{w}} E(\boldsymbol{w})$: it simply computes a gradient. There are many gradient-based optimization algorithms (see Section 3.4.2), and all of them use backpropagation when applied to MLP training.

Let us compute $\nabla_{\boldsymbol{w}} E$ for our two-layer binary classification MLP, continuing where we left at the end of Section 3.2. First, the error function decomposes as $E(\boldsymbol{w}) = \sum_{i=1}^n E_i(\boldsymbol{w})$, where $E_i(\boldsymbol{w}) = \frac{1}{2}(a^{(2)}(\boldsymbol{x}_i) - t_i)^2$. We compute $\nabla_{\boldsymbol{w}} E_i$, then sum up the results. We fix one $i \in \{1, \dots, n\}$ and drop the index and the argument $\boldsymbol{x}_i$ from the notation (for example, we write $a^{(2)}$ instead of $a^{(2)}(\boldsymbol{x}_i)$). As in the linear case, we break the derivatives into small, manageable steps. To this end, we define error (or residual) variables as

$$r^{(2)} = \frac{\partial E_i}{\partial a^{(2)}}, \qquad r_q^{(1)} = \frac{\partial E_i}{\partial a_q^{(1)}}.$$

Note that for every activation variable, there is one error variable. Moreover, the error variables determine the gradient components:

$$\begin{aligned} \nabla_{\boldsymbol{w}^{(2)}} E_i &= \left(\frac{\partial E_i}{\partial a^{(2)}} \right) \nabla_{\boldsymbol{w}^{(2)}} a^{(2)} = r^{(2)} \boldsymbol{z}^{(1)}, \quad \frac{\partial E_i}{\partial b^{(2)}} = r^{(2)}, \\ \nabla_{\boldsymbol{w}_q^{(1)}} E_i &= \left(\frac{\partial E_i}{\partial a_q^{(1)}} \right) \nabla_{\boldsymbol{w}_q^{(1)}} a_q^{(1)} = r_q^{(1)} \boldsymbol{x}, \quad \frac{\partial E_i}{\partial b_q^{(1)}} = r_q^{(1)}. \end{aligned} \tag{3.2}$$

All we need to do is to compute the errors. First, $r^{(2)} = a^{(2)} - t_i$. This is just the residual we already know from the linear case (Section 2.4.1). Finally, we need the chain rule:

$$r_q^{(1)} = \frac{\partial E_i}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial a_q^{(1)}} = r^{(2)} \frac{\partial}{\partial a_q^{(1)}} \sum_{k'=1}^{h_1} w_{k'}^{(2)} g(a_{k'}^{(1)}) = r^{(2)} w_q^{(2)} g'(a_q^{(1)}).$$

This means we compute $r_q^{(1)}$ in terms of $r^{(2)}$, multiplying it with the weight $w_q^{(2)}$ linking the output and q-th hidden unit, then by the derivative $g'(a_q^{(1)})$. Note the remarkable symmetry with the activation variables. For them, $a^{(2)}$ is computed in terms of $a_q^{(1)}$, multiplied by $w_q^{(2)}$ as well.

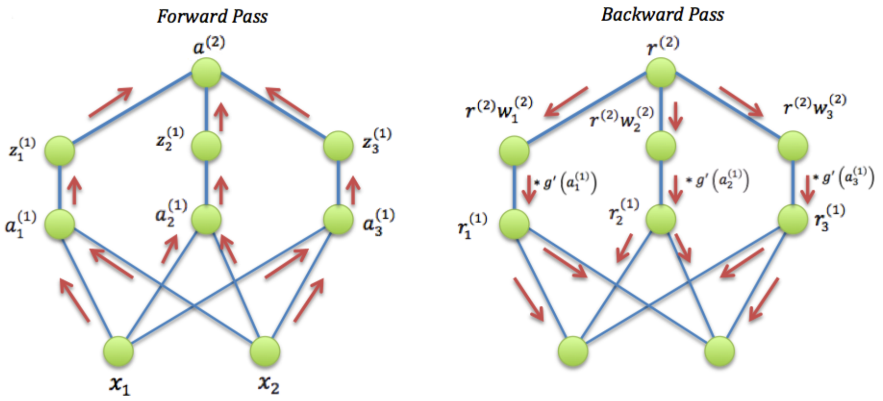


Figure 3.3: Forward pass (left) and backward pass (right) of error backpropagation algorithm for a two-layer MLP. Notice the symmetry between the two passes, and how information $g'(a_q^{(1)})$ computed during the forward pass and stored locally at each node is recycled during the backward pass.

To see the backpropagation technique in its full glory, let us consider a network of at least three layers. The relationship between gradient terms and error variables remain the same. To work out $r_q^{(1)}$, note that the q-th unit in the first layer is now connected to h_2 second layer units. We have to use the chain rule with respect to the error variables for all of them:

$$r_q^{(1)} = \sum_{j=1}^{h_2} \frac{\partial E_i}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial a_q^{(1)}} = \sum_{j=1}^{h_2} r_j^{(2)} w_{jq}^{(2)} g'(a_q^{(1)}) = g'(a_q^{(1)}) \sum_{j=1}^{h_2} w_{jq}^{(2)} r_j^{(2)}. \qquad (3.3)$$

Compare this to the equation

$$a_j^{(2)} = \sum_{q=1}^{h_1} w_{jq}^{(2)} g(a_q^{(1)}) + b_j^{(2)}$$

to appreciate the symmetry between the *forward propagation* of the activation variables and the subsequent *backward propagation* of the error variables. These two passes are illustrated in Figure 3.3. The backpropagation technique to compute the gradient term $\nabla_{\boldsymbol{w}} E_i$ for a L-layer network can be summarized as follows:

1. Forward pass: Propagate the activation variables forward through the network, from the inputs up to the output layer. At the j-th unit of layer l, maintain $g'(a_j^{(l)})$ (or alternatively $a_j^{(l)}$).
2. Compute the output residual $r^{(L)} = a^{(L)} - t_i$, in order to initialize the backward pass.
3. Backward pass: Propagate the error variables backward through the network, from the output down to the inputs, using (3.3).
4. Compute the gradient term $\nabla_{\boldsymbol{w}} E_i$ based on the error variables, as in (3.2).

The computational cost is dominated by the linear mappings, both in the forward and the backward pass, and the final gradient assembly. In each of the passes, each weight is touched exactly once (the bias parameters are not used in the backward pass, but their number is subdominant). Therefore, both passes scale linearly in the number of weights, and so does the final gradient assembly.

This is the backpropagation technique applied to gradient computations for the squared error function. Later during the course, in Chapter 8, we will consider models with several output units and different error functions. Our mantra holds true: one of the key aspects of backpropagation is that it is invariant to such changes. You have to write code for it only once in order to serve a wide variety of machine learning problems.

3.3.1 Vectorization of Error Backpropagation

Once more, vectorization will bring out the basic features even more clearly. Recall Section 3.2.1, in particular the forward equations (3.1). Here is the vectorization corresponding to the backward equations (3.3):

$$\boldsymbol{r}^{(1)} = \left(\operatorname{diag} g'(\boldsymbol{a}^{(1)})\right) (\boldsymbol{W}^{(2)})^T \boldsymbol{r}^{(2)}.$$

Forward propagation uses $\boldsymbol{W}^{(2)}$ (and $\boldsymbol{b}^{(2)}$), backward propagation the transpose $(\boldsymbol{W}^{(2)})^T$. For the gradient assembly, (3.2) becomes

$$\nabla_{\boldsymbol{w}_q^{(2)}} E_i = r_q^{(2)} \boldsymbol{z}^{(1)}, \quad \nabla_{\boldsymbol{b}^{(2)}} E_i = \boldsymbol{r}^{(2)},$$

or even more concisely:

$$\nabla_{\boldsymbol{W}^{(2)}} E_i = \boldsymbol{r}^{(2)} (\boldsymbol{z}^{(1)})^T.$$

3.4 Training a Multi-Layer Perceptron

In this section, we have a closer look at the problem of minimizing the squared error function $E(\boldsymbol{w})$ for a multi-layer perceptron ($\boldsymbol{w}$ is the vector of all weights). Since error backpropagation offers a computationally attractive way to determine the gradient $\nabla_{\boldsymbol{w}} E$, we will focus on gradient-based optimization. It will become clear that the problem $\min_{\boldsymbol{w}} E(\boldsymbol{w})$ is not easily characterized or solved, and in practice a host of "tricks of the trade" are usually required in order to get good results. We will cover some of the most important ones. Finally, for batch optimization, there are far better optimization methods than gradient descent. Covering them in any depth is out of the scope of this course, but if you use MLPs in practice, you need to know about them. Some pointers are provided below.

Suppose our MLP for binary classification has at least two layers, one linear output layer and at least one hidden layer. In (batch) gradient descent optimization, we decrease $E(\boldsymbol{w})$ along the negative gradient direction, until we find $\boldsymbol{w}_*$ so that $\nabla_{\boldsymbol{w}_*} E = \mathbf{0}$. This means that $\boldsymbol{w}_*$ is a *stationary point.* Suppose we take a tiny step in direction $\boldsymbol{d}$, to $\boldsymbol{w}_* + \varepsilon \boldsymbol{d}$. The change in error function value is

$$E(\boldsymbol{w}_* + \varepsilon \boldsymbol{d}) - E(\boldsymbol{w}_*) = \varepsilon (\nabla_{\boldsymbol{w}_*} E)^T \boldsymbol{d} + O(\varepsilon^2) = O(\varepsilon^2).$$

No matter what direction $\boldsymbol{d}$ we pick ($\|\boldsymbol{d}\| = 1$), the error function value will not change (to first order in ε). Now, for *some* error functions of specific kind, a stationary point is a global minimum point, so that $\min_{\boldsymbol{w}} E(\boldsymbol{w})$ is solved. Alas, not for our MLP problem. First, in order to check whether $\boldsymbol{w}_*$ is a local *minimum* point in the strict sense of mathematical optimization, we have to use a Taylor expansion of $E(\boldsymbol{w}_* + \varepsilon \boldsymbol{d})$ to second order:

$$E(\boldsymbol{w}_* + \varepsilon \boldsymbol{d}) - E(\boldsymbol{w}_*) = \frac{1}{2} \varepsilon^2 \boldsymbol{d}^T (\nabla \nabla_{\boldsymbol{w}_*} E)^T \boldsymbol{d} + O(\varepsilon^3),$$

where we used that $\nabla_{\boldsymbol{w}_*} E = \mathbf{0}$, so the first-order term vanishes. Here,

$$\nabla \nabla_{\boldsymbol{w}_*} E = \left[\frac{\partial^2 E}{\partial w_{*,j} \partial w_{*,k}} \right]_{j,k}$$

is the Hessian. $\boldsymbol{w}_*$ is a local minimum in the strict sense if $\boldsymbol{d}^T (\nabla \nabla_{\boldsymbol{w}_*} E)^T \boldsymbol{d} > 0$ for any unit norm direction $\boldsymbol{d}$, which guarantees $E(\boldsymbol{w}_* + \varepsilon \boldsymbol{d}) > E(\boldsymbol{w}_*)$ for small $\varepsilon > 0$. In other words, the Hessian has to be positive definite, a concept we will revisit later during the course (Section 4.2.2). More details about conditions for local minima can be found in [3, ch. 1].

There is a more serious problem. Suppose that $\boldsymbol{w}_*$ is a local minimum point in the strict sense. It will in general not be a global minimum point: $E(\boldsymbol{w}_*) > \min_{\boldsymbol{w}} E(\boldsymbol{w})$. And there are no tractable conditions for finding out how big the difference is, or how far away $\boldsymbol{w}_*$ is from a global minimum point. In Figure 3.4, we illustrate the the concepts of local and global minima, maxima, and stationary points. The squared error function for an MLP with ≥ 2 layers has many local minimum points which are suboptimal. To see why that is, consider the high degree of *non-identifiability* of the model class: for a particular weight vector $\boldsymbol{w}$, we can easily find very many different $\boldsymbol{w}'$ so that $E(\boldsymbol{w}) = E(\boldsymbol{w}')$. Here are two examples. First, the transfer function $g(a) = \tanh(a)$ is odd, $g(-a) = -g(a)$. Pick one latent unit with activation $a_q^{(l)}$ and flip the signs of all weights and bias parameters of connections feeding into the unit from below. This flips the sign of $a_q^{(l)}$, therefore of $g(a_q^{(l)})$ as well. Now, flip the signs of all weights and biases on connections coming from the unit, which compensates the sign flip of $g(a_q^{(l)})$. All outputs remain the same, and so does the error function value. Second, we can simply interchange any two units in a hidden layer, leading to a permutation of the weight vector $\boldsymbol{w}$. In short, the error function $E(\boldsymbol{w})$ for an

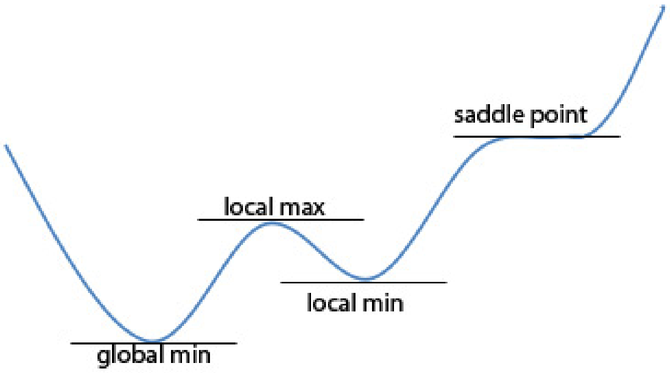


Figure 3.4: A non-convex function, such as the squared error for a multi-layer perceptron, can have local minima and maxima, as well as stationary points which are neither of the two (called saddle points). A stationary point is defined by $\nabla_{\boldsymbol{w}} E = 0$.

MLP is non-identifiable on a massive scale, which implies local minima issues in general.

While there are methods from global mathematical optimization which provably track down global minimum points, these are far too expensive to run for most real-world MLP training problems. The general consensus is to be satisfied with local minimum solutions. It is good practice to use a number of training runs from different initial points (more on initialization below), then to pick the one attaining the lowest error value. Another good idea is to pick an optimizer different from gradient descent, which is less prone to getting stuck in shallow local minima (Section 3.4.2).

A final comment, whose profound significance we will explore in more detail in later parts of this course. Do we even want to minimize $E(\boldsymbol{w})$ globally? Minimizing the squared error on our dataset seems overall a sensible objective, but it does not in general guarantee good performance on the underlying statistical problem (say, discriminating hand-written 8s from 9s). Remember the lookup table classifier from Section 2.1, it achieves zero error on the training data. Anybody who uses MLPs in practice will have run into some situations where a smaller training error $E(\boldsymbol{w})$ comes with worse overall performance on unseen patterns. This *over-fitting* problem is central to what makes machine learning challenging and interesting. We will start to understand over-fitting in Chapter 7, where we also learn about strategies how to alleviate this problem in MLP practice: regularization (Section 7.2) and early stopping (Section 7.2.1). It is also the case that larger networks with more units and weights do not necessarily imply better performance, a lesson we will learn about in Chapter 10, along with techniques to select model size. For now, it is important not to draw erroneous conclusions from the fact that straight minimization of the training squared error $E(\boldsymbol{w})$ does not always lead to optimal results. Counter-measures against over-fitting almost invariably modify, instead of abandon, the data fit criterion, and its robust numerical optimization remains at the heart of machine learning practice.

3.4.1 Gradient Descent Optimization in Practice

Recall the general structure of gradient descent minimization of $E(\boldsymbol{w})$ from Section 2.4.1. Methods differ from each other in two independent aspects. First, a starting point $\boldsymbol{w}_0$ has to be chosen. Second, the k-th update consists in choosing a direction $\boldsymbol{d}_k$ and step size $\eta_k > 0$, then updating $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \eta_k \boldsymbol{d}_k$. This choice is based on the gradient $\nabla_{\boldsymbol{w}_k} E$ or a stochastic gradient $\nabla_{\boldsymbol{w}_k} E_{i(k)}$.

Initialization (*)

We begin with the initialization issue. For model classes like MLPs, the relationship between weights $\boldsymbol{w}$ and outputs is highly non-transparent, and it is not usually possible to select a good starting point $\boldsymbol{w}_0$ in a well-informed way. It is therefore common practice to initialize $\boldsymbol{w}_0$ at random, drawing each component of $\boldsymbol{w}_0$ independently from a zero-mean Gaussian distribution (see Section 6.3) with component-dependent variances. In order to understand why the choice of these variances makes a difference, and also why we should not just start with $\boldsymbol{w}_0 = \boldsymbol{0}$, consider the following argument. A good starting point is one from which training can proceed rapidly. Recall the transfer function $g(a) = \tanh(a)$ from Section 3.2 (the argument holds for other transfer functions as well). In a nutshell, a good initial $\boldsymbol{w}_0$ is chosen so that many or most activation values $a_q^{(l)}$ lie in the area of largest curvature $|g''(a)|$. Let us see why. If a is far away from zero, $g(a)$ is saturated at $\operatorname{sgn}(a)$, with $g'(a) \approx 0$. Large coefficients of $\boldsymbol{w}_0$ imply large (absolute) activation values, and the role of $g'(a)$ in the error backpropagation formulae (3.2) implies small gradients and slow progress in general. On the other hand, for $a \approx 0$, $g(a) \approx a$ is linear. If $\boldsymbol{w}_0$ is chosen such that most activations are small (most extreme: $\boldsymbol{w}_0 = \boldsymbol{0}$), the network behaves like a linear model, and many iterations are needed to step out of this linear regime.

The upshot of this argument is that we should sample $\boldsymbol{w}_0$ in a way that most activations $a_q^{(l)}(\boldsymbol{x}_i)$ are order of unity, on average over the training sample $\{\boldsymbol{x}_i\}$ and $\boldsymbol{w}_0$ (recall that $n^{-1}E(\boldsymbol{w})$ is the average of individual errors over the training set). Assume that the data has been preprocessed to zero mean and unit covariance:

$$\frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i = \boldsymbol{0}, \quad \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^T = \boldsymbol{I}.$$

This transformation, known as *whitening*, is a common step in preprocessing pipelines (see also Section 11.1.1). Consider the first layer activation $a_q^{(1)}$. Since $\mathrm{E}[\boldsymbol{w}_0] = \boldsymbol{0}$, we have that

$$\mathrm{E}\left[a_q^{(1)}\right] = \sum_{j=1}^{d} \mathrm{E}\left[w_{qj}^{(1)}\right] \mathrm{E}[x_{i,j}] + \mathrm{E}\left[b_q^{(1)}\right] = 0.$$

For the variance,

$$\operatorname{Var}\left[a_q^{(1)}\right] = \mathrm{E}\left[\left(\sum\nolimits_{j=1}^{d} w_{qj}^{(1)} x_{i,j} + b_q^{(1)}\right)^2\right]$$
$$= \sum_{j=1}^{d} \operatorname{Var}\left[w_{qj}^{(1)}\right] \operatorname{Var}[x_{ij}] + \operatorname{Var}\left[b_q^{(1)}\right] = \sum_{j=1}^{d} \operatorname{Var}\left[w_{qj}^{(1)}\right] + \operatorname{Var}\left[b_q^{(1)}\right].$$

Here, we used that $w_{q1}^{(1)}, \dots, w_{qd}^{(1)}, b_q^{(1)}$ are independent under the distribution of $\boldsymbol{w}_0$, and that $\mathrm{Var}[x_{ij}] = 1$ due to preprocessing. In order to obtain $\mathrm{Var}\left[a_q^{(1)}\right] = 1$, we should choose $\mathrm{Var}[w_{qj}^{(1)}]$ and $\mathrm{Var}[b_q^{(1)}]$ on the order of $1/(d+1)$. While for units higher up, this argument does not exactly hold anymore (as variables become dependent), it still provides us with the right scaling for our distribution over $\boldsymbol{w}_0$. For a unit in the l-th layer, the activation $a_q^{(l)}$ receives input from h_{l-1} units below (here, $h_0 = d$), and $\mathrm{Var}[w_{qj}^{(l)}]$ and $\mathrm{Var}[b_q^{(l)}]$ should be chosen on the order of $1/(h_{l-1}+1)$.

Learning Rate. Learning with Momentum Term

Choosing the learning rate η_k seems more of an art than a science. Ultimately, the best choice is determined by the local curvature (second derivative), which is typically not computed for MLPs (although it can be obtained at surprisingly little extra effort, see [5, ch. 4.10] or [6, ch. 5.4]). In the context of online learning, the proposal $\eta_k = 1/k$ is common. For batch gradient descent, a constant learning rate often works well and is faster [5, ch. 7.5]. If assessing the Hessian is too much for you, there is a wealth of learning rate adaptation techniques from the old neural networks days [5, ch. 7.5.3].

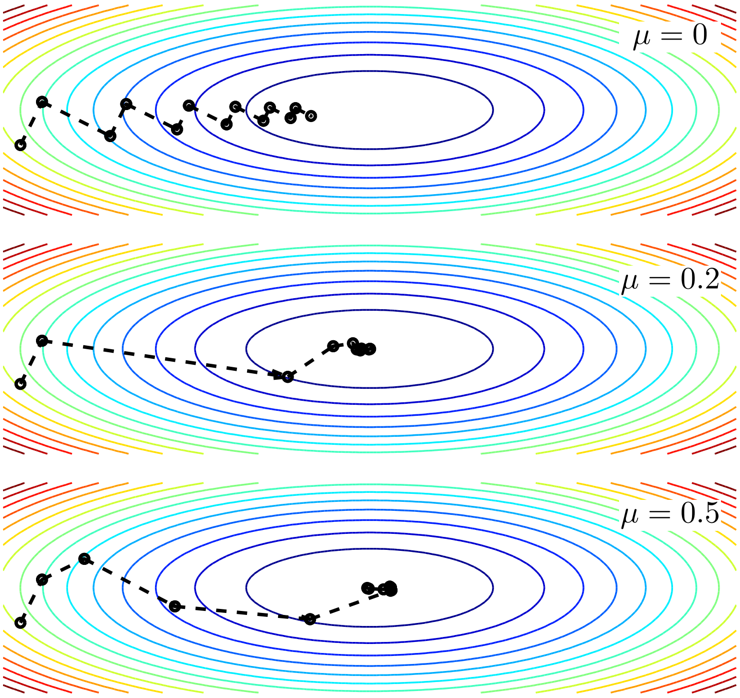


Figure 3.5: Gradient descent optimization without (top) and with momentum term (middle, bottom). We update $\Delta\boldsymbol{w}_k = -\eta(1-\mu)\nabla_{\boldsymbol{w}_k} E + \mu\Delta\boldsymbol{w}_{k-1}$, where η is selected by line minimization.

One serious problem with gradient descent is known as *zig-zagging*. If $E(\boldsymbol{w})$

is highly curved[2] around the current point $\boldsymbol{w}_k$, the steepest descent direction, which ignores the curvature information (Section 3.4), is seriously misleading. Subsequent steps along the negative gradient lead to erratic steps with slow overall progress. Zig-zagging is illustrated in Figure 3.5, top panel. The idea behind a momentum term is the observation that during a phase of zig-zagging, it would be better to move along a direction averaged over several subsequent steps, which would smooth out the erratic component, but leave useful systematic components in place. Denote by $\Delta\boldsymbol{w}_k = \boldsymbol{w}_{k+1} - \boldsymbol{w}_k$ the update done in iteration k. Gradient descent with momentum works as follows:

$$\Delta\boldsymbol{w}_k = -\eta(1-\mu)\nabla_{\boldsymbol{w}_k}E + \mu\Delta\boldsymbol{w}_{k-1}, \quad \mu \in [0,1).$$

Note that we assume a constant learning rate $\eta > 0$ for simplicity. The new update step is a convex combination of the steepest descent direction and the previous update step. Effects of momentum terms of different strength are shown in Figure 3.5. To understand what this is doing, we consider two regimes. First, in a region of low curvature of $E(\boldsymbol{w})$, the gradient will remain approximately constant over several updates. Solving $\Delta\boldsymbol{w} = -\eta(1-\mu)\nabla_{\boldsymbol{w}}E + \mu\Delta\boldsymbol{w}$ results in $\Delta\boldsymbol{w} = -\eta\nabla_{\boldsymbol{w}}E$. In a low-curvature regime, the rule does full-size steepest descent updates. On the other hand, if $E(\boldsymbol{w})$ has high curvature, then $\nabla_{\boldsymbol{w}}E$ changes erratically. In this case, subsequent $\Delta\boldsymbol{w}_k$ will tend to cancel each other out, and the effective learning rate reduces to $\eta(1-\mu) \ll \eta$ or even to $\eta(1-\mu)/(1+\mu)$, damping the oscillations. If you use momentum in practice, you can afford to experiment with larger learning rates. A momentum term is pretty obligatory with online learning by stochastic gradient descent. Stochastic gradients have erratic behaviour built in, and a momentum term is often successful in smoothing away much of the noise.

Analysis of Learning with Momentum Term (*)

Here are details for the analysis of momentum learning in the two extreme regimes just mentioned. In reality, the gradient will have both constant and oscillatory components, in which case momentum will roughly leave the former alone, but damp the latter. First, suppose that $\nabla_{\boldsymbol{w}_k}E = \nabla_{\boldsymbol{w}}E$ over many updates. To see what the updates are doing, we equate $\Delta\boldsymbol{w}_{k+1}$ and $\Delta\boldsymbol{w}_k$:

$$\Delta\boldsymbol{w} = -\eta(1-\mu)\nabla_{\boldsymbol{w}}E + \mu\Delta\boldsymbol{w} \quad \Rightarrow (1-\mu)\Delta\boldsymbol{w} = (1-\mu)(-\eta\nabla_{\boldsymbol{w}}E),$$

so that $\Delta\boldsymbol{w} = -\eta\nabla_{\boldsymbol{w}}E$: undamped gradient descent. Next, suppose that $\nabla_{\boldsymbol{w}_k}E$ is maximally oscillatory: $\nabla_{\boldsymbol{w}_k}E = (-1)^k\nabla_{\boldsymbol{w}}E$. Expanding for two steps, and using $\nabla_{\boldsymbol{w}_{k+1}}E = \nabla_{\boldsymbol{w}}E$, $\nabla_{\boldsymbol{w}_k}E = -\nabla_{\boldsymbol{w}}E$, we have

$$\Delta\boldsymbol{w}_{k+2} = -\eta(1-\mu)\nabla_{\boldsymbol{w}}E + \mu\left(\eta(1-\mu)\nabla_{\boldsymbol{w}}E + \mu\Delta\boldsymbol{w}_k\right).$$

Equating $\Delta\boldsymbol{w}_{k+2}$ and $\Delta\boldsymbol{w}_k$:

$$(1-\mu^2)\Delta\boldsymbol{w} = (1-\mu)^2(-\eta\nabla_{\boldsymbol{w}}E) \quad \Rightarrow \quad \Delta\boldsymbol{w} = \frac{1-\mu}{1+\mu}(-\eta\nabla_{\boldsymbol{w}}E).$$

Here, $(1-\mu)/(1+\mu) \ll 1$, so the osciallations are reduced substantially.

[2] In precise terms, the Hessian $\nabla\nabla_{\boldsymbol{w}}E$ has eigenvalues of very different size, or a large *condition number*.

The Netlab Package

There is a number of free MLP implementations available. In case you use `Matlab` (or the free `Octave`), the author's favourite is `Netlab`, which can be obtained from `http://www1.aston.ac.uk/eas/research/groups/ncrg/resources/netlab/`. It consists of a toolbox of functions and scripts based on the approach and techniques described in [5], but also including more recent developments in the field. `Netlab` comes with a textbook [32], which provides a host of useful advice for MLP-type machine learning in practice.

Testing the Gradient (*)

Training an MLP is tricky enough, due to local minima issues and the fact that we cannot develop a good "feeling" about what individual parameters in the network stand for. But there is one point we *can* (and *absolutely should*) ensure: that the backpropagation code for the gradient computation is bug-free. To keep the presentation simple, we focus on an error function $E(w)$ of a single weight $w \in \mathbb{R}$ only. For the general case, the recipe is repeated for each gradient component (but see below for an idea to save time).

Suppose we are at a point w, and $g(w) = E'(w) = dE/dw$ is the gradient (or derivative) there. Gradient testing works by comparing the result for $g(w)$ with *finite differences*, computed by evaluating the error function at other points w' very close to w:

$$g(w) = E'(w) \approx \tilde{g}_{1;\varepsilon}(w) := \frac{E(w+\varepsilon) - E(w)}{\varepsilon}$$

for a very small $\varepsilon > 0$. This means that we can test our gradient code by computing $\tilde{g}_{1;\varepsilon}(w)$ (which costs one forward pass), then inspecting the relative error

$$\frac{|g(w) - \tilde{g}_{1;\varepsilon}(w)|}{|g(w)|}.$$

Since finite differences and derivatives are not exactly the same, we cannot expect the error to be zero, but it should be small, in fact of the same order as ε if $g(w)$ itself is away from zero. It is important to choose ε not too small, say $\varepsilon = 10^{-8}$.

There is a more accurate *symmetric* finite-difference approximation of gradient components, about twice as costly to evaluate. As debugging is not about speed, we recommend this latter one:

$$g(w) = E'(w) \approx \tilde{g}_{2;\varepsilon}(w) := \frac{E(w+\varepsilon) - E(w-\varepsilon)}{2\varepsilon}.$$

This needs two forward passes to compute, instead of just one. Why does this work better? Let us look at the Taylor expansion of the function E at the current value w:

$$E(w \pm \varepsilon) = E(w) + E'(w)(\pm\varepsilon) + \frac{1}{2}E''(w)\varepsilon^2 + O(\varepsilon^3).$$

Notice that we expand up to second order, and that the ε^2 term does not depend on the sign in $\pm\varepsilon$. Subtracting one from the other line:

$$E(w+\varepsilon) - E(w-\varepsilon) = 2E'(w)\varepsilon + O(\varepsilon^3) \quad \Rightarrow \quad \tilde{g}_{2;\varepsilon}(w) = E'(w) + O(\varepsilon^2),$$

where we divided by 2ε. The error between $E'(w)$ and $\tilde{g}_{2;\varepsilon}(w)$ is only $O(\varepsilon^2)$, whereas you can easily confirm that the error between $E'(w)$ and $\tilde{g}_{1;\varepsilon}(w)$ is $O(\varepsilon)$, thus much larger for small ε.

If your network has lots of parameters, it can be tedious to test every gradient component separately. Far better to run the following randomized test in order to spot problems (which are then analyzed component by component), and instead test the gradient at many different points $\boldsymbol{w}$. The idea is to test *directional derivatives* along random directions[3] $\boldsymbol{d} \in \mathbb{R}^p$, $\|\boldsymbol{d}\| = 1$. Suppose we wish to test the gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} E(\boldsymbol{w})$ at the point $\boldsymbol{w}$. If $f(t) := E(\boldsymbol{w} + t\boldsymbol{d})$, $t \in \mathbb{R}$, the directional derivative of E at $\boldsymbol{w}$ along $\boldsymbol{d}$ is $f'(0) = \boldsymbol{g}^T\boldsymbol{d}$. Our finite difference approximation is

$$\frac{f(\varepsilon) - f(-\varepsilon)}{2\varepsilon} = \frac{E(\boldsymbol{w} + \varepsilon\boldsymbol{d}) - E(\boldsymbol{w} - \varepsilon\boldsymbol{d})}{2\varepsilon}.$$

We monitor the absolute difference between the directional derivative and its finite difference approximation. The gradient $\boldsymbol{g}$ is tested by doing this comparison for maybe 30 random directions. If any substantial differences are detected, we have to detect where the problem comes from. To this end, we can now test along directions $\boldsymbol{d}$ of our choice. For example, $\boldsymbol{d} = \boldsymbol{\delta}_j$ will test the j-th gradient component in isolation.

A final hint. At the debugging stage, it is not important to evaluate gradients over all your training data. Work on very small batches, or even on single data points. Also, choose a smaller network architecture, in particular a reduced number of hidden units. Bugs will show up nevertheless, and you save a lot of time. On the other hand, it is important to test gradients for a range of different values for the weights.

3.4.2 Optimization beyond Gradient Descent (*)

In this section, we focus on batch training exclusively, minimization of an error defined as empirical average over a complete training set or large batches thereof. Our arguments are not specific to the squared error function, but apply just as well to all other continuously differentiable error functions we will learn about during this course.

The simplest method for MLP batch training is gradient descent. Gradients are computed by error backpropagation, which is efficient compared to lesser alternatives, but still constitutes a substantial effort for a large network. Sometimes, simple algorithms are also good algorithms. This is not the case for gradient descent. There is a range of numerical optimization methods which supersede gradient descent in any conceivable sense. For many of them, high-quality implementations are freely available, and you can simply plug in your backpropagation code.

Basic numerical optimization is a fascinating topic. Maths like calculus and linear algebra is brought to life, doing useful things for you. Methods are based on geometric intuition and just fun to learn about. And we already know that

[3] A random direction $\boldsymbol{d}$ is sampled by drawing the components independently from a Gaussian $N(0,1)$, then normalizing the resulting vector to unit length.

optimization is center stage for machine learning. Unfortunately, treating this topic in more detail is not in the scope of this course. There are many good and readable books on optimization. For machine learning purposes, a good place to start is [5, ch. 7]. The `Netlab` package (Section 3.4.1) implements[4] the optimizers discussed there and has demos you can download for plug-and-play. Beyond, we recommend [3, 16, 29, 17] for general nonlinear programming and [8] for convex optimization.

Our discussion is mainly based on [5, ch. 7]. Consider the unconstrained optimization problem $\min_{\boldsymbol{w}} E(\boldsymbol{w})$, where $E(\boldsymbol{w})$ is continuously differentiable and lower bounded (unconstrained means that $\boldsymbol{w} \in \mathbb{R}^p$ arbitrarily), not necessarily the squared error function. On the surface, modern gradient-based optimizers have a similar structure to gradient descent. An iteration from $\boldsymbol{w}_k$ proceeds as follows:

- Evaluate the gradient $\boldsymbol{g}_k = \nabla_{\boldsymbol{w}_k} E$. Determine a search direction $\boldsymbol{d}_k$, based on $\boldsymbol{g}_k$ and information gathered in earlier iterations.
- Find $\boldsymbol{w}_{k+1}$ by a line search, approximately minimizing $f(\eta) = E(\boldsymbol{w}_k + \eta \boldsymbol{d}_k)$ for $\eta > 0$.

Beyond variants of gradient descent, the most basic algorithm of this form is *conjugate gradients*, designed originally for minimizing *quadratic* functions. Examples for quadratic minimization are linear least squares problems (Section 2.4). Gradient descent is a poor method for minimizing general quadratic functions, and even a momentum term does not help much in general. In contrast, conjugate gradients is the *optimal*[5] method for minimizing quadratics, given that one gradient is evaluated per iteration. It is easily extended to non-quadratic functions by incorporating a line search. Applied to MLP training, it tends to outperform gradient descent dramatically, at the same cost per iteration. A variant called scaled conjugate gradients avoids line searches for most updates, which can be expensive for MLPs. Methods advancing on conjugate gradients are all motivated by approximating the gold-standard method: Newton-Raphson optimization. Recall from high school that an update of this algorithm is based on approximating $E(\boldsymbol{w})$ by a quadratic $q(\boldsymbol{w})$ locally at $\boldsymbol{w}_k$, then minimizing $q(\boldsymbol{w})$ as surrogate for $E(\boldsymbol{w})$. Unfortunately, this needs computing the Hessian $\nabla\nabla_{\boldsymbol{w}_k} E$ and solving a linear system with it, which is out of scope[6] of our recipe above. However, *quasi-Newton* methods manage to build search directions over several updates which share important properties with Newton directions. Among them, limited memory quasi-Newton methods are most useful for large MLPs, and are in general widely used for many other machine learning problems as well. Finally, the Levenberg-Marquardt algorithm is specialized to minimizing squared error functions, and is maybe the most widely used general technique

[4] For serious applications, codes from numerical mathematicians should be preferred, such as packages found on `www.netlib.org/`.

[5] In fact, since minimizing a quadratic is equivalent to solving a linear system with a positive definite matrix (see Section 4.2.2), conjugate gradient is the gold-standard for iterative linear solvers as well.

[6] However, note that there are surprisingly efficient methods for computing expressions such as $(\nabla\nabla_{\boldsymbol{w}_k} E)\boldsymbol{v}$, $\boldsymbol{v}$ an arbitrary vector, by a slight extension of error backpropagation [5, ch. 4.10.7]. This would allow for applying *truncated Newton* algorithms to MLP problems, where each Newton direction is approximated by a conjugate gradient solver.

for this purpose. It approximates Newton directions directly, using a simple outer product approximation to the Hessian, together with a model trust region mechanism to control resulting errors by damping.

3.5 Renaissance of Neural Networks (*)

After massive early successes, including the “Bayesian revolution” (triggered by work of David MacKay), neural networks went underground in mainstream machine learning research, submitting to such big guns as support vector machines, convex optimization and infinite models. They are back, and stronger than ever before: new architectures (incorporating concepts from probabilistic graphical models), new learning algorithms (including Markov chain Monte Carlo sampling), new regularization techniques. The groups driving them forward win one big data competition after the other.

The fathers of this renaissance are Geoffrey Hinton, University of Toronto and Yann LeCun, New York University. Hinton introduced products of experts, contrastive divergence, revived restricted Boltzmann machines [24] and stacked them into deep belief networks. Architectures are deep today, featuring many layers and very many weights to train. Backpropagation alone works only with truly massive data, it is commonly supplemented by stage-wise (layer-by-layer) training, unsupervised pre-training and many other tricks of the trade.

So why are they back? To be fair, they were never really gone, but continued to be used heavily in industry and applications. Neural networks come with a huge number of parameters, they need massive amounts of data to be trained, as well as sufficient computing power. Both very big datasets and ample parallel computing power at low cost (graphics processing units) are available today. Neural networks, in contrast to nonparametric methods, are trained by linear passes over the data, or even in a streaming mode. Their main computational primitives, dense matrix multiplication or convolutions, are a good match for parallel computing architectures. Given enough data, their deep structure allows to learn features in a purely data-driven manner, and their superior performance in competitions is just another reminder that data is a better feature engineer than human experts. Finally, and maybe most importantly, the new breed of neural networks embraces *unsupervised learning*, learning without targets and labels (Chapter 12), and not just for pre-processing. This allows to learn from more or less unprocessed data (photos, videos, web pages, news texts), which we can gather in many orders of magnitude larger volumes than the labeled and clean stuff.

Deep neural networks are beyond the scope of this introductory course. However, the reader is strongly encouraged not to miss out on one of the hottest current trends in big data machine learning today. The textbook by Murphy [31] offers a self-contained description of the basics up to mid 2012 or so. A host of references (and other material) can be found at `http://deeplearning.net/`.

Chapter 4

Linear Regression. Least Squares Estimation

In this chapter, we introduce linear regression or curve fitting, a problem which is at least as fundamental for machine learning as classification. A running example will be the fitting of a polynomial curve to real-valued data. We will encounter over-fitting for the first time.

The optimization problem behind linear regression is (linear) least squares estimation, which has already been introduced for binary classification in Chapter 2. In this chapter, we will gain a thorough geometrical intuition about least squares estimation and learn about algorithms for solving it in practice.

4.1 Linear Regression

Not all of machine learning is classification. As we advance through the course, we will see that it is not even the most basic problem on which we can build the others: this role is played by *curve fitting*, or more general representing and learning real-valued functions. Moreover, since the statistics behind linear curve fitting is much simpler than for classification, this problem serves as prime example for introducing some of the most profound concepts of machine learning later during the course: generalization, regularization and model selection. In this section, we introduce the linear regression problem, using the example of polynomial curve fitting.

In fact, we know linear regression already from Section 2.4, where we used it for binary classification. As will become fully clear later in the course, this is a somewhat misguided[1] application, so let us start all over.

Consider the data $\mathcal{D} = \{(x_i, t_i) \mid i = 1, \dots, n\}$ in Figure 4.1. The targets are real-valued here, $t_i \in \mathbb{R}$. Curve fitting amounts to learning the real-valued

[1] This was no cheat. People do use linear regression for classification, mainly because it is so simple. We will see that classification is much better served by error functions whose minimization is not much harder to do, whether for linear methods (Chapter 2) or multi-layer perceptrons (Chapter 3).

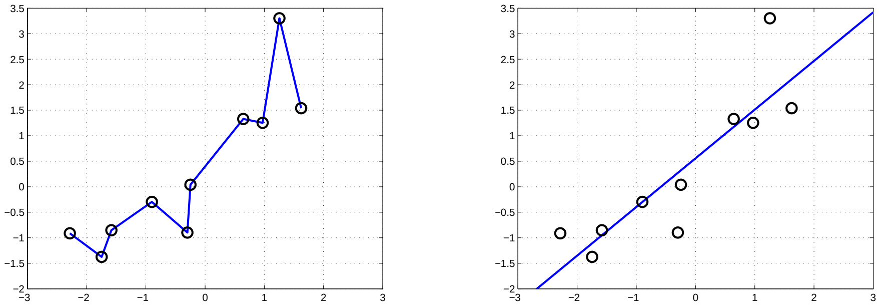


Figure 4.1: Two different ways of fitting the data $\{(x_i, t_i)\}$. Left: Piece-wise linear interpolation. Right: Linear least squares regression.

function $x \to y$ represented by this data. What does that mean? For example, we could just connect neighbouring points by straight lines and be done with it: piece-wise linear interpolation[2] (Figure 4.1, left). However, interpolation is not what curve fitting is about. Rather, we aim to represent the data as the sum of *two* functions: a *systematic curve*, smooth and simple, plus some *random errors*, highly erratic but small. We will refine and clarify this notion in Chapter 8, but for now this working definition will suffice. The rationale is not to get sidetracked by errors made during the recording of the dataset.

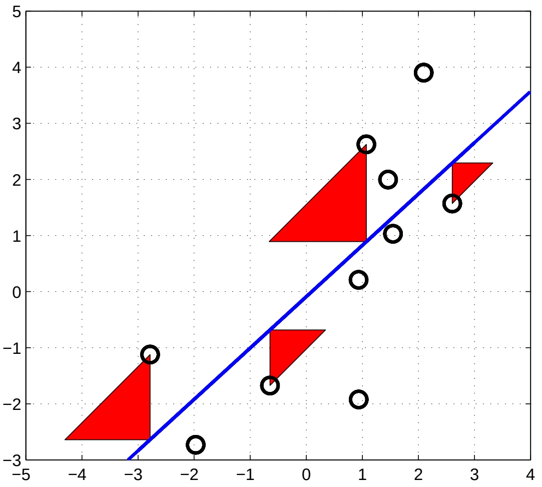


Figure 4.2: Illustration of squared error criterion for linear regression. Each triangle area corresponds to an error contribution of $(y(x_i) - t_i)^2/2$.

The simplest curve fitting technique is *linear regression* (more correctly: linear regression estimation, but the shorter term is commonly used). We assume a

[2] Interpolation differs from curve fitting in that all data points have to be represented exactly: if $y(x)$ is my interpolant, then $y(x_i) = t_i$ for all i.

linear function $y(x) = wx + b$ for the systematic part, then fit it to the data by minimizing the squared error:

$$E(w, b) = \frac{1}{2}\sum_{i=1}^{n}(y(x_i) - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(wx_i + b - t_i)^2. \tag{4.1}$$

This problem is known as *least squares estimation*. Its prediction on our data above is shown in Figure 4.1, right. The squared error criterion is illustrated in Figure 4.2. Note the rapid growth with $|y(x_i) - t_i|$: large differences are not tolerated. We solve this problem by setting the gradient w.r.t. $[w, b]^T$ to zero and solving for w, b. You should do this as an exercise, the solution is given in Section 4.1.1.

Polynomial Regression Estimation

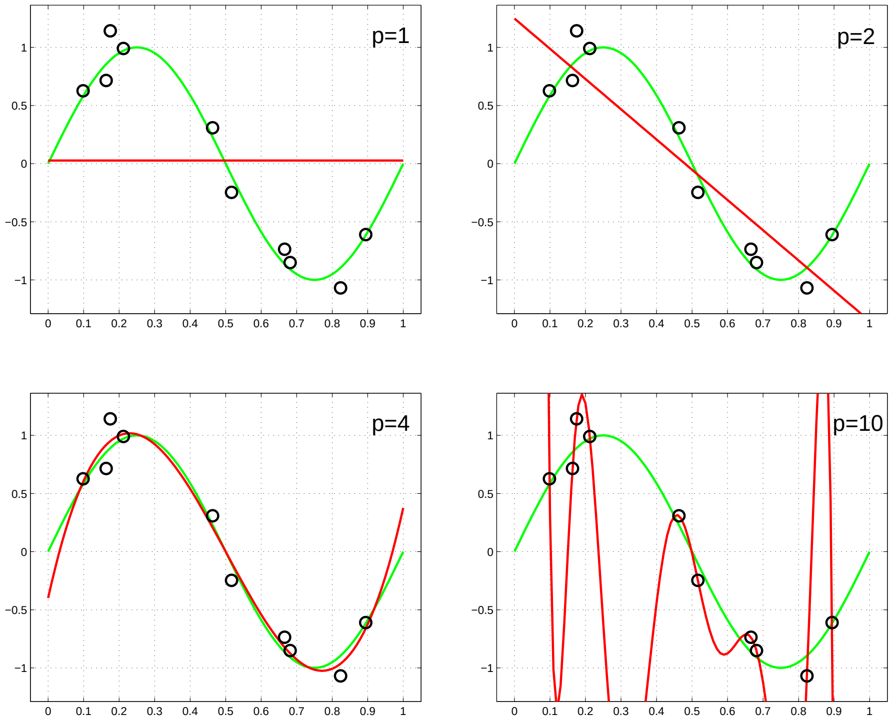


Figure 4.3: Linear regression estimation with polynomials of degree $p-1$. The generating curve (green) is $\sin(2\pi x)$, the noise is Gaussian with standard deviation 0.15.

In order to move beyond lines, we adopt an idea already introduced in Section 2.2. We define a feature map $\boldsymbol{\phi}(x) \in \mathbb{R}^p$ and employ *linear regression* with $y(x) = \boldsymbol{w}^T\boldsymbol{\phi}(x)$. Beware that some books speak of "generalized linear regression" in this case, but we do not follow this convention. Our example will be polynomial regression:

$$\boldsymbol{\phi}(x) = \begin{bmatrix} 1 \\ x \\ \vdots \\ x^{p-1} \end{bmatrix} = \left[x^{j-1}\right]_j, \quad y(x) = \boldsymbol{w}^T\boldsymbol{\phi}(x) = w_0 + w_1 x + \dots + w_{p-1}x^{p-1}.$$

The function $y(x)$ is a polynomial of maximum degree $p-1$, whose coefficients are fit by least squares estimation. The larger the number of features p, the more complex functions we are able to represent. In fact, if $p' > p$, then the class for p is strictly included in the class for p': lines are special cases of quadratic functions. We might assume that the accuracy of curve fitting improves as p grows, since we run our method with more and more flexible function classes. Indeed, if $\boldsymbol{w}_*^{(p)}$ is a minimizer of the squared error $E^{(p)}(\boldsymbol{w}^{(p)})$ (making the dimensionality p explicit in the notation for now), we have

$$E^{(p)}(\boldsymbol{w}_*^{(p)}) = E^{(p')}\left(\left[\boldsymbol{w}_*^{(p)T}, 0, \dots, 0\right]^T\right) \ge E^{(p')}(\boldsymbol{w}_*^{(p')}), \quad p < p'.$$

Our fit in terms of squared error can never get worse as p increases, and typically it improves. In Figure 4.3, we show least squares solutions using different dimensionalities p. The dataset consists of 10 points, drawn from the smooth green curve plus random errors. The fit is rather poor for $p = 1$ (constant) and $p = 2$ (line), giving rise to large errors. It improves much for $p = 4$ (cubic), representing the generating curve well. The fit for $p = 10$ provides a first example of *over-fitting*. The fit to the data is even better than for $p = 4$: the polynomial is exactly interpolating each data point. But away from the x_i locations of the training set, the prediction behaves highly erratically. It is not at all a useful representation of the generating curve. We will return to this important issue in Chapter 7.

4.1.1 Techniques: Solving Univariate Linear Regression

Our aim is to minimize the squared error $E(w, b)$ defined in (4.1). This is a special case of solving the general normal equation. It is useful as practice and to get a feeling for the properties of the solution. Denoting $y_i = y(x_i) = wx_i + b$, we have

$$\frac{\partial E}{\partial w} = \sum_{i=1}^n (y_i - t_i)x_i, \quad \frac{\partial E}{\partial b} = \sum_{i=1}^n (y_i - t_i).$$

Defining the empirical expectations

$$\langle x \rangle = n^{-1}\sum_i x_i, \; \langle x^2 \rangle = n^{-1}\sum_i x_i^2, \; \langle t\, x \rangle = n^{-1}\sum_i t_i x_i, \; \langle t \rangle = n^{-1}\sum_i t_i,$$

it is an easy exercise(!) to show that

$$n^{-1}\frac{\partial E}{\partial w} = w\langle x^2 \rangle + b\langle x \rangle - \langle t\, x \rangle, \quad n^{-1}\frac{\partial E}{\partial b} = w\langle x \rangle + b - \langle t \rangle.$$

Setting these equal to zero, and subtracting $\langle x \rangle$ times the second from the first equation gives

$$w\left(\langle x^2 \rangle - \langle x \rangle^2\right) = \langle t\, x \rangle - \langle t \rangle\langle x \rangle.$$

Now,

$$\langle x^2\rangle - \langle x\rangle^2 = n^{-1}\sum_{i=1}^n (x_i - \langle x\rangle)^2 = \mathrm{Var}[x],$$
$$\langle t\,x\rangle - \langle t\rangle\langle x\rangle = n^{-1}\sum_{i=1}^n (x_i - \langle x\rangle)(t_i - \langle t\rangle) = \mathrm{Cov}[x,t],$$

as confirmed(!) by multiplying it out (the relations hold in general, see Section 5.1.3). Therefore, the minimizer (w, b) is given by

$$w = \frac{\mathrm{Cov}[x,t]}{\mathrm{Var}[x]}, \quad b = \langle t\rangle - w\langle x\rangle.$$

4.2 Linear Least Squares Estimation

Having motivated linear least squares estimation in order to drive linear regression, let us work out the underlying optimization problem and understand properties of the solution. Geometrically, this leads to the concept of orthogonal projection. You will appreciate the foundational nature of this problem as we move through the course: it is behind concepts like conditional expectation or the bias-variance decomposition, and serves as general building block of modern nonlinear optimizers.

We continue where we left in Section 2.4, only that $t_i \in \mathbb{R}$ here, while $t_i \in \{-1, +1\}$ there. Our linear function class is given by $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$ (the bias parameter being part of $\boldsymbol{w}$ as usual, for example w_0 in polynomial regression above). We vectorize everything in terms of target vector $\boldsymbol{t} = [t_i] \in \mathbb{R}^n$ and design matrix $\boldsymbol{\Phi} = [\phi_j(\boldsymbol{x}_i)]_{ij} \in \mathbb{R}^{n\times p}$. We assume that $n \ge p$, and that $\boldsymbol{\Phi}$ has full rank p (Section 2.4.3). Also, denote by $\boldsymbol{y} = [y_i] = [y(\boldsymbol{x}_i)] \in \mathbb{R}^n$ the vector of predictions. The squared error is

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^n (y(\boldsymbol{x}_i) - t_i)^2 = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{t}\|^2 = \frac{1}{2}\left\|\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\right\|^2.$$

We determined its gradient in Section 2.4.1:

$$\nabla_{\boldsymbol{w}} E = \sum_{i=1}^n \frac{\partial E}{\partial y_i}\nabla_{\boldsymbol{w}} y_i = \sum_{i=1}^n (y_i - t_i)\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{\Phi}^T(\boldsymbol{y} - \boldsymbol{t}) = \boldsymbol{\Phi}^T\left(\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\right).$$

Let us set $\nabla_{\boldsymbol{w}} E = \boldsymbol{0}$:

$$\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{t}. \tag{4.2}$$

These are the celebrated *normal equations*: a linear system we need to solve in order to obtain $\boldsymbol{w}$. Since $\boldsymbol{\Phi}$ has full rank, it is easy to see that $\boldsymbol{\Phi}^T\boldsymbol{\Phi} \in \mathbb{R}^{p\times p}$ has full rank as well, therefore is invertible. We can write the solution as

$$\hat{\boldsymbol{w}} = \operatorname*{argmin}_{\boldsymbol{w}} E(\boldsymbol{w}) = \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^T\boldsymbol{t}. \tag{4.3}$$

Beyond simple techniques like gradient descent, many modern machine learning algorithms solve the normal equations inside, so it is important to understand how to do this well. Some advice is given in Section 4.2.3.

As an exercise, you should convince yourself that the solution for univariate linear regression of Section 4.1.1 is a special case of solving the normal equations.

4.2.1 Geometry of Least Squares Estimation

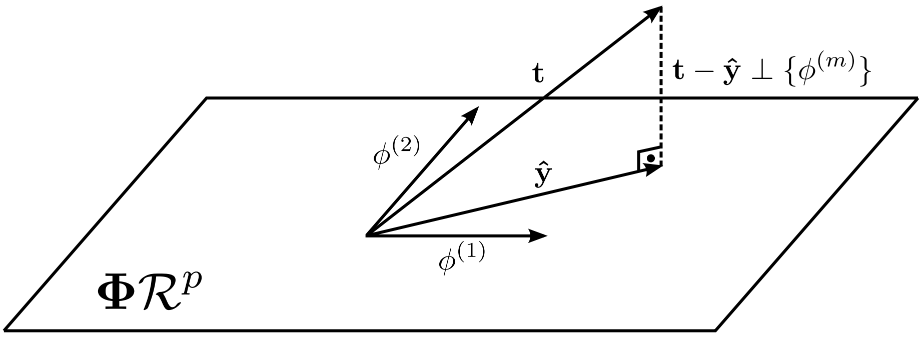

t
$\mathbf{t} - \hat{\mathbf{y}} \perp \{\phi^{(m)}\}$
$\phi^{(2)}$
$\hat{\mathbf{y}}$
$\phi^{(1)}$
$\mathbf{\Phi}\mathcal{R}^p$

Figure 4.4: Geometrical interpretation of linear least squares estimation. The vector of targets $\boldsymbol{t}$ is orthogonally projected onto the model subspace $\boldsymbol{\Phi}\mathbb{R}^p$, resulting in the prediction vector $\hat{\boldsymbol{y}} = \boldsymbol{\Phi}\hat{\boldsymbol{w}}$.

The first point to note about (4.3) is that $\hat{\boldsymbol{w}}$, and also $\hat{\boldsymbol{y}} = \boldsymbol{\Phi}\hat{\boldsymbol{w}}$, are linear maps of the vector of targets $\boldsymbol{t}$. Not only the forward mapping from weights to predictions is linear, but also the inverse, the estimator itself. What type of linear mapping is $\boldsymbol{t} \mapsto \hat{\boldsymbol{y}}$? The normal equations read $\boldsymbol{\Phi}^T(\boldsymbol{t}-\hat{\boldsymbol{y}}) = \boldsymbol{0}$. This means that the residual vector $\boldsymbol{t} - \hat{\boldsymbol{y}}$ is orthogonal to the subspace $\boldsymbol{\Phi}\mathbb{R}^p$ of possible prediction vectors (see Figure 4.4). Therefore,

$$\boldsymbol{t} = \underbrace{(\boldsymbol{t} - \hat{\boldsymbol{y}})}_{\perp \boldsymbol{\Phi}\mathbb{R}^p} + \underbrace{\hat{\boldsymbol{y}}}_{\in \boldsymbol{\Phi}\mathbb{R}^p}.$$

This decomposition of $\boldsymbol{t}$ into $\hat{\boldsymbol{y}}$ in $\boldsymbol{\Phi}\mathbb{R}^p$ and $\boldsymbol{t} - \hat{\boldsymbol{y}}$ orthogonal to $\boldsymbol{\Phi}\mathbb{R}^p$ is unique (Section 2.1.1): $\hat{\boldsymbol{y}}$ is the *orthogonal projection of* $\boldsymbol{t}$ *onto the model space* $\boldsymbol{\Phi}\mathbb{R}^p$. Try to get a feeling for this result, by drawing in $\mathbb{R}^2$. $\hat{\boldsymbol{y}}$ is the closest point to $\boldsymbol{t}$ in $\boldsymbol{\Phi}\mathbb{R}^p$, so $\boldsymbol{t} - \hat{\boldsymbol{y}}$ must be orthogonal to this subspace.

4.2.2 Techniques: Orthogonal Projection. Quadratic Functions

Let us review some general properties about orthogonal projections. Suppose $\mathcal{U}$ is a subspace of $\mathbb{R}^n$ (above, $\mathcal{U} = \boldsymbol{\Phi}\mathbb{R}^p$). Then, any $\boldsymbol{t} \in \mathbb{R}^n$ can be written uniquely as $\boldsymbol{t} = \boldsymbol{t}_{\|} + \boldsymbol{t}_{\perp}$, $\boldsymbol{t}_{\|} \in \mathcal{U}$, $(\boldsymbol{t}_{\perp}) \in \mathcal{U}^{\perp}$ (orthogonal complement, Section 2.1.1), since $\mathbb{R}^n = \mathcal{U} \oplus \mathcal{U}^{\perp}$ (direct sum). Then, $\boldsymbol{t}_{\|}$ is the orthogonal projection of $\boldsymbol{t}$ onto $\mathcal{U}$. If the columns of $\boldsymbol{\Phi}$ form a basis of $\mathcal{U}$, then $\boldsymbol{t}_{\|}$ is determined by

$$\boldsymbol{\Phi}^T(\boldsymbol{t} - \boldsymbol{t}_{\|}) = \boldsymbol{0}, \quad \boldsymbol{t}_{\|} = \boldsymbol{\Phi}\boldsymbol{w},\ \boldsymbol{w} \in \mathbb{R}^p,$$

solved by

$$\boldsymbol{t}_{\|} = \underbrace{\boldsymbol{\Phi}\left(\boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^T}_{=\boldsymbol{M}}\boldsymbol{t}.$$

$\boldsymbol{M}$ is the matrix of the orthogonal projection. Its properties are clear from our geometrical picture: $\boldsymbol{M}\boldsymbol{u} = \boldsymbol{u}$ for $\boldsymbol{u} \in \mathcal{U}$, and $\boldsymbol{M}\boldsymbol{v} = \boldsymbol{0}$ for $\boldsymbol{v} \in \mathcal{U}^\perp$. In other words, $\ker \boldsymbol{M} = \mathcal{U}^\perp$, and $\boldsymbol{M}$ has eigenvalues 1 (p dimensions) and 0 ($n-p$ dimensions) only (see Section 11.1.2).

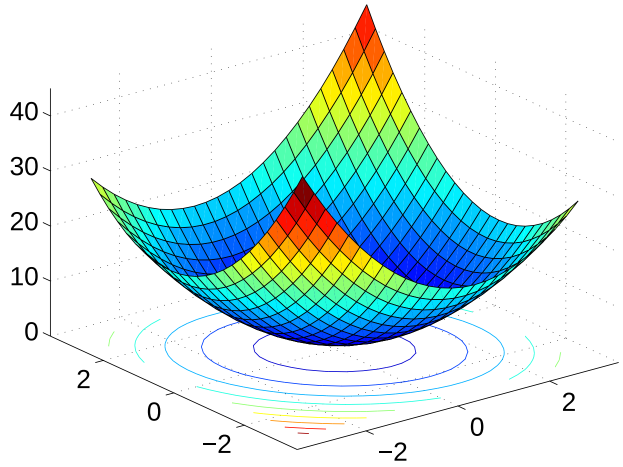


Figure 4.5: Plot of a positive definite quadratic function.

The linear least squares problem is a special case of minimizing a quadratic function, a topic which we will require several times during the course. A general quadratic function is $q(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + c$, where $\boldsymbol{A} \in \mathbb{R}^{d\times d}$, $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^d$. Convince yourself that we can always replace $\boldsymbol{A}$ by $\frac{1}{2}(\boldsymbol{A}+\boldsymbol{A}^T)$ without changing $q(\boldsymbol{x})$, so we can assume that $\boldsymbol{A}$ is a *symmetric matrix*: $\boldsymbol{A}^T = \boldsymbol{A}$. Our problem is

$$\min_{\boldsymbol{x}} \left\{ q(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + c \right\}.$$

We want to find the minimum value $q_* = \min_{\boldsymbol{x}} q(\boldsymbol{x})$ and a minimizer $\boldsymbol{x}_*$. This problem makes sense only if $q(\boldsymbol{x})$ is lower bounded. If $\boldsymbol{d} \neq \boldsymbol{0}$ is such that $\alpha = \boldsymbol{d}^T\boldsymbol{A}\boldsymbol{d} \leq 0$, then

$$q(t\boldsymbol{d}) = \frac{1}{2}t^2\alpha - t\boldsymbol{b}^T\boldsymbol{d} + c \to -\infty \quad (t \to \infty).$$

For the case $\alpha = 0$, we pick $\boldsymbol{d}$ so that $\boldsymbol{b}^T\boldsymbol{d} > 0$ (ignoring the special case that $\boldsymbol{b}^T\boldsymbol{d}$ may be zero). This means that in general[3], we require that $\boldsymbol{d}^T\boldsymbol{A}\boldsymbol{d} > 0$ for all $\boldsymbol{d} \neq \boldsymbol{0}$. A symmetric matrix with this property is called *positive definite*. It is these matrices which give rise to lower bounded quadratics, curving upwards like a bowl in all directions (Figure 4.5). To solve our problem: $\nabla_{\boldsymbol{x}} q(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = \boldsymbol{0}$, so that $\boldsymbol{x}_* = \boldsymbol{A}^{-1}\boldsymbol{b}$. A positive definite matrix $\boldsymbol{A}$ is also invertible (why?), so $\boldsymbol{x}_*$ is the unique stationary point, which is a global minimum point by virtue

[3] For the meticulous: The precise condition is $\boldsymbol{d}^T\boldsymbol{A}\boldsymbol{d} \geq 0$ for all $\boldsymbol{d} \neq \boldsymbol{0}$, and if $\boldsymbol{d}^T\boldsymbol{A}\boldsymbol{d} = 0$, then $\boldsymbol{b}^T\boldsymbol{d}$ must be zero as well.

of the Hessian $\nabla\nabla_{\boldsymbol{x}} q(\boldsymbol{x}) = \boldsymbol{A}$ being positive definite. Minimizing quadratic functions is equivalent to solving positive definite linear systems.

For moderate d (up to a few thousand), the best way to solve such a system is as follows. We use the *Cholesky decomposition* [45, ch. 6.5] $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L}$ is lower triangular ($l_{ij} = 0$ for $i < j$) and invertible. This decomposition exists if and only if $\boldsymbol{A}$ is positive definite, and it is easy to compute in $O(d^3)$. Then,

$$\boldsymbol{L}\left(\boldsymbol{L}^T\boldsymbol{x}_*\right) = \boldsymbol{b} \quad \Leftrightarrow \quad \boldsymbol{x}_* = \boldsymbol{L}^{-T}\left(\boldsymbol{L}^{-1}\boldsymbol{b}\right).$$

Here, $\boldsymbol{C}^{-T}$ is short for $(\boldsymbol{C}^{-1})^T = (\boldsymbol{C}^T)^{-1}$. Solving two systems instead of one? The point is that systems with $\boldsymbol{L}$ and $\boldsymbol{L}^T$ are easy to solve by backsubstitution, in $O(d^2)$. Hints and code pointers are given in Section 4.2.3.

4.2.3 Solving the Normal Equations (*)

How to solve the normal equations (4.2) in practice? There are many bad ways to do it, and some good ways. In this section, we focus on the best practice, obtained by numerical mathematicians over a long period. The history and details can be found in [18]. Recall that $n \geq p$ and $\operatorname{rk} \boldsymbol{\Phi} = p$. The main difficulty with solving (4.2) is that while $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ is invertible in exact arithmetic, in practice it is often just barely so: some of its eigenvalues can be very close to zero, in other words its condition number (ratio between largest and smallest eigenvalue) can be very large. For such matrices, numerical roundoff errors can get amplified dramatically. We will understand the statistical meaning of close-to-singular, or *ill-conditioned* $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ in Chapter 7, where we study mechanisms to improve the conditioning. For now, assume we really just want to solve (4.2) to best practice.

In essence, there are two different families of solvers: *direct* and *iterative* methods. As a simple rule, you use the former if n and p are not too large, otherwise you use the latter. "Not too large" depends on your hardware, but you should certainly use direct methods for n and p up to a few thousand. In contrast, if n and p are in the tens of thousands or beyond, you will have to go for iterative methods. In a nutshell, direct methods are black-box and safe to use. If a best practice direct method breaks down, your problem is hopeless. On the other hand, runtime and memory requirements of a direct method is $O(np^2)$ and $O(p^2)$ respectively, which may be prohibitive. Special structure or sparsity of $\boldsymbol{\Phi}$ is often present, but in order to use it to speed things up, you need to employ an iterative solver.

A warning up front. Maybe the *worst* way to approach the normal equations is to compute and invert $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$, something you should *never* do even with small n and p. Matrix inversion is mainly a theoretical concept, it should not be used in practice due to its inherently poor numerical properties. In the real world, we use matrix *factorizations* (such as Cholesky or QR) instead of inversion. In the case of the normal equations, we should not even compute and factorize $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$. Namely, if $\boldsymbol{\Phi}$ has a large condition number, the condition number of $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ is the square of that, which is much worse. As we will see next, the normal equations can be solved without ever computing $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$.

A best practice direct method for solving (4.2) employs the QR decomposition: $\boldsymbol{\Phi} = \boldsymbol{Q}\boldsymbol{R}$, where $\boldsymbol{Q} \in \mathbb{R}^{n\times p}$ has orthonormal columns ($\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}$), and $\boldsymbol{R} \in$

$\mathbb{R}^{p\times p}$ is upper triangular ($r_{ij} = 0$ for $i > j$) and invertible [25, 45]. The QR decomposition is computed using Householder's method. We do not have to care about this, as we simply use good numerical[4] code. Plugging this into the normal equations and using $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}$, you get $\boldsymbol{R}^T\boldsymbol{R}\boldsymbol{w} = \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{t}$, or $\boldsymbol{R}\boldsymbol{w} = \boldsymbol{Q}^T\boldsymbol{t}$ (as $\boldsymbol{R}$ is invertible). This is solved as

$$\hat{\boldsymbol{w}} = \boldsymbol{R}^{-1}(\boldsymbol{Q}^T\boldsymbol{t}),$$

by a simple algorithm called *backsubstitution* (exercise: derive backsubstitution for yourself; solution in [45, ch. 2.2]). The cost of the QR decomposition is $O(n\,p^2)$, so you see that p matters more than n. It is not necessary to store $\boldsymbol{Q}$, since $\boldsymbol{Q}^T\boldsymbol{t}$ can be computed alongside the decomposition. Note that $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ is not computed.

Let us get some geometrical intuition about the QR decomposition, linking it to the geometrical picture of Section 4.2.1. We can decompose the squared distance as

$$\|\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\|^2 = \|\boldsymbol{R}\boldsymbol{w} - \boldsymbol{Q}^T\boldsymbol{t}\|^2 + \|\boldsymbol{t} - \boldsymbol{Q}\boldsymbol{Q}^T\boldsymbol{t}\|^2. \tag{4.4}$$

This is because

$$\begin{aligned}\|\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\|^2 &= \|(\boldsymbol{Q}\boldsymbol{R}\boldsymbol{w} - \boldsymbol{Q}\boldsymbol{Q}^T\boldsymbol{t}) + (\boldsymbol{Q}\boldsymbol{Q}^T\boldsymbol{t} - \boldsymbol{t})\|^2 \\ &= \|\boldsymbol{Q}\boldsymbol{v} + (\boldsymbol{Q}\boldsymbol{Q}^T\boldsymbol{t} - \boldsymbol{t})\|^2, \quad \boldsymbol{v} = \boldsymbol{R}\boldsymbol{w} - \boldsymbol{Q}^T\boldsymbol{t}.\end{aligned}$$

Note that $\|\boldsymbol{Q}\boldsymbol{v}\|^2 = \|\boldsymbol{v}\|^2$ by the column orthonormality of $\boldsymbol{Q}$, and

$$(\boldsymbol{Q}\boldsymbol{v})^T(\boldsymbol{Q}\boldsymbol{Q}^T\boldsymbol{t} - \boldsymbol{t}) = \boldsymbol{v}^T(\boldsymbol{Q}^T\boldsymbol{t} - \boldsymbol{Q}^T\boldsymbol{t}) = 0,$$

so that the "cross-talk" vanishes (Section 2.1.1). The solution $\hat{\boldsymbol{w}}$ makes the first term in (4.4) vanish, and the second is the remaining (minimum) squared distance. Recalling Section 4.2.1,

$$\hat{\boldsymbol{y}} = \boldsymbol{\Phi}\hat{\boldsymbol{w}} = \boldsymbol{Q}\boldsymbol{R}\hat{\boldsymbol{w}} = \boldsymbol{Q}\boldsymbol{Q}^T\boldsymbol{t}$$

is the orthogonal projection of $\boldsymbol{t}$ onto the space $\boldsymbol{\Phi}\mathbb{R}^p$, and $\boldsymbol{Q}$ is an orthonormal basis of this space. The projection matrix is

$$\boldsymbol{M} = \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^T = \boldsymbol{Q}\boldsymbol{Q}^T.$$

A discussion of iterative solvers is beyond the scope of this course. For systems of the form (4.2), with $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ symmetric positive definite, the most commonly used method is *conjugate gradients* [18]. I can recommend the intuitive derivation in [5, ch. 7.7]. However, specifically for the normal equations, the `LSQR` algorithm [33] is to be preferred (code at `http://www.stanford.edu/group/SOL/software/lsqr.html`).

[4] Good numerical code is found on `www.netlib.org`. The standard package for matrix decompositions (QR, Cholesky, up to the singular value decomposition) is `LAPACK`. This code is wrapped in `Matlab`, `Octave`, `Python`, or the GNU scientific library. In contrast, "Numerical recipes for X" is not recommended.

Chapter 5

Probability. Decision Theory

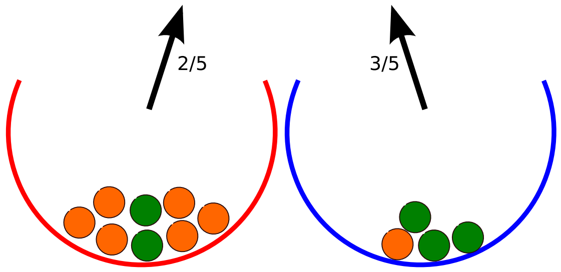
In this chapter, we refresh our knowledge about probability, the language and basic calculus behind decision making and machine learning. We also introduce concepts from decision theory, setting the stage for further developments. Raising from technical details like binary classifiers, linear functions, or numerical optimizers up towards the big picture, we will understand how decision making works in the optimal case, and which probabilistic aspects of a problem are relevant. We will see that Bayesian computations, or *inference*, are the basis for *optimal decision making.* In later chapters, we will see that *learning* is based on inference as well.

5.1 Essential Probability

Probability is the language of modern machine learning. The most important concept you need when you want to think about, formalize and automate learning and decision making, is *uncertainty*, and probability is the calculus of uncertainty. Pierre Simon Laplace, a founding father of probability theory: "Probability is nothing but common sense reduced to calculation." When humans think, act, speak, or reason, they use probabilistic statements all the time ("the chance of rain tomorrow is 20%", "it is more likely that Switzerland wins the next world cup than Austria"). There are a range of books which lucidly explain why valid reasoning needs probability [34]. Machine learning needs decision making in the presence of uncertainty about the data (measurement errors, outliers, missing cases), the parameters, and even the model itself. If we do not understand why and how probability ties all these concepts together into a consistent whole, we may end up chasing algorithms, estimators, loss functions, hypothesis classes, weighting or voting schemes, theoretical assumptions and "learning paradigms" forever, never to see the wood for all the trees.

This course is about machine learning, but let us briefly step out of this context, so that you do not get the wrong impression. The importance of probability, understanding and quantifying uncertainty and risk, far transcends machine learning, statistics, and even scientific applications. There is simply no other rational way to make sense of the data gathered today which is not based on these concepts. We are bombarded with a growing stream of ever-more sensational and poorly researched news. In order to maintain a half-way consistent world view about important topics, we have to understand the context in which these numbers live, and that is provided by probability and statistics. The most interesting jobs at cool companies (such as your favourite search engine) go to those with a profound understanding of probability and algorithms. Statistics and probability is used to decipher speech, decode communications, route data packets, predict prices, understand images, make robots explore their environment and translate natural language texts. Probability is useful but also great fun. `http://understandinguncertainty.org/` is a website dedicated to understanding probability. If you have the impression that statistics is the boring pastime of old men in tweeds, watch this movie `http://www.gapminder.org/videos/the-joy-of-stats/` and think again.

We cannot give more than a slight reminder of probability in this section, informal and on a very elementary level. This is not enough to see what it is all about and to enjoy it. The role that probability plays for machine learning should motivate you to refresh your knowledge. There are many good elementary books on this topic, for example [20, 19, 9]. The classics remain the books by Feller [14, 15].



	Box	
Fruit	**r**ed	**b**lue
apple	1/10	9/20
orange	3/10	3/20

Figure 5.1: A simple experiment to introduce probability concepts. There are two **B**oxes, a **r**ed and a **b**lue one. Each contain a certain number of **F**ruit, namely **a**pples and **o**ranges (the apples are the green balls). One of the boxes is picked with probability $P(B)$, then a fruit is drawn out if it at random, according to $P(F|B)$.

We will use an example losely based on [6, ch. 1.2]. We draw a fruit out of a box, as illustrated and explained in Figure 5.1. When thinking about a probabilistic setup, the following concepts have to be defined:

- The probability (or sample) space Ω: the set of all possible outcomes. In our example, (F, B) can take values $\Omega = \{(a, r), (o, r), (a, b), (o, b)\}$.

- The joint probability distribution P over the probability space. This is a nonnegative function on Ω which sums to one over the whole space:

$$P(\omega) \geq 0, \;\; \omega \in \Omega, \quad \sum_{\omega \in \Omega} P(\omega) = 1.$$

In our example, $P(\omega) = P(F, B)$ is given by the table in Figure 5.1.

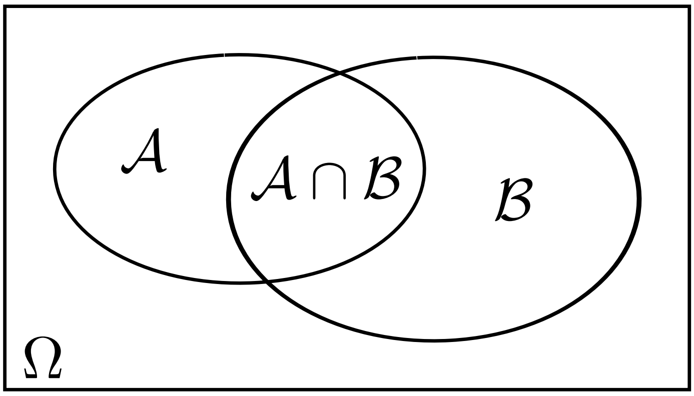


Figure 5.2: Illustration of events $\mathcal{A}$, $\mathcal{B}$ within a probability space Ω. The joint event $\mathcal{A} \cap \mathcal{B}$ is the intersection of $\mathcal{A}$ and $\mathcal{B}$, containing all outcomes ω which lie both in $\mathcal{A}$ and in $\mathcal{B}$.

At least for a finite sample space Ω, we can picture probabilities as sizes of subsets $\mathcal{A} \subset \Omega$, called *events*. In Figure 5.2, we illustrate two events $\mathcal{A}$, $\mathcal{B}$. Their intersection is the *joint* event $\mathcal{A} \cap \mathcal{B}$, the set of all outcomes both in $\mathcal{A}$ and in $\mathcal{B}$. For example, if $\mathcal{A} = \{F = a\} = \{(a, r), (a, b)\}$ and $\mathcal{B} = \{B = r\} = \{(a, r), (o, r)\}$, then $\mathcal{A} \cap \mathcal{B} = \{F = a, B = r\} = \{(a, r)\}$. The *probability* of an event $\mathcal{E}$ is obtained by summing $P(\omega)$ over all $\omega \in \mathcal{E}$:

$$P(\mathcal{E}) = \sum_{\omega \in \mathcal{E}} P(\omega).$$

For example,

$$P(\{F = a\}) = P((a, r)) + P((a, b)) = 1/10 + 9/20 = 11/20.$$

Note that both Ω and $\emptyset$ (the empty set) are events, with $P(\Omega) = 1$ and $P(\emptyset) = 0$ for any joint probability distribution P. Many rules of probability can be understood by drawing Venn diagrams such as Figure 5.2. For example,

$$P(\{\mathcal{A} \, \mathrm{or} \, \mathcal{B}\}) = P(\mathcal{A} \cup \mathcal{B}) = P(\mathcal{A}) + P(\mathcal{B}) - P(\mathcal{A} \cap \mathcal{B}).$$

For most probability experiments, the relevant events are conveniently described by *random variables*. Formally, a random variable is a function from Ω into some set, for example $\mathbb{R}$, $\{0, 1\}$, or $\{a, o\}$. This notion is so natural that we have chosen it implicitly in order to define the example in Figure 5.1: F is a random variable mapping to $\{a, o\}$, and B is a random variable mapping to $\{r, b\}$. We can define random variables in terms of others. For example $\mathrm{I}_{\{F=a \, \mathrm{or} \, B=b\}}$ is a random variable mapping into $\{0, 1\}$. It is equal to zero for the outcome $\omega = (o, r)$, equal to one otherwise.

The *joint probability distribution*, $P(F, B)$, is a complete description of the experiment. All other distributions are derived from it. First, there are *marginal* *distributions*. Suppose we want to predict which fruit will be drawn with which probability, no matter what the box. Every time you hear "no matter what B", you translate "sum over all possible values of B". The marginal distribution $P(F)$ is

$$P(F) = \sum_{B=r,b} P(F,B) = \left\{ \begin{array}{ll} 11/20 & \mid F = a \\ 9/20 & \mid F = o \end{array} \right\}.$$

$P(F)$ is a probability distribution just like $P(F,B)$ (confirm this for yourself). It can also be understood in terms of events and Venn diagrams (Figure 5.2). For our example above, $\mathcal{A} = \{F = a\}$ is the union of the disjoint events $\{F = a, B = r\} = \mathcal{A} \cap \mathcal{B}$ and $\{F = a, B \neq r\} = \mathcal{A} \cap (\Omega \setminus \mathcal{B})$.

Second, there are *conditional distributions*. Given that I picked the red box ($B = r$), which fruit will I get? This is $P(F|B = r)$, the conditional distribution of F, given that $B = r$. There are three times as many oranges as apples in the red box, so $P(F = o|B = r) = 3/4$, $P(F = a|B = r) = 1/4$. Here is a rule linking all these probabilities. I pick (F, B) according to $P(F, B)$ by first picking B according to $P(B)$, then F according to $P(F|B)$:

$$P(F, B) = P(F|B)P(B).$$

If you are given a joint distribution $P(F, B)$, you can first work out the marginals $P(B)$ and $P(F)$ by summing out over the other variable respectively, then obtain

$$P(F|B) = \frac{P(F,B)}{P(B)},\ P(B) \neq 0, \quad P(B|F) = \frac{P(F,B)}{P(F)},\ P(F) \neq 0.$$

For our Venn diagram example,

$$P(\mathcal{A}|\mathcal{B}) = P(\{F = a\}|\{B = r\}) = \frac{P(\{F = a, B = r\})}{P(\{B = r\})} = \frac{P(\mathcal{A} \cap \mathcal{B})}{P(\mathcal{B})}.$$

Given that we are in $\mathcal{B}$, the probability of being in $\mathcal{A}$ is obtained by the fractional size of the intersection $\mathcal{A} \cap \mathcal{B}$ within $\mathcal{B}$.

Note that we speak about *the* conditional distribution $P(F|B)$, even though it is really a set of distributions, $P(F|B = r)$ and $P(F|B = b)$. Storing it in a table needs as much space as the joint distribution. Note that if, for a different experiment, $P(B = r)$ was zero, then $P(F|B = r)$ would remain undefined: it does not make sense to reason about F given $B = r$ if $B = r$ cannot happen. To sum up, you only really need two rules:

- **Sum rule**: If $P(X, Y)$ is a joint distribution, the marginal distribution $P(X)$ is obtained by summing Y over all possible values:

  $$P(X) = \sum_{Y} P(X, Y).$$

  The interpretation of summing (or *marginalization*) over Y is disinterest in its value. What will the fruit be, no matter what the box?

- **Product rule**: If $P(X, Y)$ is a joint distribution with conditional distribution $P(X|Y)$ and marginal distribution $P(Y)$, then

  $$P(X, Y) = P(X|Y)P(Y).$$

> This rule is the basis for factorizations of probability distributions into simpler components.

These rules formalize two basic operations which drive probabilistic reasoning. Whenever a variable Y is observed, so we know its value (for example, it is a case in a dataset), we *condition* on it, consider *conditional distributions given* Y. Whenever a variable X, whose precise value is uncertain, is not currently of interest (not the aim of prediction), we *marginalize* over it, sum distributions over all values of X, thereby eliminate it.

There is no sense in which the variables F and B are ordered, we could write $P(B,F)$ instead of $P(F,B)$. After all, the intersection of events (sets) is not ordered either. In fact, in derivations, we will sometimes simply use $P(B,F) = P(F,B)$: the argument names matter, not their ranking. Given that, let us apply the product rule in both orderings:

$$\begin{aligned}
&P(F|B)P(B) = P(F,B) = P(B,F) = P(B|F)P(F) \quad \Rightarrow \\
&P(B|F) = \frac{P(F|B)P(B)}{P(F)}.
\end{aligned}$$

This is *Bayes' formula* (or Bayes' rule, or Bayes' theorem). This looks harmless, but substitute “cause” for B, "effect" for F, and Bayes' formula provides the ultimately solution for the inverse problem of reasoning. Suppose you show me the fruit you got: $F = o$ (an orange). What can I say about the box you picked?

$$\begin{aligned}
&P(B=r|F=o) = \frac{P(F=o|B=r)P(B=r)}{P(F=o)} = \frac{3/4 \cdot 2/5}{9/20} = 2/3, \\
&P(B=b|F=o) = 1 - P(B=r|F=o) = 1/3.
\end{aligned}$$

It is twice as likely that your box was the red one. One point should become clear even from this simple example. The fact that events happen at random and we have to be *uncertain* about them, does *not mean that they are entirely unpredictable.* It just means that we might not be able to predict them with complete certainty. Most events around you are uncertain to some degree, almost none are completely unpredictable. The key to decision making from uncertain knowledge is to quantify your probabilistic belief[1] in *dependent* variables, so that if you observe some of them, you can predict others by probability computations.

5.1.1 Independence. Conditional Independence

Let us consider two random variables X, Y with joint distribution $P(X,Y)$. Suppose we observe X. Does this tell us anything new about Y? Our knowledge about Y *without* observing X is $P(Y)$. Knowing X, this becomes $P(Y|X)$. X contains no information about Y if and only if

$$P(Y|X) = P(Y) \quad \Leftrightarrow \quad P(X,Y) = P(X)P(Y).$$

Such variables are *independent*: the joint distribution is the product of the marginals. Equivalently, the events $\mathcal{A}$ and $\mathcal{B}$ are independent if $P(\mathcal{A}|\mathcal{B}) = P(\mathcal{A})$,

[1] Belief, or “subjective probability”, is another word for probability distribution.

or $P(\mathcal{A} \cap \mathcal{B}) = P(\mathcal{A})P(\mathcal{B})$. For example, if we pick a fruit F_r from the red box and a fruit F_b from the blue box, then F_r and F_b are independent random variables. Within a machine learning problem, independent variables are good and bad. They are good, because we can treat them separately, thereby saving computing time and memory. Independence can simplify derivations a lot. They are bad, because we only learn things from dependent variables. Independence is often too strong a concept, we need something weaker. Consider three random variables X, Y, Z with joint distribution $P(X, Y, Z)$. X and Y are *conditionally independent, given* Z if they are independent under the conditional $P(X, Y|Z)$, namely if $P(X, Y|Z) = P(X|Z)P(Y|Z)$. Conditional independence constraints are useful for learning. In general, X, Y, Z are still all dependent, but the conditional independence structure can be used to simplify derivations and computations. In Figure 5.1, draw a box B, then two fruits F_1, F_2 at random from B (with replacement: you put the fruits back). Then, $P(F_1, F_2|B) = P(F_1|B)P(F_2|B)$, but $P(F_1, F_2) \neq P(F_1)P(F_2)$ (check for yourself that $P(F_2 = o|F_1 = o) = 7/12 \neq P(F_2 = o) = 9/20$). Note that it can go the other way around. Throw two dice $X, Y \in \{1, \dots, 6\}$ independently. If I tell you that $Z = X + Y = 7$, then X and Y are dependent given Z, but they are independent as such.

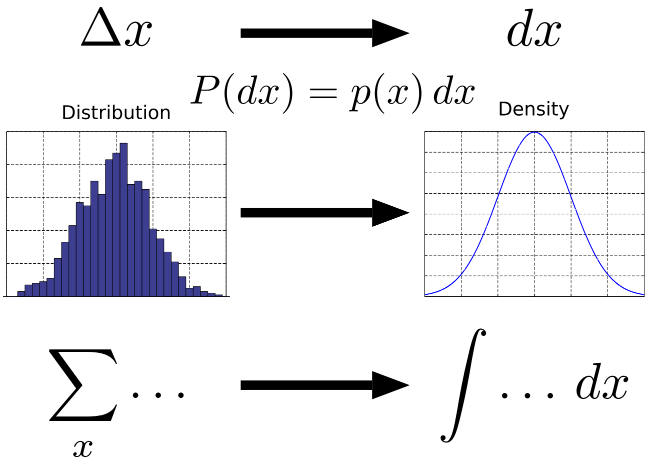


Figure 5.3: From a discrete distribution (histogram) on a finer and finer grid to a continuous probability density.

5.1.2 Probability Densities

Recall $\boldsymbol{x}_i \in \mathbb{R}^d$, $\boldsymbol{w} \in \mathbb{R}^p$. We need probability distributions over continuous variables as well. The probability space Ω is a continuous (overcountably) infinite set, and $P(\cdot)$ is a probability measure. Sums over $\omega \in \Omega$ do not make sense anymore, but from calculus we know how they are replaced by integrals. Consider a random variable $x \in \mathbb{R}$ defined on such a space. Imagine $\mathbb{R}$ being gridded up into cells of size Δx each. The interval $[x, x + \Delta x)$ is an event with probability $P([x, x + \Delta x))$. The key idea is to relate this probability "volume" (the probability of x landing in the interval) to the volume Δx of the interval itself:

$$p(x) = \lim_{\Delta x \to 0} \frac{P([x, x + \Delta x))}{\Delta x}.$$

Under technical conditions on $P(\cdot)$ and the random variable, this limit exists for all $x \in \mathbb{R}$: it is called the *probability density* of the random variable x. Just as for P, we have that

$$p(x) \geq 0, \quad x \in \mathbb{R}, \qquad \int p(x)\, dx = 1.$$

A "graphical proof" for the latter is given in Figure 5.3. Careful: the value $p(x)$ of a density at some $x \in \mathbb{R}$ can be larger than 1, in fact some densities are unbounded[2] above. The density allows us to compute probabilities over x, for example:

$$P(\{x \in [a, b]\}) = \int_a^b p(x)\, dx = \int \mathrm{I}_{\{a \leq x \leq b\}} p(x)\, dx, \quad P(\mathcal{E}) = \int \mathrm{I}_{\{x \in \mathcal{E}\}} p(x)\, dx.$$

The *cumulative distribution function* is

$$F(x) = P(\{x \in (-\infty, x]\}) = \int_{-\infty}^{x} p(t)\, dt \quad \Rightarrow \quad p(x) = F'(x).$$

In this course, what probability distributions are for discrete variables, probability densities are for continuous variables. Sum and product rule work for densities as well:

$$p(x) = \int p(x, y)\, dy, \qquad p(x, y) = p(x|y)p(y),$$

and so does Bayes rule, always replacing sums by integrals. At this point, you will start to appreciate random variables. If x and y are random variables, so are e^x and $x+y$, and computing probabilities for the latter is merely an exercise in integration.

This is about the level of formality we need in this course. However, you should be aware that we are avoiding some difficulties here which do in fact become relevant in a number of branches of machine learning, notably nonparametric statistics and learning theory. For example, it is not possible to construct a probability space on $\mathbb{R}$ so that every subset is an event. We have to restrict ourselves to measurable (or Borel) sets. Also, not every function is allowed as random variable, and not every distribution of a random variable has a probability density (a simple example: the constant variable $x = 0$ does not have a density). Things also become difficult if we consider an infinite number of random variables at the same time, for example in order to make statements about limits of random variable sequences. None of these issues are in the scope of this course. Good general expositions are given in [4, 19].

5.1.3 Expectations. Mean and Covariance

Let $x \in \mathbb{R}$ be a random variable. The *expectation* (or *mean*, or expected value) of x is

$$\mathrm{E}[x] = \sum_x x P(x)$$

[2] An example is the gamma density $p(x) = \pi^{-1/2} x^{-1/2} e^{-x} \mathrm{I}_{\{x>0\}}$.

if x is discrete with distribution $P(x)$,

$$\mathrm{E}[x] = \int x p(x)\, dx$$

if x is continuous with density $p(x)$ (always under the assumption that sum or integral exists). Note that our definition covers extensions such as

$$\mathrm{E}[f(x)] = \int f(x) p(x)\, dx,$$

since $f(x)$ is a random variable if x is one. If $\boldsymbol{x} \in \mathbb{R}^d$ is a random vector, then

$$\mathrm{E}[\boldsymbol{x}] = [\mathrm{E}[x_j]] \in \mathbb{R}^d.$$

Expectation is linear: if x, y are random variables, $\alpha \in \mathbb{R}$ a constant, then

$$\mathrm{E}[x + \alpha y] = \mathrm{E}[x] + \alpha \mathrm{E}[y].$$

If x and y are independent, then

$$\mathrm{E}[xy] = \mathrm{E}[x]\mathrm{E}[y].$$

However, the reverse[3] is not true in general. Moreover, the *conditional expectation* is

$$\mathrm{E}[x \,|\, y] = \sum_x x P(x|y)$$

or

$$\mathrm{E}[x \,|\, y] = \int x p(x|y)\, dx.$$

Note that in general, $\mathrm{E}[x \,|\, y]$ is itself a random variable (a function of y, which is random), although we can also consider expectations conditioned on an event, for example

$$\mathrm{E}[x \,|\, y = y_0] = \sum_x x P(x|y = y_0),$$

which are simply numbers.

Variance and Covariance

Picture the mean as value around which a random variable is fluctuating. By how much? The *variance* gives a good idea:

$$\mathrm{Var}[x] = \mathrm{E}\left[(x - \mathrm{E}[x])^2\right],$$

the expected squared distance of x from its mean. Another formula for the variance is

$$\mathrm{Var}[x] = \mathrm{E}\left[x^2 - 2x\mathrm{E}[x] + \mathrm{E}[x]^2\right] = \mathrm{E}[x^2] - \mathrm{E}[x]^2.$$

[3] If $\mathrm{E}[f(x)g(y)] = \mathrm{E}[f(x)]\mathrm{E}[g(y)]$ for every pair of (measurable) functions f, g, then x and y are independent.

Again,

$$\mathrm{Var}[x\,|\,y]=\mathrm{E}\left[(x-\mathrm{E}[x])^2\,|\,y\right]=\mathrm{E}[x^2\,|\,y]-\mathrm{E}[x\,|\,y]^2.$$

Mean and variance are examples of *moments* of a distribution. The *covariance* between random variables $x, y\in\mathbb{R}$ is

$$\begin{aligned}\mathrm{Cov}[x,y]&=\mathrm{E}\left[(x-\mathrm{E}[x])(y-\mathrm{E}[y])\right]=\mathrm{E}\left[xy-\mathrm{E}[x]y-x\mathrm{E}[y]+\mathrm{E}[x]\mathrm{E}[y]\right]\\&=\mathrm{E}[xy]-\mathrm{E}[x]\mathrm{E}[y].\end{aligned}$$

Note that

$$|\mathrm{Cov}[x,y]|\le\sqrt{\mathrm{Var}[x]\mathrm{Var}[y]},$$

a special case of the Cauchy-Schwarz inequality (Section 2.3). Note that

$$\mathrm{Var}[x+y]=\mathrm{Var}[x]+2\mathrm{Cov}[x,y]+\mathrm{Var}[y].$$

Namely, if $x'=x-\mathrm{E}[x]$, $y'=y-\mathrm{E}[y]$, then

$$\mathrm{Var}[x+y]=\mathrm{E}\left[(x'+y')^2\right]=\mathrm{E}[(x')^2]+2\mathrm{E}[x'y']+\mathrm{E}[(y')^2].$$

Therefore, $\mathrm{Var}[x+y]=\mathrm{Var}[x]+\mathrm{Var}[y]$ if x and y are uncorrelated, which holds in particular if they are independent. More general, if $\boldsymbol{x}\in\mathbb{R}^d$ is a random vector, we collect all $\mathrm{Cov}[x_j,x_k]$ in the *covariance matrix*

$$\mathrm{Cov}[\boldsymbol{x}]=[\mathrm{Cov}[x_j,x_k]]=\mathrm{E}\left[(\boldsymbol{x}-\mathrm{E}[\boldsymbol{x}])(\boldsymbol{x}-\mathrm{E}[\boldsymbol{x}])^T\right]=\mathrm{E}\left[\boldsymbol{x}\boldsymbol{x}^T\right]-\mathrm{E}[\boldsymbol{x}]\mathrm{E}[\boldsymbol{x}]^T.$$

More general, the cross-covariance matrix between $\boldsymbol{x}\in\mathbb{R}^d$ and $\boldsymbol{y}\in\mathbb{R}^q$ is

$$\mathrm{Cov}[\boldsymbol{x},\boldsymbol{y}]=\mathrm{E}\left[(\boldsymbol{x}-\mathrm{E}[\boldsymbol{x}])(\boldsymbol{y}-\mathrm{E}[\boldsymbol{y}])^T\right]=\mathrm{E}\left[\boldsymbol{x}\boldsymbol{y}^T\right]-\mathrm{E}[\boldsymbol{x}]\mathrm{E}[\boldsymbol{y}]^T.$$

Mean and Covariance after Linear Transformation

If $\boldsymbol{x}\in\mathbb{R}^d$ is a random vector with mean $\mathrm{E}[\boldsymbol{x}]$, covariance matrix $\mathrm{Cov}[\boldsymbol{x}]$, what are the corresponding moments of $\boldsymbol{y}=\boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{A}\in\mathbb{R}^{q\times d}$? By linearity,

$$\mathrm{E}[\boldsymbol{A}\boldsymbol{x}]=\boldsymbol{A}\mathrm{E}[\boldsymbol{x}].$$

Also,

$$\begin{aligned}\mathrm{Cov}[\boldsymbol{A}\boldsymbol{x}]&=\mathrm{E}\left[\boldsymbol{A}\boldsymbol{x}(\boldsymbol{A}\boldsymbol{x})^T\right]-\mathrm{E}[\boldsymbol{A}\boldsymbol{x}]\mathrm{E}[\boldsymbol{A}\boldsymbol{x}]^T\\&=\boldsymbol{A}\mathrm{E}\left[\boldsymbol{x}\boldsymbol{x}^T\right]\boldsymbol{A}^T-\boldsymbol{A}\mathrm{E}[\boldsymbol{x}]\mathrm{E}[\boldsymbol{x}]^T\boldsymbol{A}^T=\boldsymbol{A}\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{A}^T,\end{aligned}$$

left-multiplication by $\boldsymbol{A}$, right-multiplication by $\boldsymbol{A}^T$.

Given a dataset $\mathcal{D}=\{\boldsymbol{x}_i\mid i=1,\dots,n\}$, *empirical mean* and *empirical covariance* (or sample mean, sample covariance) are computed as

$$\hat{\boldsymbol{\mu}}=\frac{1}{n}\sum_{i=1}^n\boldsymbol{x}_i,\quad\hat{\boldsymbol{\Sigma}}=\frac{1}{n}\sum_{i=1}^n\boldsymbol{x}_i\boldsymbol{x}_i^T-\hat{\boldsymbol{\mu}}\hat{\boldsymbol{\mu}}^T.$$

If the data points are drawn independently from a distribution with mean $\mathrm{E}[\boldsymbol{x}]$, covariance $\mathrm{Cov}[\boldsymbol{x}]$, then $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\Sigma}}$ converge[4] against $\mathrm{E}[\boldsymbol{x}]$ and $\mathrm{Cov}[\boldsymbol{x}]$ as $n\to\infty$. We will gain a more precise idea about these estimators in Chapter 6.

[4] Convergence happens almost surely (with probability one), a notion of stochastic convergence. This result is called *law of large numbers*.

Finally, be aware that not all continuous random variables with a density have a variance, some do not even have a mean, since corresponding integrals diverge. For example, the Cauchy distribution with density

$$p(x) = \frac{1}{\pi(1 + (x - x_0)^2)}.$$

does not have mean or variance. The reason for this is that too much probability mass resides in the *tails* $(-\infty, -x_0] \cup [x_0, \infty)$, x_0 large. Such *heavy-tailed distributions* become important in modern statistics and machine learning, as good models for data with outliers. They can be challenging to work with.

5.2 Decision Theory

Assessing, manipulating and learning about probabilities is a means towards an end. That end is expressed in decision theory. We want to act optimally, make optimal decisions in the presence of uncertain knowledge. This is a two-stage process. First, we condition on data in order to resolve uncertainty by way of probability calculus. Second, we take a decision which minimizes expected loss. Consider an example. In a hospital, a tissue sample is taken from a patient, giving rise to an input vector $\boldsymbol{x}$. An automatic classifier $f(\boldsymbol{x})$ is to predict whether the patient has cancer ($t = 1$) or not ($t = 0$). How can we calibrate this procedure in terms of familiar units?

In order to develop decision theory, it is customary to make the assumption that the *true* probabilistic law between relevant variables ($\boldsymbol{x}$ and t in our screening example) is known exactly. First, this allows us to talk quantitatively about the best possible solution to a statistical problem. Optimal solutions can be worked out for simple setups, and they will provide additional motivation for common model assumptions. Second, a decision-theoretic analysis can clarify what are the most important aspects about a problem, and we can concentrate modelling and learning efforts on those. Some vocabulary:

- *Class-conditional distribution/density* $p(\boldsymbol{x}|t)$: The distribution of inputs $\boldsymbol{x}$, given class label t. In our example, $p(\boldsymbol{x}|t = 0)$ is the distribution of tissue sample vectors for healthy, $p(\boldsymbol{x}|t = 1)$ for cancerous patients. Beware that the class-conditional distribution is termed "likelihood" in some books (this becomes clear in Section 6.2), but this nomenclature mixes up concepts and will not be used in this course.

- *Class prior probability distribution* $P(t)$: The distribution of the class label t on its own. What is the fraction of healthy versus cancerous patients to be exposed to our screening procedure?

- *Class posterior probability distribution* $P(t|\boldsymbol{x})$: Obtained from the class-conditional and prior probabilities by Bayes' rule:

$$P(t|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t)P(t)}{p(\boldsymbol{x})}, \qquad p(\boldsymbol{x}) = \sum_t p(\boldsymbol{x}|t)P(t). \tag{5.1}$$

The same definition applies to a multi-way classification problem, where $t \in \{0, \dots, K-1\}$. Given our "apples and oranges" intuition about probability, we would proceed as follows. The setup is defined by the joint probability density $p(\boldsymbol{x}, t) = p(\boldsymbol{x}|t)P(t)$. When a patient comes in, we have to predict t from $\boldsymbol{x}$, our (un)certainty about which is quantified by the posterior $P(t|\boldsymbol{x})$, so this would be the basis for our prediction.

5.2.1 Minimizing Classification Error

A natural goal in classification is to commit as few errors as possible, in other words to choose a classification rule $f(\boldsymbol{x})$ such that the *error probability*

$$R(f) = P\left\{f(\boldsymbol{x}) \neq t\right\}$$

is as small as possible. How does such an optimal classifier look like? Consider the example in Figure 5.4, where $x \in \mathbb{R}$ and $t \in \{1, 2\}$.

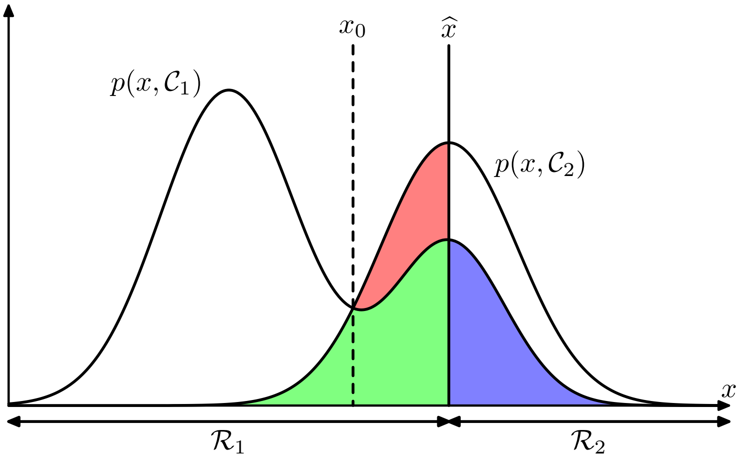


Figure 5.4: Joint distributions $p(x, \mathcal{C}_t) = p(x|t)P(t)$ for a binary classification problem with $x \in \mathbb{R}$, $\mathcal{T} = \{1, 2\}$. The classifier $f(x) = 1 + \mathrm{I}_{\{x \geq \hat{x}\}}$ has decision regions $\mathcal{H}_2 = [\hat{x}, \infty)$ and $\mathcal{H}_1 = (-\infty, \hat{x})$ (called $\mathcal{R}_2$ and $\mathcal{R}_1$ in the figure). Its error probability $R(f)$ is visualized as the combined area of the blue, green and red regions. The blue region stands for points x from class 1 being classified as $f(x) = 2$, while the union of green and red regions stands for points x from class 2 being classified as $f(x) = 1$. If we move the decision threshold away from $\hat{x}$, the combined area of green and blue regions stays the same. It symbolizes the unavoidable Bayes error R^*. On the other hand, we can reduce the red area to zero by setting $\hat{x} = x_0$, the point where the joint density functions intersect. $f^*(x) = 1 + \mathrm{I}_{\{x \geq x_0\}}$ is the Bayes-optimal rule.
Figure from [6] (used with permission).

It becomes clear from this example that the error $R(f)$ can be decomposed into several parts, some of which are intrinsic, while others can be avoided by a better choice of f. From the figure, it seems that the optimal classifier f^* follows the lower of the two curves $p(x|t)P(t)$, $t = 1, 2$, and its error R^* is the area under the curve $\min\{p(\boldsymbol{x}, t=1), p(\boldsymbol{x}, t=2)\}$, the best we can do. In particular, this minimum achievable error is not zero.

To cement our intuition, denote the space of label values by $\mathcal{T}$: $\mathcal{T} = \{0, 1\}$ for our cancer screening example, or $\mathcal{T} = \{0, \dots, K-1\}$ for K-way classification. A classifier $f(\boldsymbol{x})$ is characterized by its decision regions

$$\mathcal{H}_t = \left\{\boldsymbol{x} \;\middle|\; f(\boldsymbol{x}) = t\right\}.$$

For example, recall from Section 2.2 that the decision regions for a binary linear classifier are half-spaces in feature space. First,

$$P\left\{f(\boldsymbol{x}) \neq t\right\} = \mathrm{E}\left[\mathrm{I}_{\{f(\boldsymbol{x}) \neq t\}}\right],$$

recalling that $\mathrm{I}_{\{\mathcal{A}\}} = 1$ if $\mathcal{A}$ is true, $\mathrm{I}_{\{\mathcal{A}\}} = 0$ otherwise. The expectation is just a weighted sum/integral over the joint probability. The product rule provides the factorizations $p(\boldsymbol{x}, t) = p(\boldsymbol{x}|t)P(t) = P(t|\boldsymbol{x})p(\boldsymbol{x})$, so we can express $R(f)$ in two different ways. First,

$$\begin{aligned} R(f) &= \sum_{k \in \mathcal{T}} P(t=k) \underbrace{\int \mathrm{I}_{\{f(\boldsymbol{x}) \neq k\}} p(\boldsymbol{x}|t=k)\, d\boldsymbol{x}}_{=R(f|t=k)} \\ &= 1 - \sum_{k \in \mathcal{T}} P(t=k) \int_{\mathcal{H}_k} p(\boldsymbol{x}|t=k)\, d\boldsymbol{x}. \end{aligned}$$

The total error is the expectation of the class-conditional errors $R(f|t=k)$ over the class prior $P(t)$. This is useful in order to understand the composition of the error. Test your understanding by deriving the second equation (note that $1 - R(f)$ is the probability of getting it right). Second,

$$\begin{aligned} R(f) &= \int p(\boldsymbol{x}) \left(\sum_{k \in \mathcal{T}} \mathrm{I}_{\{f(\boldsymbol{x}) \neq k\}} P(t=k|\boldsymbol{x})\right) d\boldsymbol{x} \\ &= \int p(\boldsymbol{x}) \left(1 - P(t = f(\boldsymbol{x})|\boldsymbol{x})\right) d\boldsymbol{x}. \end{aligned}$$

In order to minimize $R(f)$, we should minimize $1 - P(t = f(\boldsymbol{x})|\boldsymbol{x})$ for every $\boldsymbol{x}$. The best possible classifier $f^*(\boldsymbol{x})$, called *Bayes-optimal classifier*, is given by

$$f^*(\boldsymbol{x}) = \operatorname*{argmax}_{t \in \mathcal{T}} P(t|\boldsymbol{x}) = \operatorname*{argmax}_{t \in \mathcal{T}} p(\boldsymbol{x}|t)P(t).$$

The second equation is due to the fact that in the definition (5.1) of the posterior $P(t|\boldsymbol{x})$, the denominator $p(\boldsymbol{x})$ does not depend on t. In Figure 5.4, we plot $p(\boldsymbol{x}|t=k)P(t=k)$ for two classes. The Bayes-optimal classifier picks $t = k$ whenever the curve for k lies above the curve for all other classes. Its decision regions are

$$\mathcal{H}^*_k = \left\{\boldsymbol{x} \;\middle|\; p(\boldsymbol{x}|t=k)P(t=k) > p(\boldsymbol{x}|t=k')P(t=k'),\ \forall k' \in \mathcal{T} \setminus \{k\}\right\}.$$

The probability of error for the Bayes-optimal classifier is called *Bayes error*:

$$R^* = R(f^*) = \int p(\boldsymbol{x}) \left(1 - \max_{k \in \mathcal{T}} P(t=k|\boldsymbol{x})\right) d\boldsymbol{x} = 1 - \mathrm{E}\left[\max_{k \in \mathcal{T}} P(t=k|\boldsymbol{x})\right].$$

In the binary case, $1 - \max_{t \in \mathcal{T}} P(t|\boldsymbol{x}) = \min_{t \in \mathcal{T}} P(t|\boldsymbol{x})$, so that

$$R^* = \mathrm{E}\left[\min_{k=0,1} P(t=k|\boldsymbol{x})\right] = \int \min_{k=0,1} P(t=k|\boldsymbol{x}) p(\boldsymbol{x})\, d\boldsymbol{x}.$$

In Figure 5.4, the Bayes error R^* is the area under the curve $\min\{p(\boldsymbol{x}, t = 1), p(\boldsymbol{x}, t = 2)\}$.

5.2.2 Discriminant Functions

We saw that the Bayes-optimal classifier compares joint probabilities $p(\boldsymbol{x}|t)P(t)$ and decides for the largest. It is most convenient to describe this procedure by way of *discriminant functions*. Consider binary classification, $t \in \mathcal{T} = \{0, 1\}$. The optimal rule decides for $t = 1$ if

$$\frac{p(\boldsymbol{x}|t=1)P(t=1)}{p(\boldsymbol{x}|t=0)P(t=0)} = \frac{p(\boldsymbol{x}|t=1)}{p(\boldsymbol{x}|t=0)} \cdot \frac{P(t=1)}{P(t=0)} > 1.$$

A product, thresholded at 1? Recall from Chapter 2 that we prefer sums which are thresholded at 0. Let us take the *logarithm*[5], a strictly increasing function:

$$y^*(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=1)}{p(\boldsymbol{x}|t=0)} + \log \frac{P(t=1)}{P(t=0)} > 0.$$

$y^*(\boldsymbol{x})$ is a Bayes-optimal discriminant function, in that thresholding it at zero provides a Bayes-optimal classifier: $f^*(\boldsymbol{x}) = \mathrm{I}_{\{y^*(\boldsymbol{x})>0\}}$. To relate this to Chapter 2, note that if our class labels were $-1, +1$ instead of $0, 1$, the relationship would be $f^*(\boldsymbol{x}) = \mathrm{sgn}(y^*(\boldsymbol{x}))$.

How about K-way classification, $K > 2$? In this case, the optimal classifier picks the maximum among K functions

$$y_k^*(\boldsymbol{x}) = \log p(\boldsymbol{x}|t=k) + \log P(t=k), \quad k = 0, \dots, K-1,$$

in that $f^*(\boldsymbol{x}) = \mathrm{argmax}_{t \in \mathcal{T}} y_t^*(\boldsymbol{x})$. As in the binary case, we could get by with $K - 1$ functions, say $y_k^*(\boldsymbol{x}) - y_0^*(\boldsymbol{x})$, $k = 1, \dots, K - 1$. However, this singles out one class ($t = 0$) arbitrarily and creates more problems than it solves, so usually K discriminant functions are employed.

5.2.3 Example: Class-conditional Cauchy Distributions

Let us work out a binary classification example. We have a uniform class prior $P(t = 0) = P(t = 1) = 1/2$ and class-conditional Cauchy densities

$$p(x|t) = \frac{1}{\pi b} \cdot \frac{1}{1 + \left(\frac{x - a_t}{b}\right)^2}, \quad b > 0.$$

The setup is illustrated in Figure 5.5. Recall from Section 5.1.3 that the Cauchy distribution is peculiar in that mean and variance do not exist. If you sample

[5] It does not matter to which basis the logarithm is taken, as long as we keep consistent. In this course, we will use the natural logarithm to Euler's basis e.

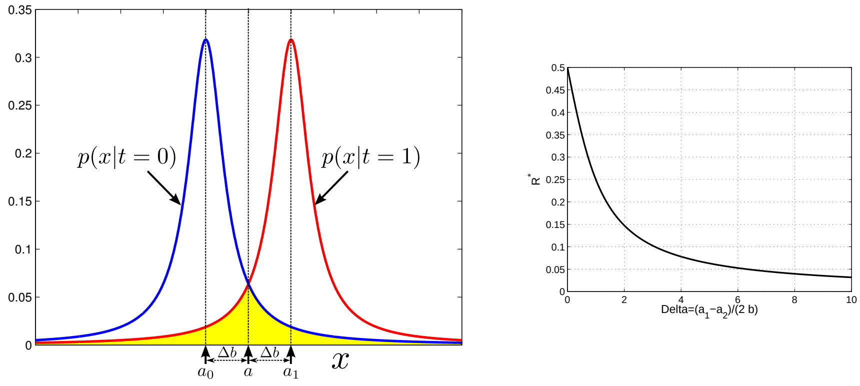


Figure 5.5: Bayes-optimal classifier and Bayes error for two class-conditional Cauchy distributions, centered at a_0 and a_1. The optimal rule thresholds at the midpoint $a = (a_0 + a_1)/2$. Since the class prior is $P(t = 0) = P(t = 1) = 1/2$, the Bayes error R^* is twice the yellow area. Right plot show R^* as function of separation parameter Δ. The slow decay of R^* is due to the very heavy tails of the Cauchy distributions.

values from it, you may encounter very large values occasionally. Assume $a_1 > a_0$ and define the midpoint $a = (a_0 + a_1)/2$. The Bayes-optimal rule is obvious by symmetry: $f^*(x) = \mathrm{I}_{\{x>a\}}$. Moreover,

$$R^* = \frac{1}{2} \sum_{k=0,1} \int_{\mathcal{H}^*_{1-k}} p(x|t=k)\, dx.$$

By symmetry, the two summands are the same, so

$$R^* = \int_{-\infty}^{a} \frac{1}{\pi b} \cdot \frac{1}{1 + \left(\frac{x-a_1}{b}\right)^2}\, dx.$$

Substitute $y = (x - a_1)/b$, and denote $\Delta = (a_1 - a_0)/(2b)$:

$$R^* = \frac{1}{\pi} \int_{-\infty}^{-\Delta} \frac{dy}{1+y^2} = \frac{1}{\pi} \left(\arctan(-\Delta) + \frac{\pi}{2} \right) = \frac{1}{2} - \frac{\arctan(\Delta)}{\pi}.$$

The Bayes error R^* is a function of the separation Δ of the classes (Figure 5.5, right). For $\Delta = 0$, the class-conditional densities are the same, so that $R^* = 1/2$. Also, $R^* \to 0$ as $\Delta \to \infty$. However, the decay is very slow, which is a direct consequence of the heavy tails of the $p(x|t)$. Any $x > a$ is classified as $f^*(x) = 1$, but even a $x \gg a$ could still come from $p(x|t = 0)$ whose probability mass far to the right is considerable.

5.2.4 Loss Functions. Minimizing Risk

Our cancer screening procedure is to be implemented at the local hospital. Knowing about decision theory, we determine the Bayes-optimal classifier $f^*(\boldsymbol{x})$.

Based on a tissue sample $\boldsymbol{x}$, if $f^*(\boldsymbol{x}) = 1$ we alert a human medical doctor. If $f^*(\boldsymbol{x}) = 0$, we send the patient home. Optimal.

Wait a moment! If $t = 0$ (no cancer), but $f^*(\boldsymbol{x}) = 1$, we waste the time of a doctor and some money for additional tests. If $t = 1$ and $f^*(\boldsymbol{x}) = 0$, a human being will die for not being treated on time. Even the most cynical health care reformer will agree that the losses are hugely imbalanced, yet our Bayes-optimal rule treats them exactly the same. Fortunately, this problem is simple to address. Along with other specifications (such as the attributes of $\boldsymbol{x}$), we have to assess a *loss function* $L(y,t)$ from $\mathcal{T} \times \mathcal{T} \to \mathbb{R}$. If our classifier predicts $f(\boldsymbol{x})$, we incur a loss of $L(f(\boldsymbol{x}),t)$, t the true label. In our cancer screening example, we could choose

$$L(0,0) = L(1,1) = 0, \quad L(1,0) = 1, \quad L(0,1) = \lambda \gg 1.$$

There is no loss for getting it right. Calling an ultimately unnecessary checkup costs 1, while a misdiagnosis costs λ. We can now calibrate λ to our needs. The goal is to minimize expected loss, called *risk*:

$$R(f) = \mathrm{E}\left[L(f(\boldsymbol{x}),t)\right].$$

The *Bayes-optimal classifier* $f^*(\boldsymbol{x})$ under loss $L(y,t)$ minimizes the risk, and $R^* = R(f^*)$ is called *Bayes risk*. Since

$$R(f) = \int p(\boldsymbol{x}) \left(\sum_{k \in \mathcal{T}} L(f(\boldsymbol{x}),k) P(t=k|\boldsymbol{x}) \right) d\boldsymbol{x},$$

the Bayes-optimal classifier is given by

$$f^*(\boldsymbol{x}) = \operatorname*{argmin}_{j \in \mathcal{T}} \sum_{k \in \mathcal{T}} L(j,k) P(t=k|\boldsymbol{x}),$$

its Bayes risk is

$$R^* = \mathrm{E}\left[\min_{j \in \mathcal{T}} \sum_{k \in \mathcal{T}} L(j,k) P(t=k|\boldsymbol{x}) \right].$$

You have probably noticed by now that minimizing the classification error is just a special case under the *zero-one loss* $L(y,t) = \mathrm{I}_{\{t \neq y\}}$: no loss for getting it right, loss 1 for an error.

How does the optimal decision rule depend on the loss function values? Let us work out the Bayes-optimal discriminant function $y^*(\boldsymbol{x})$ for our cancer screening example. It is positive (classifies 1) if

$$\begin{aligned}
&L(1,0)P(t=0|\boldsymbol{x}) + L(1,1)P(t=1|\boldsymbol{x}) < L(0,0)P(t=0|\boldsymbol{x}) + L(0,1)P(t=1|\boldsymbol{x}) \\
&\Leftrightarrow \quad (L(0,1) - L(1,1))P(t=1|\boldsymbol{x}) > (L(1,0) - L(0,0))P(t=0|\boldsymbol{x}) \quad \Leftrightarrow \\
&\log \frac{p(\boldsymbol{x}|t=1)}{p(\boldsymbol{x}|t=0)} + \log \frac{P(t=1)}{P(t=0)} > \log \frac{L(1,0) - L(0,0)}{L(0,1) - L(1,1)} = -\log \lambda \quad \Leftrightarrow \\
&y^*(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=1)}{p(\boldsymbol{x}|t=0)} + \log \frac{P(t=1)}{P(t=0)} + \log \lambda > 0.
\end{aligned}$$

The loss function values only shift the threshold of the optimal discriminant. The larger λ, the more $y^*(\boldsymbol{x}) = 1$ outputs will happen (human checkup). You might think an error is inacceptable and set $\lambda = \infty$. However, this leads to $y^*(\boldsymbol{x}) = 1$ for every patient, and the screening becomes uninformative.

Armed with loss functions, we can extend decision theory to scenarios where the prediction space (output of $f(\boldsymbol{x})$) is different from the label space $\mathcal{T}$. A common example is to allow the classifier to output "don't know". None of this adds complexity, it is left to the reader to be explored.

Finally, we need to mention some vocabulary coming from the area of statistical tests, as it is widely used in machine learning. Take our cancer screening example. Our method outputs $f(\boldsymbol{x})$, the true label is t. If $f(\boldsymbol{x}) = 1$, our vote is *positive*, if $f(\boldsymbol{x}) = 0$, it is *negative*. Then, if $f(\boldsymbol{x}) = t$, it is *true*, otherwise it is *false*. Any combination $(f(\boldsymbol{x}), t)$ is a true/false positive/negative. The two types of errors are false positive and false negative. A false positive is a checkup for a healthy patient, a false negative is sending a cancerous patient home. In most situations, false negatives are more serious than false positives.

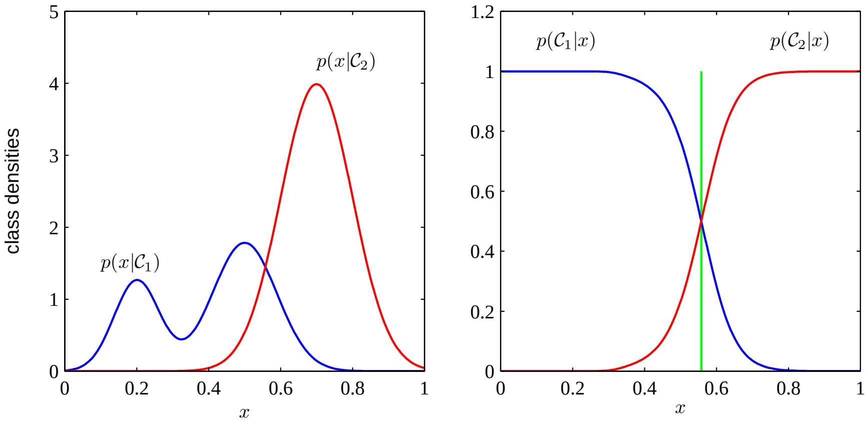


Figure 5.6: Left: Class-conditional densities for binary classification problem, where $P(t = 1) = P(t = 2)$. Right: Corresponding posterior probabilities. The green line constitutes the Bayes-optimal threshold. The complex bimodal shape of $p(x|\mathcal{C}_1) = p(x|t = 1)$ has no effect on the posterior probabilities and the optimal classifier.
Figure from [6] (used with permission).

5.2.5 Inference and Decisions

Our insights from decision theory can be summed up in the statement that optimal decision making is driven by *probabilistic inference*, the computation of posterior probabilities. The standard approach to a problem is to:

- Model the problem by assessing a complete joint distribution, such as $p(\boldsymbol{x}, t) = p(\boldsymbol{x}|t)P(t)$, as well as a loss function $L(y, t)$.
- Compute the posterior distribution $P(t|\boldsymbol{x})$.

- Make decisions so as to minimize risk (expected loss), using optimal discriminant functions based on $P(t|\boldsymbol{x})$.

However, we are in a comfortable position in this chapter: we know all of $p(\boldsymbol{x}, t)$, and $P(t|\boldsymbol{x})$ is easily computed. This is not so for most real-world situations. The true, or even a very good model is unknown. We have to learn from data, and computing posterior distributions can be very difficult. In the real world, we have to find shortcuts to the standard approach. For example, if only the posterior $P(t|\boldsymbol{x})$ enters decision making, why don't we model it directly, rather than bothering with all of $p(\boldsymbol{x}, t)$? It may well be that the precise shape of the class-conditionals $p(\boldsymbol{x}|t)$ is irrelevant for optimal decision making (Figure 5.6), in which case learning it from data is wasteful. This rationale underlies the *discriminative* (as opposed to *generative*) approach to probabilistic modelling, which we will learn about in Chapter 8. Even more economical, we could learn discriminant functions directly, without taking the detour over the posterior $P(t|\boldsymbol{x})$. On the other hand, the posterior $P(t|\boldsymbol{x})$ provides much more useful information about a problem than any single discriminant function. For one, we can evaluate our risk and act accordingly. Moreover, the combination of multiple probabilistic predictors is easy to do by rules of probability.

Chapter 6

Probabilistic Models. Maximum Likelihood

In this chapter, we introduce probabilistic modelling, the leading approach to make decision theory work in practice. We also establish the principle of maximum likelihood, the most widely used framework for deriving statistical estimators in order to learn from data. We will learn about Gaussian (or normal) distributions, the most important family of probability distributions over continuous variables. We will also introduce naive Bayes classifiers based on discrete distributions, which are widely used in the context of information retrieval and machine learning on natural language documents.

6.1 Generative Probabilistic Models

One problem with decision theory (Chapter 5) is that it assumes full knowledge of "true" distributions, which we know little about. All we have in practice is some vague ideas and data to learn the distributions from.

Consider the dataset of final grades of the PCML course in 2010, shown in Figure 6.1, left. How can we understand the gist of this data? Let us compute a *histogram*. We divide[1] the relevant range into bins of width Δx. For each bin j, we count the number of data points falling in there (say, n_j) and draw a bar of heigth $n_j/(n\Delta x)$. The normalization ensures that the total area of bars adds up to one. This way, we get an idea about the distribution of scores. However, histograms come with a few problems:

- The choice of Δx is crucial. Too large, and we miss important details. Too fine, and most bins will be empty. Histograms are smoothed out in *kernel density estimators* [6, ch. 2.5.1].
- Histograms do not work in more than five dimensions or so. The number of cells grows exponentially with the number of dimensions, and most cells

[1] For the final grades data, responses are naturally quantized at bin width 0.5, so there is little to choose in this case.

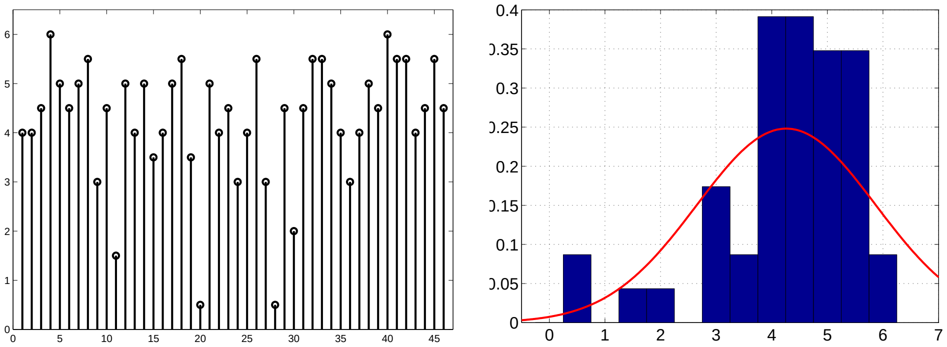


Figure 6.1: Left: Final grades from *Pattern Classification and Machine Learning* course 2010. Right: Histogram of data. Overlaid is the maximum likelihood fit for a Gaussian distribution. For this data, the responses t_i are quantized to $\{0, 0.5, 1, \dots, 5.5, 6\}$, so the bin width is $\Delta x = 0.5$.

will simply be empty. This problem of histograms and related techniques is called *curse of dimensionality*. Our exam scores are in one dimension, but how do we get a good idea about data in 25 dimensions? Or in 784 dimensions, where our MNIST digits live?

- Histograms are often not versatile enough. There are no knobs we can adjust in order to start *analyzing* the data, instead of just staring at it from one angle.

These problems sound familiar. In Chapter 4, we decided to fit simple lines or polynomials to noisy data, so to find the basics behind the chaff. Maybe we should *fit simple distributions to our data*. In Figure 6.1, right, a Gaussian is fitted to the exam results. One glance reveals the mean score as well as its approximate spread. On the other hand, it seems that the fit could be improved by employing a bimodal[2] density (two bumps). The assumptions on the density are the knobs we can play with in order to understand our data.

Fitting a probability density to data is called *density estimation*. In fact, histograms and kernel smoothers are density estimation methods. However, in this chapter, we focus on *parametric* techniques, where, just as in Chapter 4, the densities to choose from are parameterized by some $\boldsymbol{w} \in \mathbb{R}^p$.

Assumptions about Data

At this point, we need to specify our assumptions about the data. Consider a dataset $\mathcal{D} = \{x_i \mid i = 1, \dots, n\}$. The most commonly made assumption is that the data points x_i are *independently and identically distributed* (short: i.i.d.). There exists a "true" distribution, from which the x_i are drawn independently (recall independence from Section 5.1.1).

[2] We will not be concerned with multimodal densities in the present chapter, but the techniques we develop here will be the major ingredient for Gaussian mixture models in Section 12.2.

In order to build a foundation for density estimation, we go further and formulate *model assumptions*. In fact, we propose a *generative probabilistic model* for our data. In order to fit a Gaussian to our exam results $\mathcal{D} = \{x_i \mid i = 1, \dots, n\}$, we postulate that each x_i is drawn independently from a Gaussian distribution with unknown mean μ and unknown variance σ^2 (Gaussians are introduced shortly, their details do not matter at this point). This model assumption directly and uniquely leads to an optimization problem for the fitting. Let us pause a second to understand that. We *do not* come up with an algorithm, an optimization problem, or an error function to minimize. All we do is to suggest a way in which the data could have been generated, leaving our ignorance about any details in the unknown parameters. As we will see shortly, *everything else is automatic*. We just have to do the *forward* modelling, from unknown parameters to observed data. The *inverse* problem, from data back to parameters, is implied by general statistical principles.

It is important to distinguish the *i.i.d. assumption* from *model assumptions*. Model assumptions are choices we make about our model or method. They are on par with options such as “I will use a linear classifier” or “let me use an MLP with 4 layers.” These choices can be good or bad, useful or not useful, but it does not make sense to call them “right” or “wrong”. George Box, one of the pioneers of Bayesian inference, quality control and design of experiments: “All models are wrong, but some are useful.” The i.i.d. assumption[3] is a different story. It is like an axiom for much of machine learning. If it fails for our data, we may as well read tea leaves than trying to learn something from it.

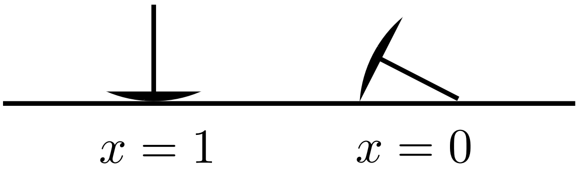


Figure 6.2: Repeatedly dropping a thumbtack on an even surface can be used to draw a sample of binary data.

6.2 Maximum Likelihood Estimation

Browsing in your drawer, you find an odd-shaped thumbtack (Figure 6.2). When you drop it on the floor, its point x is either facing up ($x = 1$) or down ($x = 0$). You get curious. What is the probability for it landing point up, $p_1 = P\{x = 1\}$?

[3] Or weaker assumptions such as ergodicity.

You throw it $n = 100$ times, thereby collecting data $\mathcal{D} = \{x_1, \dots, x_{100}\}$. We can assume that this data is i.i.d. Now, if $P\{x_1 = 1\} = p_1$, the probability of generating $\mathcal{D}$ is

$$P(\mathcal{D}|p_1) = \prod_{i=1}^{n} p_1^{x_i}(1-p_1)^{1-x_i} = p_1^{n_1}(1-p_1)^{n-n_1}, \quad n_1 = \sum_{i=1}^{n} x_i.$$

n_1 is the number of times the thumbjack lands with point facing up. The distribution of x_i is called *Bernoulli distribution* with parameter p_1. We can regard $P(\mathcal{D}|p_1)$ as the probability of the data $\mathcal{D}$ *under the model assumption*. This has nothing to do with the "true" probability of data, whatever that may be. After all, for every value of $p_1 \in [0, 1]$, we get a *different* probability $P(\mathcal{D}|p_1)$. This model probability, as a function of the parameter p_1, is called *likelihood function*. If our goal is to fit the model $P\{x_1 = 1\} = p_1$ with parameter p_1 to the i.i.d. data $\mathcal{D}$, it makes sense to *maximize the likelihood*. The *maximum likelihood estimator* (MLE) for p_1 is

$$\hat{p}_1 = \operatorname*{argmax}_{p_1 \in [0,1]} P(\mathcal{D}|p_1).$$

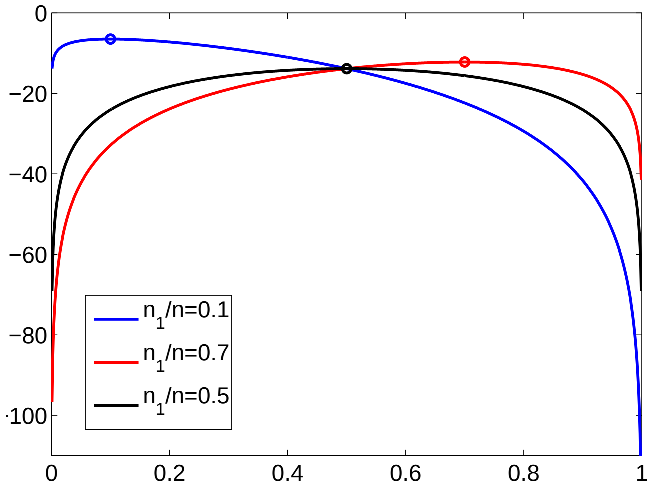


Figure 6.3: Log likelihood functions for Bernoulli distributions (thumbtack example). The sample size is $n = 20$ here. Note that the log likelihood function $\log P(\mathcal{D}|p_1)$ depends on the data $\mathcal{D}$ only through $\hat{p}_1 = n_1/n$ and n. Its mode is at $p_1 = \hat{p}_1$, the maximum likelihood estimate.

Let us solve for $\hat{p}_1$. Assume for now that $n_1 \in \{1, \dots, n-1\}$. Then, $P(\mathcal{D}|p_1) > 0$ for $p_1 \in (0, 1)$, $P(\mathcal{D}|p_1) = 0$ for $p_1 \in \{0, 1\}$, so we can assume that $p_1 \in (0, 1)$. It is always simpler to *maximize the log-likelihood* $\log P(\mathcal{D}|p_1)$ instead. The derivative is

$$\frac{d \log P(\mathcal{D}|p_1)}{dp_1} = \frac{n_1}{p_1} - \frac{n-n_1}{1-p_1} = 0 \quad \Leftrightarrow \quad n_1(1-p_1) = p_1(n-n_1) \quad \Leftrightarrow \quad p_1 = \frac{n_1}{n}.$$

$\hat{p}_1 = n_1/n$ is indeed a maximum point, the unique maximizer, and this holds for $n_1 \in \{0, n\}$ as well. The *maximum likelihood estimator* (MLE) for $p_1 = P\{x_1 =$

1} is

$$\hat{p}_1 = \frac{n_1}{n},$$

the fraction of the thumbjack pointing up over all throws. This is what we would have estimated anyway. Indeed, maximum likelihood estimation often coincides with common sense (ratios of counts) in simple situations. Bernoulli log likelihood functions for the thumbtack setup are shown in Figure 6.3.

Is this "correct"? Remember that this question is ill-defined. Is it any good then? Maximum likelihood estimation works well in many situations in practice. It comes with a well-understood asymptotic theory. Given that, we will see that it can have shortcomings as a statistical estimation technique, in particular if applied with small datasets, and we will study modifications of maximum likelihood in order to overcome these problems. For now, it is simply a surprisingly straightforward and general principle to implement density estimation.

6.3 The Gaussian Distribution

Our exam marks data is real-valued, $x_i \in \mathbb{R}$. The most important distribution over real numbers, arguably the most important distribution at all, is the *Gaussian distribution* (or *normal distribution*). It is behind the squared error function. Maximum likelihood estimation for Gaussians is maybe the most frequently used data analysis technique there is. There is virtually no machine learning work dealing with continuous variable data, for which Gaussians do not play a role. In the context of this chapter, Gaussians are important for (at least) two reasons:

- Maximum likelihood estimators for Gaussian densities coincide with sample mean and sample covariance matrix.

- For Gaussian class-conditional densities $p(\boldsymbol{x}|t)$, $t = 0, 1$, linear classifiers turn out to be Bayes-optimal classifiers (for equal covariance matrices).

Here is the probability density of a Gaussian:

$$N(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2}, \qquad \sigma^2 > 0. \tag{6.1}$$

We also sometimes write $N(\mu, \sigma^2)$ if the argument x is clear from context. We will work out in Section 6.3.2 that if $x \sim N(\mu, \sigma^2)$ (read: "x is distributed according to $N(\mu, \sigma^2)$"), then $\mu = \mathrm{E}[x]$, $\sigma^2 = \mathrm{Var}[x]$. The parameters are mean and variance of the Gaussian. Please note one thing: it is μ and σ *squared*, mean and variance. $N(0, 2)$ means variance $\sigma^2 = 2$, *not* standard deviation $\sigma = 2$. We work out several properties of the Gaussian below in this chapter, and more as we move through the course. Just one observation here:

$$-\log N(x|\mu, \sigma^2) = \frac{1}{2\sigma^2}(x-\mu)^2 + \frac{1}{2}\log(2\pi\sigma^2).$$

The negative log of a Gaussian is a *quadratic* function.

The real importance of the Gaussian becomes apparent only for multivariate distributions. In fact, Gaussians are maybe the only distributions we can tractably work with in high dimensions. The density of a Gaussian distribution of $\boldsymbol{x} \in \mathbb{R}^d$ is

$$N(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = |2\pi\boldsymbol{\Sigma}|^{-1/2} \, e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}. \qquad (6.2)$$

This looks daunting, but is nothing to worry about. $|\boldsymbol{\Sigma}|$ is the *determinant* of $\boldsymbol{\Sigma}$. Refresh your memory about determinants in Section 6.3.1. In Section 6.3.3, we work out the form of this density from the univariate Gaussian, using transformation rules which come in handy elsewhere as well. We also show there that $\boldsymbol{\mu} = \mathrm{E}[\boldsymbol{x}]$, $\boldsymbol{\Sigma} = \mathrm{Cov}[\boldsymbol{x}]$. The parameters of the multivariate Gaussian are *mean* and *covariance matrix.*

Covariance Matrices are Positive Definite

Not every matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times d}$ is allowed here. First, $\boldsymbol{\Sigma}$ must be invertible. Also, $-\log N(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is a quadratic function. Recall our discussion of quadratic functions in Section 4.2.2: $\boldsymbol{\Sigma}^{-1}$, therefore $\boldsymbol{\Sigma}$ should be symmetric. It also must be a valid covariance matrix. What does that mean? If $\boldsymbol{x} \in \mathbb{R}^d$ is a random variable with covariance $\mathrm{Cov}[\boldsymbol{x}]$ (not necessarily Gaussian), then for any $\boldsymbol{v} \in \mathbb{R}^d$:

$$0 \le \mathrm{Var}[\boldsymbol{v}^T\boldsymbol{x}] = \boldsymbol{v}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{v}.$$

A symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ is called *positive semidefinite* if

$$\boldsymbol{v}^T\boldsymbol{A}\boldsymbol{v} \ge 0 \quad \forall \boldsymbol{v} \in \mathbb{R}^d.$$

Valid covariance matrices are precisely symmetric positive semidefinite matrices. We used a stronger condition in Section 4.2.2. A symmetric matrix $\boldsymbol{A}$ is *positive definite* if

$$\boldsymbol{v}^T\boldsymbol{A}\boldsymbol{v} > 0 \quad \forall \boldsymbol{v} \in \mathbb{R}^d \setminus \{\boldsymbol{0}\}.$$

So our $\boldsymbol{\Sigma}$ is invertible and positive semidefinite. These two conditions together are equivalent to positive definite. Why? If $\boldsymbol{v}^T\boldsymbol{\Sigma}\boldsymbol{v} = 0$ for $\boldsymbol{v} \ne \boldsymbol{0}$, we must have $\boldsymbol{\Sigma}\boldsymbol{v} = \boldsymbol{0}$. Namely, for any $\boldsymbol{z} \in \mathbb{R}^d$, $\lambda \in \mathbb{R}$:

$$0 \le (\boldsymbol{z} + \lambda\boldsymbol{v})^T\boldsymbol{\Sigma}(\boldsymbol{z} + \lambda\boldsymbol{v}) = \boldsymbol{z}^T\boldsymbol{\Sigma}\boldsymbol{z} + 2\lambda\boldsymbol{z}^T\boldsymbol{\Sigma}\boldsymbol{v}.$$

This means that $\boldsymbol{z}^T\boldsymbol{\Sigma}\boldsymbol{v} = 0$, otherwise sending λ to ∞ or $-\infty$ gives a contradiction. But then, $\boldsymbol{\Sigma}$ cannot be invertible. To conclude, $\boldsymbol{\Sigma}$ in (6.2) must be a symmetric positive definite matrix.

Contour Plots of Gaussian Distributions

A Gaussian distribution[4] over $\boldsymbol{x} \in \mathbb{R}^2$ (Figure 6.4, left) can be visualized by a contour plot (Figure 6.4, right), which works like a topographic map (a contour is a curve of equal density value, or "height over sea level"). How to read such a plot? First, the point of highest density (the *mode* of the density) is the mean $\mathrm{E}[\boldsymbol{x}]$. For a Gaussian, mean and mode coincide. Next, all contours of a Gaussian

[4] In dimensions $d > 2$, we can still visualize aspects of a Gaussian by projecting it into 2D subspaces (Chapter 11).

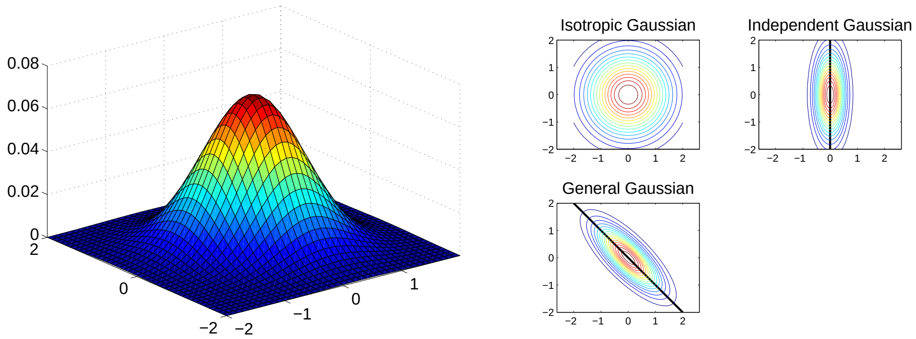


Figure 6.4: Left: Bivariate Gaussian density function. Right: Contours of Gaussian density functions. Contours are spherical for isotropic Gaussians (no preferred direction), they are aligned with the standard coordinate axes for independent Gaussians (diagonal covariance matrix). Each contour line is an ellipse, whose major axes are given by the eigenvectors of the covariance matrix.

density are ellipses. This is easy to understand: $-\log N(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is a quadratic function with positive definite $\boldsymbol{\Sigma}$ (therefore positive definite $\boldsymbol{\Sigma}^{-1}$, why?), and ellipses are solutions of positive definite quadratic equations. For the same reason, different contour lines are related by a uniform scaling transformation centered at $\boldsymbol{\mu}$. In general, the ellipses are maximally elongated along a single direction (axis). We will find in Section 11.1.1 that this principal axis $\boldsymbol{d}$, maximizing the variance $\mathrm{Var}[\boldsymbol{d}^T\boldsymbol{x}]$ over all $\|\boldsymbol{d}\| = 1$, corresponds to the maximum eigendirection of the covariance $\boldsymbol{\Sigma}$.

For now, let us distinguish between a few characteristic contour plot shapes, shown in the right panel of Figure 6.4. The upper two contour plots are mirror-symmetric with respect to the standard coordinate system anchored at $\boldsymbol{\mu}$, while the lower is not. Mirror symmetry implies that $\boldsymbol{\Sigma}^{-1}$ (and therefore $\boldsymbol{\Sigma}$) is *diagonal*. The coordinates of a random vector $\boldsymbol{x} \in \mathbb{R}^d$ with diagonal covariance $\mathrm{Cov}[\boldsymbol{x}]$ are *uncorrelated* variables. In general, the *correlation* between two variables x_j, x_k is

$$\frac{\mathrm{Cov}[x_j, x_k]}{\sqrt{\mathrm{Var}[x_j]\,\mathrm{Var}[x_k]}}.$$

Knowing about the Cauchy-Schwarz inequality, you will have no problem showing that correlations range from -1 (fully anti-correlated) over 0 (uncorrelated) to 1 (fully correlated). A glance at (6.2) reveals that for a Gaussian distribution, *uncorrelated components are independent components.* For example, if $d = 2$ and x_1, x_2 are uncorrelated, then $\boldsymbol{\Sigma}$ is diagonal, so that $\boldsymbol{\Sigma}^{-1}$ is diagonal as well, and $p(x_1, x_2) = p(x_1)p(x_2)$. This implication is specific to Gaussians and does *not* hold for other distributions in general. Both contour plots on the top depict Gaussians with independent components. For the left of these, the contours are circles: the variance along *any* direction is the same: $\mathrm{Var}[\boldsymbol{d}^T\boldsymbol{x}] = \boldsymbol{d}^T\boldsymbol{\Sigma}\boldsymbol{d}$ is the same for all $\|\boldsymbol{d}\| = 1$, which is possible only if $\boldsymbol{\Sigma}$ is a multiple of $\boldsymbol{I}$. Such a covariance structure is called *isotropic* or *spherical.* Imagine you stand at the mean and look into any direction. For an isotropic Gaussian, what you see is always the same.

Marginal Distribution of Gaussian

Given that $\boldsymbol{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, $\boldsymbol{x} \in \mathbb{R}^d$, what is the marginal distribution over a subset of the $\boldsymbol{x}$ components? Say, $\boldsymbol{x}_J = [x_j]_{j \in J}$ for $J \subset \{1, \dots, d\}$. The answer is obvious from our geometrical intuition: $\boldsymbol{x}_J$ is Gaussian again, with $\boldsymbol{x}_J \sim N(\boldsymbol{\mu}_J, \boldsymbol{\Sigma}_J)$. A more general statement is as follows. If $\boldsymbol{A} \in \mathbb{R}^{p \times d}$ has full rank $\operatorname{rk} \boldsymbol{A} = p$, $p \leq d$ and $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, then $\boldsymbol{y}$ is Gaussian again, with $\boldsymbol{y} \sim N(\boldsymbol{A}\boldsymbol{\mu}, \boldsymbol{A}\boldsymbol{\Sigma}\boldsymbol{A}^T)$. First, mean and covariance of $\boldsymbol{y}$ follow from the transformation rules of Section 5.1.3. The Gaussianity of $\boldsymbol{y}$ is shown as in Section 6.3.3: its density must have the form of (6.2). In other words, the family of Gaussian distributions is *closed* under full-rank affine linear transformations. In particular, all *marginal* distributions of a joint Gaussian distribution are Gaussian again.

A final point which you might be puzzled about. We saw that covariance matrices are positive semidefinite in general, while $\boldsymbol{\Sigma}$ in (6.2) must be positive definite. Are there Gaussians whose covariance matrix is not invertible? Consider an extreme example. For a fixed vector $\boldsymbol{z} \in \mathbb{R}^d \setminus \{\boldsymbol{0}\}$, $d > 1$, and $\alpha \sim N(0, 1)$, does $\boldsymbol{x} = \alpha \boldsymbol{z}$ have a Gaussian distribution? Its covariance would be $\mathrm{Cov}[\alpha \boldsymbol{z}] = \boldsymbol{z}\boldsymbol{z}^T$, which is not invertible. The geometry behind such *degenerate Gaussians* is simple: they are perfectly normal Gaussians, but confined to an affine subspace of $\mathbb{R}^d$. Picture them as entirely flat in $\mathbb{R}^d$. There are always directions $\boldsymbol{v} \neq \boldsymbol{0}$ along which they do not fluctuate at all: $\mathrm{Var}[\boldsymbol{v}^T \boldsymbol{x}] = 0$, in fact $\boldsymbol{v}^T \boldsymbol{x}$ is constant. In contrast, a proper Gaussian with density (6.2) always fluctuates along *all* directions. Beware that some texts simply extend the family of Gaussian distributions to these cases. However, in this course a *Gaussian* distribution has a positive definite covariance matrix and a density of the form (6.2).

Gaussians in a Non-Gaussian World (*)

In this basic course, we will mainly be concerned with linear methods and models based on Gaussian distributions. However, it is increasingly appreciated in statistical physics, statistics, machine learning, and elsewhere that many real-world distributions of interest are not Gaussian at all. Relevant buzzwords are "heavy-tailed" (or "fat tails"), "power law decay", "scale-free", "small world", etc. Real processes exhibit large jumps now and then, which are essentially ruled out in Gaussian random noise. As Gaussian distributions are simple to work with, these facts have been widely ignored until quite recently. In this sense, parts of classical statistics, economics and machine learning are founded on principles which do not fit the data.

Does this mean we waste our time learning about Gaussians? Absolutely not! The most natural way to construct realistic heavy-tailed distributions with power law decay is by mixing together Gaussians of different scales. Non-Gaussian statistics is built on top of the Gaussian world. The bottomline is this. Classical statistics use Gaussians and linear models to represent data *directly*. Modern statistics employs non-Gaussian distributions and non-linear models, which are built from Gaussians and linear mappings *inside*, mainly via the powerful concept of *latent variables* (we will explore this idea in Chapter 12). Modern methods behave non-Gaussian, but they are driven by Gaussian mathematics and corresponding numerical linear algebra as major building blocks.

6.3.1 Techniques: Determinants

The multivariate Gaussian density (6.2) features a *determinant* $|\boldsymbol{\Sigma}|$ of the matrix $\boldsymbol{\Sigma}$. It is highly recommended that you study [45, ch. 5.1], even if you think you know everything about determinants. The following is just a brief exposition, mainly taken from there.

Every square matrix $\boldsymbol{A} \in \mathbb{R}^{p\times p}$ has a determinant $|\boldsymbol{A}|$. Other texts use the notation det$\boldsymbol{A}$ to avoid ambiguities, but the $|\boldsymbol{A}|$ is most common and will be used in this course. The number $|\boldsymbol{A}| \in \mathbb{R}$ contains a lot of information about the matrix $\boldsymbol{A}$. First, $|\boldsymbol{A}| \neq 0$ if and only if $\boldsymbol{A}$ is invertible. If this is the case, then $|\boldsymbol{A}^{-1}| = 1/|\boldsymbol{A}|$. You should memorize the 2×2 case:

$$\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \Rightarrow |\boldsymbol{A}| = ad - bc, \quad \boldsymbol{A}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

The last equation holds only if $\boldsymbol{A}$ is invertible, $|\boldsymbol{A}| \neq 0$.

Some useful facts about determinants (they all follow from three simple axioms, see [45, ch. 5.1]):

- Linear in each column/row: If

  $$F(\boldsymbol{a}_1, \dots, \boldsymbol{a}_p) = |[\boldsymbol{a}_1, \dots, \boldsymbol{a}_p]|,$$

  $\boldsymbol{a}_k$ the columns of the matrix $\boldsymbol{A}$, then F is a linear function in each of its arguments (F is called multilinear). The same holds for the rows of $\boldsymbol{A}$.

- Product: If $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{p\times p}$, then $|\boldsymbol{AB}| = |\boldsymbol{A}|\,|\boldsymbol{B}|$.
  In particular: $|\boldsymbol{A}^{-1}| = 1/|\boldsymbol{A}|$, since $|\boldsymbol{I}| = 1$ and

  $$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}.$$

- Transpose: The determinant is invariant under transposition:
  $|\boldsymbol{A}^T| = |\boldsymbol{A}|$

- Triangular matrices: If a matrix $\boldsymbol{A}$ is *upper triangular*, meaning that $a_{ij} = 0$ for all $i > j$:

  $$\boldsymbol{A} = \begin{bmatrix} a_{11} & \dots & \dots & a_{1p} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{pp} \end{bmatrix},$$

  its determinant is the product of the diagonal entries:

  $$|\boldsymbol{A}| = \prod_{i=1}^{p} a_{ii}.$$

  The same holds for lower triangular matrices (and of course for diagonal matrices).

The determinant in the Gaussian density (6.2) is of a positive definite matrix $\boldsymbol{\Sigma}$. In this case, the rules provide a method for computing $|\boldsymbol{\Sigma}|$. We use the Cholesky decomposition from Section 4.2.2: $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$. Then,

$$|\boldsymbol{\Sigma}| = |\boldsymbol{L}||\boldsymbol{L}^T| = |\boldsymbol{L}|^2 = \prod_{i=1}^{p} l_{ii}^2.$$

For large matrices $\boldsymbol{\Sigma}$, it is always numerically better to compute

$$\log|\boldsymbol{\Sigma}| = 2\log|\boldsymbol{L}| = 2\sum_{i=1}^{p} \log l_{ii}.$$

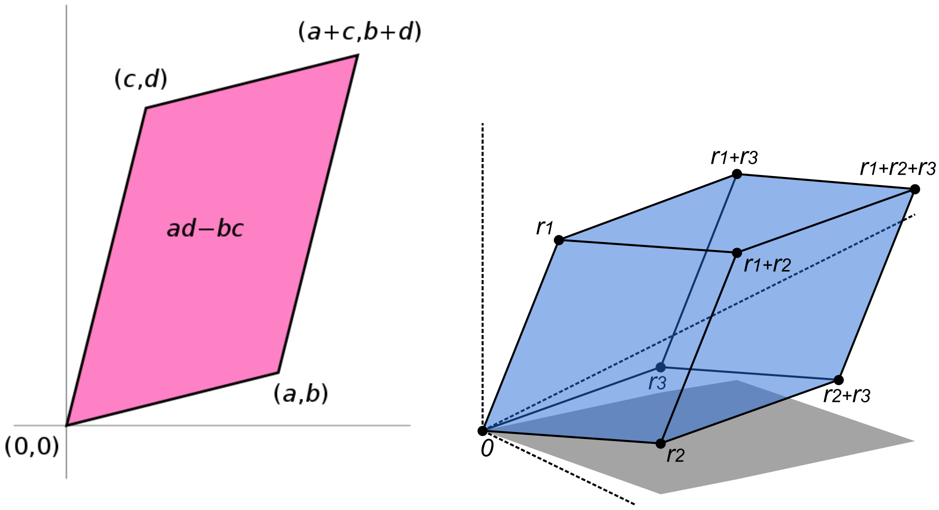


Figure 6.5: Illustration of determinant of $\boldsymbol{R} = [\boldsymbol{r}_j]$ in 2D and 3D.
Left: The absolute value of the determinant in 2D quantifies the area of the parallelogram spanned by $\boldsymbol{r}_1 = [a, b]^T$, $\boldsymbol{r}_2 = [c, d]^T$. Right: The absolute value of the determinant in 3D quantifies the volume of the parallelepiped spanned by $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, $\boldsymbol{r}_3$.
Figures from `wikipedia` (used with permission). Left: Copyright by Jitse Nielsen. Right: Copyright Claudio Rocchini, GNU free documentation license.

Finally, what is the geometrical meaning of $|\boldsymbol{A}| = F(\boldsymbol{a}_1, \dots, \boldsymbol{a}_p)$? Properties such as $|\boldsymbol{I}| = 1$ and $|\alpha\boldsymbol{A}| = \alpha^p|\boldsymbol{A}|$ hint towards *volume*, and that is correct. Consider $p = 2$. The two vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$ span a parallelogram (Figure 6.5, left), and the absolute value of $|\boldsymbol{A}| = F(\boldsymbol{a}_1, \boldsymbol{a}_2)$ is its *area*. We denote the absolute value of the determinant by $||\boldsymbol{A}||$, but typically point this out to avoid confusion. The area of a triangle? No problem, take half the determinant (why?). The same holds for $p = 3$. Now, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$ span a parallelepiped, and $||\boldsymbol{A}||$ is its *volume* (Figure 6.5, right). In the context of multivariate distributions such as the Gaussian (6.2),

$$|\boldsymbol{\Sigma}| = |\mathrm{Cov}[\boldsymbol{x}]|$$

measures the “volume of covariance”.

6.3.2 Techniques: Working with Densities (*)

Recall the Gaussian density from (6.1). Suppose we did not know what μ and σ^2 were. Let us practice some elementary manipulation of densities, not restricted to the Gaussian. If the random variable t has the density $p(t)$, then the variable $x = \mu + \sigma t$, $\sigma > 0$, has the density $\sigma^{-1} p((x-\mu)/\sigma)$. This is because expectations have to come out the same, whether we do them w.r.t. x or t. And $dx = \sigma dt$, so that

$$p(t)dt = p((x-\mu)/\sigma)\sigma^{-1}dx.$$

Confirm for yourself that the Gaussian (6.1) is obtained in this way from the standard normal density

$$N(t|0,1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2}.$$

Moreover, we know how mean and variance transform from t to x (Section 5.1.3):

$$\mathrm{E}[x] = \mu + \sigma \mathrm{E}[t], \quad \mathrm{Var}[x] = \sigma^2 \mathrm{Var}[t]. \tag{6.3}$$

So it all comes down to $N(0,1)$. Being an even function, its mean must be zero if it exists (it may not, remember the Cauchy). Let us go for the variance directly: if this exists, so does the mean (why?). We substitute $r = t^2/2$, so $dr = t\,dt$:

$$\begin{aligned}\mathrm{Var}[t] &= 2\int_0^\infty t^2 (2\pi)^{-1/2} e^{-\frac{1}{2}t^2}\,dt = (2/\pi)^{1/2}\int_0^\infty (2r)^{1/2} e^{-r}\,dr \\ &= 2\pi^{-1/2}\int_0^\infty r^{1/2} e^{-r}\,dr = 2\pi^{-1/2}\Gamma(3/2).\end{aligned}$$

The last integral involves Euler's Gamma function:

$$\Gamma(x) = \int_0^\infty r^{x-1} e^{-r}\,dr,$$

interpolating the factorial via $\Gamma(x+1) = x!$, $x \in \mathbb{N}$. In general, $\Gamma(x+1) = x\Gamma(x)$ and $\Gamma(1/2) = \sqrt{\pi}$. Therefore, $\Gamma(3/2) = \sqrt{\pi}/2$, and $\mathrm{Var}[t] = 1$. The standard normal distribution $N(0,1)$ has mean 0, variance 1, and (6.3) implies that μ is the mean, σ^2 the variance of the Gaussian (6.1).

6.3.3 Techniques: Density after Transformation (*)

Let us make sense of the multivariate Gaussian density (6.2) and show that $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ do correspond to mean and covariance. We have already firmly established the univariate Gaussian (6.1) in Section 6.3.2. We construct a random vector $\boldsymbol{t} \in \mathbb{R}^d$ with independent components, distributed as $t_j \sim N(0,1)$, so that $\mathrm{E}[\boldsymbol{t}] = \boldsymbol{0}$, $\mathrm{Cov}[\boldsymbol{t}] = \boldsymbol{I}$ (the identity). Its density is

$$N(\boldsymbol{t}|\boldsymbol{0},\boldsymbol{I}) = \prod_{j=1}^d \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t_j^2} = |2\pi\boldsymbol{I}|^{-1/2}\, e^{-\frac{1}{2}\boldsymbol{t}^T\boldsymbol{I}^{-1}\boldsymbol{t}}, \tag{6.4}$$

using that $\boldsymbol{t}^T\boldsymbol{I}^{-1}\boldsymbol{t} = \|\boldsymbol{t}\|^2 = \sum_j t_j^2$ and $|2\pi\boldsymbol{I}| = (2\pi)^d|\boldsymbol{I}| = (2\pi)^d$. Not bad. Given $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, we work backwards. Recall that $\boldsymbol{\Sigma}$ is positive definite. We use the Cholesky decomposition discussed in Section 4.2.2: $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L}$ is triangular and invertible.

If $\boldsymbol{x} = \boldsymbol{\mu}+\boldsymbol{L}\boldsymbol{t}$, how does the density $p(\boldsymbol{t})$ of $\boldsymbol{t}$ transform? The following step is not specific to Gaussian distributions. Again, we have to make sure that expectation is conserved when switching from $\boldsymbol{t}$ to $\boldsymbol{x}$. And by multivariate calculus:

$$p(\boldsymbol{t})d\boldsymbol{t} = p\left(\boldsymbol{L}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)||\boldsymbol{L}||^{-1}d\boldsymbol{x}.$$

Here, $||\boldsymbol{L}||$ denotes the absolute value of the determinant $|\boldsymbol{L}|$. This rule is easy to understand. The differentials $d\boldsymbol{x}$ and $d\boldsymbol{t}$ are infinitesimal volume elements. Picture $d\boldsymbol{t}$ as tiny hypercube. Since $\boldsymbol{t} \to \boldsymbol{x}$ involves multiplication with $\boldsymbol{L}$, the cube is transformed. What is its new volume? By our volumne intuition about the determinant, it must be $d\boldsymbol{x} = ||\boldsymbol{L}||d\boldsymbol{t}$. Plugging this into (6.4):

$$e^{-\frac{1}{2}\|\boldsymbol{t}\|^2} = e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{L}^{-T}\boldsymbol{L}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})} = e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}.$$

Moreover, $|\boldsymbol{L}^T| = |\boldsymbol{L}|$, so that $|\boldsymbol{\Sigma}| = |\boldsymbol{L}^T\boldsymbol{L}| = |\boldsymbol{L}|^2$, and the prefactor becomes $||\boldsymbol{L}||^{-1} = |\boldsymbol{\Sigma}|^{-1/2}$. All in all, the density of $\boldsymbol{x}$ is (6.2). Finally, we know how mean and covariance transform (Section 5.1.3):

$$\mathrm{E}[\boldsymbol{x}] = \boldsymbol{\mu} + \boldsymbol{L}\mathrm{E}[\boldsymbol{t}] = \boldsymbol{\mu}, \quad \mathrm{Cov}[\boldsymbol{x}] = \boldsymbol{L}\mathrm{Cov}[\boldsymbol{t}]\boldsymbol{L}^T = \boldsymbol{L}\boldsymbol{L}^T = \boldsymbol{\Sigma}.$$

We have derived (6.2) from the univariate case and shown that $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$ are mean and covariance respectively.

6.4 Maximum Likelihood for Gaussian Distributions

We are ready now to fit a Gaussian distribution $N(\mu, \sigma^2)$ to our exam results dataset. We minimize the negative log likelihood

$$L(\mu,\sigma^2) = -\log p(\mathcal{D}|\mu,\sigma^2) = \frac{1}{2}\sum_{i=1}^n \left\{(x_i-\mu)^2/\sigma^2 + \log(2\pi\sigma^2)\right\}.$$

First,

$$\frac{\partial L}{\partial \mu} = \sum_{i=1}^n (\mu - x_i)/\sigma^2 = \frac{n}{\sigma^2}(\mu - \bar{x}), \quad \bar{x} = \frac{1}{n}\sum_{i=1}^n x_i.$$

Therefore, $\hat{\mu} = \bar{x}$: the empirical mean. Note that this *is* a minimum point, since the second derivative is positive. Plugging in $\mu = \hat{\mu}$:

$$L(\sigma^2) = \frac{1}{2}\sum_{i=1}^n \left\{(x_i-\bar{x})^2/\sigma^2 + \log(2\pi\sigma^2)\right\} = \frac{n}{2}\left(S/\sigma^2 + \log(2\pi\sigma^2)\right),$$
$$S = \frac{1}{n}\sum_{i=1}^n (x_i - \bar{x})^2.$$

If $\tau = \sigma^{-2}$, then

$$L(\tau) = \frac{n}{2}\left(S\tau + \log(2\pi) - \log\tau\right), \quad \frac{\partial L}{\partial \tau} = \frac{n}{2}\left(S - \frac{1}{\tau}\right).$$

Therefore, $\hat{\tau} = 1/S$, or $\hat{\sigma}^2 = S$. This is a maximum point, since

$$\frac{\partial^2(2L/n)}{\partial\tau^2} = \frac{\partial}{\partial\tau}(S - 1/\tau) = \frac{1}{\tau^2} > 0.$$

The maximum likelihood estimator for mean and variance of a Gaussian (6.1) is

$$\hat{\mu} = \bar{x} = \frac{1}{n}\sum_{i=1}^n x_i, \quad \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^n (x_i - \bar{x})^2.$$

The Gaussian ML fit to our exam results data employs the empirical mean and variance. For multivariate data $\mathcal{D} = \{\boldsymbol{x}_i \mid i = 1, \dots, n\}$, $\boldsymbol{x}_i \in \mathbb{R}^d$, the maximum likelihood estimator is equally intuitive:

$$\hat{\boldsymbol{\mu}} = \bar{\boldsymbol{x}} = \frac{1}{n}\sum_{i=1}^n \boldsymbol{x}_i, \quad \hat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^n (\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T, \tag{6.5}$$

sample mean and covariance. The latter holds only if the sample covariance has full rank d (in particular, $n \geq d$). Otherwise, the MLE for the covariance is undefined. We derive these estimators in Section 6.4.3.

6.4.1 Gaussian Class-Conditional Distributions

Fitting bell-shaped curves to exam results is fine. But how does all this help us with classification? Let us combine decision theory (Section 5.2) with ML estimation for Gaussians. Consider a binary classification problem with input patterns $\boldsymbol{x} \in \mathbb{R}^d$ and targets $t \in \{-1, +1\}$. Decision theory provides optimal discriminants, given that we *know* the joint density $p(\boldsymbol{x}, t)$. MLE allows us to fit parametric distribution families to data. Therefore, if we make a parametric *model assumption* about what $p(\boldsymbol{x}, t)$ could be, everything else falls in place. Let us assume that the class-conditional densities are Gaussian with unit covariance matrix:

$$p(\boldsymbol{x}|t) = N(\boldsymbol{x}|\boldsymbol{\mu}_t, \boldsymbol{I}) = (2\pi)^{-d/2} e^{-\frac{1}{2}\|\boldsymbol{x}-\boldsymbol{\mu}_t\|^2}, \quad \boldsymbol{\mu}_t \in \mathbb{R}^d.$$

Moreover, $P(t = +1) = \pi_1 \in (0, 1)$, $P(t = -1) = 1 - \pi_1$. The setup is depicted in Figure 6.6. Knowing that contour lines of spherical Gaussians are circles centered at the mean, the optimal classifier assigns $\boldsymbol{x}$ to class t for the closest mean $\boldsymbol{\mu}_t$ in Euclidean distance. At least in this two-dimensional example, the Bayes-optimal decision boundary is a line orthogonal to $\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}$. We have seen this example before in Section 2.1.

In general, the optimal discriminant under this setup is

$$y^*(\boldsymbol{x}) = \log\frac{p(\boldsymbol{x}|t = +1)\,\pi_1}{p(\boldsymbol{x}|t = -1)\,(1 - \pi_1)} = -\frac{1}{2}\left(\|\boldsymbol{x} - \boldsymbol{\mu}_{+1}\|^2 - \|\boldsymbol{x} - \boldsymbol{\mu}_{-1}\|^2\right) + \log\frac{\pi_1}{1 - \pi_1}.$$

Expanding the squared distances, the $\|\boldsymbol{x}\|^2$ terms cancel each other:

$$y^*(\boldsymbol{x}) = (\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1})^T\boldsymbol{x} - \frac{1}{2}\left(\|\boldsymbol{\mu}_{+1}\|^2 - \|\boldsymbol{\mu}_{-1}\|^2\right) + \log\frac{\pi_1}{1 - \pi_1}. \tag{6.6}$$

The optimal discriminant function is *linear.* A hyperplane in input space, just like those we discussed in Chapter 2. Its (unnormalized) normal vector is $\boldsymbol{w} =$

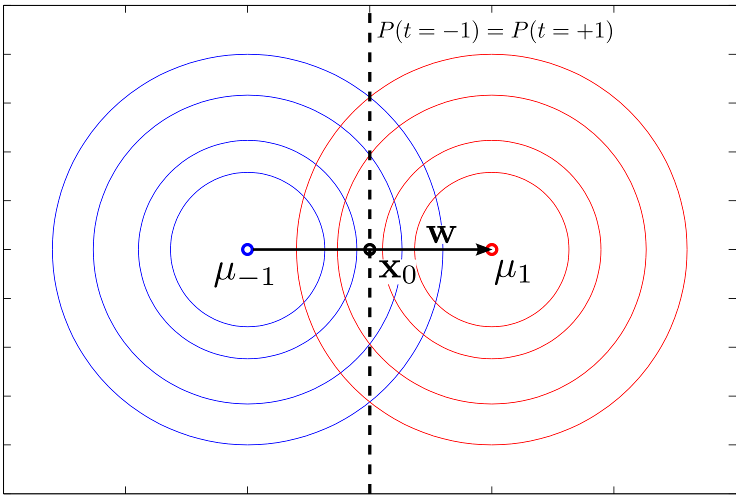


Figure 6.6: Two Gaussian class-conditional distribution with spherical covariance $\boldsymbol{I}$. The optimal discriminant is a hyperplane with normal vector $\boldsymbol{w} = \boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}$ and offset point $\boldsymbol{x}_0$. If $P(t=-1) = P(t=+1)$ (equal class priors), then $\boldsymbol{x}_0 = (\boldsymbol{\mu}_{+1} + \boldsymbol{\mu}_{-1})/2$. Otherwise, it is translated along the line through $\boldsymbol{\mu}_{-1}$ and $\boldsymbol{\mu}_{+1}$, towards the class mean whose $P(t)$ is the smaller.

$\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}$. If $c = \log\{\pi_1/(1-\pi_1)\}$, we can use $\|\boldsymbol{\mu}_{+1}\|^2 - \|\boldsymbol{\mu}_{-1}\|^2 = (\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1})^T(\boldsymbol{\mu}_{+1} + \boldsymbol{\mu}_{-1})$ to obtain

$$y^*(\boldsymbol{x}) = \boldsymbol{w}^T(\boldsymbol{x} - \boldsymbol{x}_0), \quad \boldsymbol{x}_0 = \frac{1}{2}(\boldsymbol{\mu}_{+1} + \boldsymbol{\mu}_{-1}) - \frac{c}{\|\boldsymbol{w}\|^2}\boldsymbol{w}.$$

The geometrical picture is clear (Figure 6.6). Imagine a line through the class means $\boldsymbol{\mu}_{+1}$, $\boldsymbol{\mu}_{-1}$. The optimal hyperplane is orthogonal to this line. It intersects the line at $\boldsymbol{x}_0$, which is the midpoint between the class means, $(\boldsymbol{\mu}_{+1}+\boldsymbol{\mu}_{-1})/2$, if and only if $c = 0$, or $P(t=+1) = P(t=-1) = 1/2$. For unequal class priors, $\boldsymbol{x}_0$ is obtained by translating the midpoint along the line, towards the class mean whose $P(t)$ is smaller. For this simple setup, we can compute the Bayes error analytically (Section 6.4.2). For the special case $c = 0$ (equal class priors),

$$R^* = \Phi(-\|\boldsymbol{w}\|/2), \quad \Phi(x) = (2\pi)^{-1/2}\int_{-\infty}^{x} e^{-t^2/2}\,dt.$$

Here, $\Phi(x)$ is the cumulative distribution function of the standard normal distribution $N(0,1)$: $\Phi(x) = P\{t \le x\}$, $t \sim N(0,1)$. As expected, R^* is a decreasing function of the distance $\|\boldsymbol{w}\| = \|\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}\|$ between the class means, $R^* = 1/2$ for $\boldsymbol{\mu}_{+1} = \boldsymbol{\mu}_{-1}$, and $R^* \to 0$ as $\|\boldsymbol{w}\| \to \infty$.

Maximum Likelihood Plug-in Discriminants

In order to use this in the real world, we can estimate the parameters $\boldsymbol{\mu}_{+1}$, $\boldsymbol{\mu}_{-1}$, and $\pi_1 = P(t=+1)$ from data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$, using the principle of maximum likelihood:

$$p(\mathcal{D}|\boldsymbol{\mu}_{+1},\boldsymbol{\mu}_{-1},\pi_1) = \left(\prod_{i=1}^{n} N(\boldsymbol{x}_i|\boldsymbol{\mu}_{t_i},\boldsymbol{I})\right)\pi_1^{n_1}(1-\pi_1)^{n-n_1}, \quad n_1 = \sum_i \mathrm{I}_{\{t_i=+1\}}.$$

Using our results from above,

$$\hat{\boldsymbol{\mu}}_{+1} = \frac{1}{n_1}\sum_i \mathrm{I}_{\{t_i=+1\}}\boldsymbol{x}_i, \quad \hat{\boldsymbol{\mu}}_{-1} = \frac{1}{n-n_1}\sum_i \mathrm{I}_{\{t_i=-1\}}\boldsymbol{x}_i, \quad \hat{\pi}_1 = \frac{n_1}{n}.$$

The *maximum likelihood plug-in discriminant* $\hat{y}(\boldsymbol{x})$ is given by plugging the ML estimates into (6.6):

$$\hat{y}(\boldsymbol{x}) = \hat{\boldsymbol{w}}^T\boldsymbol{x} - \frac{1}{2}\left(\|\hat{\boldsymbol{\mu}}_{+1}\|^2 - \|\hat{\boldsymbol{\mu}}_{-1}\|^2\right) + \log\frac{\hat{\pi}_1}{1-\hat{\pi}_1}, \quad \hat{\boldsymbol{w}} = \hat{\boldsymbol{\mu}}_{+1} - \hat{\boldsymbol{\mu}}_{-1}.$$

In summary, plug-in classifiers are obtained by the following schema:

- Pick a model $p(\boldsymbol{x},t|\boldsymbol{\theta}) = p(\boldsymbol{x}|t,\boldsymbol{\theta})P(t|\boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta}$. This choice determines everything else, and for real-world situations there is no uniformly optimal recipe for it. In this course, we will learn about consequences of modelling choices, so we can do them in an informed way. We need two basic properties:
  - For any fixed $\boldsymbol{\theta}$, the Bayes-optimal classifier is known and has a reasonably simple form.
  - ML density estimation is tractable for $p(\boldsymbol{x}|t,\boldsymbol{\theta})$ and $P(t|\boldsymbol{\theta})$

  In our example above, $p(\boldsymbol{x}|t,\boldsymbol{\theta}) = N(\boldsymbol{x}|\boldsymbol{\mu}_t,\boldsymbol{I})$, $P(t|\boldsymbol{\theta}) = \pi_1^{(1+t)/2}(1-\pi_1)^{(1-t)/2}$, and $\boldsymbol{\theta} = [\boldsymbol{\mu}_{+1}^T,\boldsymbol{\mu}_{-1}^T,\pi_1]^T$.
- Given training data $\mathcal{D} = \{(\boldsymbol{x}_i,t_i)\}$, estimate $\boldsymbol{\theta}$ by maximizing the likelihood, resulting in $\hat{\boldsymbol{\theta}}$.
- The ML plug-in classifier is obtained by plugging the estimated parameters $\hat{\boldsymbol{\theta}}$ into the optimal rule.

Plug-in classifiers are examples of the *generative modelling* paradigm. Our goal is to predict t from $\boldsymbol{x}$, and we get there by estimating the whole joint density by $p(\boldsymbol{x},t|\hat{\boldsymbol{\theta}})$, then use Bayes' formula to obtain the posterior $P(t|\boldsymbol{x},\hat{\boldsymbol{\theta}})$, based on which we classify. Another idea would be to estimate the posterior $P(t|\boldsymbol{x})$ directly, bypassing the modelling of inputs $\boldsymbol{x}$ altogether, which is what we do in *discriminative modelling.* We will get to the bottom of this important distinction in Chapter 8.

Equal Covariances $\neq \boldsymbol{I}$

A slightly more general case is given by the model assumptions $p(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t,\boldsymbol{\Sigma})$. We allow for a general covariance matrix $\boldsymbol{\Sigma}$, which is shared by all class-conditional distributions. To save space, let us write

$$\|\boldsymbol{v}\|_{\boldsymbol{\Sigma}}^2 := \boldsymbol{v}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{v}, \quad \|\boldsymbol{v}\|_{\boldsymbol{\Sigma}} := \sqrt{\boldsymbol{v}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{v}}.$$

The optimal discriminant function is

$$y^*(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)\,\pi_1}{p(\boldsymbol{x}|t=-1)\,(1-\pi_1)} = -\frac{1}{2}\left(\|\boldsymbol{x}-\boldsymbol{\mu}_{+1}\|_{\boldsymbol{\Sigma}}^2 - \|\boldsymbol{x}-\boldsymbol{\mu}_{-1}\|_{\boldsymbol{\Sigma}}^2\right) + c$$
$$= (\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1})^T\boldsymbol{\Sigma}^{-1}\boldsymbol{x} - \frac{1}{2}\left(\|\boldsymbol{\mu}_{+1}\|_{\boldsymbol{\Sigma}}^2 - \|\boldsymbol{\mu}_{-1}\|_{\boldsymbol{\Sigma}}^2\right) + c, \quad c = \log\frac{\pi_1}{1-\pi_1}.$$

A linear discriminant once more, with normal vector $\boldsymbol{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1})$. The difference between the means is transformed by the covariance. Roughly speaking, contributions of $\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1}$ along directions of large variance are downweighted, since the Gaussians overlap more along these directions. What about the Bayes error? We can reduce the equal covariance case to the spherical covariance case, for which we know the Bayes error already (Section 6.4.2). Namely, let $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$ the Cholesky decomposition (Section 4.2.2), and $\tilde{\boldsymbol{x}} = \boldsymbol{L}^{-1}\boldsymbol{x}$. Then, $p(\tilde{\boldsymbol{x}}|t) = N(\tilde{\boldsymbol{\mu}}_t, \boldsymbol{I})$, where $\tilde{\boldsymbol{\mu}}_t = \boldsymbol{L}^{-1}\boldsymbol{\mu}_t$. The optimal discriminating hyperplane has normal vector $\tilde{\boldsymbol{w}} = \tilde{\boldsymbol{\mu}}_{+1} - \tilde{\boldsymbol{\mu}}_{-1}$, and

$$\tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}} = (\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1})^T\boldsymbol{L}^{-T}\boldsymbol{L}^{-1}\boldsymbol{x} = (\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1})^T\boldsymbol{\Sigma}^{-1}\boldsymbol{x},$$

which is what we obtained above. This means that the Bayes error is given by the expressions derived in Section 6.4.2, replacing $\|\boldsymbol{w}\|$ by

$$\|\tilde{\boldsymbol{w}}\| = \sqrt{(\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1})} = \|\boldsymbol{\mu}_{+1}-\boldsymbol{\mu}_{-1}\|_{\boldsymbol{\Sigma}}.$$

For fixed positive definite $\boldsymbol{\Sigma}$, this norm is known as *Mahalanobis distance.* The corresponding ML plug-in classifier is obtained by estimating means $\boldsymbol{\mu}_t$, $t = -1, 1$, and covariance $\boldsymbol{\Sigma}$ by maximum likelihood (6.5). The reduction from equal to spherical covariances is unproblematic in the decision-theoretic context. However, if we lift it to plug-in rules by ML estimation, the non-spherical case can lead to substantial difficulties, in particular if n (number of data points) is not much larger than d (dimensionality of $\boldsymbol{x}$). To appreciate the nature of these difficulties, consider the case $d > n$. If the dimensionality is greater than the number of data points, the ML estimator for the covariance matrix is not even defined! This is an important general problem with generative ML approaches, we will analyze it in greater detail in Chapter 7.

What about the general case $p(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$? In this case, the optimal discriminant function is *quadratic* in $\boldsymbol{x}$. Contrary to quadratic functions used elsewhere in this course, it is *not* positive definite in general (its matrix is $\boldsymbol{\Sigma}_{+1}^{-1} - \boldsymbol{\Sigma}_{-1}^{-1}$), therefore can give rise to complex decision boundaries. In particular, decision regions need not be connected. We will not be concerned with this general case any further during this course. If you are interested, [13] has some pretty figures.

Multi-Way Classification

Finally, what about multi-way classification, $K > 2$ classes? Assume that $t \in \mathcal{T} = \{0, \dots, K-1\}$. Decision theory (Section 5.2) suggests to employ one discriminant function $y_k^*(\boldsymbol{x})$ per class,

$$y_k^*(\boldsymbol{x}) = \log\{p(\boldsymbol{x}|t=k)P(t=k)\} = -\frac{1}{2}\|\boldsymbol{x}-\boldsymbol{\mu}_k\|^2 + \log P(t=k) + C$$

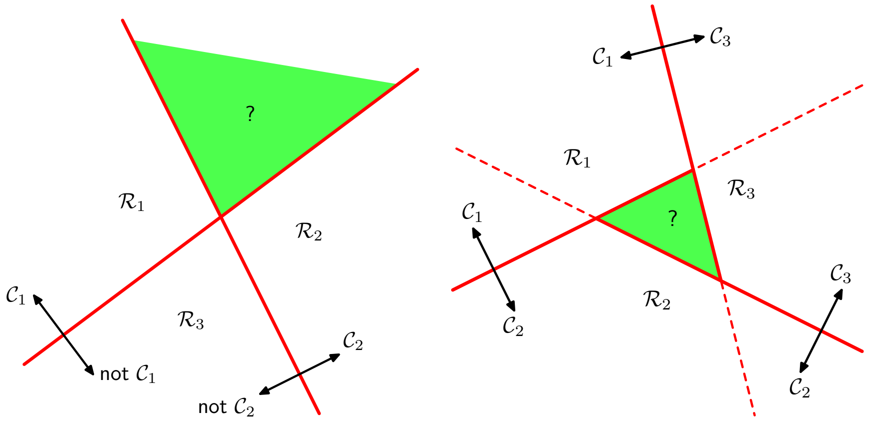


Figure 6.7: Attempts to construct a multi-way discriminant from a number of binary discriminants leads to ambiguous regions, shown in green. In the left panel, two "one-against-rest" discriminants are combined, one for class $\mathcal{C}_1$, the other for class $\mathcal{C}_2$. Both label the green region as positive. In the left panel, three discriminants are employed, one for each pair of classes. Each of them predicts the green region to belong to a different class.
Figure from [6] (used with permission).

in the spherical covariance case, where C is a constant. Given our model assumptions, the Bayes-optimal rule is $f^*(\boldsymbol{x}) = \operatorname{argmax}_{k\in\mathcal{T}} y_k^*(\boldsymbol{x})$. The ML plug-in classifier has the same form, plugging in ML estimates

$$\hat{\boldsymbol{\mu}}_k = n_t^{-1}\sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}}\boldsymbol{x}_i, \quad \hat{P}(t=k) = \frac{n_k}{n}, \quad n_k = \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}}.$$

Importantly, the optimal rule for Gaussian class-conditional densities, while defined in terms of K linear discriminant functions, is *not* a combination of binary linear *classifiers*. A lot of effort has been spent in machine learning in order to combine multi-way classifiers from binary linear ones. The most commonly used heuristic, "one-against-rest", uses K binary classifiers $f_k(\boldsymbol{x})$, discriminating $t=k$ versus $t\neq k$. There are always regions in which the binary votes of all $f_k(\boldsymbol{x})$ remain ambiguous (Figure 6.7, left). Another heuristic uses $K(K-1)/2$ binary classifiers $f_{k,k'}(\boldsymbol{x})$, $k\neq k'$, discriminating $t=k$ versus $t=k'$. Again, their votes remain ambiguous in some regions (Figure 6.7, right). More complex heuristics employ "error-correcting codes" or directed acyclic graphs. Decision theory indicates that such attempts cannot in general attain optimal performance[5].

[5] Note that if "one-against-rest" is based on optimal binary discriminant functions $y_k^*(\boldsymbol{x}) = \log\{P(t=k|\boldsymbol{x})/(1-P(t=k|\boldsymbol{x}))\}$ (instead of classifiers $f_k^*(\boldsymbol{x}) = \operatorname{sgn}(y_k^*(\boldsymbol{x}))$), the optimal K-way classifier *can* be combined as $f^*(\boldsymbol{x}) = \operatorname{argmax}_k y_k^*(\boldsymbol{x})$, since $p\mapsto \log\{p/(1-p)\}$ is increasing. However, "one-against-rest" is typically used with large margin binary classifiers, which do not provide consistent estimators of posterior class probabilities $P(t=k|\boldsymbol{x})$ (see Section 9.4).

6.4.2 Techniques: Bayes Error for Gaussian Class-Conditionals (*)

We saw in Section 6.4.1 that the Bayes-optimal classifier for Gaussian class-conditionals $p(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t, \boldsymbol{I})$ is $f^*(\boldsymbol{x}) = \mathrm{sgn}(y^*(\boldsymbol{x}))$ with a linear discriminant function $y^*(\boldsymbol{x})$ (6.6). In this section, we derive the generalization error of this rule, the Bayes error. Recall that $c = \log\{P(t = +1)/P(t = -1)\}$. The Bayes error is the sum of two parts, the first being

$$\begin{aligned} &P\left\{f^*(\boldsymbol{x}) = -1 \text{ and } t = +1\right\} \\ &= P(t = +1)P\left\{\|\boldsymbol{x} - \boldsymbol{\mu}_{+1}\|^2 > \|\boldsymbol{x} - \boldsymbol{\mu}_{-1}\|^2 + 2c \,\middle|\, t = +1\right\}, \end{aligned}$$

where $\boldsymbol{x} \sim N(\boldsymbol{\mu}_{+1}, \boldsymbol{I})$. Denote $\boldsymbol{w} = \boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}$. If $\tilde{\boldsymbol{x}} = \boldsymbol{x} - \boldsymbol{\mu}_{+1} \sim N(\boldsymbol{0}, \boldsymbol{I})$, the event is

$$\tfrac{1}{2}\|\tilde{\boldsymbol{x}}\|^2 > \tfrac{1}{2}\|\tilde{\boldsymbol{x}} + \boldsymbol{w}\|^2 + c \quad\Leftrightarrow\quad \boldsymbol{w}^T\tilde{\boldsymbol{x}} < -\frac{1}{2}\|\boldsymbol{w}\|^2 - c.$$

Since $\boldsymbol{w}^T\tilde{\boldsymbol{x}} \sim N(0, \|\boldsymbol{w}\|^2)$ (recall Section 6.3), this is

$$\Phi\left(-\frac{\|\boldsymbol{w}\|^2/2 + c}{\|\boldsymbol{w}\|}\right) = \Phi\left(-\frac{1}{2}\|\boldsymbol{w}\| - \frac{c}{\|\boldsymbol{w}\|}\right),$$

where

$$\Phi(x) = \int_{-\infty}^{x} N(t|0,1)\,dt = (2\pi)^{-1/2}\int_{-\infty}^{x} e^{-t^2/2}\,dt$$

is the cumulative distribution function of $N(0, 1)$. The second part of the error is $P\{f^*(\boldsymbol{x}) = 1 \text{ and } t = -1\}$. Due to the symmetry of the setup, this must be the same as the first if we replace indices $+1$ and -1 everywhere: $\|\boldsymbol{w}\|$ remains unchanged, c is replaced by $-c$. Therefore, the Bayes error is

$$R^* = P(t = +1)\Phi\left(-\frac{1}{2}\|\boldsymbol{w}\| - \frac{c}{\|\boldsymbol{w}\|}\right) + P(t = -1)\Phi\left(-\frac{1}{2}\|\boldsymbol{w}\| + \frac{c}{\|\boldsymbol{w}\|}\right)$$

A few sanity checks. First, $R^* \to 0$ as $\|\boldsymbol{w}\| \to \infty$. Also, $R^* = \min\{P(t = +1), P(t = -1)\}$ for $\|\boldsymbol{w}\| = 0$: if $\boldsymbol{x}$ is independent of the target t, the optimal rule uses $P(t)$ only. In the equal class prior case $P(t = +1) = P(t = -1) = 1/2$, R^* simplifies to $\Phi(-\|\boldsymbol{w}\|/2)$.

6.4.3 Techniques: MLE for Multivariate Gaussian (*)

The ML estimators for mean and covariance of a multivariate Gaussian $N(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, $\boldsymbol{x} \in \mathbb{R}^d$, are given by (6.5). In this section, we derive this result and learn to know some useful techniques.

Let $\mathcal{D} = \{\boldsymbol{x}_i \mid i = 1, \dots, n\}$, $\boldsymbol{x}_i \in \mathbb{R}^d$, and $L = -\log p(\mathcal{D}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. For the mean $\hat{\boldsymbol{\mu}}$, we can set $\nabla_{\boldsymbol{\mu}} L = \boldsymbol{0}$ and solve for $\boldsymbol{\mu}$. But since L is quadratic in $\boldsymbol{\mu}$, it is often easier to simply "see the solution" using a technique called *completing the square*:

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{2}(\boldsymbol{x}_i - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}) &= -n\bar{\boldsymbol{x}}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} + \frac{n}{2}\boldsymbol{\mu}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} + C_1 \\ &= \frac{n}{2}(\boldsymbol{\mu} - \bar{\boldsymbol{x}})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - \bar{\boldsymbol{x}}) + C_2, \end{aligned}$$

where C_1, C_2 do not depend on $\boldsymbol{\mu}$. But this looks like the quadratic we know from a Gaussian *over* $\boldsymbol{\mu}$, something like $N(\boldsymbol{\mu}|\bar{\boldsymbol{x}}, \boldsymbol{\Sigma}/n)$. We *know* this quadratic is smallest at the mean, therefore at $\bar{\boldsymbol{x}}$. Note that we dropped additive terms not involving $\boldsymbol{\mu}$ immediately: they do not influence the minimization, and there is no merit in keeping them around.

The derivation of $\hat{\boldsymbol{\Sigma}}$ is a bit more difficult. First, since $\hat{\boldsymbol{\mu}}$ does not depend on $\boldsymbol{\Sigma}$, we can plug it into L, then minimize the result w.r.t. $\boldsymbol{\Sigma}$. To this end, we will compute the gradient $\nabla_{\boldsymbol{\Sigma}} L$, set it equal to zero and solve for $\boldsymbol{\Sigma}$. The gradient w.r.t. a matrix may be a bit unfamiliar, but recall that matrix spaces are vector spaces as well. Before we start, recall the *trace* of a square matrix:

$$\operatorname{tr} \boldsymbol{A} = \sum_{j=1}^{d} a_{jj} = \mathbf{1}^T \operatorname{diag}(\boldsymbol{A}), \quad \boldsymbol{A} \in \mathbb{R}^{d \times d},$$

the sum of the diagonal entries. Unlike the determinant, the trace *is* a linear function: $\operatorname{tr}(\boldsymbol{A} + \alpha \boldsymbol{B}) = \operatorname{tr} \boldsymbol{A} + \alpha \operatorname{tr} \boldsymbol{B}$. An important result is $\operatorname{tr} \boldsymbol{BC} = \operatorname{tr} \boldsymbol{CB}$, given the product is a square matrix. Also, $\operatorname{tr} \boldsymbol{A}^T = \operatorname{tr} \boldsymbol{A}$ is obvious. Note that

$$\operatorname{tr} \boldsymbol{B}^T \boldsymbol{C} = \sum_{j,k} b_{jk} c_{jk},$$

so that $\operatorname{tr} \boldsymbol{B}^T \boldsymbol{C}$ can be seen as generalization of the Euclidean inner product to the space of matrices $\boldsymbol{B}$, $\boldsymbol{C}$ (which need not be square). A manipulation we will use frequently is

$$\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} = \operatorname{tr} \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} = \operatorname{tr} \boldsymbol{A} \boldsymbol{x} \boldsymbol{x}^T,$$

the trace of $\boldsymbol{A}$ times an outer product.

Define the sample covariance matrix as

$$\boldsymbol{S} = \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T.$$

We assume that $\boldsymbol{S}$ is invertible. As a covariance matrix, it is symmetric positive definite. The negative log likelihood as function of $\boldsymbol{\Sigma}$, where we plug in $\bar{\boldsymbol{x}}$ for $\boldsymbol{\mu}$, is

$$\sum_{i=1}^{n} \left(\log |\boldsymbol{\Sigma}| + (\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_i - \bar{\boldsymbol{x}}) \right)$$
$$= n \log |\boldsymbol{\Sigma}| + \sum_{i=1}^{n} \operatorname{tr}(\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T \boldsymbol{\Sigma}^{-1} = n \left(\log |\boldsymbol{\Sigma}| + \operatorname{tr} \boldsymbol{S} \boldsymbol{\Sigma}^{-1} \right).$$

Here, we dropped an additive constant and also the prefactor 1/2. We also used the outer product manipulation involving the trace. If $\boldsymbol{P} = \boldsymbol{\Sigma}^{-1}$, then

$$\hat{\boldsymbol{P}} = \operatorname*{argmin}_{\boldsymbol{P}} \{ f(\boldsymbol{P}) = \operatorname{tr} \boldsymbol{SP} - \log |\boldsymbol{P}| \}, \quad \hat{\boldsymbol{\Sigma}} = \hat{\boldsymbol{P}}^{-1},$$

where we used $|\boldsymbol{P}| = 1/|\boldsymbol{\Sigma}|$, so $\log |\boldsymbol{P}| = -\log |\boldsymbol{\Sigma}|$. The minimization is over positive definite matrices $\boldsymbol{P}$, but we ignore this constraint for now. Let us compute the gradient $\nabla_{\boldsymbol{P}} f = [\partial f / \partial p_{jk}] \in \mathbb{R}^{d \times d}$. Obviously, $\nabla_{\boldsymbol{P}} \operatorname{tr} \boldsymbol{SP} = \boldsymbol{S}$ (we used the symmetry of $\boldsymbol{S}$ here). Next, we will show a result which will be of independent interest later during the course. At any $\boldsymbol{P} \in \mathbb{R}^{d\times d}$ with $|\boldsymbol{P}| > 0$:

$$\nabla_{\boldsymbol{P}} \log |\boldsymbol{P}| = \boldsymbol{P}^{-T}. \tag{6.7}$$

Altogether, $\nabla_{\boldsymbol{P}} f(\boldsymbol{P}) = \boldsymbol{S} - \boldsymbol{P}^{-T} = \boldsymbol{0}$ if and only if $\boldsymbol{P} = \boldsymbol{S}^{-1}$. This solution is positive definite, so the constraint is satisfied automatically in this case. Therefore, the ML estimator for the covariance is $\hat{\boldsymbol{\Sigma}} = \boldsymbol{S}$.

To prove[6] (6.7), we employ properties of the determinant (recall Section 6.3.1).

$$\frac{\partial \log |\boldsymbol{P}|}{\partial p_{jk}} = \lim_{\varepsilon\to 0} \frac{1}{\varepsilon}\left(\log |\boldsymbol{P} + \varepsilon\boldsymbol{\delta}_j\boldsymbol{\delta}_k^T| - \log |\boldsymbol{P}|\right).$$

Here, $\boldsymbol{P} + \varepsilon\boldsymbol{\delta}_j\boldsymbol{\delta}_k^T$ denotes the matrix obtained by adding ε to element (j,k) of $\boldsymbol{P}$. Now,

$$\log |\boldsymbol{P} + \varepsilon\boldsymbol{\delta}_j\boldsymbol{\delta}_k^T| - \log |\boldsymbol{P}| = \log\left\{|\boldsymbol{P} + \varepsilon\boldsymbol{\delta}_j\boldsymbol{\delta}_k^T| \cdot |\boldsymbol{P}^{-1}|\right\} = \log |\boldsymbol{I} + \varepsilon\boldsymbol{P}^{-1}\boldsymbol{\delta}_j\boldsymbol{\delta}_k^T|.$$

Denote $\boldsymbol{v} = \boldsymbol{P}^{-1}\boldsymbol{\delta}_j$, the j-th column of the inverse. Now, $\boldsymbol{I} + \varepsilon\boldsymbol{v}\boldsymbol{\delta}_k^T$ is obtained from the identity by adding $\varepsilon\boldsymbol{v}$ to the k-th column. We know that the determinant is linear w.r.t. each column, so

$$|\boldsymbol{I} + \varepsilon\boldsymbol{v}\boldsymbol{\delta}_k^T| = |\boldsymbol{I}| + \varepsilon|\boldsymbol{M}| = 1 + \varepsilon|\boldsymbol{M}|, \quad \boldsymbol{M} = \boldsymbol{I} + (\boldsymbol{v} - \boldsymbol{\delta}_k)\boldsymbol{\delta}_k^T,$$

where $\boldsymbol{M}$ is $\boldsymbol{I}$, except the k-th column is replaced by $\boldsymbol{v}$. Plugging this in, we have

$$\frac{\partial \log |\boldsymbol{P}|}{\partial p_{jk}} = \lim_{\varepsilon\to 0} \frac{1}{\varepsilon} \log\left(1 + \varepsilon|\boldsymbol{M}|\right) = |\boldsymbol{M}|.$$

Recall from Section 6.3.1 that we can do column eliminations without changing the determinant. Eliminating all entries of $\boldsymbol{v}$ except v_k:

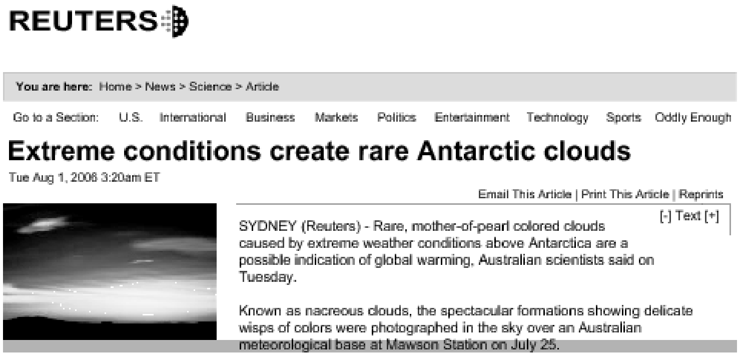
$$|\boldsymbol{M}| = \left|\boldsymbol{I} + (v_k - 1)\boldsymbol{\delta}_k\boldsymbol{\delta}_k^T\right| = v_k = (\boldsymbol{P}^{-1})_{kj}.$$

This concludes the proof.

6.5 Maximum Likelihood for Discrete Distributions

Handwritten digits? The internet and the "data deluge" provide a richer playground for machine learning today. Consider the problem of *text classification* (Figure 6.8). Given a document (for example, a news article), what is it talking about? We will concentrate on a simple K-way classification setup, where each document is to be classified according to a flat fixed-sized target set $\mathcal{T} = \{0, \dots, K-1\}$ (for example: politics, business, sports, science, movies, ...). Modern models tend to employ hierarchical grouping schemes (a document

[6] There are more direct proofs, involving the eigendecomposition of $\boldsymbol{S}$ and $\boldsymbol{P}$. Our proof just uses elementary properties of the determinant.


REUTERS
You are here: Home > News > Science > Article
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enough
Email This Article | Print This Article | Reprints
[-] Text [+]

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Figure 6.8: The Reuters RCV1 collection is a set of 800,000 documents (news articles), with about 200 words per document on average. After standard preprocessing (stop word removal), its dictionary (set of distinct words) is roughly of size 400,000. A common machine learning problem associated with this data is to classify documents into groups (for example: politics, business, sports, science, movies), which are often organized in a hierarchical fashion.

could be about politics, US politics, Barack Obama) and may associate parts of a document with different topics.

Some vocabulary. The atomic unit is the word. A document is an ordered set of words. A corpus (plural: corpora) is a dataset of documents, part of which may be labeled according to a grouping scheme. Roughly, preprocessing works as follows:

- Remove punctuation, non-text entities.
- Remove stop words: frequent words which occur in most documents and tend to carry no discriminative information. For example: a, and, is, it, be, by, for, to, …
- Stemming: Strip prefixes and endings in order to reduce words to their stem (this may not be done for certain natural language processing tasks, but is typically done for text classification).
- Build dictionary $\mathcal{C} = \{c_1, \dots, c_M\}$ of distinct words occuring somewhere in the corpus. The dictionary size is M.

Given that, we can represent a document of N words as $\boldsymbol{x} = [x_1, \dots, x_N]^T$, $x_j \in \{1, \dots, M\}$. $x_j = m$ specifies that the j-th word is c_m. In order to implement a generative classifier, we have to specify class-conditional distributions $P(\boldsymbol{x}|t = k)$. Note that the occurence of individual words can be highly indicative for one class or another. “Currency”, “Dow”, “Credit” sounds more like business than sports. To model this observation, we can use one distribution $\boldsymbol{p}^{(k)}$ over words in the dictionary $\mathcal{C}$ for each class $k = 0, \dots, K-1$. Formally,

$$\boldsymbol{p}^{(k)} \in \Delta_M = \left\{ \boldsymbol{q} \in \mathbb{R}^M \;\middle|\; q_m \geq 0 \;\; \forall m = 1, \dots, M, \;\; \sum\nolimits_{m=1}^{M} q_m = 1 \right\}.$$

The set Δ_M of all distributions over M objects is called the M-dimensional *probability simplex.* Now, if $\boldsymbol{p}^{(k)} \in \Delta_M$, $k = 0, \dots, K-1$, our generative model is

$$P(\boldsymbol{x}|N, t = k) = \prod_{j=1}^{N} p^{(k)}_{x_j}, \quad \boldsymbol{x} \in \{1, \dots, M\}^N.$$

The conditioning on the document length N is a technical point, which will play no role for the discriminant. We need it in order to ensure that $\sum_{\boldsymbol{x}} P(\boldsymbol{x}|N, t = k) = 1$. We will use $\boldsymbol{x}, N$ to represent a document, with

$$P(\boldsymbol{x}, N|t = k) = P(N) \prod_{j=1}^{N} p^{(k)}_{x_j},$$

where $P(N)$ is a distribution over $\mathbb{N}$ which does not depend on t. For our classification purposes, we can use $P(\boldsymbol{x}|N, t)$ in place of $P(\boldsymbol{x}, N|t)$, since the $P(N)$ factor cancels out in ratios like $P(\boldsymbol{x}, N|t = k)/P(\boldsymbol{x}, N|t = k')$, and these are all we ever need in the end. If you are confused at this point, simply move on and ignore the N in $P(\boldsymbol{x}|N, t)$ as a technical detail.

Imagine the build-up of $P(\boldsymbol{x}|N, t)$ for two distinct classes $t = k, k'$, say business (k) and sports (k'). For each word x_j of $\boldsymbol{x}$, we multiply $P(\boldsymbol{x}|N, t = k)$ and $P(\boldsymbol{x}|N, t = k')$ by $p^{(k)}_{x_j}$ and $p^{(k')}_{x_j}$ respectively. For example, if $c_{x_1} =$"CEO", presumably $p^{(k)}_{x_1} > p^{(k')}_{x_1}$, so the fraction $P(\boldsymbol{x}|N, t = k)/P(\boldsymbol{x}|N, t = k')$ increases. On the other hand, $c_{x_5} =$"Football" implies a decrease of the ratio by way of multiplication with $p^{(k)}_{x_5}/p^{(k')}_{x_5} < 1$. Each word contributes to the accumulation of evidence in a multiplicative fashion, *independent* of the distribution of other words. The combined parameters[7] of this model are $\boldsymbol{\theta} = [(\boldsymbol{p}^{(0)})^T, \dots, (\boldsymbol{p}^{(K-1)})^T, P(t = 0), \dots, P(t = K-1)]^T$. In practice, we deal with many documents $\boldsymbol{x}_i$ of different lengths N_i, and a representation with M factors is preferable:

$$P(\boldsymbol{x}|N, t = k) = \prod_{j=1}^{N} p^{(k)}_{x_j} = \prod_{m=1}^{M} \left(p^{(k)}_m\right)^{\phi_m(\boldsymbol{x})}, \quad \phi_m(\boldsymbol{x}) = \sum_{j=1}^{N} \mathrm{I}_{\{x_j = m\}}. \quad (6.8)$$

$\phi_m(\boldsymbol{x})$ is the number of times the word c_m occurs in document $\boldsymbol{x}$. Note that in general, the majority of the counts will be zero. Also, $\sum_{m=1}^{M} \phi_m(\boldsymbol{x}) = N$. The *feature vector* $\boldsymbol{\phi}(\boldsymbol{x}) = [\phi_m(\boldsymbol{x})] \in \mathbb{N}^M$ summarizes the information in $\boldsymbol{x}$ required to compute $P(\boldsymbol{x}|N, t = k)$ for all $k = 0, \dots, K-1$. Such summaries are called *sufficient statistics.* Given our model assumptions, this is all the information we need to know (and therefore, compute) in order to learn and predict. Whenever a model for documents $\boldsymbol{x}$ has single word occurence features as sufficient statistics, it falls under the *bag of words* assumption. The same count vector is obtained by permuting words in $\boldsymbol{x}$ arbitrarily, their ordering does not matter. It is as if we cut $\boldsymbol{x}$ into single words, put them in a bag and mix them up. This seems like a drastic assumption, but it is frequently made in information retrieval, since resulting computational simplifications are very substantial.

[7] We do not include $P(N)$ in $\boldsymbol{\theta}$, since the discriminant functions do not depend on it.

What are the optimal discriminant functions under this model?

$$
\begin{aligned}
y_k^*(\boldsymbol{x}) &= \log\left\{P(\boldsymbol{x}|N, t=k)P(t=k)\right\} = \sum_{m=1}^{M} \phi_m(\boldsymbol{x}) \log p_m^{(k)} + \log P(t=k) \\
&= (\boldsymbol{w}_k)^T \boldsymbol{\phi}(\boldsymbol{x}) + \log P(t=k), \quad \boldsymbol{w}_k = \left[\log p_m^{(k)}\right] \in \mathbb{R}^M.
\end{aligned}
\tag{6.9}
$$

Linear functions again! The weight vectors $\boldsymbol{w}_k$ correspond to log probabilities, which implies constraints[8] on the coefficients, but these are satisfied automatically in ML estimation. To discriminate between classes k and k', we would use

$$
y_k^*(\boldsymbol{x}) - y_{k'}(\boldsymbol{x}) = \sum_{m=1}^{M} \phi_m(\boldsymbol{x}) \underbrace{\log \frac{p_m^{(k)}}{p_m^{(k')}}}_{w_{k,m} - w_{k',m}} + \log \frac{P(t=k)}{P(t=k')}.
$$

The evidence for our decision is combined by summing up log odds $\log\{p_m^{(k)}/p_m^{(k')}\}$ for words c_m, each proportional to the number of occurences in $\boldsymbol{x}$. Each term depends on statistics and distributions for a single word only, there are no cross-terms.

6.5.1 Using Indicators in Maximum Likelihood Estimation

Given data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$, where $\boldsymbol{x}_i \in \{1, \dots, M\}^{N_i}$ (N_i is the length of the i-th document in our training corpus), we can estimate the model parameters $\boldsymbol{\theta}$ by maximizing the data likelihood. We will simplify the derivation of the ML estimators by systematic use of *indicator variables*, a technique which is indispensable with discrete variable models of complex structure used in modern information retrieval and machine learning. Let us rederive (6.8), using a funny way to put things:

$$
p_{x_j}^{(k)} = \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\mathrm{I}_{\{x_j = m\}}}.
$$

This identity is based on the rules[9] $a^1 = a$, $a^0 = 1$ for all $a \in \mathbb{R}$. $\mathrm{I}_{\{x_j=m\}}$ is an example for an indicator variable associated with x_j. The name for indicators in some machine learning texts is "1-of-M-coding" or "winner-takes-all-coding". Seen as a vector $[\mathrm{I}_{\{x_j=m\}}] \in \mathbb{R}^M$, exactly one component is one (namely, the x_j-th), all others are zero. On its own, the term $p_{x_j}^{(k)}$ is of course simpler than the indicator product, but if we multiply many such terms, we simply have to add up the indicators in the exponent:

$$
\prod_{j=1}^{N} \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\mathrm{I}_{\{x_j=m\}}} = \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\sum_{j=1}^{N} \mathrm{I}_{\{x_j=m\}}}.
$$

[8] Namely, $\sum_m e^{w_{k,m}} = 1$.

[9] In particular, $0^0 = 1$ everywhere in this course. This is mainly a convention to make indicators work. The argument $\lim_{x \to 0} x^x = \exp(\lim_{x \to 0} x \log x) = 1$ is also convincing.

This was pretty simple. A more complex example is given by the likelihood for our text classification model. Here, we use indicators $\mathrm{I}_{\{x_{i,j}=m\}}$, where $x_{i,j}$ is the j-th word of the i-th document, $\boldsymbol{x}_i = [x_{i,j}]$. Moreover, we use label indicators $\mathrm{I}_{\{t_i=k\}}$. The first step is to expand the likelihood, a product over $i = 1, \dots, n$, by introducing products over label values k, word values m:

$$\begin{aligned} P(\mathcal{D}|\boldsymbol{\theta}) &= \prod_{i=1}^{n} P(\boldsymbol{x}_i, N_i|t_i)P(t_i) = \prod_{i=1}^{n}\prod_{k=0}^{K-1} \left(P(\boldsymbol{x}_i|N_i, t=k)P(t=k)P(N_i)\right)^{\mathrm{I}_{\{t_i=k\}}} \\ &= \prod_{i=1}^{n} P(N_i) \prod_{k=0}^{K-1} P(t=k)^{\mathrm{I}_{\{t_i=k\}}} \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\mathrm{I}_{\{t_i=k\}} \sum_{j=1}^{N_i} \mathrm{I}_{\{x_{i,j}=m\}}}. \end{aligned}$$

The important point here is that the products over k and m are unconstrained over *all* label and word values. All constraints are encoded in the indicators. In particular, we can always interchange unconstrained products (or sums). Pulling the product over data points inside, then summing exponents instead:

$$\begin{aligned} P(\mathcal{D}|\boldsymbol{\theta}) &= C \prod_{k=0}^{K-1} P(t=k)^{\sum_i \mathrm{I}_{\{t_i=k\}}} \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\sum_i \mathrm{I}_{\{t_i=k\}} \sum_{j=1}^{N_i} \mathrm{I}_{\{x_{i,j}=m\}}}, \\ C &= \prod_i P(N_i). \end{aligned}$$

Here, $\prod_i$ and $\sum_i$ is over *all* data points. In the same way, we could write $\prod_k$ and $\prod_m$ over *all* label and word values. Dropping the range in sums and products is very commonly done in research papers, and we will sometimes follow suit to keep notations simple.

Indicator variables provide a simple and mechanical way to convert the likelihood in its original form ($\prod_i$) into the likelihood in a useful form ($\prod_k \prod_m$), directly in terms of the model parameters $P(t = k)$ and $p_m^{(k)}$. We have to accumulate the counts which appear in the exponents:

$$n_k = \sum_i \mathrm{I}_{\{t_i=k\}}, \quad N^{(k,m)} = \sum_i \mathrm{I}_{\{t_i=k\}} \sum_{j=1}^{N_i} \mathrm{I}_{\{x_{i,j}=m\}}.$$

$N^{(k,m)}$ is the number of times the word c_m occurs in the documents $\boldsymbol{x}_i$ labeled as $t_i = k$, and n_k is the number of documents labeled as $t_i = k$. With these, the log likelihood is

$$\log P(\mathcal{D}|\boldsymbol{\theta}) = \sum_{k=0}^{K-1} n_k \log P(t=k) + \sum_{k=0}^{K-1}\sum_{m=1}^{M} N^{(k,m)} \log p_m^{(k)} + \log C. \quad (6.10)$$

There is no need to drag around additive constants, and we can drop $\log C$ at will. Given the thumbtack example of Section 6.2 and common sense, you may guess the following maximizers of the log likelihood is

$$\hat{p}_m^{(k)} = \frac{N^{(k,m)}}{\sum_{m'} N^{(k,m')}}, \quad \hat{P}(t=k) = \frac{n_k}{n}, \quad (6.11)$$

the ratios of empirical counts. This is correct. We establish these maximum likelihood estimators for our text classification model in Section 6.5.3.

6.5.2 Naive Bayes Classifiers

Let us put things together. The optimal discriminant functions for known probabilities are the linear functions of (6.9). Not knowing them, we plug in their ML estimates (6.11) instead. Variants of this particular maximum likelihood plug-in classifier are widely used for document classification. While there are more advanced methods, they are also more costly to train and evaluate. Comparing against this simple and efficient baseline technique is a must. It is an example of a *naive Bayes classifier.*

A naive Bayes classifier comes with class-conditional distributions $P(\boldsymbol{x}|t)$ which have a factorizing structure. For our text classifier,

$$P(\boldsymbol{x}|N, t = k) = \prod_{j=1}^{N} p_{x_j}^{(k)} = \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\phi_m(\boldsymbol{x})}.$$

Each term in the rightmost product depends on a distinct part of the parameter vector $\boldsymbol{\theta}$ (here: a single coefficient $p_m^{(k)}$). This *factorization assumption* implies crucial simplifications. The plug-in discriminants are linear functions, which can be evaluated rapidly. More importantly, the log likelihood decomposes additively and can be maximized efficiently.

A naive Bayes classifier does not have to use word count features $\phi_m(\boldsymbol{x})$ over a dictionary, but can be run based on any feature map $\boldsymbol{\phi}(\boldsymbol{x}) = [\phi_m(\boldsymbol{x})]$ over the input point $\boldsymbol{x}$. Naive Bayes far transcends text classification and can be used with rather arbitrary input points, even discrete and continuous attributes mixed together. Our example above is special, in that the different parameter values $p_m^{(k)}$, $m = 1, \dots, M$, come together to form one distribution in Δ_M. Naive Bayes can be used with features and corresponding probabilities which are not linked in any way. For example, consider a *binary* feature map $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}) = [\tilde{\phi}_m(\boldsymbol{x})]$, where $\tilde{\phi}_m(\boldsymbol{x}) \in \{0, 1\}$ are not linearly dependent like the word counts. If $\boldsymbol{x}$ is a document once more, each $\tilde{\phi}_m(\boldsymbol{x})$ could be sensitive to a certain word pattern, indicating its presence by $\tilde{\phi}_m(\boldsymbol{x}) = 1$. A naive Bayes setup[10] would be

$$P(\boldsymbol{x}|t = k) = \prod_{m=1}^{M} \left(p_m^{(k)}\right)^{\tilde{\phi}_m(\boldsymbol{x})} \left(1 - p_m^{(k)}\right)^{1-\tilde{\phi}_m(\boldsymbol{x})}. \tag{6.12}$$

If you parse this expression with indicators in mind, you see that $p_m^{(k)}$ encodes $P\{\tilde{\phi}_m(\boldsymbol{x}) = 1 \,|\, t = k\}$. Once more, the log likelihood decouples additively over the different $p_m^{(k)}$, and we can estimate them independently of each other. Naive Bayes classification for binary features is summarized at the end of this subsection.

At this point, we should note that the term "naive Bayes" is not used consistently in the literature. There is a narrow and a more general definition, and we use the latter here. To understand the difference, consider the two examples we looked at. The narrow definition stipulates that the class-conditionals $P(\boldsymbol{x}|t = k)$ are such that, given $t = k$, the different features $\phi_m(\boldsymbol{x})$ are conditionally independent in the probabilistic sense (see Section 5.1.1). This is the

[10] For the meticulous: We should use $P(\tilde{\boldsymbol{\phi}}(\boldsymbol{x})|t = k)$ instead of $P(\boldsymbol{x}|t = k)$, in order to obtain a proper distribution. After all, $\boldsymbol{x} \leftrightarrow \tilde{\boldsymbol{\phi}}(\boldsymbol{x})$ may not be one-to-one.

case for our latter binary feature example (6.12), but it is *not* the case for the document classification setup (6.8): the constraint $\sum_{m=1}^M \phi_m(\boldsymbol{x}) = N$ obviously links the features. The more general definition of naive Bayes, adopted here, requires the class-conditionals $P(\boldsymbol{x}|t = k)$ to factorize w.r.t. the parameters $p_m^{(k)}$, so that the log likelihood decouples additively. However, the features may still be linked[11], and so are the corresponding parameters.

One final observation, in preparation of things to come. Back to the document classification example. What happens if one specific word c_m does not occur in *any* documents $\boldsymbol{x}_i$ labeled as $t_i = k$? Go back up and check for yourself. The ML estimator $\hat{\boldsymbol{p}}^{(k)}$ will have $\hat{p}_m^{(k)} = 0$ in this case. Under this distribution, it is *impossible* that any document of class k ever contains c_m. In other words, suppose I come along with a document $\boldsymbol{x}_*$ which contains c_m at least once: $\phi_m(\boldsymbol{x}_*) > 0$. Then, the ML plug-in discriminant function $\hat{y}_k(\boldsymbol{x})$ of the form (6.9) is

$$\hat{y}_k(\boldsymbol{x}_*) = \phi_m(\boldsymbol{x}_*) \log \hat{p}_m^{(k)} + \dots = -\infty.$$

The hypothesis $t = k$ is entirely ruled out for $\boldsymbol{x}_*$, due to the absence of a *single* word c_m from its training data. Does this matter? Yes, very much so. Natural language distributions over words are extremely heavy-tailed, meaning that new unseen words pop up literally all the time. The fact that some c_m does not occur in documents for some class will happen with high probability for any real-world corpus. We will analyze this serious shortcoming of ML plug-in rules in Chapter 7, where we will learn how it can be alleviated.

Summary: Naive Bayes Classification for Binary Features

Suppose that $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}) = [\tilde{\phi}_m(\boldsymbol{x})]$, where $\tilde{\phi}_m(\boldsymbol{x}) \in \{0, 1\}$. Different from word count features, the different $\tilde{\phi}_m$ can be on or off independent of each other. Naive Bayes classification is based on the model (6.12). This means that given $t = k$, the features $\tilde{\phi}_m(\boldsymbol{x})$ are conditionally independent. The classifier comes with parameters $p_m^{(k)} \in [0, 1]$, $m = 1, \dots, M$, $k = 0, \dots, K-1$, one for each feature and class. These are estimated by maximum likelihood. If there are n_k input points $\boldsymbol{x}_i$ labeled as $t_i = k$, and $n = \sum_{k=0}^{K-1} n_k$, then

$$\hat{p}_m^{(k)} = \frac{\sum_{i=1}^n \mathrm{I}_{\{t_i = k\}} \tilde{\phi}_m(\boldsymbol{x}_i)}{n_k},$$

the fraction of points $\boldsymbol{x}_i$ of class k with $\tilde{\phi}_m(\boldsymbol{x}_i) = 1$. The ML estimator for $P(t)$ is

$$\hat{P}(t = k) = \frac{n_k}{n}.$$

The trained classifier uses $\hat{P}(t = k)$ and $\hat{P}(\boldsymbol{x}|t = k)$, the latter is (6.12) with $\hat{p}_m^{(k)}$ plugged in. For example, the (posterior) probability of class $\tilde{k}$ for a new point $\boldsymbol{x}$ is computed by first computing

$$\hat{P}(\boldsymbol{x}|t = k)\hat{P}(t = k) = \left(\prod_{m=1}^M (\hat{p}_m^{(k)})^{\tilde{\phi}_m(\boldsymbol{x})} (1 - \hat{p}_m^{(k)})^{1 - \tilde{\phi}_m(\boldsymbol{x})} \right) \frac{n_k}{n}.$$

[11] For the meticulous: This linkage is typically a *linear* one (linear equality constraints), expressed in not overly many constraints. It is possible to write down models with decoupling log likelihood and intricate nonlinear constraints between the features. Such models would not be called "naive Bayes" anymore.

Then, using Bayes' formula:

$$\hat{P}(t=\tilde{k}|\boldsymbol{x}) = \frac{\hat{P}(\boldsymbol{x}|t=\tilde{k})\hat{P}(t=\tilde{k})}{\sum_k \hat{P}(\boldsymbol{x}|t=k)\hat{P}(t=k)}.$$

In practice, we have to use log in order to convert products into sums, otherwise we produce overflow or underflow. It is easiest to work in terms of discriminant functions:

$$\begin{aligned}\hat{y}_k(\boldsymbol{x}) &= \log\left\{\hat{P}(\boldsymbol{x}|t=k)\hat{P}(t=k)\right\} \\ &= \sum_{m=1}^{M}\left(\tilde{\phi}_m(\boldsymbol{x})\log\hat{p}_m^{(k)} + (1-\tilde{\phi}_m(\boldsymbol{x}))\log(1-\hat{p}_m^{(k)})\right) + \log\frac{n_k}{n}.\end{aligned}$$

The naive Bayes classifier decides for the class $\operatorname{argmax}_k \hat{y}_k(\boldsymbol{x})$. Moreover,

$$\hat{P}(t=\tilde{k}|\boldsymbol{x}) = \frac{e^{\hat{y}_{\tilde{k}}(\boldsymbol{x})}}{\sum_k e^{\hat{y}_k(\boldsymbol{x})}}.$$

6.5.3 Techniques: Maximizing Discrete Log Likelihoods (*)

In this section, we establish the maximum likelihood estimators (6.11) for the document classification setup. In fact, the log likelihood decomposes additively in $K+1$ separate terms, one for $P(t) \in \Delta_K$, and K for $\boldsymbol{p}^{(k)} \in \Delta_M$, $k = 0,\dots,K-1$. They are instances of ML estimation for a *multinomial* distribution. Suppose $\mathcal{D} = \{x_1,\dots,x_n\}$, $x_i \in \{1,\dots,L\}$, is sampled independently from $\boldsymbol{p} \in \Delta_L$ (model assumption). The log likelihood is

$$\mathcal{L} = \log P(\mathcal{D}|\boldsymbol{p}) = \sum_{l=1}^{L} n_l \log p_l, \quad n_l = \sum_{i=1}^{n} \mathrm{I}_{\{x_i=l\}}.$$

The maximum likelihood estimation problem is $\max_{\boldsymbol{p}\in\Delta_L} \log P(\mathcal{D}|\boldsymbol{p})$. For $L=2$, we can parameterize $\boldsymbol{p}$ by a single parameter, and the solution was obtained in Section 6.2. In the general case, we could use the vanilla technique of Lagrange multipliers (see Appendix A). However, let us do things differently, learning about one of the most fundamental inequalities in mathematics in passing. First,

$$\mathcal{L} = n\sum_{l=1}^{L}\hat{p}_l \log p_l, \quad \hat{p}_l = \frac{n_l}{n}.$$

Note that $\hat{\boldsymbol{p}} = [\hat{p}_l] \in \Delta_L$ just like $\boldsymbol{p}$, since $\sum_l n_l = n$. What you should take away from this section is the following:

$$\boldsymbol{q} = \operatorname*{argmax}_{\boldsymbol{p}\in\Delta_L}\left\{F_{\boldsymbol{q}}(\boldsymbol{p}) = \sum\nolimits_{l=1}^{L} q_l \log p_l\right\}, \quad \boldsymbol{q} \in \Delta_L. \tag{6.13}$$

If you are given a distribution $\boldsymbol{q} \in \Delta_L$ and seek the maximizer of $F_{\boldsymbol{q}}(\boldsymbol{p})$ for $\boldsymbol{p} \in \Delta_L$, the unique answer is $\hat{\boldsymbol{p}} = \boldsymbol{q}$. This problem appears over and over again[12]

[12] It holds just as well for distributions over continuous variables, even though our proof here only covers the discrete finite case.

in machine learning. Whenever you recognize this pattern, you immediately *know* the solution: no need for Lagrange multipliers. Sometimes, the best way to derive pattern recognition methods is pattern recognition!

Let us use (6.13) to establish the ML estimators (6.11). How often do you spot our pattern in the log likelihood (6.10)? $K+1$ times. First, $L \to K$, $q_k \to n_k/n$, $p_k \to P(t=k)$ (multiply and divide by n to obtain frequencies n_k/n). Solution: $\hat{P}(t=k) = \hat{p}_k = q_k = n_k/n$. Next, fix $k = 0, \dots, K-1$, and let $N^{(k)} = \sum_m N^{(k,m)}$. $L \to M$, $q_m \to N^{(k,m)}/N^{(k)}$, $p_m \to p_m^{(k)}$ (multiply and divide by $N^{(k)}$ to obtain frequencies). Solution: $\hat{p}_m^{(k)} = \hat{p}_m = q_m = N^{(k,m)}/N^{(k)}$.

In order to establish our pattern rule (6.13), let us look at the difference

$$\mathrm{D}[\boldsymbol{q} \,\|\, \boldsymbol{p}] := F_{\boldsymbol{q}}(\boldsymbol{q}) - F_{\boldsymbol{q}}(\boldsymbol{p}) = \sum_{l=1}^{L} q_l \log q_l - \sum_{l=1}^{L} q_l \log p_l = \sum_{l=1}^{L} q_l \log \frac{q_l}{p_l}, \quad \boldsymbol{q}, \boldsymbol{p} \in \Delta_L.$$

This function of two distributions over the same set is called *relative entropy* (or *Kullback-Leibler divergence*, or also “cross-entropy” in the neural networks literature). We need to show that $\mathrm{D}[\boldsymbol{q} \,\|\, \boldsymbol{p}] \geq 0$ for any $\boldsymbol{q}, \boldsymbol{p} \in \Delta_L$, and that $\mathrm{D}[\boldsymbol{q} \,\|\, \boldsymbol{p}] = 0$ only if $\boldsymbol{q} = \boldsymbol{p}$. This is the *information inequality* (or Gibbs inequality), one of the most powerful “inequality generators” in mathematics. It holds for general probability distributions, a general proof is found in [11].

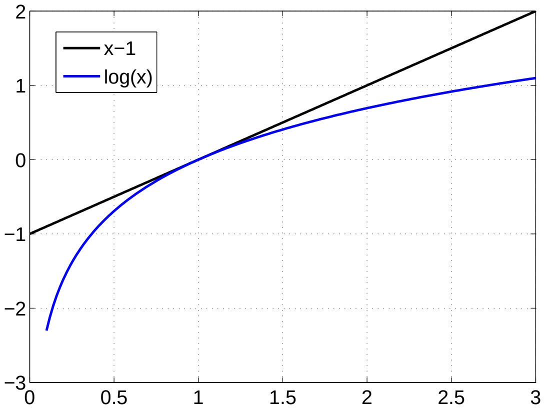


Figure 6.9: $\log(x)$ is upper bounded by $x-1$. This bound is used in order to prove the information (or Gibbs) inequality.

Let us prove the information inequality for discrete finite distributions $\boldsymbol{q}, \boldsymbol{p} \in \Delta_L$. The proof is based on the inequality $\log x \leq x - 1$, which holds for all $x > 0$ (Figure 6.9). Namely, if $f(x) = x - 1 - \log x$ (continuously differentiable), then $f(1) = 0$, $f'(x) = 1 - 1/x \geq 0$ if and only if $x \geq 1$, so that $f(x) \geq 0$ for all $x > 0$. In fact, $|f'(x)| > 0$ for $x \neq 1$, so that $f(x) = 0$ if and only if $x = 1$. If we define $\log 0 = -\infty$, then $\log x \leq x - 1$ holds for all $x \geq 0$. Given that,

$$-\mathrm{D}[\boldsymbol{q} \,\|\, \boldsymbol{p}] = \mathrm{E}_{\boldsymbol{q}}\left[\log \frac{p_l}{q_l}\right] \leq \mathrm{E}_{\boldsymbol{q}}\left[\frac{p_l}{q_l} - 1\right] = \mathrm{E}_{\boldsymbol{p}}[1] - \mathrm{E}_{\boldsymbol{q}}[1] = 0.$$

Here, $\mathrm{E}_{\boldsymbol{q}}[f(l)] = \sum_l q_l f(l)$ denotes expectation over $\boldsymbol{q}$. This proves the inequality. Now, suppose that $\mathrm{D}[\boldsymbol{q} \,\|\, \boldsymbol{p}] = 0$. We need to show that $q_l = p_l$ for all $l = 1, \dots, L$. Suppose that $q_{l_*} \neq p_{l_*}$ for some l_* with $q_{l_*} \neq 0$. Then, $\varepsilon = f(p_{l_*}/q_{l_*}) > 0$, and

$$-\mathrm{D}[\boldsymbol{q} \,\|\, \boldsymbol{p}] = \mathrm{E}_{\boldsymbol{q}}\left[\log \frac{p_l}{q_l}\right] \leq \mathrm{E}_{\boldsymbol{q}}\left[\frac{p_l}{q_l} - 1 - \varepsilon \mathrm{I}_{\{l=l_*\}}\right] = -\varepsilon q_{l_*} < 0,$$

a contradiction. Therefore, $p_l = q_l$ for all $q_l \neq 0$. Since both $\boldsymbol{q}$ and $\boldsymbol{p}$ sum to 1, we must have $p_l = q_l$ for all $l = 1, \dots, L$. This completes the proof.

Chapter 7

Generalization. Regularization

In this chapter, we introduce the concept of generalization and shed some light on the phenomenon of over-fitting, which can negatively affect estimation-based learning techniques. We will study regularization as a simple and frequently effective remedy against over-fitting. Finally, MAP estimation is introduced as general way of regularizing ML estimation, employing prior distributions over model parameters.

7.1 Generalization

The world around us is complex. And the closer we look, the more details we see. Arguably, for a model to stand any chance making interesting predictions, it ought to reflect this complexity: many variables, high-dimensional feature maps, a great depth of hidden layers separated by nonlinearities, so that training data can be fit with high precision. What could be wrong with that?

Certainly, there will be computational issues. For highly detailed models, maximum likelihood estimation can be a hard problem to solve. But leaving these issues aside, there is a fundamental problem: *generalization.* Understanding generalization is arguably the single most important lesson we will learn in this course. Without a solid understanding of this concept, there can be no valid statistics or useful machine learning. Its relevance goes far beyond statistics and machine learning, essentially governing all natural sciences: Occam's razor dictates that we should always favour the *simplest* model appropriate for our data, not the most complex one.

We start with some definitions which link back to decision theory (Section 5.2) and discriminants (Chapter 2). Suppose you are given some data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$, $t_i \in \{-1, +1\}$, and your goal is binary classification: finding a classifier $f(\boldsymbol{x})$, mapping to labels $\{-1, +1\}$, which predicts well. What does "predict well" mean? We can give several answers. After reading Chapter 2, you might say: $f(\boldsymbol{x})$ is a good classifier if it does well on the training data $\mathcal{D}$. In other words, its *training error*

$$\hat{R}_n(f) = \hat{R}_n(f; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \mathrm{I}_{\{f(\boldsymbol{x}_i) \neq t_i\}}$$

is small. For example, if $\mathcal{D}$ is linearly separable in some feature space we chose, then the perceptron algorithm outputs a linear classifier with $\hat{R}_n(f) = 0$. However, after reading about decision theory (Section 5.2), you might give a different answer. After all, there is the *i.i.d. assumption* (Section 6.1): the data $\mathcal{D}$ is drawn independently from some underlying "true" distribution with joint density $p^*(\boldsymbol{x}, t)$. If we knew this distribution, the optimal classifier would be the one which minimizes the *generalization error* (or *test error*)

$$R(f) = P^* \left\{ f(\boldsymbol{x}) \neq t \right\} = \mathrm{E}^* \left[\mathrm{I}_{\{f(\boldsymbol{x}) \neq t\}} \right].$$

Of course, we don't know $p^*(\boldsymbol{x}, t)$, we only know the data $\mathcal{D}$. But we could use $\mathcal{D}$ in order to learn about $p^*(\boldsymbol{x}, t)$, using any number of subtle ideas (an example is probabilistic modelling, leading to maximum likelihood plug-in rules; see Chapter 6), and this may well lead to a classifier $f(\boldsymbol{x})$ which does not minimize the training error $\hat{R}_n(f)$ well, but attains a small test error $R(f)$. *Training error $\hat{R}_n(f)$ and test error $R(f)$ are different numbers, which in extreme cases can have little relationship with each other.* It is perfectly possible to attain very small, even zero, training error and at the same time run up a large test error. This nightmare scenario for statistical machine learning is called *over-fitting*. This is all not very deep and fairly intuitive. But here is the interesting part. It is possible to predict under which circumstances over-fitting is likely to occur. Moreover, there are automatic techniques to guard against it, one of which we will study in this chapter.

As far as over-fitting is concerned, there is nothing special about the training error as a learning statistic. We will see that maximum likelihood estimation is equally affected, where the statistic to minimize is the negative log likelihood.

Before we look into over-fitting and what to do about it, let us clarify our goals. The correct answer above is the second: we wish to find a predictor with as small a *test error* as possible. The catch with this goal is that it is in general impossible to attain. We do not know $p^*(\boldsymbol{x}, t)$, but only have a finite dataset $\mathcal{D}$ drawn from it. The next best idea seems to select a classifier which minimizes the *training error*, a statistic we can compute on $\mathcal{D}$. This idea is a good one in general, it works well in many cases. Yet training error minimization has some problems which can make it fail poorly in certain relevant situations. Understanding and mediating some of these problems is the subject of this chapter.

7.1.1 Over-fitting

We have already encountered over-fitting at several places. In Section 4.1, polynomial curve fitting gave absurd, yet interpolating results for too high a polynomial degree. In Section 4.2.3, we noted potential difficulties when solving the normal equations of linear regression. In Section 6.4.1, we mentioned that maximum likelihood plug-in classification can run into trouble if Gaussian class-conditionals come with a full covariance matrix to be estimated. Finally, at the end of Section 6.5, we observed an extreme sensitivity of our naive Bayes document classifier to zero word counts. In this section, we expose the commonalities between these issues.

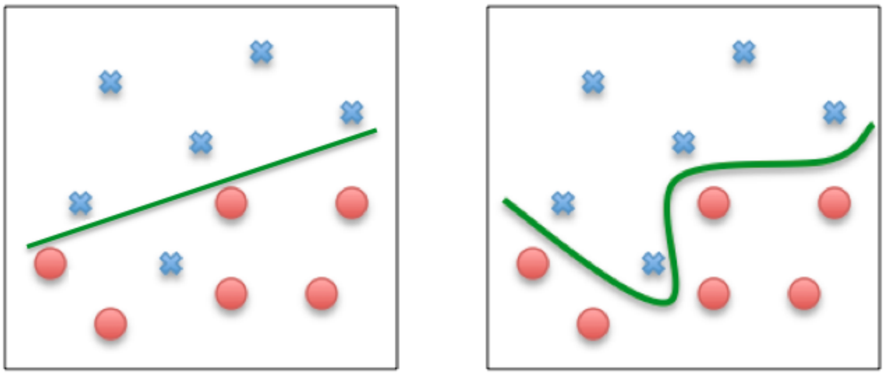


Figure 7.1: Example of over-fitting for binary classification. The simple linear discriminant on the left errs on a single pattern. In order to drive the traning error to zero, a more complex nonlinear discriminant is required (right). Given the limited amount of data, the latter solution is less likely to generalize well.

Over-fitting comes about due to a mismatch[1] between amount of training data on the one, choice of model parameterization and learning method on the other hand. It has several aspects. First, a certain model parameterization and learning procedure (for example, minimizing the training error for linear discriminants) may not result in a unique solution, at least in practice. This aspect is linked to *non-identifiability* (or *ill-posedness*) and to *ill-conditioning*. Non-identifiability is easy to understand. If the family of classifiers you learn with is so large than many different candidates attain the minimum training error (say, zero), then the training error alone remains silent about how to choose among them, its minimization does not identify a unique solution. Ill-conditioning is slightly more subtle. You should think about it as "non-identifiability about to happen". It is often closely linked to numerical inaccuracy. Examples below will make this clear. A second aspect of over-fitting is that often the best predictors in hindsight, which minimize the test error, are *not* among those which minimize the training error. Remember our discussion of curve fitting in Section 4.1. The data is a stochastic sample from the "true" distribution, its points are typically obscured by random noise. Solutions which minimize the training error are often those which fit the noise on top of the systematic signal an optimal predictor would uncover. The same comments apply to classification as well (Figure 7.1; see also Section 9.2.1).

Recall polynomial curve fitting from Section 4.1, an instance of linear regression. A polynomial $y(x) = w_0 + w_1 x + \dots + w_{p-1} x^{p-1}$ of degree $p-1$ is fit to n data points $\mathcal{D} = \{(x_i, t_i) \mid i = 1, \dots, n\}$ by way of minimizing the squared error.

[1] As the name suggests, over-fitting is contingent on the fact that a model is *fit* to data in the first place. In the *Bayesian statistics* approach to machine learning, over-fitting is ruled out up front, at least in principle. Bayesian machine learning will not feature much in this basic course, but see [30, 6].

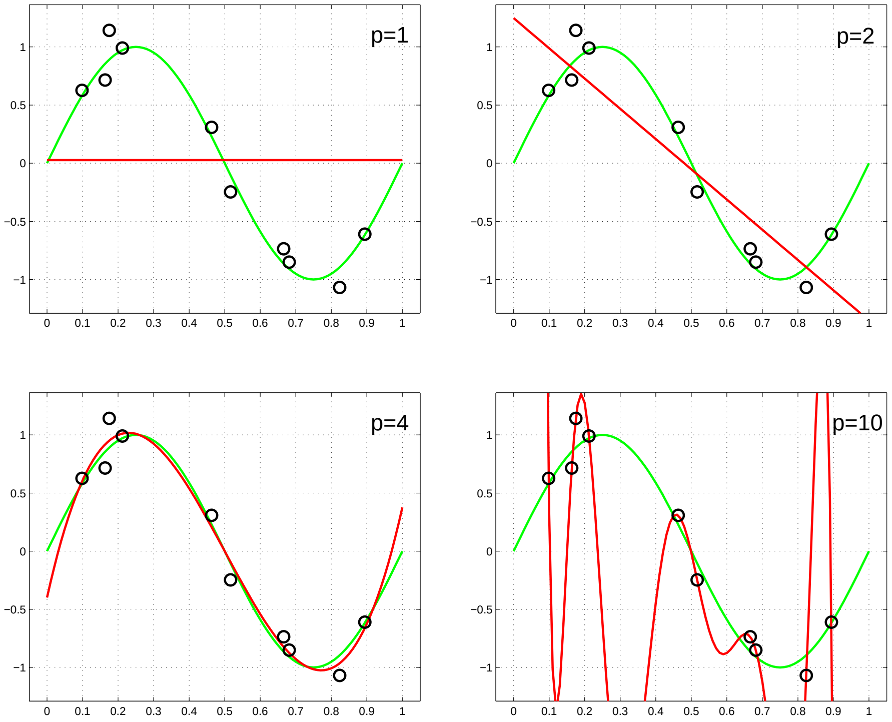


Figure 7.2: Linear regression estimation with polynomials of degree $p-1$. The generating curve (green) is $\sin(2\pi x)$, the noise is Gaussian with standard deviation 0.15.

In Figure 7.2, these least squares solutions are plotted for $n = 10$ data points and different numbers p of free parameters. The data comes from a smooth curve, yet the targets t_i are obscured by additive noise. In the presence of noise, a good predictor should refrain from interpolating the points. However, as p grows close to n, it is the interpolants which minimize the training error, no matter how erratic they behave elsewhere. For $p = n$, the training error drops to zero, all training points are fit exactly. For $p > n$, we face a non-identifiable problem: infinitely many polynomials interpolate the data with training error zero. However, even for $p \approx n$, $p \leq n$, the least squares solutions behave terribly. As noted in Section 4.2.3, this is due to *ill-conditioning*. For $p \leq n$, the design matrix $\boldsymbol{\Phi} \in \mathbb{R}^{n \times p}$ typically has full rank p, and the system matrix $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ of the normal equations is invertible. However, in particular for large p and n, some of its eigenvalues are very close to zero. Geometrically speaking, for some directions $\boldsymbol{d}$, $\boldsymbol{d}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{d} = \|\boldsymbol{\Phi}\boldsymbol{d}\|^2 \approx 0$. This means that our data remains silent about contributions of the weight vector along $\boldsymbol{d}$. Such matrices are called ill-conditioned, and solving systems with them tends to produce solutions with large coefficients, which in turn give rise to highly erratic polynomials. In short, over-fitting occurs for least squares polynomial regression if the data is noisy and the number of parameters p is overly close to n. We can do little about noise, but in Section 7.2 we will learn to know remedies against ill-conditioning and non-identifiability. In Figure 7.2, we can also observe the opposite problem of *under-fitting*. Clearly, constant ($p = 1$) or affine ($p = 2$) functions are insufficient to describe the systematic part of the data well. Given the data $\mathcal{D}$, how can we choose p so that fits are most likely to generalize well and avoid both over- and under-fitting? We will take up this *model selection* problem in Chapter 10. To summarize, the root of over-fitting in least squares polynomial regression is that for p of a certain size, the n data points remain nearly silent about certain directions $\boldsymbol{d}$, and the training error is minimized only by growing the weight vector dramatically along $\boldsymbol{d}$.

Our problems with the naive Bayes bag of words document classifier of Section 6.5 come from the unruly importance attached to word counts being zero (say, word c_m does not occur in data for class k). Dictionaries for natural language tasks grow rapidly with corpus size, and there are usually many classes. As many zero count events happen simply by chance, the infinite sensitivity attached to them is plain wrong. Once more, this is an over-fitting effect. For a large dictionary size M and many classes K, there are very many parameters $p_m^{(k)}$ and too little data to fit all of them by maximum likelihood. Small (but nonzero) probabilities in particular are not determined well by training data, but have a large effect on predictions. A simple remedy is called *Laplace smoothing*: add 1 to each count $N^{(k,m)}$ and use the modified (or "smoothed") counts to estimate the word probabilities. This mitigates the zero counts artefact of straight ML naive Bayes without any substantial negative influence.

Finally, recall the ML plug-in rule for Gaussian class-conditionals, where $\boldsymbol{x} \in \mathbb{R}^p$ (Section 6.4.1). Typically, class-conditional data has a pronounced covariance structure, which is not captured by spherical distributions $P(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t, \boldsymbol{I})$. This affects classification performance. If classes spread unequally in different directions, they will overlap more along certain direction than a spherical covariance fit would make us believe. General assumptions $P(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t, \boldsymbol{\Sigma})$ should improve things, but now we have to estimate a full covariance matrix $\boldsymbol{\Sigma}$ of about p^2 parameters from our data. If the training set size n is not much larger than p, plugging in the ML estimator $\hat{\boldsymbol{\Sigma}}$ (Section 6.4) can lead to poor classification performance. Once more, the culprits are directions of small variance which are underestimated in $\hat{\boldsymbol{\Sigma}}$, and which exercise a large effect on the final predictor.

To sum up, over-fitting happens in the absence of large enough training sets, given all our choices. If you can get more[2] data, do so by any means. However, there is a pattern in the examples discussed above. *Small* variations or probabilities are typically underestimated from limited data. With little room to move, they might even be set to zero (for example, a rare word c_m may occur zero times in training documents for some class k). These estimation effects happen by chance, since our data is a random sample. Nevertheless, they exert a very strong influence on the final predictor. Viewed this way, over-fitting is an artefact of learning methodology applied to small samples, and in the next section, we discuss a remedy. Beyond, over-fitting may come from non-optimal choice of model size and parameterization. In Chapter 10, we will learn about techniques to assess the suitability our our model choices, and ways to validate learned predictors.

[2] However, with more data, you might also want to explore more complex and realistic models. Over-fitting will not go away with the "data deluge".

7.2 Regularization

Simple learning techniques like training error minimization or maximum likelihood estimation can run into serious trouble, collectively termed over-fitting. Will we have to sacrifice their simple geometrical structure and efficient learning algorithms and do something else altogether? Will we have to painstakingly sift through data, identify smallish counts and treat them by hand-tuned heuristics? We don't. It turns out that with a simple modification of the standard techniques, we can alleviate some of the most serious over-fitting issues. Importantly, this modification does not add any computational complexity. In fact, many algorithms behave better and may converge faster in that case. This idea is called *regularization* (or *penalization*).

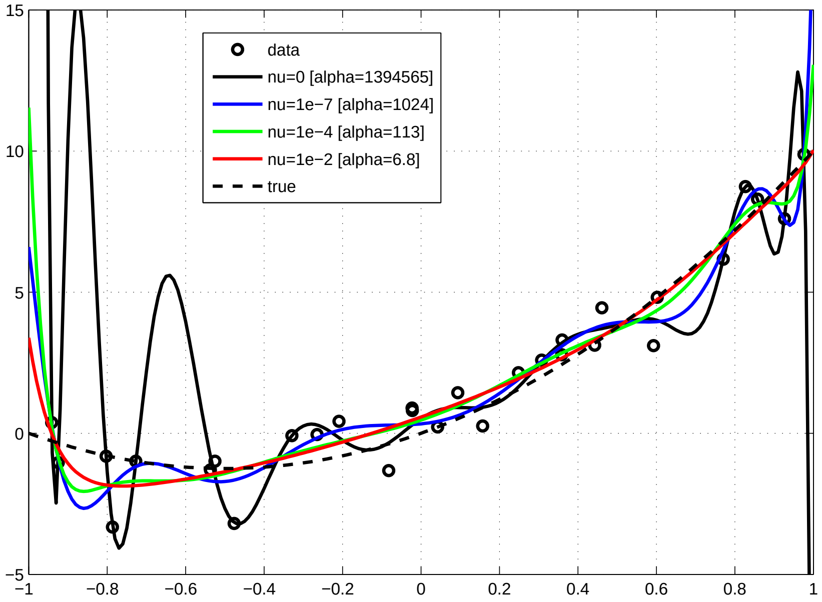


Figure 7.3: Regularized polynomial curve fitting. $n = 30$ data points were drawn from a smooth curve (dashed) plus Gaussian noise. Polynomials of degree $p-1$ are used for the fitting, where $p = 20$. The black curve is the standard least squares solution without regularization. The other curves are regularized least squares solutions with different regularization parameter values ν. The α values are $\alpha(\nu) = \|\hat{\boldsymbol{w}}_\nu\|$, where $\hat{\boldsymbol{w}}_\nu = \operatorname{argmin}_{\boldsymbol{w}} E_\nu(\boldsymbol{w})$. Notice the extreme size of $\alpha(0)$ for the non-regularized solution ($\nu = 0$). Its erratic behaviour is smoothed out in what amounts to better tradeoffs between data fit and curve complexity.

Recall how over-fitting manifests itself in polynomial curve fitting. Since some directions are almost unconstrained by the data, contributions of the weight vector $\boldsymbol{w}$ can become large along these. Not required to pay for it, least squares estimation uses these degrees of freedom in order to closely fit all training points. As these are noisy, large weights are required, and the least squares fit behaves very erratically elsewhere (Figure 7.3, black curve). A remedy is to make LS estimation pay for using large weights, no matter along which directions. We can do so by adding an extra penalty term $(\nu/2)\|\boldsymbol{w}\|^2$ to the squared error, where $\nu \geq 0$, ending up with a different criterion function

$$E_\nu(\boldsymbol{w}) = \underbrace{\frac{1}{2}\|\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\|^2}_{\text{error function}} + \underbrace{\frac{\nu}{2}\|\boldsymbol{w}\|^2}_{\text{regularization term}}. \tag{7.1}$$

The procedure of finding the weights $\boldsymbol{w}$ by minimizing $E_\nu(\boldsymbol{w})$ is known as (Tikhonov) *regularized least squares estimation* (or penalized least squares estimation). $E_\nu(\boldsymbol{w})$ is the sum of the usual squared error function and a second term, called *regularization term* (or regularizer, or penalization term, or penalizer). The constant $\nu \geq 0$ is known as *regularization constant.* In Figure 7.3, regularized least squares estimates $\hat{\boldsymbol{w}}_\nu^T \boldsymbol{\phi}(\boldsymbol{x})$ are shown for different values of ν, where $\hat{\boldsymbol{w}}_\nu = \operatorname{argmin}_{\boldsymbol{w}} E_\nu(\boldsymbol{w})$. Obviously, curves becomes smoother, the larger ν. This makes sense. The larger ν, the more price to pay (in terms of size of $E_\nu(\boldsymbol{w})$) for large weights. Therefore, $\|\hat{\boldsymbol{w}}_\nu\|$ will decrease as ν increases. On the other hand, the training squared error alone increases as ν increases. We establish these points rigorously in Section 7.2.2.

The regularized least squares problem implies modified normal equations:

$$\left(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \nu\boldsymbol{I}\right)\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{t}.$$

This is because the quadratic term $(1/2)\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w}$ in $E(\boldsymbol{w})$ is extended by adding $(\nu/2)\boldsymbol{w}^T\boldsymbol{w}$. Recalling Section 4.2.3, we see that regularization also improves the conditioning of this problem. As analyzed in Section 7.2.2, the effect of regularization is to add ν to all eigenvalues of $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$, in particular to lift the tiny ones at the lower end of the spectrum. Regularization does not only smooth out solutions to make them behave less erratically, regularized problems can also be solved more robustly in practice.

Moreover, our analysis in Section 7.2.2 shows that regularization tackles overfitting at its roots. Comparing the standard least squares solution $\hat{\boldsymbol{w}}_0$ with $\hat{\boldsymbol{w}}_\nu$, it is not just that $\|\hat{\boldsymbol{w}}_\nu\|$ is smaller than $\|\hat{\boldsymbol{w}}_0\|$. $\hat{\boldsymbol{w}}_\nu$ is shrunk compared to $\hat{\boldsymbol{w}}_0$ along all directions $\boldsymbol{d}$, but shrinkage is most pronounced along such directions $\boldsymbol{d}$ which are least determined by the data. The large effect of such poorly estimated directions on the least squares solution is diminished.

How do we choose the regularization parameter ν? In a way, this is just one more model choice, along with the number p of weights or aspects of the feature map $\boldsymbol{\phi}(\boldsymbol{x})$, and model selection techniques discussed in Chapter 10 can be applied. Since ν is a single parameter with an obvious effect on the smoothness of the prediction, another common approach in data analysis is to inspect curves for different values of ν and to choose it by hand.

To conclude, in order to regularize an estimation method which is based on minimizing an error function (for example the squared error for linear regression, or the training error or perceptron error function for a linear classifier), we add a regularization term which grows with the sizes of the weights. Due to the presence of this term, regularized estimation will prefer a solution with small weights (and therefore smooth functions) if it fits the training data equally well or even a little less well than another solution with large weights. No matter what the error function, by far the most commonly used regularizer is the squared

Euclidean norm $(\nu/2)\|\boldsymbol{w}\|^2$. If this term is used, we speak of *Tikhonov regularization*. The same idea is behind Wiener filtering in signal processing, ridge regression in statistics, and weight decay regularization in the neural networks literature.

Tricks of the trade of regularization in practice are numerous and out of the scope of this course. If you are working with multi-layer perceptrons, you should be aware of them. Both [6, ch. 5.5] and [5, ch. 9.2] give a good overview.

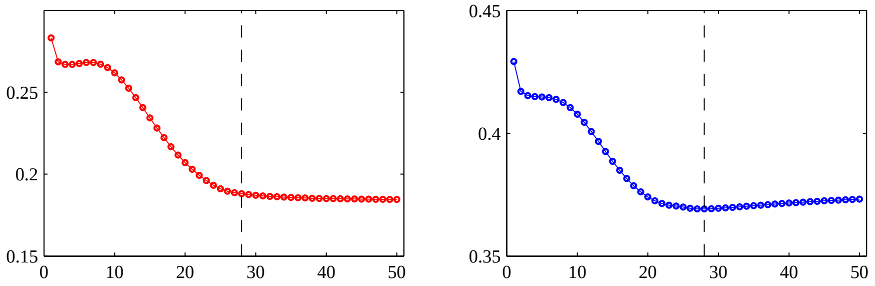


Figure 7.4: Illustration of over-fitting and early stopping. Shown are error on the training dataset (left) and on an independent validation dataset not used for training (right), for a MLP applied to a regression problem (details in [6], Figure 5.12), the horizontal unit is number of gradient descent iterations. The training error curve is monotonically decreasing as expected. In contrast, the validation error curve drops only up to a point, after which it increases. Early stopping corresponds to monitoring the error on a validation set, terminating MLP training once this statistic starts to increase.
Figure from [6] (used with permission).

7.2.1 Early Stopping

There are other techniques to keep over-fitting at bay. One simple technique, *early stopping*, is frequently used with multi-layer perceptrons or other neural network models. Early stopping is based on monitoring an estimate of the test error $R(f)$ alongside training (minimization of $\hat{R}_n(f)$). To do so, we *hold out* some of our data exclusively for the purpose of validation, split our data into a *training* set $\mathcal{D}_T$ and a *validation* set $\mathcal{D}_V$. Since the latter is never used for training, the empirical error $\hat{R}(\hat{f};\mathcal{D}_V)$ provides a reliable estimate of $R(\hat{f})$ even as $\hat{f}$ is fitted to $\mathcal{D}_T$. A typical MLP training run is shown in Figure 7.4. By definition, the training error $\hat{R}(\hat{f},\mathcal{D}_T)$ (left panel) decreases monotonically. In contrast, the validation error $\hat{R}(\hat{f};\mathcal{D}_V)$ does so only up to a point, after which it begins to increase. We can stop the MLP training early at this point, since any further decrease in training error does not lead to a decrease in $\hat{R}(\hat{f};\mathcal{D}_V)$.

Compared to regularization by adding a complexity penalty term(for example, a Tikhonov squared norm), early stopping has the advantage of not changing the standard training procedure at all, so existing code does not have to be modified. In contrast to penalization, whose success relies on a good choice of the regularization constant, early stopping is free of parameters. Moreover, it can sometimes speed up training. Its chief drawback is that a part of the data has to be sacrificed for validation and cannot be used to learn the weights. It is tempting to choose only a few points for $\mathcal{D}_V$, but in this case, $\hat{R}(\hat{f};\mathcal{D}_V)$ does not represent the test error $R(f)$ well enough, which defies the whole purpose. Typically, at least 20% of the data should be used for validation. Moreover, early stopping can be difficult to use and is hard to automatize. It can happen that $\hat{R}(\hat{f};\mathcal{D}_V)$ increases for some iterations, then continues to drop. Finally, notice that early stopping does not modify the training procedure as such, therefore does not help with ill-conditioning, apart from stopping at the first sign of things going wrong.

7.2.2 Regularized Least Squares Estimation (*)

In this section, we obtain insight into Tikhonov regularization for least squares estimation by working out some properties. We focus on linear regression (for example, polynomial curve fitting). The minimizer of the regularized squared error is given by

$$\hat{\boldsymbol{w}}_\nu = \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \nu\boldsymbol{I}\right)^{-1}\boldsymbol{\Phi}^T\boldsymbol{t}.$$

The main message is this. The norm of $\hat{\boldsymbol{w}}_\nu$ is shrunk towards zero as ν gets larger. This shrinkage does not happen uniformly, but $\hat{\boldsymbol{w}}_0$ (the standard least squares solution) is shrunk more along directions which are less well determined by training data. This shrinkage alleviates the erratic behaviour of the least squares estimate, it has to be paid for by an increase in the training set squared error.

Let us compare the standard least squares solution $\hat{\boldsymbol{w}}_0$ against the regularized $\hat{\boldsymbol{w}}_\nu$. To do so, we will expand the weight vectors in the eigenbasis of the relevant system matrix $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$. Let

$$\boldsymbol{\Phi}^T\boldsymbol{\Phi} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$$

the eigendecomposition of $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$. Here, $\boldsymbol{U} = [\boldsymbol{u}_j] \in \mathbb{R}^{p\times p}$ is an orthonormal matrix, $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$, whose columns $\boldsymbol{u}_j$ are the eigenvectors, and $\boldsymbol{\Lambda} = \operatorname{diag}[\lambda_j]$ is a diagonal matrix of the eigenvalues $\lambda_1 \le \dots \le \lambda_p$. If you are feeling lost about all things eigen, you might want to skip to Section 11.1.2, or otherwise return to the present section at a later time. Now, $\{\boldsymbol{u}_j\}$ is an orthonormal basis of $\mathbb{R}^p$, so we can represent each $\hat{\boldsymbol{w}}_\nu$ as a linear combination of the eigenvectors:

$$\hat{\boldsymbol{w}}_\nu = \sum_{j=1}^{p}\beta_{\nu,j}\boldsymbol{u}_j, \quad \hat{y}_\nu(\boldsymbol{x}) = \hat{\boldsymbol{w}}_\nu^T\boldsymbol{\phi}(\boldsymbol{x}) = \sum_{j=1}^{p}\beta_{\nu,j}\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{u}_j.$$

The predictor $\hat{y}_\nu(\boldsymbol{x})$ is a weighted sum of the basis functions $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{u}_j$, each being the inner product between the feature map $\boldsymbol{\phi}(\boldsymbol{x})$ and the eigendirection $\boldsymbol{u}_j$. At least for large p, these basis functions differ dramatically when we evaluate them at the training points $\{\boldsymbol{x}_i\}$ only. Namely,

$$\lambda_j = \boldsymbol{u}_j^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{u}_j = \|\boldsymbol{\Phi}\boldsymbol{u}_j\|^2.$$

For large p, the smallest eigenvalue λ_1 is typically very close to zero, so that $\boldsymbol{\Phi}\boldsymbol{u}_1 \approx \boldsymbol{0}$, therefore $\boldsymbol{\phi}(\boldsymbol{x}_i)^T\boldsymbol{u}_1 \approx 0$ for all $i = 1,\dots,n$. In contrast, the basis function $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{u}_p$ for the largest eigenvalue makes sizeable contributions at the training points. Importantly, these differences manifest themselves at the training points only: as a polynomial with weights of unit norm, $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{u}_1$ is of healthy size elsewhere.

What does this mean? If $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{u}_1$ is to contribute significantly to the fit of the training data, then $\beta_{0,1}$ has to be very large, which is precisely what happens for the least squares solution $\hat{\boldsymbol{w}}_0$, in its effort to minimize the training error. But then, $\hat{y}_0(\boldsymbol{x})$ sports a very large component $\beta_{0,1}\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{u}_1$, which explains its erratic behaviour away from the training data. To understand what regularization does about this, we need to work out the $\beta_{\nu,j}$ coefficients.

$$\begin{aligned}\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \nu\boldsymbol{I} &= \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U} + \nu\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{U}\left(\boldsymbol{\Lambda} + \nu\boldsymbol{I}\right)\boldsymbol{U}^T,\\ \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \nu\boldsymbol{I}\right)^{-1} &= \boldsymbol{U}\left(\boldsymbol{\Lambda} + \nu\boldsymbol{I}\right)^{-1}\boldsymbol{U}^T\end{aligned}$$

and

$$\begin{aligned}\hat{\boldsymbol{w}}_\nu &= \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \nu\boldsymbol{I}\right)^{-1}\boldsymbol{\Phi}^T\boldsymbol{t} = \boldsymbol{U}\left(\boldsymbol{\Lambda} + \nu\boldsymbol{I}\right)^{-1}\tilde{\boldsymbol{y}} = \sum_{j=1}^n \frac{\tilde{y}_j}{\lambda_j + \nu}\boldsymbol{u}_j,\\ \tilde{\boldsymbol{y}} &= \boldsymbol{U}^T\boldsymbol{\Phi}^T\boldsymbol{t}.\end{aligned}$$

Therefore,

$$\beta_{0,j} = \frac{\tilde{y}_j}{\lambda_j}, \qquad \beta_{\nu,j} = \frac{\tilde{y}_j}{\lambda_j + \nu} = \frac{\lambda_j}{\lambda_j + \nu}\beta_{0,j}.$$

Regularization with $\nu > 0$ transforms $\beta_{0,j}$ (standard least squares) to $\beta_{\nu,j}$. Obviously, $|\beta_{\nu,j}| < |\beta_{0,j}|$, but the amount of shrinkage depends strongly on λ_j:

$$\frac{\lambda_j}{\lambda_j + \nu} = \left\{\begin{array}{ll} \approx 1 & \mid \lambda_j \gg \nu \text{ (well determined by data)} \\ \ll 1 & \mid \lambda_j \ll \nu \text{ (poorly determined by data)} \end{array}\right\}.$$

Therefore, regularization precisely counteracts the erratic behaviour of the standard least squares solution. Put differently, $\beta_{0,j}$ is inversely proportional to λ_j, explaining the very large contributions of the smallest eigendirections to $\hat{\boldsymbol{w}}_0$ and $\hat{y}_0(\boldsymbol{x})$. Regularization exchanges the denominator by $\lambda_j + \nu$, uniformly limiting the influence of these poorly determined directions. In contrast, for well determined directions with $\lambda_j \gg \nu$, regularization hardly makes a difference.

The expansion of $\hat{\boldsymbol{w}}_\nu$ in the orthonormal eigendirections $\boldsymbol{u}_j$ implies that for $0 \le \nu_1 < \nu_2$:

$$\|\hat{\boldsymbol{w}}_{\nu_2}\|^2 = \sum_{j=1}^n \beta_{\nu_2,j}^2 = \sum_{j=1}^n \frac{\tilde{y}_j^2}{(\lambda_j + \nu_2)^2} < \sum_{j=1}^n \frac{\tilde{y}_j^2}{(\lambda_j + \nu_1)^2} = \sum_{j=1}^n \beta_{\nu_1,j}^2 = \|\hat{\boldsymbol{w}}_{\nu_1}\|^2.$$

The first equality uses the orthonormality: $\boldsymbol{u}_i^T\boldsymbol{u}_j = \mathrm{I}_{\{i=j\}}$. On the other hand, the squared training error increases with ν:

$$\|\boldsymbol{\Phi}\hat{\boldsymbol{w}}_{\nu_2} - \boldsymbol{t}\|^2 > \|\boldsymbol{\Phi}\hat{\boldsymbol{w}}_{\nu_1} - \boldsymbol{t}\|^2.$$

To establish this, assume that the opposite holds. Then,

$$\begin{aligned}E_{\nu_1}(\hat{\boldsymbol{w}}_{\nu_2}) &= E_0(\hat{\boldsymbol{w}}_{\nu_2}) + \frac{\nu_1}{2}\|\hat{\boldsymbol{w}}_{\nu_2}\|^2 < E_0(\hat{\boldsymbol{w}}_{\nu_2}) + \frac{\nu_2}{2}\|\hat{\boldsymbol{w}}_{\nu_1}\|^2\\ &\le E_0(\hat{\boldsymbol{w}}_{\nu_1}) + \frac{\nu_2}{2}\|\hat{\boldsymbol{w}}_{\nu_1}\|^2 = E_{\nu_1}(\hat{\boldsymbol{w}}_{\nu_1}),\end{aligned}$$

which contradicts the optimality of $\hat{\boldsymbol{w}}_{\nu_1}$. To conclude, a stronger penalty term leads to strictly smaller weights and a smoother predictor $\hat{y}_\nu(\boldsymbol{x})$, but also implies a worse training set fit.

7.3 Maximum A-Posteriori Estimation

Regularization can help to alleviate over-fitting problems which can affect linear regression and classification. How does this principle generalize to maximum likelihood estimation? We could add a Tikhonov regularizer to the negative log likelihood and see what we get. Apart from being unfounded, this approach creates several problems. First, a quadratic penalizer does not make much sense on probability distributions or covariance matrices. Second, the simple and elegant MLE solutions (ratios of counts, empirical covariance) do not carry over to such a modification, and we would have to solve tedious optimization problems. In this section, we introduce a probabilistic viewpoint on regularization of ML estimation, which not only helps to construct regularizers in an informed way, but whose modified optimization problems are typically solved in much the same way as their MLE counterparts. This framework is called *maximum a-posteriori (MAP) estimation*, and regularizers correspond to negative log *prior distributions* over parameters.

Recall the first ML estimation example we came across earlier, featuring the thumbtack you found in your drawer (Section 6.2). Curious about the probability of landing point up, $p_1 = P\{x = 1\}$, you could use ML estimation: throw it $n = 100$ times, collect data $\mathcal{D} = \{x_1, \dots, x_{100}\}$, and maximize the likelihood function

$$P(\mathcal{D}|p_1) = (p_1)^{n_1}(1 - p_1)^{n - n_1}, \quad n_1 = \sum_{i=1}^{n} x_i,$$

resulting in the ML estimator $\hat{p}_1 = n_1/n$. However, having enjoyed some training in physics, you take a good look at the thumbtack and convince yourself that its shape implies that $p_1 > 1/2$ is substantially more probable than $p_1 < 1/2$ ("if not, I will eat my hat"). It is not that you dare to pin down the value p_1 from first principles. But you are more or less certain about some properties of p_1. You could formulate your thoughts in a probability distribution over p_1.

Wait a second. "More probable", "probability distribution"? p_1 is not random, it's just a parameter we don't know. Recall our introduction of probability in Section 5.1. We do not care whether p_1 is "random" or just an unknown parameter (whatever this distinction may mean). We are *uncertain* about it, and that is enough. *We encode our uncertainty in the precise value of p_1 by treating it as random variable.* In other words, we can maintain a distribution over p_1 in the same way as we maintain one over x. The latter is a conditional distribution $P(x|p_1)$. If you plug in data $\mathcal{D}$, this becomes the *likelihood* $P(\mathcal{D}|p_1) = \prod_i P(x_i|p_1)$. The former[3] is $p(p_1)$, called *prior distribution* over p_1.

If this vocabulary reminds you of decision theory (Section 5.2), you are on the right track. Back there, we predict a class label t from an observed input point $\boldsymbol{x}$ as follows. We know the class-conditional distributions $p(\boldsymbol{x}|t)$ and the class

[3] $p(p_1)$ is a density, since $p_1 \in [0, 1]$ is continuous.

prior $P(t)$. We determine the class posterior $P(t|\boldsymbol{x}) = p(\boldsymbol{x}|t)P(t)/p(x)$ by Bayes' formula, then predict $f^*(\boldsymbol{x}) = \text{argmax}_t\, P(t|\boldsymbol{x})$. The normalization by $p(x)$ does not matter, so that $f^*(\boldsymbol{x}) = \text{argmax}_t\, p(\boldsymbol{x}|t)P(t)$. Here, we observe data $\mathcal{D}$ and want to predict the probability p_1. We determine the *posterior distribution*

$$p(p_1|\mathcal{D}) = \frac{P(\mathcal{D}|p_1)p(p_1)}{P(\mathcal{D})}, \quad P(\mathcal{D}) = \int P(\mathcal{D}|p_1')p(p_1')\, dp_1'$$

and predict (or estimate)

$$\hat{p}_1^{\text{MAP}} = \underset{p_1\in[0,1]}{\text{argmax}}\, p(p_1|\mathcal{D}) = \underset{p_1\in[0,1]}{\text{argmax}}\, \{P(\mathcal{D}|p_1)p(p_1)\},$$

the maximum point of the posterior, the *maximum a-posteriori (MAP) estimator*. How does this differ from ML estimation? As usual, we minimize the negative log:

$$-\log\{P(\mathcal{D}|p_1)p(p_1)\} = \underbrace{-\log P(\mathcal{D}|p_1)}_{\text{error function}} + \underbrace{-\log p(p_1)}_{\text{regularization term}}.$$

The MAP criterion is the sum of the negative log likelihood and the negative log prior. The first quantifies data fit and plays the role of an error function. The second is a regularizer. MAP estimation is a form of regularized estimation, obtained by choosing the negative log prior as regularizer.

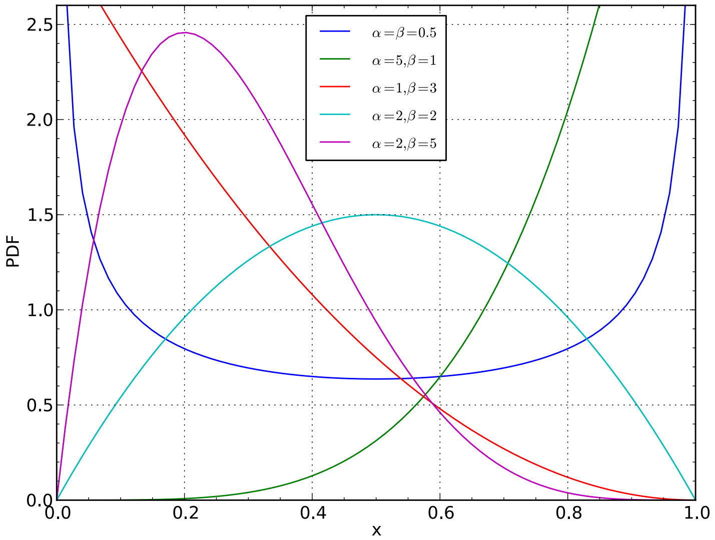


Figure 7.5: Density functions of the Beta distribution Beta(α, β) for different values (α, β).
Figures from `wikipedia` (used with permission). Copyright by Krishnavedala, Creative Commons Attribution-Share Alike 3.0 Unported license.

What is a good prior $p(p_1)$? In principle, we can choose any distribution we like. The choice of a prior is simply a part of the choice of a probabilistic model. But certain families of models are easier to work with than others, and the same holds for prior distributions. For our thumbtack example, a prior from the *Beta* family $\mathrm{Beta}(\alpha, \beta)$ is most convenient:

$$p(p_1|\alpha,\beta) = \frac{1}{B(\alpha,\beta)}(p_1)^{\alpha-1}(1-p_1)^{\beta-1}\mathrm{I}_{\{p_1\in[0,1]\}}, \quad B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)},$$
$$\alpha,\beta > 0.$$

Here, $\Gamma(x)$ is Euler's Gamma function (Section 6.3.2). Before we move on, a general hint about working with probability distributions. *Never write, or even try to remember, the normalization constants.* What matters is the part depending on p_1. Memorize $p(p_1|\alpha,\beta) \propto (p_1)^{\alpha-1}(1-p_1)^{\beta-1}$ as $\mathrm{Beta}(\alpha,\beta)$ density. Here, $A \propto B$ reads "*A proportional to B*", or $A = CB$ for some constant $C > 0$. If you need the normalization constant at all (you usually do not), you can look it up. In particular, keeping normalization constants up to date during a derivation of a posterior is a waste of time and an unnecessary source of mistakes.

Back to $\mathrm{Beta}(\alpha,\beta)$. It is a distribution over a probability $p_1 \in [0,1]$. Its mean is $\mathrm{E}[p_1] = \alpha/(\alpha+\beta)$. Density functions for a range of (α,β) values are shown in Figure 7.5. The larger the sum $\alpha+\beta$, the more peaked the distribution. The density is bounded above only if $\alpha \geq 1$, $\beta \geq 1$. $\mathrm{Beta}(1,1)$ is a special case you already know: the uniform distribution over $[0,1]$. For $\alpha > 1$, $\beta > 1$, the unique mode (maximum point) of the density is $(\alpha-1)/(\alpha+\beta-2)$, which is always further away from $1/2$ than the mean, unless $\alpha = \beta$. $\mathrm{Beta}(\alpha,\beta)$ is symmetric around $1/2$ if and only if $\alpha = \beta$. What is special about this family? You may have noticed the similarity between $p(p_1|\alpha,\beta)$ and the likelihood $P(\mathcal{D}|p_1)$. Indeed,

$$\begin{aligned}P(\mathcal{D}|p_1)p(p_1|\alpha,\beta) &\propto (p_1)^{n_1}(1-p_1)^{n-n_1} \times (p_1)^{\alpha-1}(1-p_1)^{\beta-1} \\ &\propto (p_1)^{\alpha+n_1-1}(1-p_1)^{\beta+n-n_1-1}.\end{aligned}$$

In our first exercise in dropping normalization constants, we directly conclude that the posterior $p(p_1|\mathcal{D})$ must be $\mathrm{Beta}(\alpha+n_1, \beta+n-n_1)$. That is because $p(p_1|\mathcal{D}) \propto (p_1)^{\alpha+n_1-1}(1-p_1)^{\beta+n-n_1-1}$, and there is only one density of this form (of course, we have to check that both $\alpha+n_1$ and $\beta+n-n_1$ are positive). The posterior is Beta again! All the data is doing is changing the constants to $\alpha_n = \alpha+n_1$, $\beta_n = \beta+n-n_1$. Since we know what the mode of a Beta density is, we can look up the MAP estimator:

$$\hat{p}_1^{\mathrm{MAP}} = \frac{\alpha+n_1-1}{\alpha+\beta+n-2}.$$

This can be seen as convex combination between the ML estimate $\hat{p}_1$ and the MAP estimate based on the prior $p(p_1)$ alone:

$$\begin{aligned}\hat{p}_1^{\mathrm{MAP}} &= \frac{\alpha+\beta-2}{\alpha+\beta-2+n}\cdot\frac{\alpha-1}{\alpha+\beta-2} + \frac{n}{\alpha+\beta-2+n}\cdot\frac{n_1}{n} \\ &= \kappa\cdot\frac{\alpha-1}{\alpha+\beta-2} + (1-\kappa)\cdot\hat{p}_1, \quad \kappa = \frac{\alpha+\beta-2}{\alpha+\beta-2+n}.\end{aligned}$$

If the sample size n is not too large compared to $\alpha+\beta-2$, the estimator is pulled away from $\hat{p}_1$ towards the prior mode. For example, to incorporate your insight that p_1 should rather be $> 1/2$, you would specify $\alpha > \beta$, and $\hat{p}_1$ is shifted towards the prior mode $> 1/2$.

Figuring out thumbtack probabilities is just a simple example of what turns out to be a very useful framework. It is often much easier to formulate one's prior belief about a parameter in a distribution then to come up with a useful regularizer from scratch. Examples are given in Section 7.3.1, where we rederive Laplace smoothing as MAP estimation and show how to regularize the MLE for a Gaussian covariance matrix.

ML estimation can often be interpreted as MAP estimation with a particular prior. In the thumbtack example, MLE corresponds to MAP estimation with prior Beta(1, 1): $p(p_1) = \mathrm{I}_{\{p_1 \in [0,1]\}}$, the uniform distribution. This is a fairly benign assumption, and MLE is unproblematic in this case. But in other cases (Section 7.3.1), MLE's lack of a prior runs the estimation method into serious trouble, and MAP regularization can save the day.

Final Comments for Curious Readers (*)

Finally, some more advanced comments, which can be skipped at first reading. ML estimation was motivated by treating the likelihood $P(\mathcal{D}|p_1)$ as a scoring function to assess data fit, whose maximization w.r.t. p_1 intuitively makes sense. However, the posterior $p(p_1|\mathcal{D})$ is a *distribution* over p_1. Why should the posterior be particularly well represented by its mode? Why not for example the mean $\mathrm{E}[p_1 \,|\, \mathcal{D}]$? Our analogy with decision theory is useful, in that it motivates the role of the prior $p(p_1)$, but it is not perfect. p_1 is not a class label from a finite set, but a continuous probability. Why not output all of $p(p_1|\mathcal{D})$ as result, given that it is just a Beta distribution with two parameters? These questions are resolved in the Bayesian approach to machine learning, the ultimately correct way to implement decision theory from finite data, whereas ML or MAP estimation are computationally attractive shortcuts. Bayesian machine learning will not feature much in this course, but [30, 6] provide good introductions.

If MAP corresponds to regularized ML estimation, then Beta(α, β) implies the following regularizer:

$$\begin{aligned}
&-\log p(p_1|\alpha,\beta) = -(\alpha-1)\log p_1 - (\beta-1)\log(1-p_1) \\
&= (\alpha+\beta-2)\left\{-q_1\log p_1 - (1-q_1)\log(1-p_1)\right\}, \quad q_1 = \frac{\alpha-1}{\alpha+\beta-2},
\end{aligned}$$

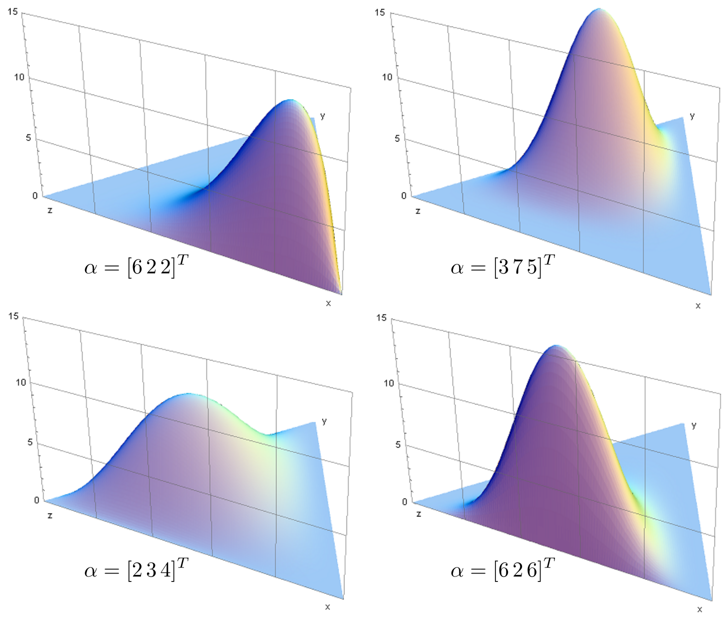
dropping an additive constant. Here, q_1 is the mode of $p(p_1|\alpha,\beta)$. This is certainly not a quadratic function of p_1. What is it then? The attentive reader may remember the pattern from Section 6.5.3, which allows us to write

$$-\log p(p_1|\alpha,\beta) = (\alpha+\beta-2)\mathrm{D}[(q_1, 1-q_1) \,\|\, (p_1, 1-p_1)],$$

dropping another additive constant. We can interpret $\alpha+\beta-2$ as regularization constant. The larger $\alpha+\beta-2$, the more peaked the regularizer is around the mode q_1. The special role of this constant for the convex combination of $\hat{p}_1^{\mathrm{MAP}}$ from q_1 and the MLE $\hat{p}_1$ becomes apparent now. Second, the regularization term is simply the relative entropy between the prior mode q_1 and p_1, viewed as binary distributions.

7.3.1 Examples of Conjugate Prior Distributions (*)

In this section, we provide further examples of MAP estimation. We focus on models where ML estimation is prone to over-fitting (Section 7.1), and MAP estimation can be seen as a particular form of regularization. Recall the thumbtack example. Using a Beta prior distribution proved convenient there: no matter what the data, the posterior is Beta again. Such prior distributions are called *conjugate* (for the likelihood), and the examples in this section are of this type as well.



$\alpha = [6\,2\,2]^T$ $\alpha = [3\,7\,5]^T$

$\alpha = [2\,3\,4]^T$ $\alpha = [6\,2\,6]^T$

Figure 7.6: Density plots of Dirichlet distributions over the probability simplex Δ_3, with corners $x = [1,0,0]$, $y = [0,1,0]$, $z = [0,0,1]$. Note how $\sum_m \alpha_m$ determines the concentration.
Figure from `wikipedia` (used with permission). Released into public domain by en:User:ThG (no license).

Recall our naive Bayes document classifier from Section 6.5. Its over-fitting issues have been discussed in Section 7.1.1, where we noted a surprisingly simple and widely used fix: *Laplace smoothing*, add 1 to each count and move on. Compare this to MAP estimation for the thumbtack example, where we added α and β to the counts n_1 and $n - n_1$. Maybe Laplace smoothing is MAP estimation with a prior much like Beta? Abstracting from naive Bayes details, suppose that

$$\boldsymbol{p} \in \Delta_M = \left\{ \boldsymbol{q} \;\middle|\; q_m \geq 0,\ m = 1, \dots, M,\ \ \sum\nolimits_{m=1}^{M} q_m = 1 \right\}.$$

Δ_M is the M-dimensional probability simplex (Section 6.5). What the Beta distribution is for Δ_2, the *Dirichlet distribution* is for general Δ_M:

$$\mathrm{Dir}(\boldsymbol{p}|\boldsymbol{\alpha}) \propto \prod_{m=1}^{M} (p_m)^{\alpha_m - 1} \mathrm{I}_{\{\boldsymbol{p}\in\Delta_M\}}, \quad \alpha_m > 0,\ m = 1, \dots, M.$$

The normalization constant is not important for our purposes here. Densities for different Dirichlet distributions over Δ_3 are shown in Figure 7.6. Its properties parallel those of the Beta. The mean is $\mathrm{E}[\boldsymbol{p}|\boldsymbol{\alpha}] = \boldsymbol{\alpha}/(\boldsymbol{1}^T\boldsymbol{\alpha})$, and $\boldsymbol{1}^T\boldsymbol{\alpha} = \sum_{m=1}^{M}\alpha_m$ determines the concentration of the density. $\mathrm{Dir}(\boldsymbol{p}|\boldsymbol{1})$ is the uniform distribution over Δ_M. If any $\alpha_m < 1$, the density grows unboundedly as $\boldsymbol{p} \to \boldsymbol{\delta}_m$. If all $\alpha_m > 1$, the density has a unique mode at $(\boldsymbol{\alpha} - \boldsymbol{1})/(\boldsymbol{1}^T\boldsymbol{\alpha} - M)$. Most importantly, the Dirichlet family is conjugate for the likelihood $P(\mathcal{D}|\boldsymbol{p}) = \prod_{m=1}^{M}(p_m)^{N^{(m)}}$ of i.i.d. data sampled from $\boldsymbol{p}$. If the prior is $p(\boldsymbol{p}) = \mathrm{Dir}(\boldsymbol{p}|\boldsymbol{\alpha})$, then the posterior

$$p(\boldsymbol{p}|\mathcal{D}) \propto \prod_{m=1}^{M} (p_m)^{N^{(m)}+\alpha_m-1} \mathrm{I}_{\{\boldsymbol{p}\in\Delta_M\}} \propto \mathrm{Dir}(\boldsymbol{p}|\boldsymbol{\alpha} + [N^{(m)}])$$

is Dirichlet again, with counts $[N^{(m)}]$ added to $\boldsymbol{\alpha}$. If $N = \sum_m N^{(m)}$ is the sum of counts, the concentration grows by N. The MAP estimator is

$$\hat{\boldsymbol{p}}^{\mathrm{MAP}} = \frac{1}{\boldsymbol{1}^T\boldsymbol{\alpha} + N - M}\left[N^{(m)} + \alpha_m - 1\right].$$

This is the same as the ML estimator if $\boldsymbol{\alpha} = \boldsymbol{1}$, the uniform distribution over Δ_M. On the other hand, Laplace smoothing is obtained by setting $\boldsymbol{\alpha} = 2\boldsymbol{1}$. Compared to the MLE, the MAP estimator is shrunk towards $\boldsymbol{1}/M \in \Delta_M$, which alleviates over-fitting issues due to zero counts. Variants of Laplace smoothing used in practice correspond to MAP estimation with different $\boldsymbol{\alpha}$, which may even be learned from data.

The Wishart Distribution (*)

Our second example is concerned with covariance estimation. To keep things simple, assume that data $\boldsymbol{x}_i \in \mathbb{R}^p$ is drawn i.i.d. from $N(\boldsymbol{0}, \boldsymbol{\Sigma})$, and we wish to estimate the covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{p\times p}$. The ML estimator is

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i\boldsymbol{x}_i^T.$$

Note that this lacks the $-\bar{\boldsymbol{x}}\bar{\boldsymbol{x}}^T$ term of the standard sample covariance, since the mean is fixed to zero here. If p is comparable to the sample size n, then $\hat{\boldsymbol{\Sigma}}$ is ill-conditioned, and the ML plug-in classifier is adversely affected (Section 7.1.1). One remedy is to use MAP estimation instead, with a Wishart prior distribution. Let us focus on the precision matrix $\boldsymbol{P} = \boldsymbol{\Sigma}^{-1}$. The likelihood is

$$p(\mathcal{D}|\boldsymbol{P}) \propto |\boldsymbol{P}|^{n/2} e^{-\frac{1}{2}\sum_{i=1}^{n}\boldsymbol{x}_i^T\boldsymbol{P}\boldsymbol{x}_i} = |\boldsymbol{P}|^{n/2} e^{-\frac{n}{2}\operatorname{tr}\hat{\boldsymbol{\Sigma}}\boldsymbol{P}}.$$

Here, we used manipulations involving the trace which were introduced in Section 6.4.3. The *Wishart* family contains distributions over symmetric positive definite matrices $\boldsymbol{P} \succ \boldsymbol{0}$ (Section 6.3). The Wishart distribution with $\alpha > p-1$ degrees of freedom and mean $\boldsymbol{V} \succ \boldsymbol{0}$ has the density

$$W(\boldsymbol{P}|\boldsymbol{V},\alpha) \propto |\boldsymbol{P}|^{(\alpha-p-1)/2} e^{-\frac{\alpha}{2}\operatorname{tr}\boldsymbol{V}^{-1}\boldsymbol{P}} \mathrm{I}_{\{\boldsymbol{P}\succ\boldsymbol{0}\}}.$$

This is of course conjugate to our Gaussian likelihood for the precision matrix. For the Wishart prior $p(\boldsymbol{P}) = W(\boldsymbol{I},\alpha)$ with mean $\boldsymbol{I}$, the posterior is

$$\begin{aligned} p(\boldsymbol{P}|\mathcal{D}) &\propto |\boldsymbol{P}|^{(\alpha+n-p-1)/2} e^{-\frac{\alpha}{2}\operatorname{tr}\boldsymbol{P} - \frac{n}{2}\operatorname{tr}\hat{\boldsymbol{\Sigma}}\boldsymbol{P}} \\ &\propto |\boldsymbol{P}|^{(\alpha+n-p-1)/2} e^{-\frac{\alpha+n}{2}\operatorname{tr}\left(\frac{\alpha}{\alpha+n}\boldsymbol{I} + \frac{n}{\alpha+n}\hat{\boldsymbol{\Sigma}}\right)\boldsymbol{P}}, \end{aligned}$$

Wishart with $\alpha+n$ degrees of freedom and mean $(\alpha\boldsymbol{I} + n\hat{\boldsymbol{\Sigma}})/(\alpha+n)$. Its mode is the inverse of

$$\hat{\boldsymbol{\Sigma}}^{\mathrm{MAP}} = \frac{1}{\alpha+n-p-1}\left(\alpha\boldsymbol{I} + n\hat{\boldsymbol{\Sigma}}\right),$$

derived as in Section 6.4.3. If $\lambda_1 \le \cdots \le \lambda_p$ are the eigenvalues of $\hat{\boldsymbol{\Sigma}}$, then $\hat{\boldsymbol{\Sigma}}^{\mathrm{MAP}}$ has the same eigenvectors, but eigenvalues

$$\lambda_j^{\mathrm{MAP}} = \frac{\alpha + n\lambda_j}{\alpha+n-p-1}.$$

In particular, $\lambda_1^{\mathrm{MAP}} \ge \alpha/(\alpha+n-p-1)$. Eigenvalues are bounded away from zero, alleviating the over-fitting problems of the plug-in classifier. To conclude, the MAP estimator with a Wishart prior with mean $\boldsymbol{I}$ is obtained from the standard sample covariance matrix by regularizing the eigenspectrum. Such *spectral regularizers* are widely used in machine learning.

Chapter 8

Conditional Likelihood. Logistic Regression

The machine learning methods we encountered so far can be classified into two different groups. We can pick some error function and function class, then optimize for the error-minimizing hypothesis within the class. In order to alleviate over-fitting problems, the error function can be augmented by a regularization term. Examples for this approach include perceptron classification, least squares estimation or multi-layer perceptron learning. Alternatively, we can capture our uncertainty in relevant variables by postulating a probabilistic model, then use probability calculus for prediction or decision making. The most important ideas in this context are maximum likelihood estimation (Chapter 6) and maximum a-posteriori estimation (Section 7.3). Both paradigms come with strenghts and weaknesses. Probabilistic modelling is firmly grounded on decision theory (Section 5.2), and it is more "user-friendly". As noted in Section 5.1, it is natural for us to formulate our ideas about a setup in terms of probabilities and conditional independencies among them, while constructing some regularizer or feature map from scratch is unintuitive business. On the other hand, a probabilistic model forces us to specify each and every relationship between variables in a sound way, which can be more laborious than fitting some classifier to the data.

In this chapter, we will work out a model-based perspective on methodology in the first group, thereby unifying the paradigms under the umbrella of probabilistic modelling. To this end, we will introduce discriminative models and conditional maximum likelihood estimation, providing a common basis for linear classification and regression, least squares estimation, and training MLPs. We will understand what the squared error means in probabilistic terms and devise alternative error functions which can work better in practice. Discriminative modelling does for error function minimization what generative modelling does for data analysis. It provides a natural and automatic route from prior knowledge about the structure of a problem and the noise corrupting our observations to the optimization problems to be solved in practice.

8.1 Conditional Maximum Likelihood

We do like the squared error function

$$E_{\mathrm{sq}}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y(\boldsymbol{x}_i;\boldsymbol{w}) - t_i)^2. \tag{8.1}$$

Its gradient is easily computed, and we can run gradient descent optimization (Section 2.4.1). If $y(\boldsymbol{x}_i;\boldsymbol{w})$ is linear in the weights $\boldsymbol{w}$, minimizing $E_{\mathrm{sq}}(\boldsymbol{w})$ (least squares estimation) corresponds to orthogonal projection onto the linear mode space (Section 4.2.1), for which we can use robust and efficient algorithms from numerical mathematics (Section 4.2.3). For multi-layer perceptrons, the gradient can be computed by error backpropagation. Why would we ever use anything else? As we will see in this section, the squared error function is not always a sensible choice in practice. In many situations, we can do much better by optimizing other error functions. Switching from squared error to something else, we should be worried about sacrificing the amazing properties of least squares estimation. However, as we will see in the second half of this chapter, such worries are unfounded. The minimization of most frequently used error functions can be reduced to running least squares estimation a few times with reweighted data points. MLP training works pretty much the same way with all these alternatives, in particular error backpropagation remains essentially the same.

Where do alternative error functions come from? How do we choose a good error function for a problem out there? We will address this question in much the same spirit as in the last two chapters: we let ourselves be guided by decision theory and probabilistic modelling. We will discover a second way of doing maximum likelihood estimation and learn about the difference between generative and discriminative (or diagnostic) modelling.

8.1.1 Issues with the Squared Error Function

In this section, we work through some examples, demonstrating why the squared error (8.1) can be improved upon in certain situations. Recall that we introduced $E_{\mathrm{sq}}(\boldsymbol{w})$ in Section 2.4 for training a binary classifier, even before discussing the classical application of least squares linear regression (Section 4.1). Is $E_{\mathrm{sq}}(\boldsymbol{w})$ a good error function for binary classification? *No, it is not!* Let us see why. Consider a binary classification dataset $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1,\dots,n\}$, where $t_i \in \{-1,+1\}$, for which we wish to train a linear classifier $f(\boldsymbol{x}) = \mathrm{sgn}(y(\boldsymbol{x}))$, $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$. The only other classification error function we know up to now is the perceptron error

$$E_{\mathrm{perc}}(\boldsymbol{w}) = \sum_{i=1}^{n} g\left(-t_i y(\boldsymbol{x}_i)\right), \quad g(z) = z\mathrm{I}_{\{z\geq 0\}},$$

derived in Section 2.4.2, so let us compare the two. Before we do that, one comment is necessary in the light of Chapter 2. We do not have to assume in this comparison that the dataset $\mathcal{D}$ is linearly separable. The perceptron algorithm from Section 2.3 will fail for a non-separable set, but *other* algorithms[1] minimize $E_{\text{perc}}(\boldsymbol{w})$ properly for any dataset.

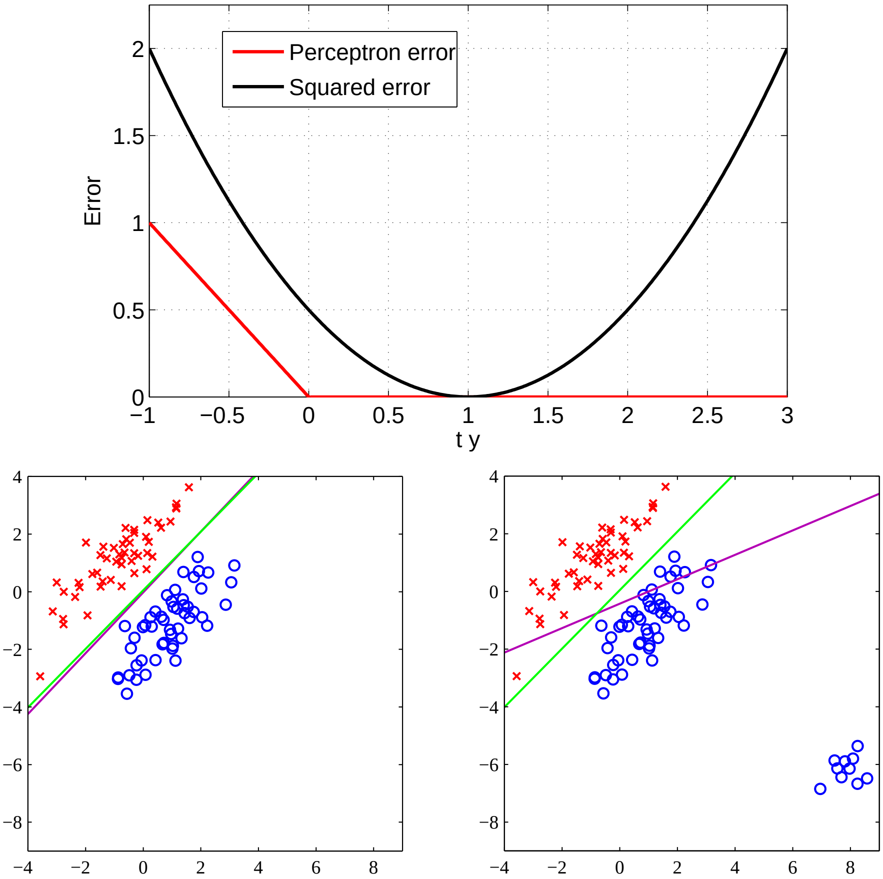


Figure 8.1: Top: Perceptron and squared error for binary classification, shown as function of ty. Notice the rapid increase of the squared error on *both* sides of $ty = 1$. Bottom: Two binary classification datasets. On both sides, we show the least squares discriminant (squared error; magenta line) and the logistic regression discriminant (logistic error; green line). The logistic error is introduced in Section 8.2. For now, think of it as a smooth approximation of the perceptron error. Bottom right: The least squares fit is adversely affected by few additional datapoints in the lower right, even though they are classified correctly by a large margin. Figures on the bottom from [6] (used with permission).

Note that the perceptron error per pattern is a function of $t_i y(\boldsymbol{x}_i)$, which is positive if and only if $y(\cdot)$ classifies $(\boldsymbol{x}_i, t_i)$ correctly. We can also get the squared error into this form (using that $t_i^2 = 1$):

$$E_{\text{sq}}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y(\boldsymbol{x}_i) - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(t_i y(\boldsymbol{x}_i) - 1)^2.$$

[1] $E_{\text{perc}}(\boldsymbol{w})$ is convex and lower bounded. It can be minimized by solving a linear program, which you may derive for yourself.

We can compare the error function by plotting their contribution for pattern $(\boldsymbol{x}_i, t_i)$ as function of $t_i y_i$, where $y_i = y(\boldsymbol{x}_i)$ (Figure 8.1, top). As far as similarities go, both error functions are lower bounded by zero and attain zero only for correctly classified patterns ($y_i = t_i$ for squared error). But there are strong differences. The perceptron error does not penalize correctly classified patterns at all, which is why the perceptron algorithm never updates on such. Also, $E_{\text{perc}}(\boldsymbol{w})$ grows gracefully (linearly) with $-t_i y_i$ for misclassified points. In contrast, the squared error grows quadratically in $t_i y_i$ on *both* sides of 1. Misclassified patterns ($t_i y_i \le 0$) are penalized much more than for the perceptron error. The behaviour of the squared error right of 1 is much more bizarre. $t_i y_i > 1$ means that $y(\cdot)$ classifies $(\boldsymbol{x}_i, t_i)$ correctly with a large margin (Section 2.3.3). That is a good thing, but the squared error penalizes it! In fact, it penalizes it the more, the larger the margin is! This leads to absurd situations, as depicted in Figure 8.1, bottom right. This irrational[2] behaviour of the squared error, applied to binary classification, can lead to serious problems in practice. *Minimizing the squared error is not a useful procedure to train a classifier.* In this section, we learn about alternatives which are only marginally more costly to run and require about the same coding efforts.

Why don't we simply always use the perceptron error? While it is widely used, it comes with problems of its own. It is not differentiable everywhere, so most gradient-based optimizers do not work without subtle modifications. Also, $E_{\text{perc}}(\boldsymbol{w})$ does not assign any error to correctly classified points $(\boldsymbol{x}_i, t_i)$, which means that all separating hyperplanes are equally optimal under this criterion. A different and more compelling reason for using other error functions is discussed in Section 8.2.2.

8.1.2 Squared Error and Gaussian Noise

Recall how we motivated curve fitting in Section 4.1. The targets $t_i \in \mathbb{R}$ arise as sum of a systematic part $y(\boldsymbol{x}_i)$, where $y(\cdot)$ is a smooth curve, plus random noise: $t_i = y(\boldsymbol{x}_i) + \varepsilon_i$. In the spirit of Chapter 6, let us write down what we mean by that and formalize our model assumptions. By the i.i.d. assumption, the variables ε_i are distributed independently according to a *noise distribution* with density $p_\varepsilon(\varepsilon)$. Put differently, the conditional distribution of a target t_i, given the underlying clean curve value $y_i = y(\boldsymbol{x}_i)$, is $p(t_i|y_i) = p_\varepsilon(t_i - y_i)$. Suppose you were given the underlying curve $y(\cdot)$ and access to the noise distribution $p_\varepsilon(\cdot)$. Then, for any $\boldsymbol{x}_i$, you could generate a corresponding target t_i by drawing $\varepsilon_i \sim p_\varepsilon$, then $t_i = y(\boldsymbol{x}_i) + \varepsilon_i$. Generate data, given model and parameters? Sounds like a *likelihood function*:

$$p(\{t_i\}|y(\cdot), \{\boldsymbol{x}_i\}) = \prod_{i=1}^{n} p(t_i|y(\boldsymbol{x}_i)) = \prod_{i=1}^{n} p_\varepsilon(t_i - y(\boldsymbol{x}_i)).$$

If $y(\boldsymbol{x}) = y(\boldsymbol{x}; \boldsymbol{w})$ is parameterized in terms of weights $\boldsymbol{w}$ (for example, a linear function $\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$ or a MLP), then

$$p(\{t_i\}|\boldsymbol{w}, \{\boldsymbol{x}_i\}) = \prod_{i=1}^{n} p(t_i|y(\boldsymbol{x}_i; \boldsymbol{w})).$$

[2] In fact, any sensible error function for binary classification should be nonincreasing on $t_i y_i$. Certainly, it should not grow quadratically!

This likelihood function differs from what we saw in Chapter 6. We do not generate all of the data $\mathcal{D}$, but only the targets $\boldsymbol{t} = [t_i]$. It is a *conditional likelihood*: we condition on the input points. This makes sense. After all, our goal is to predict the function $\boldsymbol{x} \to t$ at inputs $\boldsymbol{x}_*$ not seen in $\mathcal{D}$, and we do not need the distribution of $\boldsymbol{x}$ for that. In the sequel, we will typically drop the conditioning on the input points $\{\boldsymbol{x}_i\}$ from the notation:

$$p(\boldsymbol{t}|\boldsymbol{w}) = \prod_{i=1}^{n} p(t_i|y_i), \quad y_i = y(\boldsymbol{x}_i; \boldsymbol{w}).$$

As noted in Section 6.3, random errors are often captured well by a Gaussian distribution. Let us choose $p(t|y) = N(y, \sigma^2)$, meaning that each ε_i is drawn from $N(0, \sigma^2)$, where σ^2 is the *noise variance*. Here is a surprise. The negative log conditional likelihood for Gaussian noise is the squared error function:

$$\begin{aligned} -\log p(\boldsymbol{t}|\boldsymbol{w}) &= \sum_{i=1}^{n} \frac{1}{2\sigma^2}(y(\boldsymbol{x}_i;\boldsymbol{w}) - t_i)^2 + C = \frac{1}{\sigma^2} E_{\mathrm{sq}}(\boldsymbol{w}) + C, \\ C &= \frac{n}{2}\log(2\pi\sigma^2). \end{aligned} \tag{8.2}$$

Least squares estimation of $\boldsymbol{w}$, minimizing $E_{\mathrm{sq}}(\boldsymbol{w})$, is equivalent to maximizing a Gaussian conditional likelihood.

This observation has several important consequences. First, it furnishes us with *statistical* intuition about the squared error. Independent of computational benefits and simplicity, it is the correct error function to minimize if residuals away from a smooth curve behave more or less like Gaussian noise. On the other hand, if t_i are classification labels and $y(\boldsymbol{x})$ a discriminant, or if the problem is curve fitting, but we expect large errors to occur with substantial probability, then other criteria can lead to far better solutions, and we should make the additional effort to implement estimators for them. Second, just as squared error and Gaussian noise are related by the now familiar $-\log(\cdot)$ transform, the same holds for other such pairs. It is fruitful to walk this bridge in both ways. Given an exotic error function, it may correspond to a negative log conditional likelihood, which provides a clear idea about the scales of residuals to which the error is most sensitive. And if a conditional density $p(t_i|y_i)$ captures properties of the random errors we expect for our data, we have to search no further for a sensible error function to minimize: $\sum_{i=1}^{n} -\log p(t_i|y_i)$.

8.1.3 Conditional Maximum Likelihood

Probabilistic modelling is all about formulating one's assumptions. If the task is to predict a target t from an input pattern $\boldsymbol{x}$, as in classification or regression, it may be most convenient to model the relationship $\boldsymbol{x} \to t$ directly, not bothering with the distribution of $\boldsymbol{x}$ itself. A powerful way to do so is to postulate a systematic function $\boldsymbol{x} \to y$, subsequently obscured by conditionally i.i.d. random noise (see Figure 8.2). In other words, our conditional model has two parts: a deterministic function $y(\boldsymbol{x})$ and a noise distribution (or density) $p(t|y)$. The underlying function $y(\boldsymbol{x})$ is parameterized in terms of weights $\boldsymbol{w}$, and our goal is to learn $\boldsymbol{w}$ from data. For example, in least squares linear regression, we use

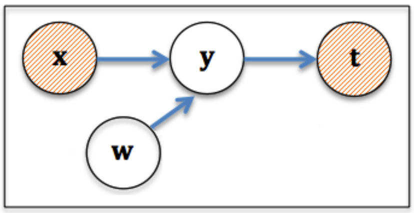


Figure 8.2: Graphical model for a conditional likelihood. $y = y(\boldsymbol{x})$ is a linear combination of a weight vector $\boldsymbol{w}$ and a feature map $\boldsymbol{\phi}(\boldsymbol{x})$. The observed target t is then sampled from $P(t|y)$. While y is real-valued, t can be discrete (for example, $t \in \{-1,+1\}$ in binary classification).

linear functions $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$ and a Gaussian noise model $p(t|y) = N(y,\sigma^2)$. In our examples so far, $y(\boldsymbol{x})$ was a univariate function (a "clean" curve in regression, or a discriminant function in binary classification), and noise was additive ($t_i = y_i + \varepsilon_i$, where $\varepsilon_i \sim p_\varepsilon(\cdot)$ i.i.d. and independent of y_i), but these are special cases. In the sequel of this chapter, we will learn to know a non-additive noise model for binary classification. Its extension to multi-way classification will necessitate a multivariate systematic mapping $\boldsymbol{y}(\boldsymbol{x})$.

Given such a setup, we can fit the conditional model to data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1,\dots,n\}$ by maximizing the *conditional likelihood*

$$p(\boldsymbol{t}|\boldsymbol{w}) = \prod_{i=1}^{n} p(t_i|y_i), \quad y_i = y(\boldsymbol{x}_i;\boldsymbol{w}).$$

In practice, we minimize the negative log conditional likelihood $-\log p(\boldsymbol{t}|\boldsymbol{w})$ instead. Much as in Chapter 6, the main advantage of conditional maximum likelihood is that we do not have to come up with loss functions, estimators or algorithms for every new problem we face. Instead, we simply phrase what we know and observe about the setting in probabilistic terms, from which all these details follow automatically. However, conditional modelling and estimation is clearly different from the joint maximum likelihood procedure of Chapter 6. We will get to the bottom of this distinction in Section 8.2.3, after working through an illustrative example.

Finally, the attentive reader may ask whether parameters $\boldsymbol{w}$ to be learned are confined to the systematic part $y(\boldsymbol{x})$. After all, the noise model $p(t|y)$ may come with parameter as well, for example the noise variance σ^2 of the Gaussian $N(y,\sigma^2)$. In general, the weights $\boldsymbol{w}$ may parameterize the noise model as well. However, parameters such as the noise variance σ^2 are not typically lumped together with $\boldsymbol{w}$, but kept separate as so called *hyperparameters*. The reason for this is that we cannot estimate them by maximum likelihood without running into the over-fitting problem (Chapter 7). For example, we can drive (8.2) to $-\infty$ by choosing $\boldsymbol{w}$ such that $y_i = t_i$ for all i (interpolant), and $\sigma^2 \to 0$. Another example for a hyperparameter is the degree p in polynomial regression (Section 7.1.1). Choosing hyperparameters belongs to the realm of model selection, which we will learn about in Chapter 10. For now, we consider these parameters fixed and given, and concentrate our learning efforts on the weights (or primary parameters) $\boldsymbol{w}$.

8.2 Logistic Regression

The squared error function is not useful for training a binary discriminant function. Neither are we completely happy with the perceptron error function (Section 8.1.1). In this section, we employ the conditional likelihood viewpoint to derive an error function with improved properties.

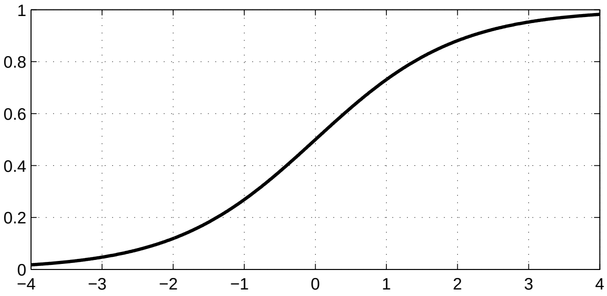


Figure 8.3: Logistic function $\sigma(v) = 1/(1+e^{-v})$, the transfer function implied by the logistic regression likelihood.

Consider the binary classification problem with data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$, where $t_i \in \{-1, +1\}$. Suppose we employ linear functions $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$. How to choose a sensible noise model $P(t|y)$, where $t \in \{-1, +1\}$ and $y \in \mathbb{R}$? Given such a model, which aspect of the conditional distribution $P(t|\boldsymbol{x})$ does $y(\boldsymbol{x})$ represent? Once more, decision theory comes to the rescue. Recall our derivation of the optimal discriminant function $y^*(\boldsymbol{x})$ for spherical Gaussian class-conditional distributions (Section 6.4.1). We found that $y^*(\boldsymbol{x})$ is linear, representing the *log odds ratio*

$$y^*(\boldsymbol{x}) = \log \frac{P(t=1|\boldsymbol{x})}{P(t=-1|\boldsymbol{x})}. \tag{8.3}$$

What we did *not* do back then was to work out $P(t|\boldsymbol{x})$ in terms of $y^*(\boldsymbol{x})$. Let us do that now. Using that $P(t=1|\boldsymbol{x}) + P(t=-1|\boldsymbol{x}) = 1$:

$$\begin{aligned} & e^{y^*(\boldsymbol{x})}\left(1 - P(t=1|\boldsymbol{x})\right) = P(t=1|\boldsymbol{x}) \\ \Leftrightarrow\ & P(t=1|\boldsymbol{x}) = \frac{e^{y^*(\boldsymbol{x})}}{1+e^{y^*(\boldsymbol{x})}} = \sigma(y^*(\boldsymbol{x})), \qquad \sigma(v) = \frac{1}{1+e^{-v}}. \end{aligned}$$

$\sigma(v)$ is called the *logistic function* (Figure 8.3). It may remind you of the $\tanh(a)$ transfer function we used with MLPs (Section 3.2). Indeed, you will have no problem confirming that $\tanh(a) = 2\sigma(2a) - 1$: while $\tanh(a)$ is symmetric w.r.t. the origin, $\sigma(v)$ is symmetric w.r.t. $(0, 1/2)$. Since $\sigma(-v) = 1 - \sigma(v)$, we can write compactly:

$$P(t|\boldsymbol{x}) = P(t|y^*(\boldsymbol{x})) = \sigma(t y^*(\boldsymbol{x})), \quad t \in \{-1, +1\}.$$

$P(t|\boldsymbol{x})$ depends on $\boldsymbol{x}$ only through $y^*(\boldsymbol{x})$, so this is the noise model we have been looking for: $P(t|y) = \sigma(ty)$.

This went a bit fast, so let us repeat what we did here. We used a simple generative setup of binary classification (two Gaussian class-conditional distributions with unit covariance matrix) in order to specify a noise model $P(t|y)$ for conditional maximum likelihood. Given the joint setup $p(\boldsymbol{x}, t) = p(\boldsymbol{x}|t)P(t)$, we worked out the posterior $P(t|\boldsymbol{x})$ and the corresponding optimal discriminant function $y^*(\boldsymbol{x})$ in Section 6.4.1. Two things happen. First, $y^*(\boldsymbol{x})$ turns out to be linear. Second, the posterior distribution $P(t|\boldsymbol{x})$ can be expressed as $P(t|y^*(\boldsymbol{x}))$ for a conditional distribution $P(t|y)$, $y \in \mathbb{R}$. In probabilistic terms, t and $\boldsymbol{x}$ are conditionally independent given $y^*(\boldsymbol{x})$. The "two spherical Gaussian" generative setup exhibits the separation into systematic part and random noise required for conditional modelling (Section 8.1.3). The systematic part $y^*(\boldsymbol{x})$ is linear, while the noise model turns out to be logistic: $P(t|y) = \sigma(ty)$.

There is nothing special about the setup with spherical Gaussian class-conditional distributions $p(\boldsymbol{x}|t)$. The logistic link between $y^*(\boldsymbol{x})$ and $P(t|y)$ is simply an inversion of the definition of the log odds ratio (8.3) in terms of $P(t|y)$. Our conditional likelihood requirements are satisfied whenever the log odds ratio is linear in the parameters $\boldsymbol{w}$ to be learned, and this happens for a range of other generative models as well. An example is the naive Bayes document classification model for $K = 2$ classes (Section 6.5).

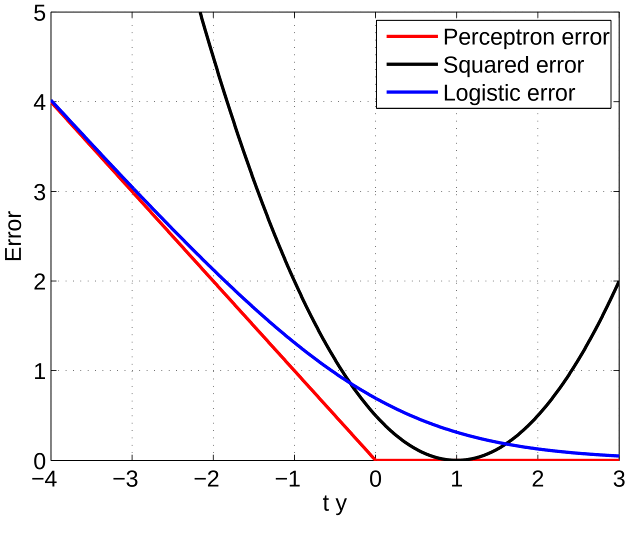


Figure 8.4: Binary classification error functions used in perceptron learning, least squares discrimination and logistic regression.

The corresponding error function per data point is

$$-\log P(t_i|y_i) = -\log \sigma(t_i y_i) = \log\left(1 + e^{-t_i y_i}\right),$$

giving rise to the *logistic error function*

$$E_{\log}(\boldsymbol{w}) = -\log P(\boldsymbol{t}|\boldsymbol{w}) = \sum_{i=1}^{n} \log\left(1 + e^{-t_i y_i}\right), \quad y_i = y(\boldsymbol{x}_i; \boldsymbol{w}). \tag{8.4}$$

Minimizing $E_{\log}(\boldsymbol{w})$ is an instance of conditional maximum likelihood, known as *logistic regression* (if $y(\boldsymbol{x};\boldsymbol{w})$ is linear in $\boldsymbol{w}$). The nomenclature reinforces a point we made in Section 4.1. Even though our goal is classification, mapping $\boldsymbol{x}$ to $t \in \{-1,+1\}$, the basic relationship is a real-valued curve $\boldsymbol{x} \to y$, which is fitted to data ("regression") through the logistic link. As we will see in Section 8.2.2, this procedure can do more for us than mere discrimination. Moreover, the embedded real-valued function is the basis for a very efficient training algorithm, which reduces logistic regression to a few (reweighted) calls of classical linear regression (Section 8.2.4).

In Figure 8.4, the logistic error is compared to the squared error $E_{\mathrm{sq}}(\boldsymbol{w})$ and the perceptron error $E_{\mathrm{perc}}(\boldsymbol{w})$, plotting their contribution per data point $(\boldsymbol{x}, t)$ as function of ty. Clearly, $E_{\log}(\boldsymbol{w})$ does not share the erratic behaviour of the squared error. It decreases monotonically with ty, encouraging classification with large margins. In fact, the logistic and perceptron error functions behave the same for large $|ty|$, positive or negative. The logistic error improves[3] upon the perceptron error by being continuously differentiable everywhere. It also encourages large margins, as patterns which are correctly classified just so still imply a substantial penalty. A further more important advantage of $E_{\log}$ over E_{perc} is discussed in Section 8.2.2.

8.2.1 Gradient Descent Optimization

In this section, we show how to compute the gradient of the logistic error (8.4) w.r.t. the weights $\boldsymbol{w}$, which allows us to minimize $E_{\log}(\boldsymbol{w})$ by gradient descent (Section 2.4.1). We begin with the linear case, $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$. Recall that

$$E_{\log}(\boldsymbol{w}) = \sum_{i=1}^{n} E_i(\boldsymbol{w}), \quad E_i(\boldsymbol{w}) = -\log\sigma(t_i y_i) = \log\left(1 + e^{-t_i y_i}\right).$$

One observation up front. $E_{\log}(\boldsymbol{w})$ is a convex[4], lower bounded function of $\boldsymbol{w}$. Convexity is a key concept in numerical optimization and machine learning, which we will need in Chapter 9. Some definitions and links are given in Section 9.1.1. For our purposes here, convexity implies that each local minimum is a global minimum, so that gradient descent optimization will detect a global minimum point upon convergence[5]. The gradient computation decomposes as $\nabla_{\boldsymbol{w}} E_{\log} = \sum_{i=1}^{n} \nabla_{\boldsymbol{w}} E_i$. For standard gradient descent, we determine a descent direction by summing up $\boldsymbol{g}_i = \nabla_{\boldsymbol{w}} E_i$, while for stochastic gradient descent we

[3] Differentiability can also be a disadvantage, as it does not encourage sparsity of the predictor (see Chapter 9).

[4] Namely, $y_i \mapsto \log(1 + e^{-t_i y_i})$ is an instance of the `logsumexp` function, which is convex [8], and y_i is linear in $\boldsymbol{w}$.

[5] There is one catch with the latter statement, which we will clarify in Section 8.3.2. For a linearly separable training dataset, logistic regression can drive $\|\boldsymbol{w}\|$ to very large values at ever decreasing gains in $E_{\log}(\boldsymbol{w})$.

use $\boldsymbol{g}_i$ for single patterns $(\boldsymbol{x}_i, t_i)$ (Section 2.4.2). Before we begin, recall the simple form of the gradient for the *squared* error (8.1) from Section 2.4.1:

$$\nabla_{\boldsymbol{w}} \frac{1}{2}(y_i - t_i)^2 = (y_i - t_i)\nabla_{\boldsymbol{w}} y_i = (y_i - t_i)\boldsymbol{\phi}(\boldsymbol{x}_i).$$

The gradient is the sum of feature vectors $\boldsymbol{\phi}(\boldsymbol{x}_i)$, each weighted by the residual $y_i - t_i$.

Recall that $t_i \in \{-1, +1\}$. Let $\tilde{t}_i = (t_i + 1)/2 \in \{0, 1\}$, and define $\pi_i = P(t_i = 1|y_i) = \sigma(y_i) \in (0, 1)$. In order to work out the gradient, note that

$$\sigma'(v) = \frac{e^{-v}}{(1 + e^{-v})^2} = \sigma(v)\frac{e^{-v}}{1 + e^{-v}} = \sigma(v)\sigma(-v) = \sigma(v)(1 - \sigma(v)).$$

Therefore,

$$\frac{\partial}{\partial y_i} - \log \sigma(t_i y_i) = \frac{-t_i \sigma(t_i y_i)\sigma(-t_i y_i)}{\sigma(t_i y_i)} = -t_i \sigma(-t_i y_i) = \pi_i - \tilde{t}_i,$$

and

$$\nabla_{\boldsymbol{w}} E_i = (\pi_i - \tilde{t}_i)\nabla_{\boldsymbol{w}} y_i = (\pi_i - \tilde{t}_i)\boldsymbol{\phi}(\boldsymbol{x}_i), \quad \pi_i = \sigma(y_i).$$

This has the *same* form as for the squared error, only that the residuals are redefined: $\pi_i - \tilde{t}_i$ instead of $y_i - t_i$. For the squared error, a pattern $(\boldsymbol{x}_i, t_i)$ does not contribute much to the gradient if and only if $y_i - t_i \approx 0$, or $y_i \approx t_i$. As noted in Section 8.1.1, this does not make much sense for binary classification. For example, $t_i = +1$, $y_i = 5$ is penalized heavily, even though the pattern is classified correctly with large margin. For the logistic error, the residual is redefined in a way which makes sense for binary classification. A pattern does not contribute much if and only if $\pi_i = P(t_i = 1|y_i) \approx \tilde{t}_i$, namely if the pattern is classified correctly with high confidence. If $t_i = +1$, $y_i = 5$, then $\pi_1 \approx 1$ and $\tilde{t}_i = 1$, so that $(\boldsymbol{x}_i, t_i)$ contributes very little to $\nabla_{\boldsymbol{w}} E_{\text{log}}$. Using the vectorized notation of Section 2.4.1:

$$\nabla_{\boldsymbol{w}} E_{\text{log}} = \sum_{i=1}^{n} (\pi_i - \tilde{t}_i)\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{\Phi}^T(\boldsymbol{\pi} - \tilde{\boldsymbol{t}}), \quad \boldsymbol{\pi} = [\pi_i],\ \tilde{\boldsymbol{t}} = [\tilde{t}_i]. \tag{8.5}$$

If we limit ourselves to gradient descent optimization, there is no computational difference between the two error functions, and the logistic error should be the preferred option.

Logistic Error for MLPs

Recall multi-layer perceptrons (MLPs) from Chapter 3. Back then, we minimized the squared error $E_{\text{sq}}(\boldsymbol{w})$ by gradient descent, where the gradient could be computed efficiently by error backpropagation (Section 3.3). Suppose we want to train an MLP for a binary classification task, such as 8s versus 9s on the MNIST digits. The squared error is bad for classification, whether we use linear discriminants, MLPs, or anything else, so let us use the logistic error for our MLP instead. Surprisingly, this switch does not introduce any additional complications, neither in the implementation nor in computations. Recall our derivation of error backpropagation in Section 3.3, and consider a network with L layers, whose output activation is $a^{(L)}(\boldsymbol{x}_i)$ for input $\boldsymbol{x}_i$. Let us compare the i-th pattern's contribution to the error functions and their respective gradient computations:

$$E_{\text{sq},i}(\boldsymbol{w}) = \frac{1}{2}\left(a^{(L)}(\boldsymbol{x}_i) - t_i\right)^2, \quad E_{\log,i}(\boldsymbol{w}) = -\log\sigma\left(t_i a^{(L)}(\boldsymbol{x}_i)\right).$$

The corresponding residuals at the uppermost layer are

$$r^{(L)}_{\text{sq},i} = a^{(L)}(\boldsymbol{x}_i) - t_i, \quad r^{(L)}_{\log,i} = \sigma\left(a^{(L)}(\boldsymbol{x}_i)\right) - \tilde{t}_i.$$

Recall that backpropagation decomposes into forward pass, computation of the output residual, backward pass and gradient assembly. Suppose you have code up and running for training MLPs by minimizing the squared error. The only change you have to make in order to minimize the logistic error instead, is to replace the output residuals $a^{(L)}(\boldsymbol{x}_i) - t_i$ by $\sigma(a^{(L)}(\boldsymbol{x}_i)) - \tilde{t}_i$. This insight holds for other (continuously differentiable) error functions besides E_{sq} and $E_{\log}$ just as well. You can and should keep your MLP code generic. For each new error function to be supported, all you need to work out is which output residuals it gives rise to. All dominating computations, such as forward and backward pass, are independent of the error function used. In short, error backpropagation stays exactly the same, the backward pass is simply initialized with different errors (or residuals).

8.2.2 Estimating Posterior Class Probabilities (*)

Recall our comparison between logistic and perceptron error above. A compelling reason to prefer $E_{\log}(\boldsymbol{w})$ over $E_{\text{perc}}(\boldsymbol{w})$ is understood by noting that for our decision-theoretic setup, the link between the optimal discriminant $y^*(\boldsymbol{x})$ and $P(t|\boldsymbol{x})$ is one-to-one. In other words, the posterior distribution $P(t|\boldsymbol{x})$ is a fixed and known function of $y^*(\boldsymbol{x})$. If we employ the logistic link for conditional maximum likelihood, we cannot only learn to discriminate well, but also estimate the posterior probabilities $P(t|\boldsymbol{x})$. There are numerous reasons why we would want to do the latter, some of which were summarized in Section 5.2.5. After all, the Bayes-optimal classifier $f^*(\boldsymbol{x})$ decides $f^*(\boldsymbol{x}) = 1$ in the same way for $P(t = 1|\boldsymbol{x}) = 0.51$ and $P(t = 1|\boldsymbol{x}) = 0.99$, while knowledge of $P(t|\boldsymbol{x})$ may lead to different decisions in these cases, and is clearly more informative in general. For instance, class probability estimates are essential in the cancer screening example of Section 5.2.4.

The representation of $f^*(\boldsymbol{x})$ in terms of a discriminant function, $f^*(\boldsymbol{x}) = \operatorname{sgn}(y^*(\boldsymbol{x}))$, is obviously not unique. When we talked about "the" optimal discriminant $y^*(\boldsymbol{x})$ above, we meant the log odds ratio (8.3), but there are many others. For some of them, we can reconstruct the posterior probability $P(t|\boldsymbol{x})$ from $y^*(\boldsymbol{x})$ for each input point $\boldsymbol{x}$, for others we cannot. An example for the first class of discriminant functions is the log odds ratio:

$$y^*(\boldsymbol{x}) = \log\frac{P(t = +1|\boldsymbol{x})}{P(t = -1|\boldsymbol{x})} \quad \Rightarrow \quad P(t|\boldsymbol{x}) = \sigma(t y^*(\boldsymbol{x})).$$

An example for the second class is $f^*(\boldsymbol{x})$ itself. This is an optimal discriminant function, since $f^*(\boldsymbol{x}) = \mathrm{sgn}(f^*(\boldsymbol{x}))$. On the other hand, $f^*(\boldsymbol{x})$ contains no more information about $P(t|\boldsymbol{x})$ than whether it is larger or smaller than $1/2$. If our goal is to estimate posterior class probabilities, we have to choose an error function whose minimizer tends to a "probability-revealing" discriminant function at least in principle, for a growing number of training data points and discriminant functions to choose from.

Giving an operational meaning to "at least in principle" is the realm of learning theory, which is not in the scope of this lecture. But a necessary condition for an error function to allow for posterior probability estimation is easily understood by the concept of *population minimizers*. Recall the setup of decision theory from Section 5.2. Suppose we are given an error function $\mathcal{E}(y(\cdot)) = n^{-1}\sum_{i=1}^{n} g(t_i, y(\boldsymbol{x}_i))$. Then, $y^*(\boldsymbol{x})$ is a population minimizer if

$$y^*(\boldsymbol{x}) \in \operatorname*{argmin}_{y} \mathrm{E}\left[g(t,y) \;\middle|\; \boldsymbol{x}\right]$$

for almost all $\boldsymbol{x}$. Intuitively, if we use more and more training data and allow for any kind of function $y(\boldsymbol{x})$, we should end up with a population minimizer. An error function allows for estimating posterior probabilities only if all its population minimizers $y^*(\boldsymbol{x})$ are probability-revealing, in that $P(t = +1|\boldsymbol{x})$ is a fixed function of $y^*(\boldsymbol{x})$.

For the logistic error function, $g_{\log}(t,y) = \log(1+e^{-ty})$, and the unique population minimizer is the log odds ratio (8.3), which is probability-revealing. On the other hand, the perceptron error function performs poorly w.r.t. this test. One of its population minimizers is $y^*(\boldsymbol{x}) \equiv 0$. This comment may paint an overly pessimistic picture of the perceptron criterion, which often works well if applied with linear functions $y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$ under constraints such as $\|\boldsymbol{w}\| = 1$ (which rules out the all-zero solution). But if E_{perc} is used in less restrictive settings, it often does lead to solutions with a tiny margin (see Section 2.3.3), from which posterior probabilities cannot be estimated. As we will see in Chapter 9, one way of addressing these problems of E_{perc} is to insist on a maximally large margin. However, as shown in Section 9.4, the resulting maximum margin perceptron (or support vector machine) does not allow for the consistent estimation of posterior class probabilities either.

To sum up, the logistic error function, being the negative log conditional likelihood under the logistic noise model $P(t|y) = \sigma(ty)$, supports the consistent estimation of posterior class probabilities, while the perceptron error does not. It is interesting to note that the squared error (8.1), for all its shortcomings with binary classification, shares this beneficial property with $E_{\log}$. We will see in Section 10.1.3 that its population minimizer is the conditional expectation $\mathrm{E}[t \mid \boldsymbol{x}]$, which is obviously probability-revealing in the case of binary classification.

8.2.3 Generative and Discriminative Models

Armed with the example of logistic regression, we are now ready to highlight the fundamental difference between joint and conditional likelihood maximization, between generative and discriminative modelling. Recall our motivation of logistic regression by way of decision theory for spherical Gaussian class-conditional distributions. However, there are two different routes now from decision-theoretic insight to a real-world classifier trained on data $\mathcal{D}$ by maximum likelihood estimation. We can proceed as in Section 6.4.1, estimating the parameters of class-conditional and class prior distributions by joint maximum likelihood and plugging them into the log odds ratio discriminant. Or we estimate the discriminant parameters directly by conditional maximum likelihood (Section 8.1.3).

For the first option, $p(\boldsymbol{x}|t,\boldsymbol{\theta}_{\text{gen}}) = N(\boldsymbol{x}|\boldsymbol{\mu}_t,\boldsymbol{I})$ and $P(t|\boldsymbol{\theta}_{\text{gen}}) = \pi_1^{(1+t)/2}(1-\pi_1)^{(1-t)/2}$, with parameters $\boldsymbol{\theta}_{\text{gen}} = [(\boldsymbol{\theta}_{-1})^T,(\boldsymbol{\theta}_{+1})^T,\pi_1]^T$. We estimate those by maximizing the *joint likelihood*

$$\max_{\boldsymbol{\theta}_{\text{gen}}}\left\{\prod_{i=1}^n p(\boldsymbol{x}_i,t_i|\boldsymbol{\theta}_{\text{gen}}) = \prod_{i=1}^n p(\boldsymbol{x}_i|t_i,\boldsymbol{\theta}_{\text{gen}})P(t_i|\boldsymbol{\theta}_{\text{gen}})\right\},$$

then plug the solution $\hat{\boldsymbol{\theta}}_{\text{gen}}$ into the logs odds ratio

$$\hat{y}_{\text{gen}}(\boldsymbol{x};\hat{\boldsymbol{\theta}}_{\text{gen}}) = \left(\hat{\boldsymbol{\mu}}_{+1}-\hat{\boldsymbol{\mu}}_{-1}\right)^T\boldsymbol{x}+\hat{b} = (\hat{\boldsymbol{w}}_{\text{gen}})^T\boldsymbol{\phi}(\boldsymbol{x}),\ \hat{\boldsymbol{w}}_{\text{gen}} = \begin{bmatrix}\hat{\boldsymbol{\mu}}_{+1}-\hat{\boldsymbol{\mu}}_{-1}\\ \hat{b}\end{bmatrix},$$

$$\hat{b} = -\frac{1}{2}\left(\|\hat{\boldsymbol{\mu}}_{+1}\|^2-\|\hat{\boldsymbol{\mu}}_{-1}\|^2\right)+\log\frac{\hat{\pi}_1}{1-\hat{\pi}_1}.$$

For the second option, we parameterize the log odds ratio directly as $y_{\text{dsc}}(\boldsymbol{x};\boldsymbol{\theta}_{\text{dsc}}) = (\boldsymbol{w}_{\text{dsc}})^T\boldsymbol{\phi}(\boldsymbol{x})$, so that $\boldsymbol{\theta}_{\text{dsc}} = \boldsymbol{w}_{\text{dsc}}$, and fit it to the data by maximizing the *conditional likelihood*

$$\max_{\boldsymbol{\theta}_{\text{dsc}}}\left\{\prod_{i=1}^n P(t_i|\boldsymbol{\theta}_{\text{dsc}},\boldsymbol{x}_i) = \prod_{i=1}^n \sigma\left(t_i y_{\text{dsc}}(\boldsymbol{x}_i;\boldsymbol{\theta}_{\text{dsc}})\right)\right\}.$$

Even though these two classification methods share the same decision-theoretic motivation and result in $P(t|\boldsymbol{x})$ estimates of the same functional form, they can behave very differently in practice. The weights $\hat{\boldsymbol{w}}_{\text{dsc}}$ in conditional maximum likelihood are estimated directly, while the weights $\hat{\boldsymbol{w}}_{\text{gen}}$ are assembled from $\hat{\boldsymbol{\theta}}_{\text{gen}}$, which are twice as many parameters estimated in a different way. In Figure 8.5, the generative ML plug-in rule is compared to discriminative logistic regression. The two methods clearly produce different results. They constitute examples of different modelling and learning paradigms:

- Generative modelling: Devise a joint model of all variables of the problem domain, containing in particular the input point $\boldsymbol{x}$. For classification, the joint model would be

  $$p(\boldsymbol{x},t|\boldsymbol{\theta}) = p(\boldsymbol{x}|t,\boldsymbol{\theta})P(t|\boldsymbol{\theta}).$$

  Learn parameters by joint maximum likelihood,

  $$\max_{\boldsymbol{\theta}}\prod_{i=1}^n p(\boldsymbol{x}_i,t_i|\boldsymbol{\theta}),$$

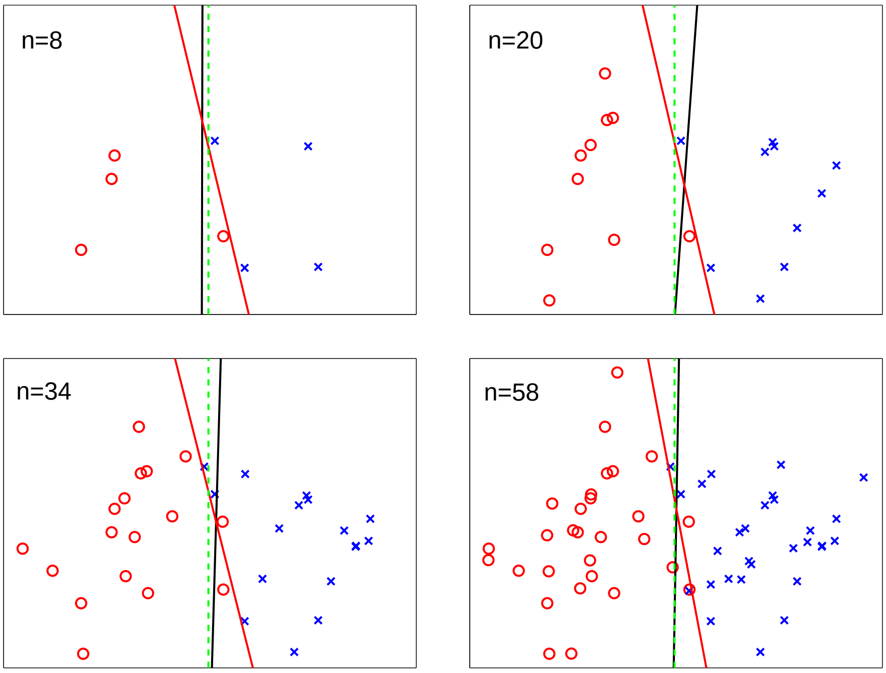


Figure 8.5: ML plug-in classifier for spherical Gaussian class-conditional distributions (black) versus logistic regression by conditional maximum likelihood (red), for a growing number n of data points. The true data distribution consists of two equi-probable Gaussians with spherical covariance (optimal decision boundary in dashed green), which favours the plug-in rule. Note the different behaviour of the two methods, in particular for small training set sizes.

or a maximum a-posteriori variant (Section 7.3). Plug the maximizer $\hat{\boldsymbol{\theta}}$ into the log odds ratio discriminant function, or into the derived posterior

$$P(t|\boldsymbol{x}, \hat{\boldsymbol{\theta}}) \propto p(\boldsymbol{x}|t, \hat{\boldsymbol{\theta}}) P(t|\hat{\boldsymbol{\theta}})$$

in order to predict posterior class probabilities.
We mainly focussed on the generative modelling approach in Chapter 6. The nomenclature is due to the fact that given a generative model, a complete dataset could be sampled (or generated) from it. In particular, the input point $\boldsymbol{x}$ is modelled.

- Discriminative modelling (or diagnostic modelling): Devise a conditional model of the variable(s) to be predicted, conditional on variables which will always be given at prediction time. For classification, the target t is modelled conditioned on the input point $\boldsymbol{x}$:

  $$P(t|\boldsymbol{x}, \boldsymbol{\theta}).$$

  Learn parameters by conditional maximum likelihood,

  $$\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} P(t_i|\boldsymbol{x}_i, \boldsymbol{\theta}),$$

or a maximum a-posteriori variant (Section 8.3.2). Plug the maximizer $\hat{\boldsymbol{\theta}}$ directly into $P(t|\boldsymbol{x}, \boldsymbol{\theta})$ in order to predict posterior class probabilities. The discriminative modelling approach and conditional maximum likelihood was introduced in the present chapter. The name "discriminative" is somewhat misleading (and "diagnostic" is preferred in statistics), since regression and other setups are covered just as well. As seen in Section 8.1.2, least squares estimation is an example of conditional maximum likelihood for a discriminative model with Gaussian noise. Note that a discriminative model does not say anything about the generation of input points $\boldsymbol{x}$.

The most important point to note about generative and discriminative modelling is that they constitute two different options we have for addressing a problem such as classification or regression, each coming with strengths and weaknesses. For a particular application, it is not usually obvious a priori which of the two will work better. However, some general statements can be made. Discriminative modelling is more direct and often leads to models with less parameters to estimate. After all, the distribution $p(\boldsymbol{x})$ over input points is not represented at all. Most "black-box" classification or curve fitting methods are based on discriminative models, even though the feature map $\boldsymbol{\phi}(\boldsymbol{x})$ still has to be specified. On the other hand, it is often much simpler to encode structural prior knowledge about a task in a generative "forward" model (as emphasized in Section 6.1) than to guess a sensible form for $P(t|\boldsymbol{x})$. Moreover, training a generative model is often simpler and more modular. A general advantage of generative over discriminative models is that cases with corrupted or partly missing input point $\boldsymbol{x}_i$ can be used rather easily with the former (using techniques discussed in Chapter 12), but typically have to be discarded with the latter. It is possible to combine the two paradigms in different ways, a topic which is not in the scope of this course.

8.2.4 Iteratively Reweighted Least Squares (*)

As noted in Section 8.1.1, least squares estimation is not a useful approach to binary classification. Nevertheless, it tends to be used rather frequently in machine learning circles towards this end. As seen in Section 8.2.1, if you limit yourself to gradient descent, there is really no reason to prefer E_{sq} over E_{log}. On the other hand, least squares estimation can in general be solved much more efficiently by advanced solvers from numerical mathematics, for which code is publicly available (Section 4.2.3). In this section, we discuss an algorithm which globally minimizes the logistic error function (8.4) by solving a short sequence of reweighted least squares problems. In other words, this algorithm reduces logistic regression training to a few calls of least squares estimation. As we will see, the implementational effort on top of LSE is rather minor.

The algorithm we will work out is called *iteratively reweighted least squares* (IRLS). It is an instance of the Newton-Raphson algorithm, motivated in Section 3.4.2. The application to logistic regression is also called Fisher scoring in the statistics literature. We will derive it as a reduction to least squares estimation. Recall from Chapter 4 that LSE is equivalent to the minimization of the quadratic function $E_{\text{sq}}(\boldsymbol{w})$ with positive definite system matrix $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$. We wish to minimize $E_{\text{log}}(\boldsymbol{w})$ which is not quadratic, so we cannot hope for a direct reduction to LSE. The next best idea is to proceed iteratively. Suppose we are at the point $\boldsymbol{w}$. The locally closest quadratic fit to $E_{\log}(\boldsymbol{w}')$ is given by the Taylor approximation

$$\begin{aligned} E_{\log}(\boldsymbol{w}') \approx q_{\boldsymbol{w}}(\boldsymbol{w}') :=& E_{\log}(\boldsymbol{w}) + (\nabla_{\boldsymbol{w}} E_{\log})^T (\boldsymbol{w}' - \boldsymbol{w}) \\ &+ \frac{1}{2}(\boldsymbol{w}' - \boldsymbol{w})^T (\nabla\nabla_{\boldsymbol{w}} E_{\log}) (\boldsymbol{w}' - \boldsymbol{w}). \end{aligned}$$

Here, $\nabla\nabla_{\boldsymbol{w}} E_{\log}$ is the Hessian (matrix of second derivatives) at $\boldsymbol{w}$. The idea behind Newton-Raphson is to minimize the surrogate $q_{\boldsymbol{w}}(\boldsymbol{w}')$ instead of $E_{\log}(\boldsymbol{w}')$, updating $\boldsymbol{w}$ to the quadratic minimizer. We will see that $\nabla\nabla_{\boldsymbol{w}} E_{\log}$ is positive definite, so the quadratic minimizer $\boldsymbol{w}'$ is given by

$$(\nabla\nabla_{\boldsymbol{w}} E_{\log}) (\boldsymbol{w}' - \boldsymbol{w}) = -\nabla_{\boldsymbol{w}} E_{\log}.$$

We already worked out the gradient (8.5) in Section 8.2.1. For the Hessian, we employ the same strategy. First, it will be the sum of contributions $\nabla\nabla_{\boldsymbol{w}} E_i$, one for each data point. Second, we can use the chain rule once more. If $\boldsymbol{y} = [y_i] = \boldsymbol{\Phi}\boldsymbol{w}$, then

$$\begin{aligned} \nabla_{\boldsymbol{w}} E_i &= \frac{\partial E_i}{\partial y_i} \nabla_{\boldsymbol{w}} y_i = \frac{\partial E_i}{\partial y_i} \boldsymbol{\phi}(\boldsymbol{x}_i) \\ \Rightarrow \nabla\nabla_{\boldsymbol{w}} E_i &= (\nabla_{\boldsymbol{w}} y_i) \frac{\partial E_i^2}{\partial^2 y_i} (\nabla_{\boldsymbol{w}} y_i)^T = \frac{\partial E_i^2}{\partial^2 y_i} \boldsymbol{\phi}(\boldsymbol{x}_i)\boldsymbol{\phi}(\boldsymbol{x}_i)^T. \end{aligned}$$

Therefore, the Hessian is

$$\nabla\nabla_{\boldsymbol{w}} E_{\log} = \sum_{i=1}^n \kappa_i \boldsymbol{\phi}(\boldsymbol{x}_i)\boldsymbol{\phi}(\boldsymbol{x}_i)^T = \boldsymbol{\Phi}^T (\operatorname{diag} \boldsymbol{\kappa}) \boldsymbol{\Phi}, \quad \kappa_i = \frac{\partial E_i^2}{\partial^2 y_i}.$$

Moreover, since $\partial E_i / \partial y_i = \sigma(y_i) - \tilde{t}_i$, then

$$\kappa_i = \frac{\partial}{\partial y_i} \left(\sigma(y_i) - \tilde{t}_i \right) = \sigma(y_i)\sigma(-y_i) = \pi_i(1 - \pi_i) > 0,$$

since $\pi_i = \sigma(y_i) > 0$. In fact, $\kappa_i \in (0, 1/4]$. The Hessian is positive definite, which provides another proof of the convexity of $E_{\log}(\boldsymbol{w})$ (see Section 9.1.1).

Better still, both Hessian and gradient are simply reweighted versions of the corresponding entities in standard least squares estimation. If $\nabla_{\boldsymbol{w}} E_{\log} = \boldsymbol{\Phi}^T \boldsymbol{\xi}$, $\xi_i = \sigma(y_i) - \tilde{t}_i$, then

$$\begin{aligned} q_{\boldsymbol{w}}(\boldsymbol{w} + \boldsymbol{d}) &= \boldsymbol{\xi}^T \boldsymbol{\Phi} \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T \boldsymbol{\Phi}^T (\operatorname{diag} \boldsymbol{\kappa}) \boldsymbol{\Phi} \boldsymbol{d} + C_1 \\ &= \frac{1}{2} (\boldsymbol{\Phi}\boldsymbol{d} - \boldsymbol{f})^T (\operatorname{diag} \boldsymbol{\kappa}) (\boldsymbol{\Phi}\boldsymbol{d} - \boldsymbol{f}) + C_2 \\ &= \frac{1}{2} \sum_{i=1}^n \kappa_i \left(\boldsymbol{d}^T \boldsymbol{\phi}(\boldsymbol{x}_i) - f_i \right)^2 + C_2, \quad f_i = -\xi_i / \kappa_i, \end{aligned}$$

where C_1, C_2 are constants. $\min_{\boldsymbol{d}} q_{\boldsymbol{w}}(\boldsymbol{w} + \boldsymbol{d})$ is a least squares estimation problem, where each data point $\boldsymbol{x}_i$ is associated with a pseudotarget f_i, and the contribution of $(\boldsymbol{x}_i, f_i)$ is weighted by $\kappa_i \in (0, 1/4]$. Publicly available LSE codes typically allow for such weighting. The solution $\boldsymbol{d}_*$ is called *Newton direction*, $q_{\boldsymbol{w}}(\boldsymbol{w}')$ is minimized for $\boldsymbol{w}' = \boldsymbol{w} + \boldsymbol{d}_*$. This observation explains the naming of IRLS, which is solved by iterating over a sequence of reweighted least squares estimation problems.

If you ever implement IRLS in practice, you should note one more detail. The derivation above suggests to update $\boldsymbol{w}$ to $\boldsymbol{w} + \boldsymbol{d}_*$ at the end of an iteration, as this is the minimizer of the quadratic fit $q_{\boldsymbol{w}}(\boldsymbol{w}')$. In practice, it works better to employ a line search:

$$\boldsymbol{w}' \leftarrow \boldsymbol{w} + \lambda_* \boldsymbol{d}_*, \qquad \lambda_* = \operatorname*{argmin}_{\lambda > 0} E_{\log}(\boldsymbol{w} + \lambda \boldsymbol{d}_*).$$

In other words, we search for a minimum point along the line segment $\{\boldsymbol{w} + \lambda \boldsymbol{d}_* \mid \lambda > 0\}$ determined by the Newton direction. The minimum does not have to be found to high accuracy, so that very few evaluations of $E_{\log}$ (and possibly its derivative) along the line are sufficient. It is common practice to start the line search with $\lambda = 1$ and accept the full Newton step if this leads to sufficient descent. Details can be found in [3].

8.3 Discriminative Models

In this section, we discuss further examples and extensions of discriminative modelling. First, we show how to extend logistic regression to more than two classes. Next, we extend conditional maximum likelihood to conditional maximum a-posteriori (MAP) estimation and draw a bridge to regularized estimation.

8.3.1 Multi-Way Logistic Regression

We developed logistic regression in Section 8.2 for the case of two classes. How about the general case of $K \geq 2$ classes? We can let ourselves being guided by decision theory in much the same way. We derived a generative maximum likelihood approach to multi-way classication in Section 6.4.1, which employed K discriminant functions

$$y_k^*(\boldsymbol{x}) = (\boldsymbol{w}_k)^T \boldsymbol{\phi}(\boldsymbol{x}) = \log P(t = k|\boldsymbol{x}) + C(\boldsymbol{x}), \quad k = 0, \dots, K-1.$$

Here, $C(\boldsymbol{x})$ does not depend on k. In the case of spherical Gaussian class-conditional distributions discussed back then, we had $\boldsymbol{\phi}(\boldsymbol{x}) = [\boldsymbol{x}^T, 1]^T$. What matters is that the log posterior $\log P(t = k|\boldsymbol{x})$ is a linear function of $\boldsymbol{\phi}(\boldsymbol{x})$ plus a part which does not depend on the class label k. The implied classification rule was

$$f^*(\boldsymbol{x}) = \operatorname*{argmax}_{k=0,\dots,K-1} y_k^*(\boldsymbol{x}) = \operatorname*{argmax}_{k=0,\dots,K-1} P(t = k|\boldsymbol{x}),$$

since neither the addition of $C(\boldsymbol{x})$ nor the increasing $\exp(\cdot)$ transform changes the maximizing value for k. How do we obtain $P(t|\boldsymbol{x})$ from the $y_k^*(\boldsymbol{x})$? We take the exponential, then normalize the result to sum to one:

$$e^{y_k^*(\boldsymbol{x})} = P(t=k|\boldsymbol{x})e^{C(\boldsymbol{x})} \quad \Rightarrow \quad P(t=k|\boldsymbol{x}) = \frac{e^{y_k^*(\boldsymbol{x})}}{\sum_{\tilde{k}} e^{y_{\tilde{k}}^*(\boldsymbol{x})}} = \sigma_k\left(\boldsymbol{y}^*(\boldsymbol{x})\right),$$

$$\sigma_k(\boldsymbol{v}) = \frac{e^{v_k}}{\sum_{\tilde{k}} e^{v_{\tilde{k}}}} = e^{v_k - \operatorname{lsexp}(\boldsymbol{v})}, \quad \operatorname{lsexp}(\boldsymbol{v}) = \log \sum\nolimits_{\tilde{k}} e^{v_{\tilde{k}}}.$$

$\boldsymbol{\sigma}(\boldsymbol{v}) = [\sigma_k(\boldsymbol{v})] \in \Delta_K$ is called *softmax mapping* (or multivariate logit mapping in statistics). Recall from Section 6.5 that Δ_K denotes the simplex of probability distributions over $\{0,\dots,K-1\}$. The softmax mapping is not one-to-one, since $\boldsymbol{\sigma}(\boldsymbol{v}+\alpha\mathbf{1}) = \boldsymbol{\sigma}(\boldsymbol{v})$ for any $\alpha \in \mathbb{R}$, reflecting the fact that we can always add any fixed $C(\boldsymbol{x})$ to our discriminant functions $y_k^*(\boldsymbol{x})$ and still obtain the same posterior probabilities. The softmax mapping is conveniently defined in terms of the `logsumexp` function $\operatorname{lsexp}(\boldsymbol{v})$, a convex function we previously encountered in Section 8.2.1.

Once more, our motivation of the softmax link for multi-way classification is not limited to spherical Gaussian generative models, but holds more generally. At this point, we take the same step as in Section 8.2. We define a *discriminative* model for $P(t|\boldsymbol{x})$ as

$$P(t=k|\boldsymbol{x},\boldsymbol{w}) = \sigma_k\left([y_{\tilde{k}}(\boldsymbol{x};\boldsymbol{w}_k)]\right), \quad y_k(\boldsymbol{x};\boldsymbol{w}_k) = (\boldsymbol{w}_k)^T\boldsymbol{\phi}(\boldsymbol{x}), \quad \boldsymbol{w} = \begin{bmatrix} \boldsymbol{w}_0 \\ \vdots \\ \boldsymbol{w}_{K-1} \end{bmatrix}.$$

Suppose we are given some multi-way classification data $\mathcal{D} = \{(\boldsymbol{x}_i,t_i) \mid i = 1,\dots,n\}$, where $t_i \in \{0,\dots,K-1\}$. What is the negative log conditional likelihood $-\log P(\boldsymbol{t}|\boldsymbol{w})$, the generalization of the logistic error function (8.4) to K classes? Recall from Section 6.5 that the use of *indicators* can simplify expressions substantially. With this in mind, let us define $\tilde{\boldsymbol{t}}_i = [\tilde{t}_{ik}] \in \{0,1\}^K$, where $\tilde{t}_{ik} = \mathrm{I}_{\{t_i=k\}}$, or $\tilde{\boldsymbol{t}}_i = \boldsymbol{\delta}_{t_i}$. $\tilde{\boldsymbol{t}}_i$ has a one in component t_i, zeros elsewhere. The representation of $\{t_i\}$ in terms of the $\tilde{\boldsymbol{t}}_i$ is called *1-of-K coding.* Using the techniques from Section 6.5:

$$P(t_i|\boldsymbol{x}_i,\boldsymbol{w}) = P(t_i|\boldsymbol{y}_i) = \prod_{k=0}^{K-1} \sigma_k(y_{ik})^{\tilde{t}_{ik}} = \exp\left(\sum_{k=0}^{K-1} \tilde{t}_{ik}y_{ik} - \operatorname{lsexp}(\boldsymbol{y}_i)\right),$$

$$\boldsymbol{y}_i = [y_{ik}], \quad y_{ik} = y_k(\boldsymbol{x}_i;\boldsymbol{w}_k).$$

In this derivation, we used that $\sum_k \tilde{t}_{ik} = 1$. The negative log conditional likelihood, or K-way logistic error function, is

$$-\log P(\boldsymbol{t}|\boldsymbol{w}) = E_{\log}(\boldsymbol{w}) = \sum_{i=1}^{n} \underbrace{\left\{\operatorname{lsexp}(\boldsymbol{y}_i) - (\tilde{\boldsymbol{t}}_i)^T\boldsymbol{y}_i\right\}}_{=:E_i(\boldsymbol{w})}, \quad \boldsymbol{y}_i = [y_k(\boldsymbol{x}_i;\boldsymbol{w}_k)].$$

This is still a sum of independent terms $E_i(\boldsymbol{w})$, one for each data point $(\boldsymbol{x}_i,t_i)$, but it couples the entries of each $\boldsymbol{y}_i \in \mathbb{R}^K$. It is also a convex function, due to the convexity of $\operatorname{lsexp}(\cdot)$. We can understand the structure of this error function by noting that $\operatorname{lsexp}(\boldsymbol{v})$ is a "soft" approximation to $\max_k v_k$:

$$\log\sum_k e^{v_k} = M + \log\sum_k e^{v_k - M} \in (M, M+\log K], \quad M = \max_k v_k.$$

Namely, $v_k - M \le 0$, so that $e^{v_k - M} \le 1$, while $\sum_k e^{v_k - M} > 1$, since $v_k = M$ for at least one k. The approximation is close if $\max_k v_k$ is larger than the other entries by some margin, and in this case $\boldsymbol{\sigma}(\boldsymbol{v}) \approx \boldsymbol{\delta}_{\mathrm{argmax}_k v_k}$, which explains the "softmax" nomenclature. Therefore,

$$E_i(\boldsymbol{w}) = \mathrm{lsexp}(\boldsymbol{y}_i) - (\tilde{\boldsymbol{t}}_i)^T \boldsymbol{y}_i \approx \max_k y_{ik} - y_{i(t_i)}.$$

The error is close to zero if $y_{i(t_i)} = \max_k y_{ik}$, i.e. if $(\boldsymbol{x}_i, t_i)$ is classified correctly, while an error is penalized linearly by the distance between predicted $\max_k y_{ik}$ and desired $y_{i(t_i)}$.

Gradient of Multi-Way Logistic Error

In order to minimize $-\log P(\boldsymbol{t}|\boldsymbol{w})$, we need to work out its gradient w.r.t. $\boldsymbol{w}$. To this end, we use a remarkable property of $\mathrm{lsexp}(\boldsymbol{v})$:

$$\frac{\partial}{\partial v_k} \mathrm{lsexp}(\boldsymbol{v}) = \frac{e^{v_k}}{\sum_{\tilde{k}} e^{v_{\tilde{k}}}} = \sigma_k(\boldsymbol{v}) \quad \Rightarrow \quad \nabla_{\boldsymbol{v}} \mathrm{lsexp}(\boldsymbol{v}) = \boldsymbol{\sigma}(\boldsymbol{v}).$$

Therefore,

$$\nabla_{\boldsymbol{y}_i} E_i = \boldsymbol{\sigma}(\boldsymbol{y}_i) - \tilde{\boldsymbol{t}}_i,$$

an expression which is precisely analogous to what we found in the binary classification case. Using the chain rule,

$$\nabla_{\boldsymbol{w}_k} E_{\mathrm{log}} = \sum_{i=1}^{n} (\sigma_k(\boldsymbol{y}_i) - \tilde{t}_{ik}) \nabla_{\boldsymbol{w}_k} y_{ik} = \sum_{i=1}^{n} (\sigma_k(\boldsymbol{y}_i) - \tilde{t}_{ik}) \boldsymbol{\phi}(\boldsymbol{x}_i).$$

Note that

$$\sum_{k=0}^{K-1} \nabla_{\boldsymbol{w}_k} E_i = \left\{ \sum\nolimits_{k=0}^{K-1} (\sigma_k(\boldsymbol{y}_i) - \tilde{t}_{ik}) \right\} \boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{0},$$

since $\sum_k \sigma_k(\boldsymbol{y}_i) = \sum_k \tilde{t}_{ik} = 1$. The contribution $\boldsymbol{\phi}(\boldsymbol{x}_i)$ is distributed between the gradients $\nabla_{\boldsymbol{w}_k} E_{\mathrm{log}}$ in a zero-sum fashion: a positive example for class t_i, a negative example for all other classes $k \ne t_i$.

Let us apply our new found K-way logistic error function to training a multilayer perceptron for K-way classification. We need K output layer activations $a_k^{(L)}(\boldsymbol{x})$, one for each class. Also, $\boldsymbol{a}^{(L)}(\boldsymbol{x}) = [a_k^{(L)}(\boldsymbol{x})]$. Much like in Section 8.2.1, we only have to modify error backpropagation in a minor way. Fix pattern $(\boldsymbol{x}_i, t_i)$. The output residuals form a vector

$$\boldsymbol{r}_i^{(L)} = \boldsymbol{\sigma}\left(\boldsymbol{a}^{(L)}(\boldsymbol{x}_i)\right) - \tilde{\boldsymbol{t}}_i$$

now. The backward pass starting at output activation $a_k^{(L)}(\boldsymbol{x}_i)$ is seeded with the error $r_{i,k}^{(L)}$.

8.3.2 Conditional Maximum A-Posteriori Estimation (*)

In this chapter, we found novel interpretations of error function minimization, such as least squares estimation, in terms of conditional likelihood. Recall from Chapter 7 that such procedures can run into over-fitting problems, which prevents them from generalizing well to unseen test data. A general remedy, *regularization*, was found to alleviate over-fitting artefacts substantially. An example is Tikhonov regularized least squares estimation, where a penalty term $(\beta/2)\|\boldsymbol{w}\|^2$ is added to the squared error function. As seen in Section 7.3, regularization has a clean probabilistic interpretation as maximum a-posteriori (MAP) estimation in *generative* models which would normally give rise to *joint* maximum likelihood plug-in rules. But what about regularized LSE?

In this section, we close the circle by showing how regularization in *conditional* maximum likelihood for *discriminative* models can be interpreted as conditional maximum a-posteriori estimation. Recall our reformulation of LSE in Section 8.1.2 as conditional likelihood maximization, where $p(\boldsymbol{t}|\boldsymbol{w}) = N(\boldsymbol{t}|\boldsymbol{y}, \sigma^2\boldsymbol{I})$, where $y_i = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)$, or $\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{w}$. In the spirit of Section 7.3, the Tikhonov regularizer $(\beta/2)\|\boldsymbol{w}\|^2$ corresponds to a Gaussian *prior distribution* on the weight vector, $p(\boldsymbol{w}) = N(\boldsymbol{w}|\boldsymbol{0}, \beta^{-1}\boldsymbol{I})$. The posterior distribution is

$$p(\boldsymbol{w}|\boldsymbol{t}) \propto p(\boldsymbol{t}|\boldsymbol{w})p(\boldsymbol{w}),$$

and the conditional MAP estimator is its mode:

$$\hat{\boldsymbol{w}}_{\text{MAP}} = \underset{\boldsymbol{w}}{\operatorname{argmax}}\, p(\boldsymbol{w}|\boldsymbol{t}) = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left\{-\log p(\boldsymbol{t}|\boldsymbol{w}) - \log p(\boldsymbol{w})\right\}.$$

You should confirm for yourself that the second expression is σ^{-2} times the regularized least squares criterion (7.1) up to a constant, if $\nu = \beta\sigma^2$. The probabilistic conditional MAP interpretation of Tikhonov-regularized least squares is as follows. Or model assumptions are twofold. First, the noise is additive Gaussian with variance σ^2. Second, the clean curve $y(\boldsymbol{x})$ is a linear function of weights $\boldsymbol{w}$ with a Gaussian prior $N(\boldsymbol{w}|\boldsymbol{0}, \beta^{-1}\boldsymbol{I})$, which keeps the weights uniformly small.

MAP for Logistic Regression

MAP estimation is useful for logistic regression just as well. In fact, let us consider what happens with the logistic error function (8.4) for a linearly separable dataset $\mathcal{D}$. In this case, there exists some weight vector $\boldsymbol{w}_1$ so that $t_i y(\boldsymbol{x}_i; \boldsymbol{w}_1) = t_i(\boldsymbol{w}_1)^T\boldsymbol{\phi}(\boldsymbol{x}_i) > 0$ for all $i = 1, \dots, n$. But then,

$$\sigma\left(t_i y(\boldsymbol{x}_i; \alpha\boldsymbol{w}_1)\right) = \sigma\left(\alpha t_i(\boldsymbol{w}_1)^T\boldsymbol{\phi}(\boldsymbol{x}_i)\right) \to 1 \;\; (\alpha \to \infty),$$

and $E_{\log}(\alpha\boldsymbol{w}_1) \to 0$ as $\alpha \to \infty$. While the logistic error is lower bounded by zero, the only way to converge against zero is to scale $\|\boldsymbol{w}\|$ ever larger. Changing α leaves the separating hyperplane invariant, but $P(t|\boldsymbol{x}, \boldsymbol{w})$ viewed along the direction $\boldsymbol{w}_1$ converges to a hard step function as $\alpha \to \infty$, a telltale sign of over-fitting. A simple remedy is to do MAP estimation instead, using a Gaussian prior $p(\boldsymbol{w}) = N(\boldsymbol{0}, \beta^{-1}\boldsymbol{I})$. Having to pay for large $\|\boldsymbol{w}\|$, the fitting method will refrain from over-saturating the posterior probabilities. As noted in Section 7.2.2, a beneficial side effect of employing a regularizer (or prior distribution) is to improve the conditioning of the underlying optimization problem. This holds both for regularized LSE, where the quadratic function becomes “more positive definite”, and for penalized logistic regression, where the regularizer improves the condition number of the Hessian matrices (Section 8.2.4). For multi-way logistic regression (Section 8.3.1), it is common practice to use an independent Gaussian prior

$$p(\boldsymbol{w}) = \prod_{k=0}^{K-1} p(\boldsymbol{w}_k) = \prod_{k=0}^{K-1} N(\boldsymbol{w}_k | \boldsymbol{0}, \beta_k^{-1} \boldsymbol{I}).$$

Just like the variance σ^2 of the Gaussian noise model, the prior parameter β is a hyperparameter (Section 8.1.3). Choosing good hyperparameter values is an instance of model selection (see Chapter 10).

Techniques: Gaussian Posterior Distribution (*)

By relating conditional MAP with regularized least squares estimation, we convinced ourselves that the posterior distribution $p(\boldsymbol{w}|\boldsymbol{t}) \propto p(\boldsymbol{t}|\boldsymbol{w})p(\boldsymbol{w})$ has the mode (point of maximum density)

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi} + \nu \boldsymbol{I}\right)^{-1} \boldsymbol{\Phi}^T \boldsymbol{y},$$

the minimizer of the regularized squared error (7.1). But what is the posterior distribution? Let us fill in a hole we left in our survey of the Gaussian in Section 6.3. First,

$$p(\boldsymbol{w}|\boldsymbol{t}) \propto p(\boldsymbol{t}|\boldsymbol{w})p(\boldsymbol{w}) \propto \exp\left(-\frac{1}{2\sigma^2}\|\boldsymbol{t} - \boldsymbol{\Phi}\boldsymbol{w}\|^2 + \frac{\beta}{2}\|\boldsymbol{w}\|^2\right).$$

Recall our timesaving trick from Section 7.3: when working out a distribution (or density) over $\boldsymbol{w}$, we ignore all multiplicative terms which do not depend on $\boldsymbol{w}$. Note the absence of 2π and determinant terms in our derivation. Writing $\nu = \beta\sigma^2$, we have that $p(\boldsymbol{w}|\boldsymbol{t}) \propto e^{-\frac{1}{2\sigma^2}q(\boldsymbol{w})}$, where

$$q(\boldsymbol{w}) = \|\boldsymbol{t} - \boldsymbol{\Phi}\boldsymbol{w}\|^2 + \nu\|\boldsymbol{w}\|^2 = \boldsymbol{w}^T \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi} + \nu \boldsymbol{I}\right) \boldsymbol{w}^T - 2\boldsymbol{t}^T \boldsymbol{\Phi}\boldsymbol{w} + C.$$

C is some constant, we could work it out, but we don’t, as it does not depend on $\boldsymbol{w}$. In the sequel, instead of writing C_1, C_2, … for all sorts of constants we don’t care about anyway, we call all of them C (even though the value may change from line to line). Now, $q(\boldsymbol{w})$ is a quadratic function, and we begin to suspect that maybe $p(\boldsymbol{w}|\boldsymbol{t})$ is Gaussian after all. Indeed, if $\boldsymbol{\Sigma} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi} + \nu \boldsymbol{I})^{-1}$, then

$$\begin{aligned} q(\boldsymbol{w}) &= \boldsymbol{w}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{w} - 2\hat{\boldsymbol{w}}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{w} + C = (\boldsymbol{w} - \hat{\boldsymbol{w}})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{w} - \hat{\boldsymbol{w}}) + C \\ \Rightarrow\ p(\boldsymbol{w}|\boldsymbol{t}) &\propto e^{-\frac{1}{2\sigma^2}q(\boldsymbol{w})} \propto N\left(\boldsymbol{w} \,\middle|\, \hat{\boldsymbol{w}}, \sigma^2 \boldsymbol{\Sigma}\right). \end{aligned}$$

We first completed the square (Section 6.4.3), then matched the result to the form of a Gaussian density (6.2). If we find such a match, we can just read off mean and covariance matrix. The posterior distribution is a Gaussian, its mode $\hat{\boldsymbol{w}}$ is also its mean (we already know that from Section 6.3), and its covariance matrix is $\sigma^2\boldsymbol{\Sigma}$. In other words, a Gaussian prior $p(\boldsymbol{w})$ is conjugate for a Gaussian likelihood $p(\boldsymbol{t}|\boldsymbol{w}) = N(\boldsymbol{t}|\boldsymbol{w}, \sigma^2\boldsymbol{I})$ (Section 7.3.1). One more closedness property for the amazing Gaussian family:

- Closed under full-rank affine linear transformations (Section 6.3), in particular under marginalization.
- Closed under conditioning and Bayes formula (this section).

This list is by no means exhaustive. If you recall the sum and product rule from Section 5.1, you note that all these operations keep us within the Gaussian family, which is one reason why it is so frequently used in practice. An example which throws you out of the family: if $x \sim N(0, 1)$, then x^2 is not a Gaussian random variable.

Chapter 9

Support Vector Machines

In this chapter, we explore kernel methods for binary classification, first and foremost the support vector machine. These are nonparametric methods, like nearest neighbour (Section 2.1), which can be regarded as regularized linear methods in huge, even infinite-dimensional feature spaces. We first encountered the margin of a dataset in Chapter 2, in the context of the perceptron algorithm. In this chapter, we will gain a deeper understanding of this concept.

This chapter is mathematically somewhat more demanding than previous ones, but we will move slowly and build intuition. The work will be very worthwile. Support vector machines are not just among the most powerful "black box" classifiers, with high impact[1] on many applications, they also seeded a link between machine learning and convex optimization which has been and continues to be enormously fruitful, far beyond SVMs.

9.1 Maximum Margin Perceptron Learning

In this chapter, we will be concerned with binary classification throughout. Our goal will be to train a classifier on data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$, where $t_i \in \{-1, +1\}$. In this section, we will concentrate on linear classifiers $f(\boldsymbol{x}) = \mathrm{sgn}(y(\boldsymbol{x}))$, $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + b$. Note that we make the bias parameter explicit in this chapter, for reasons that will become clear soon. We will also assume that $\mathcal{D}$ is linearly separable: there exists some $\boldsymbol{w}$, b, so that $t_i y(\boldsymbol{x}_i) > 0$ for all $i = 1, \dots, n$. We will relax both assumptions in sections to come, so the end result will be a powerful nonlinear classification method, which can tolerate training errors in the interest of better generalization: the support vector machine (SVM).

The story starts with a closer look at the perceptron learning algorithm. This is a good point to revisit Section 2.3, and in particular Section 2.3.3. We know that the perceptron algorithm converges finitely whenever $\mathcal{D}$ is linearly separable,

[1] They are easily en par with neural networks in that respect. SVMs are far easier to use and are based on much more robust training algorithms, properties which are highly appreciated in most application fields.

which is equivalent to a positive margin $\gamma_{\mathcal{D}}$. Quite literally, the margin quantifies the room to move between different separating hyperplanes. The perceptron convergence theorem (Theorem 2.1) bounds the number of updates in terms of $1/\gamma_{\mathcal{D}}^2$. Intuitively, the larger the fraction of separating hyperplanes among all hyperplanes, the easier it is to find one of the former. However, the perceptron algorithm stops once it finds *any* separating hyperplane. If $\gamma_{\mathcal{D}} > 0$, there are infinitely many such solutions. Which of them is the best? Since Chapter 7 we know that the answer to this question is not realizable, requiring knowledge of the "true" data distribution, while all we have is the data $\mathcal{D}$.

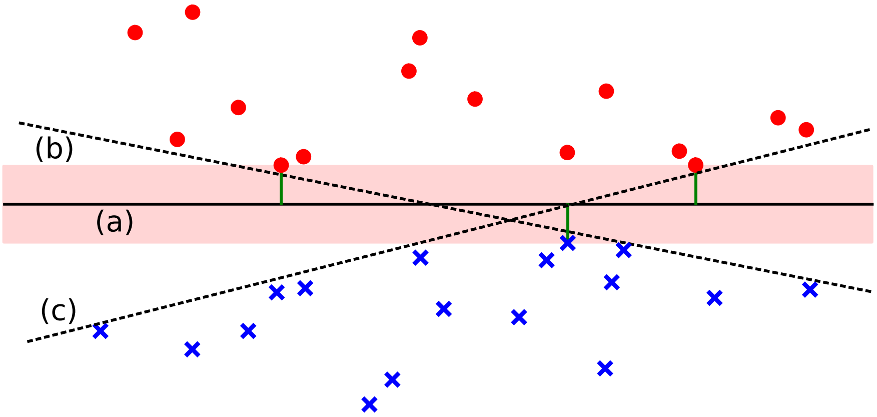


Figure 9.1: Different separating hyperplanes for binary classification data. (a) is the largest margin hyperplane, while (b) and (c) are potential solutions for the perceptron learning algorithm.

Fine, but ask yourself which of the solutions in Figure 9.1 you would pick if you had to. Would you go for (b) or (c), sporting tiny distances to some of the datapoints? Or would you choose (a), which allows for maximum room to move? The principle behind our intuitive preference for (a) is as follows. Our basic assumption about $\mathcal{D}$ is that it is an i.i.d. random sample. If we could repeat the sampling process, we would get another set $\mathcal{D}'$ whose overall structure resembled that of $\mathcal{D}$, but details would be different. With this in mind, it makes sense to search for a discriminant function exhibiting *stability* to small changes in the individual data points. For example, suppose that each pattern $\boldsymbol{x}_i$ is slightly displaced, leading to equally slight displacements of the $\boldsymbol{\phi}(\boldsymbol{x}_i)$. A stable solution would in general remain unchanged. Given this particular notion of stability, the best hyperplane is the one whose distance to the nearest training pattern is maximum. We should look for the discriminant function with maximum margin.

Problem: Among many separating hyperplanes, which one shall I choose?
Approach: The hyperplane with largest margin $\gamma_{\mathcal{D}}(\boldsymbol{w}, b)$ exhibits maximum stability against small $\boldsymbol{x}_i$ displacements.

For much of this chapter, we will use a slightly redefined version of the margin for unnormalized patterns (Section 2.3.3):

$$\gamma_{\mathcal{D}}(\boldsymbol{w}, b) = \min_{i=1,\dots,n} \frac{t_i(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i) + b)}{\|\boldsymbol{w}\|}. \tag{9.1}$$

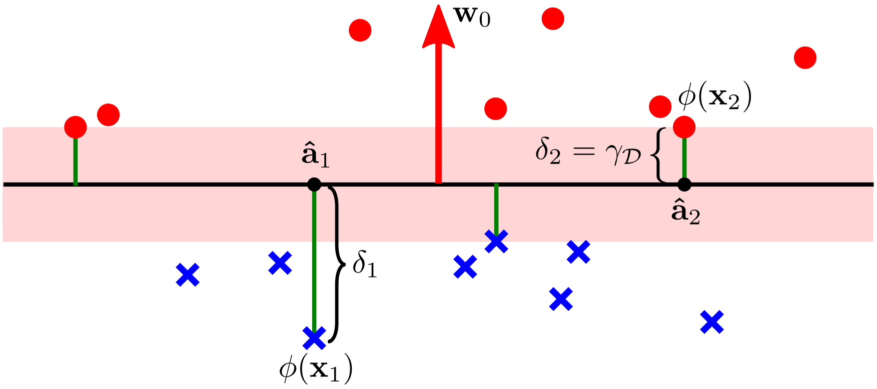


Figure 9.2: The signed distance δ_i of a pattern $\boldsymbol{\phi}(\boldsymbol{x}_i)$ to a hyperplane with unit normal vector $\boldsymbol{w}_0$ is defined by $\boldsymbol{\phi}(\boldsymbol{x}_i) = \hat{\boldsymbol{a}}_i + \delta_i t_i \boldsymbol{w}_0$, where $\hat{\boldsymbol{a}}_i$ is the orthogonal projection of $\boldsymbol{\phi}(\boldsymbol{x}_i)$ onto the hyperplane. The margin of the hyperplane w.r.t. a dataset is the smallest signed distance over all datapoints. The hyperplane is separating the data iff its margin is positive. The maximum margin over all hyperplanes is the margin $\gamma_{\mathcal{D}}$ of the dataset.

Recall the geometrical picture of the margin from Section 2.3.3 and Figure 9.2. For any i, $t_i(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)+b)/\|\boldsymbol{w}\|$ is the signed distance δ_i of $\boldsymbol{\phi}(\boldsymbol{x}_i)$ from the hyperplane (negative if the point is misclassified). Namely, let $\hat{\boldsymbol{a}}_i$ be the orthogonal projection of $\boldsymbol{\phi}(\boldsymbol{x}_i)$ onto the hyperplane. Then, $\boldsymbol{w}^T\hat{\boldsymbol{a}}_i + b = 0$, since $\hat{\boldsymbol{a}}_i$ is on the plane. Moreover, to get to $\boldsymbol{\phi}(\boldsymbol{x}_i)$, we march to $\hat{\boldsymbol{a}}_i$, then move δ_i along the unit normal vector $\boldsymbol{w}_0 = \boldsymbol{w}/\|\boldsymbol{w}\|$: $\boldsymbol{\phi}(\boldsymbol{x}_i) = \hat{\boldsymbol{a}}_i + \delta_i t_i \boldsymbol{w}_0$. Multiplying with $\boldsymbol{w}^T$:

$$\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{w}^T\hat{\boldsymbol{a}}_i + \delta_i t_i\|\boldsymbol{w}\| = -b + \delta_i t_i \|\boldsymbol{w}\| \quad \Leftrightarrow \quad \delta_i = \frac{t_i(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)+b)}{\|\boldsymbol{w}\|}.$$

Here, we used $\boldsymbol{w}^T\boldsymbol{w}_0 = \|\boldsymbol{w}\|$ and $1/t_i = t_i$. The margin is the *smallest signed distance to any of the training cases.* If $(\boldsymbol{w}, b)$ describes a separating hyperplane, you can remove “signed”. A difference[2] to the margin concept in Section 2.3.3 is that here, we normalize $\boldsymbol{w}$ only, but leave b unregularized. Moreover, we do not insist on normalizing the feature vectors $\boldsymbol{\phi}(\boldsymbol{x}_i)$ here, even though this is typically done in SVM practice (see end of Section 9.2.3).

The *maximum margin perceptron learning* problem (also known as optimal perceptron learning problem) is given by

$$\max_{\boldsymbol{w},b}\left\{\gamma_{\mathcal{D}}(\boldsymbol{w},b) = \min_{i=1,\dots,n}\frac{t_i(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)+b)}{\|\boldsymbol{w}\|}\right\}. \tag{9.2}$$

We maximize the minimum margin per pattern, where the minimum is over the data points, the maximum over the hyperplane. The solution to this problem is the margin $\gamma_{\mathcal{D}}$ for the dataset $\mathcal{D}$. As in Section 2.3.3, we can interpret $2\gamma_{\mathcal{D}}$ as the maximum width of a slab of parallel separating hyperplanes in between datapoints of the two classes (the light-red region in Figure 9.2). Note that at least one degree of freedom is left unspecified in this problem. If $(\boldsymbol{w}_*, b_*)$ is a solution, then so is $(\beta\boldsymbol{w}_*, \beta b_*)$ for any $\beta > 0$.

[2] For the margin concept used in this chapter, we can translate all datapoints by a common

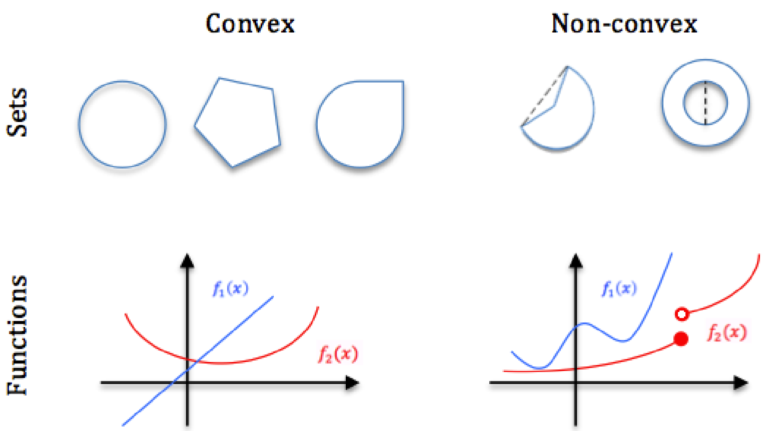


Figure 9.3: Examples of convex and non-convex sets (top panel), as well as convex and non-convex functions (bottom panel).

9.1.1 A Convex Optimization Problem

A basic understanding of convex sets, functions, and convex optimization is essential for working in machine learning today, whether in research or applications. The book of Boyd and Vandenberghe [8] is highly recommended, it will help you develop geometrical intuition, and its many examples convey the breadth of convex optimization in applications today. You should certainly read[3] Sections 2.1 and 3.1. A set $\mathcal{S} \subset \mathbb{R}^p$ is convex if all points on the line segment between any two $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{S}$ are contained in $\mathcal{S}$: $\lambda\boldsymbol{a} + (1-\lambda)\boldsymbol{b} \in \mathcal{S}$ for any $\lambda \in [0,1]$. A real-valued function $f : \mathcal{S} \to \mathbb{R}$ is convex if its domain $\mathcal{S}$ is convex, and if

$$f(\lambda\boldsymbol{a} + (1-\lambda)\boldsymbol{b}) \le \lambda f(\boldsymbol{a}) + (1-\lambda) f(\boldsymbol{b}), \quad \boldsymbol{a}, \boldsymbol{b} \in \mathcal{S},\ \lambda \in [0,1].$$

If you plot the function, then the line segment between $f(\boldsymbol{a})$ and $f(\boldsymbol{b})$ lies permanently above the curve $f(\lambda\boldsymbol{a} + (1-\lambda)\boldsymbol{b})$, $\lambda \in [0,1]$. Examples of convex (non-convex) sets and functions are given in Figure 9.3. Convex functions can be seen as generalizations of linear functions (which are convex), where "$\le$" would be "$=$". We already met convex functions several times in this course. The most elementary class of convex functions beyond linear ones are quadratics with positive semidefinite covariance matrix (Section 6.3, Section 4.2.2). More general, if a function $f(\boldsymbol{x})$ is twice differentiable everywhere, then it is convex if and only if its Hessian $\nabla\nabla_{\boldsymbol{x}} f$ is positive semidefinite everywhere. Both the logistic and the perceptron error function are convex (Chapter 8), the latter is an example for a convex function which is not differentiable everywhere. Maybe the most important practical consequence of convexity is that every local minimum point of $f(\boldsymbol{x})$ must be a global minimum point as well (you should prove this for yourself). Finally, a *convex optimization problem* has the form

$$\min_{\boldsymbol{x} \in \mathcal{S}} f(\boldsymbol{x}),$$

offset vector without changing the value of $\gamma_{\mathcal{D}}$.

[3] The book is available online at `www.stanford.edu/~boyd/cvxbook.html`.

where $\mathcal{S} \subset \mathbb{R}^p$ is convex and $f(\boldsymbol{x})$ is a convex function on $\mathcal{S}$.

The maximum margin perceptron learning problem (9.2) is not convex as it stands, since the criterion $\gamma_{\mathcal{D}}(\boldsymbol{w}, b)$ is not convex. Our goal in this section is to show that a version of this problem corresponds to a convex optimization problem with a unique solution.

The first step is to eliminate the minimum over $i = 1, \dots, n$ by introducing linear constraints. To this end, we introduce another variable $\tilde{\gamma}$ and rewrite (9.2) as

$$\max_{\boldsymbol{w}, b, \tilde{\gamma}} \frac{\tilde{\gamma}}{\|\boldsymbol{w}\|}, \quad \text{subj. to } t_i(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq \tilde{\gamma}, \quad i = 1, \dots, n.$$

Make sure to understand why the two problems are equivalent before moving on. In fact, for fixed $(\boldsymbol{w}, b)$, the optimum choice is $\tilde{\gamma} = \min_i t_i(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b)$. At this point, we play out the card of being able to rescale $[\boldsymbol{w}^T, b]^T$ without changing anything. This means that there is one parameter too much, which can be fixed to an arbitrary positive value. Let us simply choose $\tilde{\gamma}$ to be that parameter, and fix $\tilde{\gamma} = 1$.

$$\max_{\boldsymbol{w}, b} \frac{1}{\|\boldsymbol{w}\|}, \quad \text{subj. to } t_i(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq 1, \quad i = 1, \dots, n.$$

Finally, instead of maximizing $1/\|\boldsymbol{w}\|$, we can just as well minimize the quadratic $(1/2)\|\boldsymbol{w}\|^2$. We end up with

$$\min_{\boldsymbol{w}, b} \frac{1}{2}\|\boldsymbol{w}\|^2, \qquad \text{subj. to } t_i(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq 1, \quad i = 1, \dots, n. \tag{9.3}$$

This is a convex optimization problem. For one, $\|\boldsymbol{w}\|^2$ is a convex function. Moreover, the constraints determine the set to be optimized over, called the *feasible set*. Each such affine constraint defines a halfspace in $\mathbb{R}^{p+1}$ where $[\boldsymbol{w}^T, b]^T$ lives (Section 2.2), and the intersection of halfspaces, if not empty, defines a convex set (please prove this for yourself). Why is the feasible set not empty? Because we assume that the dataset $\mathcal{D}$ is linearly separable.

Problem: Finding the largest margin hyperplane is not a convex problem.
Approach: Convert problem into an equivalent convex quadratic program.
Key step: Fix scale of $(\boldsymbol{w}_*, b_*)$, whose size does not matter.

Our convex problem (9.3) is an example of a *quadratic program*. You may be familiar with linear programs, which sport linear criteria to be minimized w.r.t. linear constraints (the feasible set being a convex polytope, a bounded intersection of linear halfspaces). Quadratic programs are defined by a positive semidefinite quadratic criterion subject to linear constraints. We will defer the question of how to solve (9.3) to Section 9.3. First, we will be concerned with lifting the simplifying assumptions made at the beginning of this section.

9.2 Support Vector Machines

In the previous section, we turned the idea of stability and maximum margin perceptron learning into a convex optimization problem (9.3). However, this formulation has two shortcomings shared with the perceptron algorithm. First, it works only for linearly separable datasets $\mathcal{D}$. Second, we are limited to linear discriminants $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + b$ in some finite-dimensional feature space, $\boldsymbol{w} \in \mathbb{R}^p$. In this section, we will learn to know remedies for both shortcomings. At the end, we will not only have derived support vector machines, but also gained a much broader perspective on "kernel" variants of regularized estimation.

9.2.1 Soft Margins

How can we modify (9.3) in order to tolerate classification errors, while penalizing them? It is helpful to look at methods which already provide this feature, such as for example logistic regression (Section 8.2). Writing $y_i = y(\boldsymbol{x}_i) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b$, the constraints in (9.3) are $t_i y_i \geq 1$. Compare this to the logistic error per case in terms of $t_i y_i$ (Figure 9.5). For $t_i y_i \geq 1$, this is fairly close to zero, while it grows linearly with $-t_i y_i$. Maybe we should relax the hard constraint $t_i y_i \geq 1$ into something which implies a penalty linear in $(1 - t_i y_i)$. To do so, we introduce nonnegative *slack variables* $\xi_i \geq 0$, one for each pattern, giving rise to the *maximum soft margin perceptron learning* problem

$$\begin{aligned} &\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n} \xi_i, \quad \boldsymbol{\xi} = [\xi_i], \\ &\text{subj. to } \; t_i(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad i = 1, \dots, n. \end{aligned} \tag{9.4}$$

This is the quadratic program underlying the *support vector machine* (SVM), as devised by Cortes and Vapnik [10]. The parameter $C > 0$ is a hyperparameter, whose choice is an instance of model selection (see Chapter 10). Consider what happens at an optimal solution $(\boldsymbol{w}_*, b_*, \boldsymbol{\xi}_*)$, where we must have $\xi_{*,i} = \max\{1 - t_i y_{*,i}, 0\}$ (why?). If $\xi_{*,i} = 0$, then $t_i y_{*,i} \geq 1$ and the case is classified correctly with a margin, just as before. If $\xi_{*,i} > 0$, the pattern lies within the margin area ($ty(\boldsymbol{x}) < 1$), for $\xi_{*,i} > 1$ it is even misclassified. We pay for this flexibility by $C\xi_{*,i}$ in the criterion value. Note that due to this argument, the added penalty $\sum_i \xi_{*,i}$ can be seen as an upper bound on the number of training errors.

Problem: Margin constraints on *all* patterns can be too stringent.
No solution for linearly non-separable data (outliers).
Approach: Maximize a soft margin instead. Allow margin constraints to be violated, but impose (linear) costs for each violation.

The problem (9.4) is often called *soft margin SVM* problem, while our previous variant (9.3) becomes the *hard margin SVM* problem. Note that the hard margin problem is a special[4] case of the soft margin problem, where $C = \infty$. The soft margin extension can also lead to an improved solution for a linearly separable dataset $\mathcal{D}$. To understand this point, let us define the *soft margin* as $1/\|\boldsymbol{w}_*\|$ for an optimal solution $(\boldsymbol{w}_*, b_*, \boldsymbol{\xi}_*)$ of (9.4), in analogy with Section 9.1.1. Note that this quantity depends on C. If $\mathcal{D}$ is separable, the soft margin for $C = \infty$ coincides with the normal (hard) margin. In geometrical terms, the soft margin is the closest distance to the hyperplane of any pattern i with $\xi_{*,i} = 0$. It

[4] In general, the soft margin solution will coincide with the hard margin solution from some finite $C < \infty$ onwards.

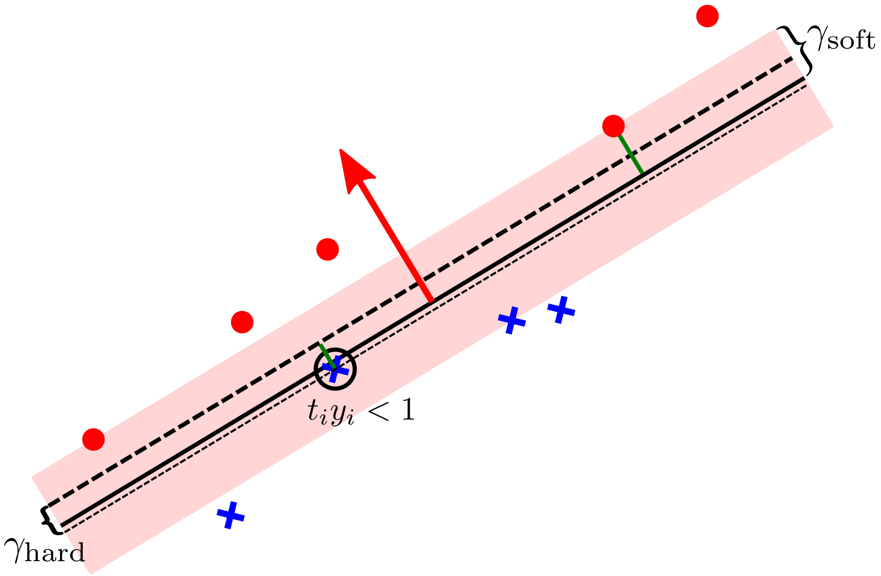


Figure 9.4: The soft margin support vector machine allows for a larger soft margin γ_{soft} at the expense of violating the large margin constraints $t_i y_i \geq 1$ for certain patterns. Different from the hard margin SVM, it can be applied to datasets which are not linearly separable.

is therefore no smaller, and typically larger, than the hard margin. Figure 9.4 illustrates the consequences of optimizing the more general soft margin. Intuitively, we extend the width of our slab (light-red region), thus the stability of the solution, at the expense of engulfing a few patterns. To conclude, the soft margin SVM is more generally useful than its hard margin special case, and we will mainly be concerned with the former in the remainder of this chapter.

The soft margin SVM can be understood as a Tikhonov-regularized estimator, directly related to conditional maximum a-posteriori estimation (Section 8.3.2). To this end, we eliminate the slack variables in (9.4) by using the hinge function $[x]_+ := \max\{x, 0\}$ and divide the criterion by C, arriving at the equivalent problem

$$\min_{\boldsymbol{w},b} \frac{1}{2C}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n}[1 - t_i y_i]_+, \quad y_i = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i) + b.$$

This problem combines the *hinge error function*

$$E_{\text{svm}}(\boldsymbol{w}, b) = \sum_{i=1}^{n}[1 - t_i y_i]_+, \quad y_i = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i) + b,$$

with the Tikhonov regularizer $(2C)^{-1}\|\boldsymbol{w}\|^2$. We can compare E_{svm} with the perceptron and logistic error (Figure 9.5). It corresponds to the perceptron error $[-t_i y_i]_+$ shifted to the right in order to enforce a discrimination with margin. Unlike the logistic error, it is exactly zero for $t_i y_i \geq 1$. The consequences of this fact will be highlighted in Section 9.4.

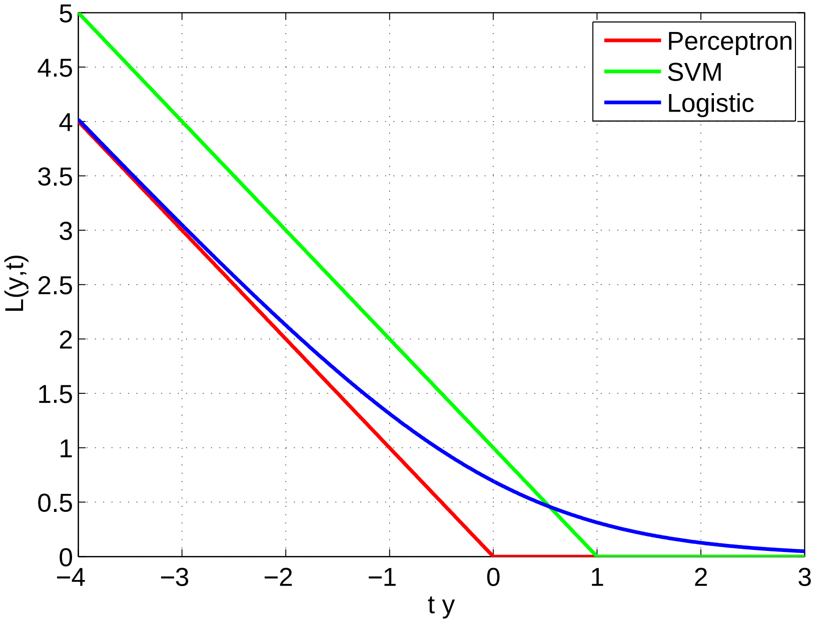


Figure 9.5: Error per pattern i as function of $t_i y_i$, for perceptron, logistic and SVM hinge error.

As a final comment, some readers may be puzzled about the choice of 1 in $[1 - t_i y_i]_+$, why not 2 or 10^{-10}? This choice is indeed arbitrary in the following sense. Suppose we had chosen $\varepsilon > 0$ instead of 1. Then, a solution $(\boldsymbol{w}_*, b_*)$ to the hard margin SVM problem corresponds to the solution $(\boldsymbol{w}_*/\varepsilon, b_*/\varepsilon)$ of (9.3) in the original form. For the soft margin SVM, we also have to scale C. A solution $(\boldsymbol{w}_*, b_*, \boldsymbol{\xi}_*)$ of the ε-modified problem with C is equivalent to a solution $(\boldsymbol{w}_*/\varepsilon, b_*/\varepsilon, \boldsymbol{\xi}_*/\varepsilon)$ of the original problem with C/ε (please confirm these points for yourself). This makes sense, since C has to be measured in the units of $y(\boldsymbol{x})$.

9.2.2 Feature Expansions. Representer Theorem

In this section, we prepare the ground for a powerful nonlinear extension not only of SVMs, but of all other regularized estimation techniques we encountered so far (logistic regression, perceptron), by way of "kernelization". This property is often presented as "magical" or a "trick", while it is a natural property of estimation methods based on linear functions. Consider some regularized estimation problem with $\nu > 0$:

$$\min_{\boldsymbol{w},b} \left\{ E(\boldsymbol{w}, b) = \sum_{i=1}^{n} E_i(t_i, y_i) + \frac{\nu}{2}\|\boldsymbol{w}\|^2 \right\}, \quad y_i = y(\boldsymbol{x}_i) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i) + b. \quad (9.5)$$

We assume that the criterion is lower bounded and has globally optimal solutions. What we will show is that there is always an optimal solution $(\boldsymbol{w}_*, b_*)$ for which we can write

$$\boldsymbol{w}_* = \sum_{i=1}^{n} \alpha_{*,i} \boldsymbol{\phi}(\boldsymbol{x}_i).$$

The weight vector $\boldsymbol{w}_*$ can be written as linear combination of the feature vectors $\boldsymbol{\phi}(\boldsymbol{x}_i)$. In other words, $\boldsymbol{w}_* = \boldsymbol{\Phi}^T \boldsymbol{\alpha}_*$. Writing $\boldsymbol{w}_*$ in terms of $\boldsymbol{\alpha}_*$ is known as *dual representation.* The consequence of this so called *representer theorem* is that we may as well optimize over $\boldsymbol{\alpha}$, with $\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{\alpha}$, without loss in optimality. This statement is trivial for $p \le n$, so for the rest of this section we will assume that $p > n$ (more features than datapoints), and that $\nu > 0$ in (9.5).

We begin with a method which does not use Tikhonov regularization, the perceptron algorithm of Section 2.3. Recall from Algorithm 1 that we start with $\boldsymbol{w} \leftarrow \boldsymbol{0}$, and each update has the form $\boldsymbol{w} \leftarrow \boldsymbol{w} + t_i\boldsymbol{\phi}(\boldsymbol{x}_i)$. This means that we can just as well maintain a dual representation $\boldsymbol{\alpha} \in \mathbb{R}^n$, start with $\boldsymbol{\alpha} \leftarrow \boldsymbol{0}$, and update $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + t_i\boldsymbol{\delta}_i$. The perceptron algorithm is ready for “kernelization”. Note that the dual representation can even be a good idea in this case for $p < n$. If $\mathcal{D}$ has a sizeable margin, the perceptron algorithm will only ever update on a few patterns, and the dual vector $\boldsymbol{\alpha}$ remains sparse, thus can be stored and handled efficiently.

How about linear classification by minimizing the squared error E_{sq}? Consider the stochastic gradient variant from Section 2.4.2. Upon visiting pattern i, we update

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}E_i = \boldsymbol{w} - \eta(y_i - t_i)\boldsymbol{\phi}(\boldsymbol{x}_i),$$

which again gives rise to a dual representation, in which we update $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \eta(y_i - t_i)\boldsymbol{\delta}_i$. We derive the underlying dual optimization problem at the end of this section.

How about *any* problem of the form (9.5)? If you read Section 4.2.2 on orthogonal projection, you have all the tools together. The criterion (9.5) consists of two parts. The error function depends on $(\boldsymbol{w}, b)$ through $\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{w} + b\boldsymbol{1}$ only, while the regularizer is $(\nu/2)\|\boldsymbol{w}\|^2$. We will show the following. For any $(\boldsymbol{w}, b)$ giving rise to $\boldsymbol{y}$, we can find some $\tilde{\boldsymbol{w}} = \boldsymbol{\Phi}^T\boldsymbol{\alpha}$ so that $\boldsymbol{\Phi}\tilde{\boldsymbol{w}} + b\boldsymbol{1} = \boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{w} + b\boldsymbol{1}$ and $\|\tilde{\boldsymbol{w}}\|^2 \le \|\boldsymbol{w}\|^2$. We just need to apply this to an optimal solution in order to establish the representer theorem. Recall from Section 4.2.2 that $\boldsymbol{w} = \boldsymbol{w}_\| + \boldsymbol{w}_\perp$ uniquely, where $\boldsymbol{w}_\| \in \boldsymbol{\Phi}^T\mathbb{R}^n$ (therefore $\boldsymbol{w}_\| = \boldsymbol{\Phi}^T\boldsymbol{\alpha}$ for some $\boldsymbol{\alpha} \in \mathbb{R}^n$) and $\boldsymbol{w}_\perp$ is orthogonal to $\boldsymbol{\Phi}^T\mathbb{R}^n$, meaning that $\boldsymbol{\Phi}\boldsymbol{w}_\perp = \boldsymbol{0}$. Therefore,

$$\boldsymbol{y} = \boldsymbol{\Phi}\left(\boldsymbol{w}_\| + \boldsymbol{w}_\perp\right) + b\boldsymbol{1} = \boldsymbol{\Phi}\boldsymbol{w}_\| + b\boldsymbol{1}.$$

Moreover, by the Pythagorean theorem (Section 2.1.1):

$$\|\boldsymbol{w}\|^2 = \|\boldsymbol{w}_\|\|^2 + \|\boldsymbol{w}_\perp\|^2 \ge \|\boldsymbol{w}_\|\|^2.$$

The claim follows with $\tilde{\boldsymbol{w}} = \boldsymbol{w}_\|$. The intuition behind the representer theorem is that the contribution $\boldsymbol{w}_\perp$ orthogonal to $\boldsymbol{w}_\|$ cannot help in decreasing the error (it does not affect $\boldsymbol{y}$), but it adds to the regularization costs.

To conclude, for a wide range of Tikhonov[5] regularized estimation problems of the form (9.5), including penalized least squares, MAP estimation for logistic regression (Section 8.3.2) and soft margin support vector machines, the optimal solution can be represented in terms of $\boldsymbol{w}_* = \boldsymbol{\Phi}^T\boldsymbol{\alpha}_* = \sum_{i=1}^n \alpha_{*,i}\boldsymbol{\phi}(\boldsymbol{x}_i)$. This fact is valuable if $p > n$, as it allows us to optimize over $(\boldsymbol{\alpha}, b)$ instead of $(\boldsymbol{w}, b)$.

[5] You will have no problems to confirm that the representer theorem holds more generally for regularizers of the form $\mathcal{R}(\|\boldsymbol{w}\|)$, where $\mathcal{R}(v)$ is nondecreasing for $v \ge 0$.

Problem: I want to use more features p than datapoints n.
Can I save time and storage?
Approach: You can! The representer theorem allows to optimize over $\boldsymbol{\alpha} \in \mathbb{R}^n$ instead of $\boldsymbol{w} \in \mathbb{R}^p$.

Dual Representation of Linear Least Squares (*)

Let us work out the dual representation for linar least squares regression, namely (9.5) with $E_i = (y_i - t_i)^2/2$. Note that for this example, we deviate from the policy here and include b into $\boldsymbol{w}$, appending 1 to $\boldsymbol{\phi}(\boldsymbol{x})$. The other case leads to the same conclusion, but is more messy to derive. Starting from Section 7.2, we have that

$$\left(\nu \boldsymbol{I} + \boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)\boldsymbol{w}_* = \boldsymbol{\Phi}^T\boldsymbol{y} \quad \Rightarrow \quad \boldsymbol{w}_* = \boldsymbol{\Phi}^T\boldsymbol{\alpha}_*, \quad \boldsymbol{\alpha}_* = \frac{1}{\nu}\left(\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}_*\right).$$

Plugging in $\boldsymbol{\Phi}\boldsymbol{w}_* = \boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\alpha}_*$, then

$$\nu\boldsymbol{\alpha}_* = \boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\alpha}_* \quad \Rightarrow \quad \left(\nu\boldsymbol{I} + \boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)\boldsymbol{\alpha}_* = \boldsymbol{y}.$$

Note the remarkable duality in this formulation. The system for the dual representation $\boldsymbol{\alpha}_*$ is obtained by replacing $\boldsymbol{\Phi}^T\boldsymbol{\Phi} \in \mathbb{R}^{p\times p}$ with $\boldsymbol{\Phi}\boldsymbol{\Phi}^T \in \mathbb{R}^{n\times n}$, and the right hand side $\boldsymbol{\Phi}^T\boldsymbol{y}$ with $\boldsymbol{y}$. Ultimately, this simple classical relationship is the basis for the "kernel trick". We will work out details in Section 9.2.3. This example provides an important lesson about penalized linear regression. If we encounter $p > n$, we solve for $\boldsymbol{\alpha}_*$ (a $n \times n$ linear system) rather than for $\boldsymbol{w}_*$ directly (a $p \times p$ linear system). Computations scale superlinearly only in the smaller of p and n.

Finally, what happens to the representer theorem for unregularized criteria, (9.5) without the Tikhonov regularizer? Should we still use a dual representation for $\boldsymbol{w}$? The answer is yes, although some clarification is needed. If $p > n$ and (9.5) comes without regularizer, then the problem has infinitely many optimal solutions. In particular, we can always add any multiple of $\boldsymbol{w}_\perp$ to $\boldsymbol{w}$ without changing the criterion at all. To see why dual representations are still possible, in fact constitute a very sensible choice, note that (9.5) without regularizer is obtained by setting $\nu = 0$. For any $\nu > 0$, the representer theorem provides an optimal solution $\boldsymbol{w}_*^{(\nu)} = \boldsymbol{\Phi}^T\boldsymbol{\alpha}_*^{(\nu)}$. By continuity, $\boldsymbol{\alpha}_*^{(\nu)}$ converges to $\boldsymbol{\alpha}_*^{(0)}$, and $\boldsymbol{w}_*^{(0)} = \boldsymbol{\Phi}^T\boldsymbol{\alpha}_*^{(0)}$ is a solution of the unregularized problem. Moreover, if $\operatorname{rk}\boldsymbol{\Phi} = n$, then among the infinitely many solutions, $\boldsymbol{w}_*^{(0)}$ is the one with smallest norm $\|\boldsymbol{w}_*\|$, simply because this holds true for any $\nu > 0$. For the regularized LSE example above, we have that

$$\boldsymbol{w}_*^{(0)} = \boldsymbol{\Phi}^T\boldsymbol{\alpha}_*^{(0)} = \boldsymbol{\Phi}^T\left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1}\boldsymbol{y},$$

the so called Moore-Penrose pseudoinverse solution. Beware that computing $\boldsymbol{w}_*^{(0)}$ in practice can be challenging. If you really do not want to regularize your problem at all, you should be familiar with best practice methods discussed in Section 4.2.3.

9.2.3 Kernel Methods

We have all the tools together now to make an exciting step. Let us summarize our findings. We are interested in regularized estimation problems of the form (9.5), where $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + b$ is linear, examples include the soft margin SVM and MAP for logistic regression. Here is a mad idea. Suppose we use a huge number of features p, maybe even infinitely many. Before figuring out how this could be done, let us first see whether this makes any sense in principle. After all, we have to store $\boldsymbol{w} \in \mathbb{R}^p$ and evaluate $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^p$. Do we? In the previous section, we learned that we can always *represent* $\boldsymbol{w} = \boldsymbol{\Phi}^T \boldsymbol{\alpha}$, where $\boldsymbol{\alpha} \in \mathbb{R}^n$, and our dataset is finite. Moreover, the error function in (9.5) depends on

$$\boldsymbol{y} = \boldsymbol{\Phi} \boldsymbol{w} + b = \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\alpha} + b$$

only, and $\boldsymbol{\Phi}\boldsymbol{\Phi}^T$ is just an $\mathbb{R}^{n \times n}$ matrix. Finally, the Tikhonov regularizer is given by

$$\frac{\nu}{2} \|\boldsymbol{w}\|^2 = \frac{\nu}{2} \left\| \boldsymbol{\Phi}^T \boldsymbol{\alpha} \right\|^2 = \frac{\nu}{2} \boldsymbol{\alpha}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\alpha},$$

it also only depends on $\boldsymbol{\Phi}\boldsymbol{\Phi}^T$. Finally, once we are done and found $(\boldsymbol{\alpha}_*, b_*)$, where $\boldsymbol{w}_* = \boldsymbol{\Phi}^T \boldsymbol{\alpha}_*$, we can predict on new inputs $\boldsymbol{x}$ with

$$y^*(\boldsymbol{x}) = \boldsymbol{w}_*^T \boldsymbol{\phi}(\boldsymbol{x}) + b_* = \boldsymbol{\alpha}_*^T \boldsymbol{\Phi} \boldsymbol{\phi}(\boldsymbol{x}) + b_*.$$

We need finite quantities only in order to make our idea work, namely the matrix $\boldsymbol{\Phi}\boldsymbol{\Phi}^T$ during training, and the mapping $\boldsymbol{\Phi}\boldsymbol{\phi}(\boldsymbol{x})$ for predictions later on, to be evaluated at finitely many $\boldsymbol{x}$. This is the basic observation which makes *kernelization* work.

The entries of $\boldsymbol{\Phi}\boldsymbol{\Phi}^T$ are $\boldsymbol{\phi}(\boldsymbol{x}_i)^T \boldsymbol{\phi}(\boldsymbol{x}_j)$, while $[\boldsymbol{\Phi}\boldsymbol{\phi}(\boldsymbol{x})] = [\boldsymbol{\phi}(\boldsymbol{x}_i)^T \boldsymbol{\phi}(\boldsymbol{x})]$. We can write

$$K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}'),$$

a *kernel function*. It is now clear that given the kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$, we never need to access the underlying $\boldsymbol{\phi}(\boldsymbol{x})$. In fact, we can forget about the dimensionality p and vectors of this size altogether. What makes $K(\boldsymbol{x}, \boldsymbol{x}')$ a kernel function? It must be the inner product in some feature space, but what does that imply? Let us work out some properties. First, a kernel function is obviously symmetric: $K(\boldsymbol{x}', \boldsymbol{x}) = K(\boldsymbol{x}, \boldsymbol{x}')$. Second, consider some arbitrary set $\{\boldsymbol{x}_i\}$ of n input points and construct the *kernel matrix* $\boldsymbol{K} = [K(\boldsymbol{x}_i, \boldsymbol{x}_j)] \in \mathbb{R}^{n \times n}$. Also, denote $\boldsymbol{\Phi} = [\boldsymbol{\phi}(\boldsymbol{x}_1), \dots, \boldsymbol{\phi}(\boldsymbol{x}_n)]^T \in \mathbb{R}^{n \times p}$. Then,

$$\boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha} = \boldsymbol{\alpha}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\alpha} = \left\| \boldsymbol{\Phi}^T \boldsymbol{\alpha} \right\|^2 \geq 0.$$

In other words, the kernel matrix $\boldsymbol{K}$ is positive semidefinite (see Section 6.3). This property defines kernel functions. $K(\boldsymbol{x}, \boldsymbol{x}')$ *is a kernel function if the kernel matrix* $\boldsymbol{K} = [K(\boldsymbol{x}_i, \boldsymbol{x}_j)]$ *for any finite set of points* $\{\boldsymbol{x}_i\}$ *is symmetric positive semidefinite.* An important subfamily are the *infinite-dimensional* or *positive definite kernel functions.* A member $K(\boldsymbol{x}, \boldsymbol{x}')$ of this subfamily is defined by all its kernel matrices $\boldsymbol{K} = [K(\boldsymbol{x}_i, \boldsymbol{x}_j)]$ being positive definite for any set $\{\boldsymbol{x}_i\}$ of any size. In particular, all kernel matrices are invertible. As we will see shortly, it is positive definite kernel functions which give rise to infinite-dimensional feature spaces, therefore to nonlinear kernel methods.

Problem: Can I use astronomically many features p? How about $p = \infty$?
Approach: No problem! As long as you can efficiently compute the *kernel function* $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}')$, the representer theorem saves the day.

Hilbert Spaces and All That (*)

Before we give examples of kernel functions, a comment for meticulous readers (all others can safely skip this paragraph and move to the examples). How can we even talk about $\boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x})$ if $p = \infty$? Even worse, what is $\boldsymbol{\Phi} \in \mathbb{R}^{n \times p}$ in this case? In the best case, all this involves infinite sums, which may not converge. Rest assured that all this can be made rigorous within the framework of Hilbert function and functional spaces. In short, infinite dimensional vectors become functions, their transposes become functionals, and matrices become linear operators. A key result is Mercer's theorem for positive semidefinite kernel functions, which provides a construction for a feature map. However, with the exception of certain learning-theoretical questions, the importance of all this function space mathematics for down-to-earth machine learning is very limited. Historically, the point about the efforts of mathematicians like Hilbert, Schmidt and Riesz was to find conditions under which function spaces could be treated in the same simple way as finite-dimensional vector spaces, working out analogies for positive definite matrices, quadratic functions, eigendecomposition, and so on. Moreover, function spaces governing kernel methods are of the particularly simple reproducing kernel Hilbert type, where common pathologies like "delta functions" do not even arise. You may read about all that in [39] or other kernel literature, it will not play a role in this course. Just one warning which you will not find spelled out much in the SVM literature. The "geometry" in huge or infinite-dimensional spaces is dramatically different from anything we can draw or imagine. For example, in Mercer's construction of $K(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j \geq 1} \phi_j(\boldsymbol{x}) \phi_j(\boldsymbol{x}')$, the different feature dimensions $j = 1, 2, \dots$ are by no means on equal terms, as far as concepts like distance or volume are concerned. For most commonly used infinite-dimensional kernel functions, the contributions $\phi_j(\boldsymbol{x}) \phi_j(\boldsymbol{x}')$ rapidly become extremely small, and only a small number of initial features determine most of the predictions. A good intuition about kernel methods is that they behave like (easy to use) linear methods of flexible dimensionality. As the number of data points n grows, a larger (but finite) number of the feature space dimensions will effectively be used.

Examples of Kernel Functions

Let us look at some examples. Maybe the simplest kernel function is $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$, the standard inner product. Moreover, for any finite-dimensional feature map $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^p$ ($p < \infty$), $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}')$ is a kernel function. Since any kernel matrix of this type can at most have rank p, such kernel functions are positive semidefinite, but not positive definite. However, even for finite-dimensional kernels, it can be much simpler to work with $K(\boldsymbol{x}, \boldsymbol{x}')$ directly than to evaluate $\boldsymbol{\phi}(\boldsymbol{x})$. For example, recall polynomial regression estimation from Section 4.1, giving rise to a polynomial feature map $\boldsymbol{\phi}(x) = [1, x, \dots, x^r]^T$ for $x \in \mathbb{R}$. Now, if $\boldsymbol{x} \in \mathbb{R}^d$ is multivariate, a corresponding polynomial feature map would consists of very many features. Is there a way around their explicit representation? Consider the *polynomial kernel*

$$K(\boldsymbol{x}, \boldsymbol{x}') = \left(\boldsymbol{x}^T \boldsymbol{x}'\right)^r = \sum_{j_1,\dots,j_r} \left(x_{j_1} \dots x_{j_r}\right) \left(x'_{j_1} \dots x'_{j_r}\right).$$

For example, if $d = 3$ and $r = 2$, then

$$K(\boldsymbol{x}, \boldsymbol{x}') = \left(x_1 x'_1 + x_2 x'_2 + x_3 x'_3\right)^2 = x_1^2 (x'_1)^2 + x_2^2 (x'_2)^2 + x_3^2 (x'_3)^2 \\ + 2(x_1 x_2)(x'_1 x'_2) + 2(x_1 x_3)(x'_1 x'_3) + 2(x_2 x_3)(x'_2 x'_3),$$

a feature map of which is

$$\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_3^2 \\ \sqrt{2} x_1 x_2 \\ \sqrt{2} x_1 x_3 \\ \sqrt{2} x_2 x_3 \end{bmatrix}.$$

If $\boldsymbol{x} \in \mathbb{R}^d$, $K(\boldsymbol{x}, \boldsymbol{x}')$ is evaluated in $O(d)$, independent of r. Yet it is based on a feature map $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^{d^r}$, whose dimensionality[6] scales exponentially in r. A variant is given by

$$K(\boldsymbol{x}, \boldsymbol{x}') = \left(\boldsymbol{x}^T \boldsymbol{x}' + \varepsilon\right)^r, \quad \varepsilon > 0,$$

which can be obtained by replacing $\boldsymbol{x}$ by $[\boldsymbol{x}^T, \sqrt{\varepsilon}]^T$ above. The feature map now runs over all subranges $1 \le j_1 \le \dots \le j_k \le d$, $0 \le k \le r$.

A frequently used infinite-dimensional (positive definite) kernel is the *Gaussian* (or radial basis function, or squared exponential) kernel:

$$K(\boldsymbol{x}, \boldsymbol{x}') = e^{-\frac{\tau}{2} \|\boldsymbol{x} - \boldsymbol{x}'\|^2}, \quad \tau > 0. \tag{9.6}$$

We establish it as a kernel function in Section 9.2.4. The Gaussian is an example of a stationary kernel, these depend on $\boldsymbol{x} - \boldsymbol{x}'$ only. We can weight each dimension differently:

$$K(\boldsymbol{x}, \boldsymbol{x}') = e^{-\frac{1}{2} \sum_{j=1}^d \tau_j (x_j - x'_j)^2}, \quad \tau_1, \dots \tau_d > 0.$$

Free parameters in kernels are hyperparameters, much like C in the soft margin SVM or the noise variance σ^2 in Gaussian linear regression, choosing them is a model selection problem (Chapter 10).

Choosing the right kernel is much like choosing the right model. In order to do it well, you need to know your options. Kernels can be combined from others in many ways, [6, ch. 6.2] gives a good overview. It is also important to understand statistical properties implied by a kernel. For example, the Gaussian kernel produces extremely smooth solutions, while other kernels from the Matérn family are more flexible. Most books on kernel methods will provide some overview, see also [41].

One highly successful application domain for kernel methods concerns problems where input points $\boldsymbol{x}$ have combinatorial structure, such as chains, trees, or

[6] More economically, we can run over all $1 \le j_1 \le \dots \le j_r \le d$.

graphs. Applications range from bioinformatics over computational chemistry to structured objects in computer vision. The rationale is that it is often simpler and much more computationally efficient to devise a kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$ than a feature map $\boldsymbol{\phi}(\boldsymbol{x})$. This field was seeded by independent work of David Haussler [22] and Chris Watkins.

A final remark concerns normalization. As noted in Chapter 2 and above in Section 9.1, it is often advantageous to use normalized feature maps $\|\boldsymbol{\phi}(\boldsymbol{x})\| = 1$. What does this mean for a kernel?

$$K(\boldsymbol{x}, \boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}) = 1.$$

Therefore, a kernel function gives rise to a normalized feature map if its diagonal entries $K(\boldsymbol{x}, \boldsymbol{x})$ are all 1. For example, the Gaussian kernel (9.6) is normalized. Moreover, if $K(\boldsymbol{x}, \boldsymbol{x}')$ is a kernel, then so is

$$\frac{K(\boldsymbol{x}, \boldsymbol{x}')}{\sqrt{K(\boldsymbol{x}, \boldsymbol{x}) K(\boldsymbol{x}', \boldsymbol{x}')}}$$

(see Section 9.2.4), and the latter is normalized. It is a good idea to use normalized kernels in practice.

9.2.4 Techniques: Properties of Kernels (*)

In this section, we review a few properties of kernel functions and look at some more examples. The class of kernel functions has formidable closedness properties. If $K_1(\boldsymbol{x}, \boldsymbol{x}')$ and $K_2(\boldsymbol{x}, \boldsymbol{x}')$ are kernels, so are cK_1 for $c > 0$, $K_1 + K_2$ and $K_1 K_2$. You will have no problem confirming the the first two. The third is shown at the end of this section. Moreover, $f(\boldsymbol{x}) K_1(\boldsymbol{x}, \boldsymbol{x}') f(\boldsymbol{x}')$ is a kernel function as well, for any $f(\boldsymbol{x})$. This justifies kernel normalization, as discussed at the end of Section 9.2.3. If $K_r(\boldsymbol{x}, \boldsymbol{x}')$ is a sequence of kernel functions converging pointwise to $K(\boldsymbol{x}, \boldsymbol{x}') = \lim_{r \to \infty} K_r(\boldsymbol{x}, \boldsymbol{x}')$, then $K(\boldsymbol{x}, \boldsymbol{x}')$ is a kernel function as well. Finally, if $K(\boldsymbol{x}, \boldsymbol{x}')$ is a kernel and $\boldsymbol{\psi}(\boldsymbol{y})$ is some mapping into $\mathbb{R}^d$, then $(\boldsymbol{y}, \boldsymbol{y}') \mapsto K(\boldsymbol{\psi}(\boldsymbol{y}), \boldsymbol{\psi}(\boldsymbol{y}'))$ is a kernel as well.

Let us show that the Gaussian kernel (9.6) is a valid kernel function. First, $(\boldsymbol{x}^T \boldsymbol{x}')^r$ is a kernel for every $r = 0, 1, 2, \dots$, namely the polynomial kernel from Section 9.2.3. By the way, $K(\boldsymbol{x}, \boldsymbol{x}') = 1$ is a kernel function, since its kernel matrices $\mathbf{1}\mathbf{1}^T$ are positive semidefinite. Therefore,

$$K_r(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j=0}^{r} \frac{1}{j!} (\boldsymbol{x}^T \boldsymbol{x}')^j$$

are all kernels, and so is the limit $e^{\boldsymbol{x}^T \boldsymbol{x}'} = \lim_{r \to \infty} K_r(\boldsymbol{x}, \boldsymbol{x}')$. More general, if $K(\boldsymbol{x}, \boldsymbol{x}')$ is a kernel, so is $e^{K(\boldsymbol{x}, \boldsymbol{x}')}$. Now,

$$e^{-\frac{\tau}{2}\|\boldsymbol{x}-\boldsymbol{x}'\|^2} = e^{-\frac{\tau}{2}\|\boldsymbol{x}\|^2} e^{\tau \boldsymbol{x}^T \boldsymbol{x}'} e^{-\frac{\tau}{2}\|\boldsymbol{x}'\|^2}.$$

The middle is a kernel, and we apply our normalization rule with $f(\boldsymbol{x}) = e^{-\frac{\tau}{2}\|\boldsymbol{x}\|^2}$. The Gaussian kernel is infinite-dimensional (positive definite), although we will not show this here.

Another way to think about kernels is in terms of *covariance functions*. A random process is a set of random variables $a(\boldsymbol{x})$, one for each $\boldsymbol{x} \in \mathbb{R}^d$. Its covariance function is

$$K(\boldsymbol{x}, \boldsymbol{x}') = \mathrm{Cov}[a(\boldsymbol{x}), a(\boldsymbol{x}')] = \mathrm{E}\left[(a(\boldsymbol{x}) - \mathrm{E}[a(\boldsymbol{x})])(a(\boldsymbol{x}') - \mathrm{E}[a(\boldsymbol{x}')])\right].$$

Covariance functions are kernel functions. For some set $\{\boldsymbol{x}_i\}$, let $\boldsymbol{a} = [a(\boldsymbol{x}_i) - \mathrm{E}[a(\boldsymbol{x}_i)]] \in \mathbb{R}^n$ be a random vector. Then, for any $\boldsymbol{v} \in \mathbb{R}^n$:

$$\boldsymbol{v}^T \boldsymbol{K} \boldsymbol{v} = \boldsymbol{v}^T \mathrm{E}\left[\boldsymbol{a}\boldsymbol{a}^T\right] \boldsymbol{v} = \mathrm{E}\left[(\boldsymbol{v}^T \boldsymbol{a})^2\right] \geq 0.$$

Finally, there are some symmetric functions which are not kernels. One example is

$$K(\boldsymbol{x}, \boldsymbol{x}') = \tanh\left(\alpha \boldsymbol{x}^T \boldsymbol{x}' + \beta\right).$$

In an attempt to make SVMs look like multi-layer perceptrons, this non-kernel was suggested and is shipped to this day in many SVM toolboxes[7]. Running the SVM with “kernels” like this spells trouble. Soft margin SVM is a convex optimization problem only if kernel matrices are positive semidefinite, codes will typically crash if that is not the case. A valid “neural networks” kernel is found in [47], derived from the covariance function perspective.

Finally, why is $K_1(\boldsymbol{x}, \boldsymbol{x}')K_2(\boldsymbol{x}, \boldsymbol{x}')$ a kernel? This argument is for interested readers only, it can be skipped at no loss. We have to show that for two positive semidefinite kernel matrices $\boldsymbol{K}_1, \boldsymbol{K}_2 \in \mathbb{R}^{n \times n}$ the Schur (or Hadamard) product $\boldsymbol{K}_1 \circ \boldsymbol{K}_2 = [K_1(\boldsymbol{x}_i, \boldsymbol{x}_j)K_2(\boldsymbol{x}_i, \boldsymbol{x}_j)]$ (Section 2.4.3) is positive semidefinite as well. To this end, we consider the Kronecker product $\boldsymbol{K}_1 \otimes \boldsymbol{K}_2 = [K_1(\boldsymbol{x}_i, \boldsymbol{x}_j)\boldsymbol{K}_2] \in \mathbb{R}^{n^2 \times n^2}$. This is positive semidefinite as well. Namely, we can write $\boldsymbol{K}_1 = \boldsymbol{V}_1 \boldsymbol{V}_1^T$, $\boldsymbol{K}_2 = \boldsymbol{V}_2 \boldsymbol{V}_2^T$, then

$$\boldsymbol{K}_1 \otimes \boldsymbol{K}_2 = (\boldsymbol{V}_1 \otimes \boldsymbol{V}_2)(\boldsymbol{V}_1 \otimes \boldsymbol{V}_2)^T.$$

But the Schur product is a square submatrix of $\boldsymbol{K}_1 \otimes \boldsymbol{K}_2$, a so called minor. In other words, for some index set $J \subset \{1, \dots, n^2\}$ of size $|J| = n$: $\boldsymbol{K}_1 \circ \boldsymbol{K}_2 = (\boldsymbol{K}_1 \otimes \boldsymbol{K}_2)_J$, so that for any $\boldsymbol{v} \in \mathbb{R}^n$:

$$\boldsymbol{v}^T (\boldsymbol{K}_1 \circ \boldsymbol{K}_2) \boldsymbol{v} = \boldsymbol{z}^T (\boldsymbol{K}_1 \otimes \boldsymbol{K}_2) \boldsymbol{z} \geq 0, \quad \boldsymbol{z} = \boldsymbol{I}_{\cdot, J} \boldsymbol{v} \in \mathbb{R}^{n^2}.$$

The same proof works to show that the positive definiteness of $\boldsymbol{K}_1$, $\boldsymbol{K}_2$ implies the positive definiteness of $\boldsymbol{K}_1 \circ \boldsymbol{K}_2$, a result due to Schur.

9.2.5 Summary

Let us summarize the salient points leading up to support vector machine binary classification. We started with the observation that for a linearly separable dataset, many different separating hyperplanes result in zero training error. Among all those potential solutions, which could arise as outcome of the perceptron algorithm, there is one which exhibits maximum stability against small displacements of patterns $\boldsymbol{x}_i$, by attaining the maximum margin $\gamma_{\mathcal{D}}(\boldsymbol{w}, b)$. Pursuing this lead, we addressed a number of problems:

[7] It is even on Wikipedia (`en.wikipedia.org/wiki/Support_vector_machine`).

- Maximizing the margin does not look like a convex optimization problem. However, exploiting the fact that the size of $(\boldsymbol{w}, b)$ does not matter (only the direction does), we can find an equivalent convex program formulation: minimize a positive definite quadratic function, subject to linear constraints, a quadratic program. This is the *hard margin support vector machine*. Each margin constraint is enforced in a hard, non-negotiable manner.

- Real data often contains outliers, arising through measurement noise or labeling errors. They can result in overly small margins or even render a training dataset linearly non-separable. In practice, we should enforce stability, yet at the same time tolerate a small number of margin violations to not get sidetracked by outliers. Inspired by logistic regression, we introduced the concept of a "soft margin", allowing each margin constraint to be violated, but requesting a linear cost for such slack. The resulting *soft margin support vector machine* comes in two equivalent forms. First, we can represent costs to be paid as additional *slack variables* ξ_i, extending the quadratic program in a natural way. Second, the soft margin SVM problem corresponds to minimizing the piecewise linear *hinge error function* plus a Tikhonov regularizer, which allows for comparisons with logistic regression.

- Once Tikhonov regularization is used, we can in principle use linear discriminants with many more features p than training datapoints n. Maybe the most surprising twist in this chapter is that we can do so and do not even have to pay for it. The *representer theorem* and the *kernel trick* allows us to reformulate Tikhonov-regularized estimation problems in only n dual variables, *no matter what p is*. We never have to evaluate the feature map $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^p$, but can get by with *kernel* evaluations $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}')$. In other words, we can formulate our estimation problem in terms of a kernel function up front, ignoring the feature map behind. Resulting discriminant functions come in the form of kernel expansions:
$$y(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i K(\boldsymbol{x}, \boldsymbol{x}_i) + b.$$
This is a weighted sum of kernel functions $K(\cdot, \boldsymbol{x}_i)$ placed on each datapoint.

- We have motivated kernel methods as arising from linear functions in a feature space given by $\boldsymbol{\phi}(\boldsymbol{x})$, so that $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}')$. However, in practice we choose a kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$ and work with it without ever having to know the underlying feature map $\boldsymbol{\phi}(\boldsymbol{x})$. In fact, for any given kernel $K(\boldsymbol{x}, \boldsymbol{x}')$, there are *many* feature maps giving rise to it, so it is not even possible to uniquely go from $K(\boldsymbol{x}, \boldsymbol{x}')$ to $\boldsymbol{\phi}(\boldsymbol{x})$.

At this point, we could plug the kernel expansion in terms of $\boldsymbol{\alpha}$ and b into the soft margin SVM problem (9.4), then solve the resulting quadratic program in $(\boldsymbol{\alpha}, b, \boldsymbol{\xi})$. Indeed, this is how we would proceed in order to "kernelize" logistic regression. However, in case of the SVM, some more work leads to a simpler optimization problem, whose properties provide important insights into the influence of single patterns on the final solution. We will derive this *dual* problem in the following section.

9.3 Solving the Support Vector Machine Problem

In this section, we will learn how to solve the soft margin SVM learning problem (9.4), using the concept of Lagrange duality. We will keep our exposition simple and use graphical intuition rather than proofs, but there is no shortage of literature to fill in these gaps [8, 39].

The soft margin SVM is the solution of a convex problem (9.4) with a convex criterion and linear inequality constraints. It is posed in terms of $p+n+1$ parameters and n constraints. Maybe the most powerful tool to address convex programs of this kind is *Lagrange duality*, a technique to derive a second convex optimization problem of the same kind, called the *dual* problem, which (a) has the same optimal solution than the *primal* problem we start from, and (b) operates roughly on as many parameters as the number of primal constraints. As we will see, the dual problem for the soft margin SVM is a quadratic program in n parameters, *independent* of p. In fact, if we did not know about the representer theorem and kernel expansions, we would reinvent it for the SVM in this way.

In order to keep the exposition as simple as possible, we will derive the soft margin SVM dual problem in an intuitive and somewhat informal way. While this is sufficient for our purposes here, Lagrange duality for general convex problems is a core concept in machine learning today, with applications far beyond support vector machines. For this reason, we provide a more complete account of Lagrange multipliers and Lagrange duality in Appendix A.

Our starting point is the formulation of the soft margin SVM problem in terms of the hinge error function:

$$\min_{\boldsymbol{w},b}\left\{\Phi_P = \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} C[1-t_i y_i]_+\right\}, \quad y_i = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)+b.$$

This is an unconstrained optimization problem, so why don't we proceed as so many times before, solving $\nabla_{\boldsymbol{w},b}\Phi_P = \boldsymbol{0}$ for $\boldsymbol{w}$ and b? This strategy fails badly in this case, since Φ_P is not differentiable. Our goal must be to replace Φ_P by continuously differentiable functions, for which we can make use of the first-order stationary condition. The key idea is sketched in Figure 9.6. The non-differentiable term $C[1-t_i y_i]_+$ is lower bounded by the linear $\alpha_i(1-t_i y_i)$, as long as $\alpha_i \in [0, C]$. This family of linear lower bounds is tight, in that

$$C[1-t_i y_i]_+ = \max_{\alpha_i\in[0,C]} \alpha_i(1-t_i y_i).$$

The new variables $\boldsymbol{\alpha} = [\alpha_i]$ represent the slopes of the linear bounds, they come with two-sided linear inequality constraints. Given these variables, our problem becomes

$$\tilde{p}_* = \min_{\boldsymbol{w},b}\ \max_{\{0\le\alpha_i\le C\}}\left\{L(\boldsymbol{w},b,\boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n}\alpha_i(1-t_i y_i)\right\}.$$

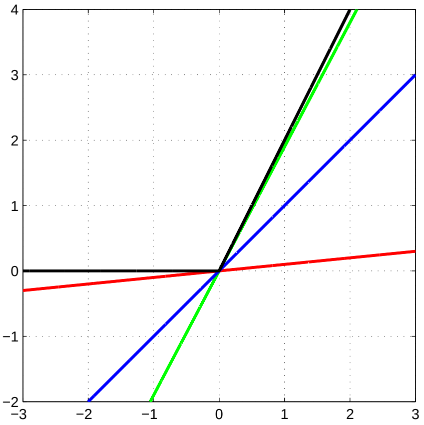


Figure 9.6: Hinge function $v \mapsto C[v]_+$, $C = 2$, where $v = 1 - ty$ (black), along with some linear lower bounds $v \mapsto \alpha v$ ($\alpha = 0.1$, red; $\alpha = 1$, blue; $\alpha = 1.9$, green). These are global lower bounds of the hinge function for any $\alpha \in [0, C]$.

At the expense of introducing new variables α_i, one for each soft margin constraint, the inner criterion L is continuously differentiable. This is called the *primal problem*, and $\tilde{p}_*$ is called the primal value. Notice the min-max structure of this problem: what we are after is a *saddlepoint*, minimizing L w.r.t. the *primal variables* $\boldsymbol{w}$, b, while at the same time maximizing L w.r.t. the new *dual variables* $\boldsymbol{\alpha}$.

These are not the only saddlepoints, we might just as well look at

$$\tilde{d}_* = \max_{\{0 \le \alpha_i \le C\}} \min_{\boldsymbol{w}, b} L(\boldsymbol{w}, b, \boldsymbol{\alpha}).$$

Here, the unconstrained minimization over $\boldsymbol{w}$, b is inside, the maximization over dual variables is outside. This is called the *dual problem*. How do these two problems relate to each other? First of all, we always have

$$\tilde{d}_* \le \tilde{p}_*.$$

After all, this is how we constructed things. For each fixed $\boldsymbol{\alpha}$ with $\alpha_i \in [0, C]$:

$$\min_{\boldsymbol{w}, b} L(\boldsymbol{w}, b, \boldsymbol{\alpha}) \le \min_{\boldsymbol{w}, b} \max_{\{0 \le \alpha'_i \le C\}} L(\boldsymbol{w}, b, \boldsymbol{\alpha}').$$

The dual value $\tilde{d}_*$ bounds the primal value $\tilde{p}_*$ from below. This fact is called *weak duality*. While weak duality holds for *any* primal problem, convex or not, more is true for our soft margin SVM problem. There, we have *strong duality*, in that the two saddlepoint values are identical:

$$\max_{\{0 \le \alpha_i \le C\}} \min_{\boldsymbol{w}, b} L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \tilde{d}_* \overset{!}{=} \tilde{p}_* = \min_{\boldsymbol{w}, b} \max_{\{0 \le \alpha_i \le C\}} L(\boldsymbol{w}, b, \boldsymbol{\alpha}).$$

This means that we can solve the original primal problem by solving the dual problem instead, which will turn out to be simpler in the case of soft margin SVM.

Let us solve the dual problem. Denote $\boldsymbol{\phi}_i := \boldsymbol{\phi}(\boldsymbol{x}_i)$. The inner criterion is

$$L = \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^n \alpha_i(1 - t_i y_i), \quad y_i = \boldsymbol{w}^T\boldsymbol{\phi}_i + b, \;\; i = 1, \dots, n.$$

The criterion for the dual problem is obtained by minimizing over the primal variables $(\boldsymbol{w}, b)$:

$$\Phi_D(\boldsymbol{\alpha}) = \min_{\boldsymbol{w},b} L(\boldsymbol{w}, b, \boldsymbol{\alpha}).$$

Finally an unconstrained and differentiable problem: we can apply our "set the gradient to zero" procedure! First,

$$\nabla_{\boldsymbol{w}} L = \boldsymbol{w} - \sum_{i=1}^n \alpha_i t_i \boldsymbol{\phi}_i = \mathbf{0} \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^n \alpha_i t_i \boldsymbol{\phi}_i.$$

Here is the feature expansion we were waiting for! Also,

$$\nabla_b L = -\sum_{i=1}^n \alpha_i t_i = 0 \quad \Rightarrow \quad \sum_{i=1}^n \alpha_i t_i = 0.$$

This means that the dual problem has an equality constraint. Plugging these into the criterion, we obtain the dual optimization problem:

$$\max_{\boldsymbol{\alpha}} \left\{ \Phi_D(\boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_i \alpha_i(1 - t_i y_i) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i\alpha_j t_i t_j \boldsymbol{\phi}_i^T\boldsymbol{\phi}_j \right\},$$
$$\text{subj. to } \;\alpha_i \in [0, C], \;\; i = 1, \dots, n, \quad \sum_i \alpha_i t_i = 0. \tag{9.7}$$

We used that

$$\sum_i \alpha_i t_i y_i = \sum_i \alpha_i t_i \left(\boldsymbol{\phi}_i^T\boldsymbol{w} + b\right) \overset{(1)}{=} \sum_i \alpha_i t_i \boldsymbol{\phi}_i^T\boldsymbol{w} \overset{(2)}{=} \|\boldsymbol{w}\|^2,$$

since (1) $\sum_i \alpha_i t_i = 0$ and (2) $\boldsymbol{w} = \sum_i \alpha_i t_i \boldsymbol{\phi}_j$. The dual problem is indeed simpler than the primal. First, it depends on n variables only, not on $p+n+1$. It is convex, since $-\Phi_D(\boldsymbol{\alpha})$ is a positive semidefinite quadratic. Finally, its feasible set is simply the intersection of the hypercube $[0, C]^n$ with the hyperplane $\boldsymbol{\alpha}^T\boldsymbol{t} = 0$. It is naturally kernelized (Section 9.2.3). Picking a kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$, we have that

$$K_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}_i^T\boldsymbol{\phi}_j, \quad \boldsymbol{K} = [K_{ij}] \in \mathbb{R}^{n\times n}.$$

In terms of this kernel matrix, the dual criterion becomes

$$\Phi_D(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i t_i K_{ij} t_j \alpha_j = \mathbf{1}^T\boldsymbol{\alpha} - \frac{1}{2}\boldsymbol{\alpha}^T(\operatorname{diag}\boldsymbol{t})\boldsymbol{K}(\operatorname{diag}\boldsymbol{t})\boldsymbol{\alpha}.$$

Moreover, if $\boldsymbol{\alpha}_*$ is the solution of the dual problem, then $\boldsymbol{w}_* = \sum_{i=1}^n \alpha_{*i} t_i \boldsymbol{\phi}_i$ solves the primal, and we obtain the discriminant

$$y^*(\boldsymbol{x}) = (\boldsymbol{w}_*)^T\boldsymbol{\phi}(\boldsymbol{x}) = \sum_{i=1}^n \alpha_{*i} t_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b_*. \tag{9.8}$$

Support Vectors

There is a property we have not yet used, which links primal and dual variables at the saddlepoint. By working it out, we will complete our derivation of the soft margin SVM dual problem. First, we will find how to solve for b_*. Second, we will obtain an important classification of patterns $(\boldsymbol{x}_i, t_i)$ in terms of values of α_i, which will finally clarify the catchy name "support vectors". To gain some intuition, consider the soft margin SVM solution in Figure 9.7. Three different things can happen for a pattern $(\boldsymbol{x}_i, t_i)$. First, it may be classified correctly with at least the margin, in that $t_i y_i \geq 1$. In this case, the optimal solution does not really depend on it. We could remove the pattern from $\mathcal{D}$ and still get the same solution. The solution is not supported by these vectors. Second, it may lie precisely on the margin, $t_i y_i = 1$. These ones are important, they provide the essential support. Finally, for the soft margin SVM, there may be patterns in the margin area or even misclassified, $t_i y_i < 1$. They support the solution as well, because we pay for them, and removing them may allow us to increase the soft margin.

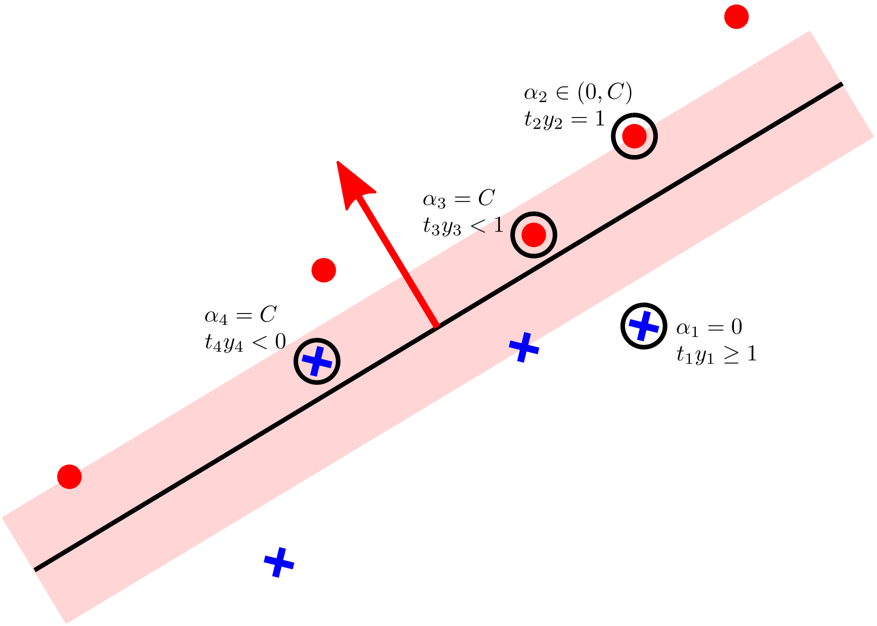


Figure 9.7: Different types of patterns for soft margin SVM solution. $(\boldsymbol{x}_1, t_1)$ is classified correctly with margin, $t_1 y_1 \geq 1$. The SV solution does not depend on it. $(\boldsymbol{x}_2, t_2)$ is an essential support vector and lies directly on the margin, $t_2 y_2 = 1$. Both $(\boldsymbol{x}_3, t_3)$ and $(\boldsymbol{x}_4, t_4)$ are bound support vectors, $\alpha_3 = \alpha_4 = C$. $(\boldsymbol{x}_3, t_3)$ lies in the margin area, while $(\boldsymbol{x}_4, t_4)$ is misclassified ($t_4 y_4 < 0$).

Suppose now that we are at a saddlepoint of the dual problem, dropping the "*" subscript for easier notation. Recall how we got the α_i into the game above:

$$C[1 - t_i y_i]_+ = \max_{\alpha_i \in [0,C]} \alpha_i (1 - t_i y_i).$$

A glance at Figure 9.6 reveals that α_i is linked with y_i (and therefore with $\boldsymbol{w}$, b)

through the requirement that the lower bound has to touch the hinge function at the argument $1 - t_i y_i$. As illustrated in Figure 9.8, there are three different cases:

- $\alpha_i = 0$. In this case, $1 - t_i y_i \leq 0$ (Figure 9.8, left). These are points which are classified correctly with at least the margin. Since $\alpha_i = 0$, the solution indeed does not depend on them. They are not support vectors.
- $\alpha_i \in (0, C)$. In this case, $1 - t_i y_i = 0$ (Figure 9.8, middle). These points lie directly on the margin. They are *essential support vectors*.
- $\alpha_i = C$. Then, $1 - t_i y_i \geq 0$ (Figure 9.8, right). These points lie in the margin area or may even be misclassified. They are called *bound support vectors*, since α_i is bound to the maximum value C.

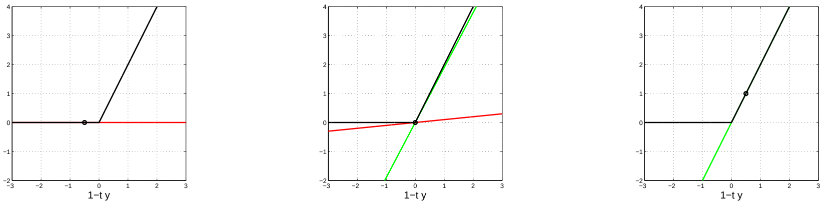


Figure 9.8: Left: No support vector ($1 - t_i y_i \leq 0$, $\alpha_i = 0$). Middle: Essential support vector ($1 - t_i y_i = 0$, $0 < \alpha_i < C$). Right: Bound support vector ($1 - t_i y_i \geq 0$, $\alpha_i = C$).

If $(\boldsymbol{x}_i, t_i)$ is not a support vector, its $\alpha_i = 0$ and it does not appear in the kernel expansion (9.8) of $y^*(\boldsymbol{x})$. If this happens for many patterns, the kernel expansion (9.8) is *sparse*, and a lot of time can be saved both when predicting on future test data and during training. Sparsity is a major attractive point about support vector machines in practice (see Section 9.4).

Finally, the essential support vectors allow us to determine the solution for b. Denote $\mathcal{S} = \{i \mid \alpha_i \in (0, C)\}$. For these, let $\tilde{y}_i = \sum_j \alpha_j t_j K(\boldsymbol{x}_j, \boldsymbol{x}_i) = y_i - b$. We know that $1 = t_i y_i = t_i(\tilde{y}_i + b)$, so that

$$b = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} (t_i - \tilde{y}_i) \,.$$

Note we could also just compute b from any single $i \in \mathcal{S}$, but the above formula averages out numerical errors.

Most SVM codes in use today solve this dual problem, making use of the so called *Karush-Kuhn-Tucker* (KKT) conditions we just derived. A particularly simple algorithm is sequential minimal optimization (SMO) [35], where pairs (α_i, α_j) are updated in each iteration. There are some good and some not so good SVM packages out there. An example for a good code is `LIBSVM` at `www.csie.ntu.edu.tw/∼cjlin/libsvm/`.

This concludes our derivation of the soft margin SVM optimization problem in both primal and dual form. Our derivation is a special case of the much more general and powerful framework of Lagrange multipliers and Lagrange duality, which we discuss in Appendix A.

9.4 Support Vector Machines and Kernel Logistic Regression (*)

In this chapter, we worked out support vector machines and learned how to kernelize estimation techniques based on linear functions. For the SVM, kernelization is a consequence of working with the dual problem (Section 9.3), while for general regularized estimators, it is justified by the representer theorem (Section 9.2.2). So what's the deal about SVMs? In this final section, we will address this question by relating the soft margin SVM to kernel logistic regression for binary classification.

Briefly, the main attraction of SVMs over regularized conditional likelihood estimators is *sparsity*, while a major drawback of SVMs is their inability to consistently estimate posterior class probabilities. These two aspects are linked in a fascinating way, unfortunately out of scope here. The latter may seem like a small problem, but it is the ultimate reason behind the appaling difficulties of generalizing SVMs to multi-way classification in a sensible way, while this can be done easily for logistic regression (Section 8.3.1).

Sparsity

The solution $(\boldsymbol{\alpha}, b)$ to a soft margin SVM problem is sparse if most patterns are not support vectors and most of the coefficients of $\boldsymbol{\alpha}$ are zero. This does not always happen, but it happens frequently in practice. A beneficial aspect of sparsity is that the final predictor $y^*(\boldsymbol{x})$ (9.8) can be evaluated efficiently on test data. In fact, we only have to store the support vectors for prediction purposes. In many real-world applications, it is very important to be able to predict quickly, and sparsity is a substantial asset then. Even better, sparsity tends to help with the training as well. Most SVM solvers exploit sparsity in one way or another, which often lets them cope with dataset sizes n for which the storage of the full kernel matrix $\boldsymbol{K}$ is out of the question. An extreme example is SMO (Section 9.3), which is very slow without, yet highly effective with sparsity.

One problem with SVM sparsity is that it cannot be controlled in a sensible way, nor can we easily say up front how sparse our solution will be. To some extent, sparsity can be linked to the Bayes error of a problem (details in [39]), but the link is not perfect. It is sometimes advocated to regulate sparsity by the choice of C in (9.4). It is true that cranking up C can lead to sparser solutions, but that is not what this parameter is for. C should be chosen solely with one goal in mind, namely to attain the best possible generalization error. There are alternative methods which allow the degree of sparsity to be regulated, for example the relevance vector machine [6, ch. 7.2] or the informative vector machine [28]. However, when it comes to combining sparsity with computational efficiency and robustness, there is still no match for the SVM.

Estimating Posterior Class Probabilities

Recall from Section 8.2.2 that a conditional maximum likelihood method such as logistic regression can estimate class posterior probabilities $P(t = +1|\boldsymbol{x})$, since its population minimizer, the log odds ratio, is probability-revealing. In this section, we consider regularized and kernelized logistic regression, which corresponds to the problem

$$\min_{\boldsymbol{\alpha},b} \sum_{i=1}^{n} \log\left(1 + e^{-t_i y(\boldsymbol{x}_i)}\right) + \frac{\nu}{2}\boldsymbol{\alpha}^T \boldsymbol{K}\boldsymbol{\alpha}, \quad y(\boldsymbol{x}) = \sum_{j=1}^{n} \alpha_j K(\boldsymbol{x}_j, \boldsymbol{x}) + b. \qquad (9.9)$$

Here, $\boldsymbol{K}$ is the kernel matrix. Other than in the SVM dual problem, the α_i can be negative (in fact, α_i plays the role of $t_i\alpha_i$ in the SVM problem). For kernels such as the Gaussian (9.6) and a certain decay of $\nu = \nu_n$ with training set size n, the minimizers $\sum_j \alpha_{*,j} K(\boldsymbol{x}_j, \cdot) + b_*$ of (9.9) converge to the log odds ratio $\log\{P(t = +1|\boldsymbol{x})/P(t = -1|\boldsymbol{x})\}$ for the true data distribution almost surely[8]. In other words, posterior class probabilities $P(t = +1|\boldsymbol{x})$ can be estimated consistently with regularized kernel logistic regression.

How about the soft margin SVM? First, it is not a conditional maximum a-posteriori technique, mainly since the hinge loss $[1 - t_i y_i]_+$ does not correspond to a negative log likelihood [40]. A way of understanding the SVM in terms of probabilities is shown in [26], but this is markedly different from MAP estimation. It should therefore not come as a surprise that posterior class probabilities cannot be estimated consistently with the SVM. The population minimizer for the hinge loss is

$$\begin{aligned} y^*(\boldsymbol{x}) &= \operatorname*{argmin}_y \mathrm{E}\left[[1 - ty]_+ \;\middle|\; \boldsymbol{x}\right] \\ &= \left\{ \begin{array}{ll} +1 & \mid\; P(t = +1|\boldsymbol{x}) > P(t = -1|\boldsymbol{x}) \\ -1 & \mid\; P(t = +1|\boldsymbol{x}) < P(t = -1|\boldsymbol{x}) \\ \in [-1, +1] & \mid\; P(t = 1|\boldsymbol{x}) = P(t = -1|\boldsymbol{x}) \end{array} \right\}. \end{aligned}$$

The proof is left to the reader. Note that $y^*(\boldsymbol{x})$ can take any value in $[-1, +1]$ if $P(t = 1|\boldsymbol{x}) = P(t = -1|\boldsymbol{x})$. Now, $y^*(\boldsymbol{x})$ is certainly not probability-revealing. For each $\boldsymbol{x}$, it merely contains the information whether $P(t = +1|\boldsymbol{x}) > P(t = -1|\boldsymbol{x})$ or $P(t = +1|\boldsymbol{x}) < P(t = -1|\boldsymbol{x})$, but nothing else. Indeed, Bartlett and Tewari [2] demonstrated rigorously that $P(t = +1|\boldsymbol{x})$ cannot be estimated consistently by SVMs. The reason for this is precisely the *sparsity* of SVMs. The failure of SVMs does not just happen on some unusual data distributions, but is a direct implication of margin maximization. Ironically, despite this clear result, SVMs are used to "estimate class probabilities" to this day, using [36] or more elaborate variants. This cannot be justified as an approximation, it is just plain wrong. If we want a sparse classifier trained by convex optimization, we can use the SVM. If our goal is to estimate class probabilities for automatic decision making (Section 5.2), the SVM is not a trustworthy option, and we should use other methods such as kernelized logistic regression, trained by the IRLS algorithm (Section 8.2.4).

We close this section with noting that there is a probabilistic counterpart to support vector machines, going far beyond MAP estimation for logistic regression: *Gaussian process methods*. Kernels are covariance functions there (Section 9.2.4), which give rise to Gaussian priors over functions $y(\cdot)$. GP methods

[8] This is a classical result, which follows for example from Corollary 3.62 and example 3.66 in [44].

are out of scope of this basic course, but do constitute an important part of modern machine learning, with applications ranging from computer vision over robotics, spatial statistics to bioinformatics. If you are interested, start with [48] or the textbook [37]. A useful website is `www.gaussianprocess.org`.

Chapter 10

Model Selection and Evaluation

The central problem of machine learning is generalization. We built a model for our problem at hand and trained it on some data. How well will it work on future test data? How can we estimate its generalization performance? How complex should our model be, how much should we regularize? In this chapter, we develop an understanding of such questions. We will learn about the training-validation-test set paradigm of machine learning and about cross-validation as a general technique for selecting complexity parameters of a model.

10.1 Bias, Variance and Model Complexity

Recall the fundamental concept of *generalization* from Chapter 7. Decision theory (Section 5.2) tells us how to act if the "true" distribution underlying our problem is known: we should minimize risk, or expected loss. Normally, we don't know the truth, but have access to a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \dots, n\}$ sampled from it. Our goal must now be to make use of $\mathcal{D}$ in order to approximate the non-realizable decision-theoretic procedure as closely as possible. In particular, we would want to know when things go wrong. We already identified a major problem, over-fitting, and studied regularization techniques in order to alleviate it (Chapter 7). But that is not the whole story. How do we decide between different models, given $\mathcal{D}$? For example, what is the best total degree in polynomial regression (Section 4.1)? The best noise variance in linear regression with Gaussian noise (Section 8.1.3)? How do we choose the parameter ν of a Tikhonov regularizer (Section 7.2), or prior parameters in MAP estimation (Section 8.3.2)? All of these parameters have in common that they determine the model's expressiveness or *complexity*. They cannot be chosen by minimizing an error function on $\mathcal{D}$, since more complex models always result in better training data fits. In this chapter, we will be concerned with how to estimate the (test) risk from data samples. Two major reasons for doing so are:

- Model selection: How do we choose between models or learning methods of different complexity (for example, hyperparameters or regularization constants)?

- Model assessment: How will the finally chosen method behave on future test data?

10.1.1 Validation and Test Data

The simplest, and ultimately the only 100% safe way of estimating the test error is to use independent data which is never used for training, and to use it only once. We distinguish between *training data*, *validation data* and *test data*. We already know what the training data is for. The test data is kept in a vault and only used at the very end, for model assessment or to compare final predictors against each other. In a serious machine learning study, we are not allowed to go back and change things after the test data has been used in any way. Otherwise, the test set takes the role of a training set and test errors will be underestimated: our method will look better than it really is.

The validation set is used for model selection, as explored in detail in Section 10.2. Briefly, we compare trained predictors $\hat{y}_\nu(\cdot)$ for different values of a complexity parameter ν by estimating their respective test errors using the validation dataset, then choosing the value ν for which the test risk estimate is lowest. This works because training and validation set are independent, so that training and test risk estimation cannot influence each other. It is difficult to give a general rule on how to choose respective sizes of training, test and validation set. A typical split may be 50% for training, 25% for testing and 25% for validation. It is good practice to split a given database at random, or permute cases up front, as there is often some particular ordering imposed on the original set.

An obvious problem with this procedure is waste of data. We need to hold out a substantial fraction of available data, which may just as well be used in order to train a better model. In many applications, high quality labeled data is difficult and expensive to come by. In the remainder of this chapter, we will mainly be interested in ways to understand and estimate generalization performance by *using the training dataset only*. It is important to note that these complement, but do not replace hold out dataset procedures. Which of them we use in practice, must depend on reliability standards for model selection and assessment as well as the size of available data in comparison to model complexity.

10.1.2 A Simple Example

Let us start with an artificial polynomial curve fitting example, reminiscent of Section 4.1 and Section 7.1.1. In Figure 10.1, top, we show two datasets of 10 points each. The data $\mathcal{D}_L$ in the left panel is closely fitted by a line, while the data $\mathcal{D}_R$ on the right is equally closely fitted by a polynomial of degree 7. In terms of goodness of the fit, these two examples are equivalent. However, most of us would agree that on the left, we discovered underlying linear structure, while the result on the right is probably arbitrary. Intuitively, there are so few

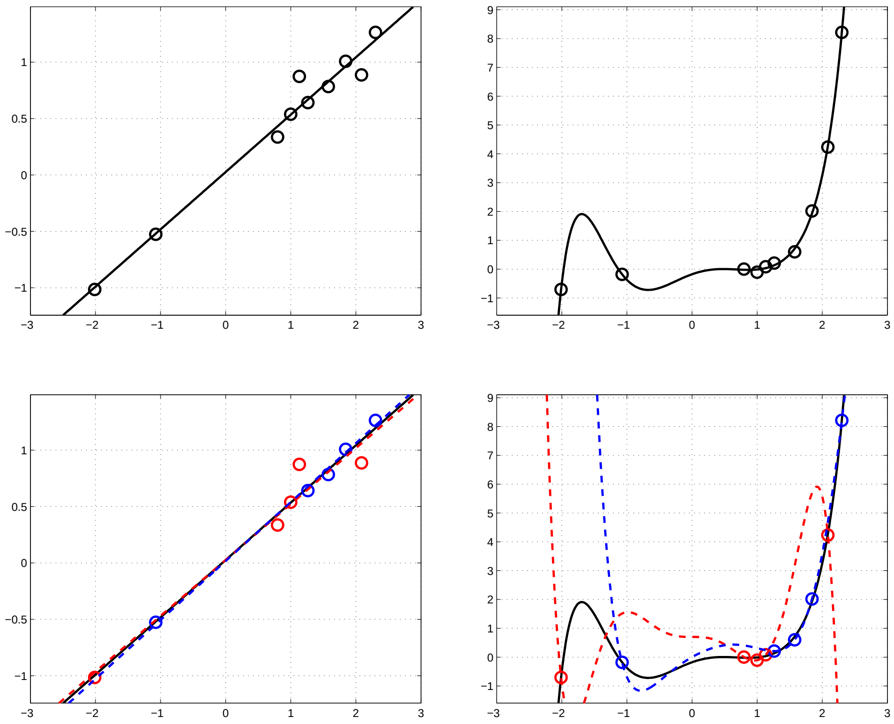


Figure 10.1: Two datasets of size 10, closely fitted by polynomials of degree 1 (left) and degree 7 (right). Top row: Least squares fits exhibit similar squared error in either case. Bottom left: Degree 1 polynomials fitted to subsets of size 5 are close to the fit for the whole set. Bottom right: Degree 7 polynomials fitted to subsets of size 5 are very different from each other and from the fit for the whole set.

10 point datasets well described by lines, compared to what could happen by chance, that the left fit is highly likely to witness underlying structure. On the other hand, polynomials of degree 6 can closely fit a substantial fraction of all 10 point datasets, and the fit on the right is likely to arise just by chance. This intuition is still too vague to be useful in practice, so let us try to be more clear. Suppose we obtained another independent sample of size 10 on either side, $\mathcal{D}'_L$ and $\mathcal{D}'_R$, from the same distributions respectively. On the left, we would expect the previous line to be a good fit for the new sample $\mathcal{D}'_L$ as well. Put differently, the best line fits of $\mathcal{D}_L$ and $\mathcal{D}'_L$ should agree closely. On the right, it seems much more likely that the best degree 6 fits of $\mathcal{D}_R$ and $\mathcal{D}'_R$ are very different from each other. In short, if we could resample the data, we would expect much more variability on the right than on the left side. Why do we expect that? Since data is sampled independently from the same distribution, we can split $\mathcal{D}_L$ randomly into two parts of 5 points each (Figure 10.1, bottom). The best line fits to each subsample are very close to each other and to the best fit of the whole $\mathcal{D}_L$. On the right side, a similar split of $\mathcal{D}_R$ leads to entirely different degree 6 polynomial fits. This simple example conveys two ideas which we will

develop in this chapter.

- Generalization behaviour can be understood and analyzed by considering the training sample $\mathcal{D}$ as a random variable which can be resampled in principle. The analysis focusses on both average value (mean) and variability (variance) of prediction errors w.r.t. $\mathcal{D}$. In particular, high variance is a telltale sign of over-fitting.
- In general, estimating the test error of a predictor requires independent data not used for training. However, it may be possible to estimate average and variability of prediction errors by splitting the single training sample $\mathcal{D}$ into several parts, using some for training and others for evaluation.

To be more specific, we introduce the (test) risk

$$R(\hat{y}_\nu|\mathcal{D}) = \mathrm{E}\left[L(\hat{y}_\nu(\boldsymbol{x}), t) \;\middle|\; \mathcal{D}\right]$$

and the empirical (or training) risk

$$\hat{R}_n(\hat{y}_\nu|\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} L(\hat{y}_\nu(\boldsymbol{x}_i), t_i),$$

where $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \,|\, i = 1, \dots, n\}$ is a training dataset drawn i.i.d. from the (unknown) true law, $\hat{y}_\nu(\cdot)$ is the predictor trained on $\mathcal{D}$ for a complexity parameter ν, and $(\boldsymbol{x}, t)$ in the definition of $R(\hat{y}_\nu|\mathcal{D})$ is another (independent) test case from the same distribution. Both test and training risk depend on the sample $\mathcal{D}$, because $\hat{y}_\nu(\cdot)$ depends on $\mathcal{D}$. However, the training risk $\hat{R}_n(\hat{y}_\nu|\mathcal{D})$ depends on $\mathcal{D}$ twice: it is used to select the predictor $\hat{y}_\nu(\cdot)$ *and* to quantify its risk. We already know that minimizing the empirical risk $\hat{R}_n(\hat{y}_\nu|\mathcal{D})$ is not necessarily a good way of choosing ν, since we may run into the overfitting problem. On the other hand, we cannot minimize the true risk $R(\hat{y}_\nu|\mathcal{D})$, because we do not know the true underlying distribution. In this chapter, we analyze the relationship between risk $R(\hat{y}_\nu|\mathcal{D})$ and empirical risk $\hat{R}_n(\hat{y}_\nu|\mathcal{D})$ more closely, with the aim of finding more reliable empirical estimators of $R(\hat{y}_\nu|\mathcal{D})$.

10.1.3 Bias-Variance Decomposition

Consider a curve estimation setup. $\hat{y}(\cdot)$ is fit to data $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \,|\, i = 1, \dots, n\}$, $t_i \in \mathbb{R}$, with the aim of minimizing the population squared error

$$\mathcal{E} = \mathrm{E}\left[L(\hat{y}(\boldsymbol{x}), t)\right] = \frac{1}{2}\mathrm{E}\left[\mathrm{E}\left[(\hat{y}(\boldsymbol{x}) - t)^2 \;\middle|\; \boldsymbol{x}\right]\right].$$

If you are unsure about the notation, you might want to revisit Chapter 5, in particular Section 5.2. Our convention is that a conditional expectation $\mathrm{E}[\mathcal{A}|\mathcal{B}]$ is over all variables in the expression $\mathcal{A}$ which do not appear in $\mathcal{B}$. For example, the expectation above is over $(\boldsymbol{x}, t)$ from the true law, which in the second expression is decomposed into conditional expectation over t given $\boldsymbol{x}$ inside and marginal expectation over $\boldsymbol{x}$ outside. We will worry about the dependence of

$\hat{y}(\cdot)$ on the training sample $\mathcal{D}$ in a moment. For now, let us find out what the optimal solution for $\hat{y}(\cdot)$ would be. Namely,

$$\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-t)^2\right]=\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-y_{\mathrm{opt}}(\boldsymbol{x})+y_{\mathrm{opt}}(\boldsymbol{x})-t)^2\right]$$
$$=\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-y_{\mathrm{opt}}(\boldsymbol{x}))^2\right]+2\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-y_{\mathrm{opt}}(\boldsymbol{x}))(y_{\mathrm{opt}}(\boldsymbol{x})-t)\right]+\mathrm{E}\left[(y_{\mathrm{opt}}(\boldsymbol{x})-t)^2\right],$$

where $y_{\mathrm{opt}}(\boldsymbol{x})$ does not depend on t. The middle term is "cross-talk", it vanishes if we choose $y_{\mathrm{opt}}(\boldsymbol{x})=\mathrm{E}[t\,|\,\boldsymbol{x}]$:

$$\begin{aligned}\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-t)^2\right]&=\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-\mathrm{E}[t\,|\,\boldsymbol{x}])^2\right]+\mathrm{E}\left[(\mathrm{E}[t\,|\,\boldsymbol{x}]-t)^2\right]\\&=\mathrm{E}\left[(\hat{y}(\boldsymbol{x})-\mathrm{E}[t\,|\,\boldsymbol{x}])^2\right]+\mathrm{E}\left[\mathrm{Var}[t\,|\,\boldsymbol{x}]\right].\end{aligned}\tag{10.1}$$

You should test your understanding on this derivation. Why does the cross-talk term vanish under this particular choice of $y_{\mathrm{opt}}(\boldsymbol{x})$? Why can the final term be written as expected conditional variance? Is this the same as $\mathrm{Var}[t]$? If not[1], why not, what is missing?

The expression (10.1) is uniquely minimized by $\hat{y}(\boldsymbol{x})=\mathrm{E}[t\,|\,\boldsymbol{x}]$, the conditional expectation of t given $\boldsymbol{x}$. In other words, $\mathrm{E}[t\,|\,\boldsymbol{x}]$ is the population minimizer under the squared loss function (see Section 8.2.2), the optimal solution to the curve fitting problem, which requires knowledge of the true underlying law. The term $\mathrm{E}[\mathrm{Var}[t\,|\,\boldsymbol{x}]]$ is a lower bound on the population squared error, we cannot perform better than this. For example, suppose the data is generated as $t_i=y_*(\boldsymbol{x}_i)+\varepsilon_i$, where $y_*(\cdot)$ is an underlying curve, and the ε_i are i.i.d. noise variables with zero mean and variance σ^2 (Section 4.1). Then, $\mathrm{E}[t\,|\,\boldsymbol{x}]=\mathrm{E}[y_*(\boldsymbol{x})+\varepsilon\,|\,\boldsymbol{x}]=y_*(\boldsymbol{x})$ and $\mathrm{E}[\mathrm{Var}[t\,|\,\boldsymbol{x}]]=\sigma^2$. For this additive noise setup, the conditional expectation recovers the underlying curve perfectly, while the minimum unresolvable error is equal to the noise variance.

Our argument so far is decision-theoretic much like in Section 5.2, but dealing with curve fitting instead of classification. In practice, $\hat{y}(\cdot)$ is learned from data $\mathcal{D}$, without knowledge of the true law. Let us fix $\boldsymbol{x}$ and use the quadratic decomposition once more, this time taking expectations over the sample $\mathcal{D}$:

$$\mathrm{E}\left[(\hat{y}(\boldsymbol{x}|\mathcal{D})-\mathrm{E}[t\,|\,\boldsymbol{x}])^2\;\middle|\;\boldsymbol{x}\right]=\mathrm{E}\left[(\hat{y}(\boldsymbol{x}|\mathcal{D})-y_{\mathrm{avg}}(\boldsymbol{x})+y_{\mathrm{avg}}(\boldsymbol{x})-\mathrm{E}[t\,|\,\boldsymbol{x}])^2\;\middle|\;\boldsymbol{x}\right],$$

where $y_{\mathrm{avg}}(\boldsymbol{x})$ does not depend on $\mathcal{D}$. Using the same argument as above, the cross-talk vanishes for $y_{\mathrm{avg}}(\boldsymbol{x})=\mathrm{E}[\hat{y}(\boldsymbol{x}|\mathcal{D})\,|\,\boldsymbol{x}]$, the expected prediction curve, and

$$\mathrm{E}\left[(\hat{y}(\boldsymbol{x}|\mathcal{D})-\mathrm{E}[t\,|\,\boldsymbol{x}])^2\;\middle|\;\boldsymbol{x}\right]=\underbrace{(\mathrm{E}[\hat{y}(\boldsymbol{x}|\mathcal{D})\,|\,\boldsymbol{x}]-\mathrm{E}[t\,|\,\boldsymbol{x}])^2}_{\mathrm{Bias}^2}+\underbrace{\mathrm{Var}\left[\hat{y}(\boldsymbol{x}|\mathcal{D})\;\middle|\;\boldsymbol{x}\right]}_{\mathrm{Variance}}.$$

This is the *bias-variance decomposition* for regression with squared loss. Note that it holds pointwise at each $\boldsymbol{x}$. A further expectation over $\boldsymbol{x}$ provides a bias-variance decomposition of the expected test risk:

$$\mathrm{E}[R(\hat{y}_\nu|\mathcal{D})]=\mathrm{E}\left[(\mathrm{E}[\hat{y}(\boldsymbol{x}|\mathcal{D})\,|\,\boldsymbol{x}]-\mathrm{E}[t\,|\,\boldsymbol{x}])^2\right]+\mathrm{E}\left[\mathrm{Var}\left[\hat{y}(\boldsymbol{x}|\mathcal{D})\;\middle|\;\boldsymbol{x}\right]\right]+\mathrm{E}\left[\mathrm{Var}[t\,|\,\boldsymbol{x}]\right]$$

[1] Answer: $\mathrm{Var}[t]=\mathrm{E}[\mathrm{Var}[t\,|\,\boldsymbol{x}]]+\mathrm{Var}[\mathrm{E}[t\,|\,\boldsymbol{x}]]$. Why?

into average squared bias, average variance and intrinsic noise variance. While the third term is constant (and typically unknown), there is a tradeoff between the first two terms which can be explored by varying the model complexity (regularization parameter ν in the example above).

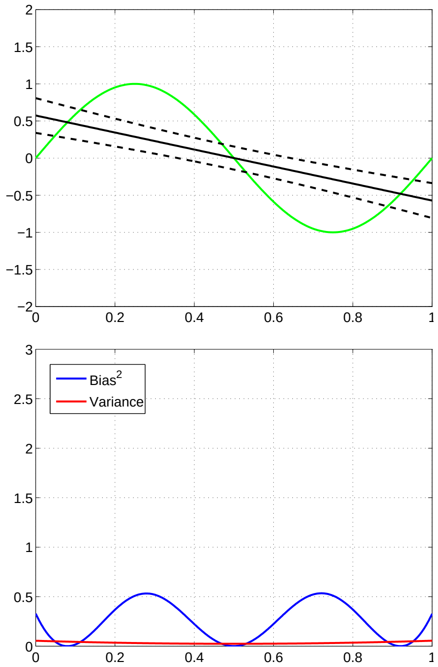


Figure 10.2: Bias and variance for least squares polynomial curve fitting. 1000 samples of size 12 each were drawn from $y_*(x) = \sin(2\pi x)$ plus $N(0, 1/16)$ noise, where x_i are drawn uniformly from $[0, 1]$. Each sample includes 0 and 1 among the x_i, to avoid high variance due to extrapolation. Note that this favours the higher-order polynomials.
Left column: Each sample is fitted by a line $y(x) = w_1 x + w_0$. The expected fit $\mathrm{E}[\hat{y}(x|\mathcal{D}) \,|\, x]$ (upper left; solid black) is rather poor, giving rise to substantial squared bias (lower left; blue). On the other hand, the variance across samples (lower left; red) is small, in that most fitted lines are close to the average. Right column: Each sample is fitted by a polynomial $y(x) = w_6 x^6 + \cdots + w_1 x + w_0$ of degree 6. The expected fit is very good (small squared bias), but the variance across samples (lower right; red) is large. Notice that these curves represent sample averages of the population quantities $\mathrm{E}[\hat{y}(x|\mathcal{D}) \,|\, x]$ and $\mathrm{Var}[\hat{y}(x|\mathcal{D}) \,|\, x]$ (averaged over 1000 samples).

10.1.4 Examples of Bias-Variance Decomposition

In this section, we work through a number of examples in order to develop an intuition about bias-variance decompositions. We will use the squared error throughout. The basic message is this. A simple model tends to give rise to large bias, but small variance, while a complex model often exhibits small bias and large variance. In Figure 10.2, we revisit the curve fitting problem of Section 4.1.

Data is sampled as $t_i = y_*(x_i) + \varepsilon_i$, $x_i \in [0, 1]$, $\varepsilon_i \sim N(0, \sigma^2)$, where $y_*(\cdot)$ is a smooth curve. We draw 1000 samples $\mathcal{D}$ of size 12 each and plot both average curve $\mathrm{E}[\hat{y}(x|\mathcal{D}) \,|\, x]$ and standard deviation $\mathrm{Var}[\hat{y}(x|\mathcal{D}) \,|\, x]^{1/2}$ in the upper row, squared bias and variance in the lower row. In each column, we use different function classes for regression: lines $y(x) = w_1 x + w_0$ on the left, polynomials of degree six on the right. The model on the left is less complex than the one on the right. The underlying $y_*(\cdot)$ is not well represented by any line, explaining the large bias on the left. On the other hand, the two parameters w_1, w_0 are robustly estimated even from little data, so the variance is small. In contrast, the average over six-degree polynomials represents $y_*(x) = \mathrm{E}[t \,|\, x]$ closely, so the bias on the right is small. However, as higher-order polynomials fit the erratic noise on top of the signal (over-fitting), the variance over samples $\mathcal{D}$ is large. Neither of these choices realize a good tradeoff between squared bias and variance.

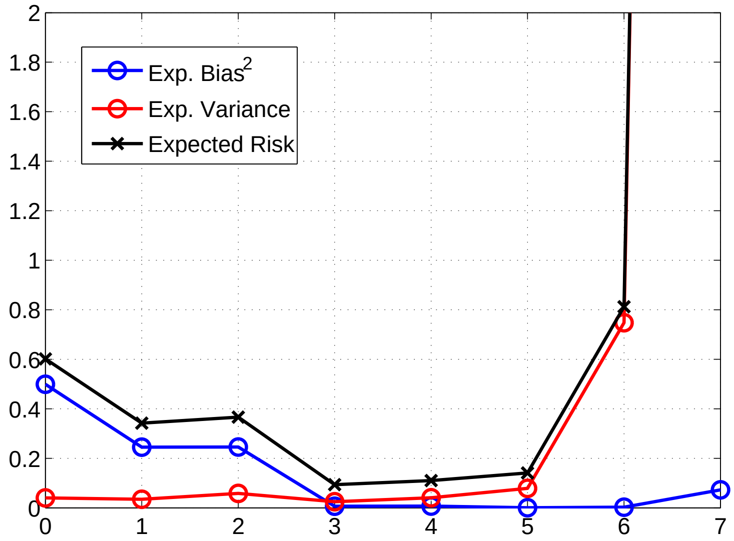


Figure 10.3: Bias-variance decomposition for least squares polynomial curve fitting. Shown are average squared bias $\mathrm{E}[(\mathrm{E}[\hat{y}(x|\mathcal{D}) \,|\, x] - \mathrm{E}[t \,|\, x])^2]$, average variance $\mathrm{E}[\mathrm{Var}[\hat{y}(x|\mathcal{D}) \,|\, x]]$ and expected test risk (sum of former two plus noise variance) as function of polynomial degree, estimated from the same data as in Figure 10.2. The best tradeoff between squared bias and variance (on average over x) is obtained for degree 3 polynomials.

In Figure 10.3, we repeat the same polynomial curve fitting setup (1000 training samples $\mathcal{D}$ of size 12 each), showing average squared bias, average variance and expected test risk as function of the polynomial degree ($p - 1$, where p is the number of weights). Here is a bias-variance tradeoff in action. As p (and therefore the model complexity) increases, the bias decreases, while the variance increases. The expected test risk is smallest for degree 3 ($p = 4$).

Regularized Least Squares Estimation (*)

One way to scale effective model complexity is regularization. Consider Tikhonov regularized linear least squares estimation (Section 7.2), based on functions $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$. Suppose that the true law is given by $t = y_*(\boldsymbol{x}) + \varepsilon$, where ε is independent zero-mean noise of variance σ^2. Moreover, let $\hat{y}_\nu(\boldsymbol{x}) = (\hat{\boldsymbol{w}}_\nu)^T \boldsymbol{\phi}(\boldsymbol{x})$ be the regularized least squares fit to the training sample $\mathcal{D}$, i.e. $\hat{\boldsymbol{w}}_\nu$ is the minimizer of (7.1). Note that $\mathrm{E}[\hat{y}_\nu(\boldsymbol{x}) \,|\, \boldsymbol{x}] = \mathrm{E}[\hat{\boldsymbol{w}}_\nu]^T \boldsymbol{\phi}(\boldsymbol{x})$. How does the choice of the regularization constant ν scale the bias-variance tradeoff? The simplest case is that the true curve is linear itself, $y_*(\boldsymbol{x}) = \boldsymbol{w}_*^T \boldsymbol{\phi}(\boldsymbol{x})$. It is well known that the standard least squares estimator is unbiased:

$$\mathrm{E}\left[\hat{\boldsymbol{w}}_0\right] = \mathrm{E}\left[\left(\boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^T\left(\boldsymbol{\Phi}\boldsymbol{w}_* + \boldsymbol{\varepsilon}\right)\right] = \mathrm{E}\left[\left(\boldsymbol{\Phi}^T\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w}_*\right] = \boldsymbol{w}_*.$$

In this case, the bias vanishes for $\nu = 0$, while being positive in general for $\nu > 0$. For general curves $y_*(\boldsymbol{x})$, we can show that the average bias on the training sample is nondecreasing in ν:

$$\mathrm{E}\left[\|\boldsymbol{t} - \boldsymbol{\Phi}\hat{\boldsymbol{w}}_{\nu_1}\|^2\right] \le \mathrm{E}\left[\|\boldsymbol{t} - \boldsymbol{\Phi}\hat{\boldsymbol{w}}_{\nu_2}\|^2\right], \quad \nu_1 < \nu_2.$$

This is because for any fixed sample $\mathcal{D}$, the training error is nonincreasing in ν (Section 7.2). Regularization is a means to decrease variance at the expense of increasing the bias.

Ensemble Methods (*)

A final example concerns ensembles (or committees) of predictors, a common technique for variance reduction. Suppose we use a number L of different learning methods, giving rise to predictors $\hat{y}_l(\boldsymbol{x})$ on a training sample $\mathcal{D}$. For example, the $\hat{y}_l(\boldsymbol{x})$ could be multi-layer perceptrons of different architecture, initialized in different ways. Or all $\hat{y}_l(\boldsymbol{x})$ could use the same method, but work on different[2] subsets of $\mathcal{D}$. If applied to data based on the underlying curve $y_*(\boldsymbol{x})$, we can write $\hat{y}_l(\boldsymbol{x}) = y_*(\boldsymbol{x}) + \hat{\varepsilon}_l(\boldsymbol{x})$ for $l = 1, \dots, L$, where $\hat{\varepsilon}_l(\boldsymbol{x})$ denotes the error committed by the l-th predictor. In the sequel, we concentrate on errors at a fixed $\boldsymbol{x}$, but average quantities can always be obtained by a further expectation. Let

$$\mathcal{E}_l(\boldsymbol{x}) = \mathrm{E}\left[(\hat{y}_l(\boldsymbol{x}) - y_*(\boldsymbol{x}))^2 \;\middle|\; \boldsymbol{x}\right] = \mathrm{E}\left[\hat{\varepsilon}_l(\boldsymbol{x})^2 \;\middle|\; \boldsymbol{x}\right].$$

Given these L methods, we can pick one of them for prediction. Doing so, the average error is

$$\mathcal{E}_{\mathrm{avg}}(\boldsymbol{x}) = \frac{1}{L}\sum_{l=1}^{L} \mathcal{E}_l(\boldsymbol{x}) = \frac{1}{L}\sum_{l=1}^{L} \mathrm{E}\left[\hat{\varepsilon}_l(\boldsymbol{x})^2 \;\middle|\; \boldsymbol{x}\right].$$

Instead, suppose we use an ensemble of *all* of them,

$$\hat{y}_{\mathrm{ens}}(\boldsymbol{x}) = \frac{1}{L}\sum_{l=1}^{L} \hat{y}_l(\boldsymbol{x}),$$

[2] A popular technique in statistics, not discussed in this course, is to use *bootstrap* resampling: each $\hat{y}_l(\boldsymbol{x})$ is trained on a sample $\mathcal{D}_l$ of the same size as $\mathcal{D}$, obtained from $\mathcal{D}$ by sampling *with* replacement. The rationale behind bootstrap resampling is discussed in [21, ch. 7].

whose error is

$$\begin{aligned}\mathcal{E}_{\text{ens}}(\boldsymbol{x}) &= \mathrm{E}\left[\left(L^{-1}\sum\nolimits_{l=1}^{L}(y_*(\boldsymbol{x})+\hat{\varepsilon}_l(\boldsymbol{x}))-y_*(\boldsymbol{x})\right)^2 \,\middle|\, \boldsymbol{x}\right] \\ &= \mathrm{E}\left[\left(L^{-1}\sum\nolimits_{l=1}^{L}\hat{\varepsilon}_l(\boldsymbol{x})\right)^2 \,\middle|\, \boldsymbol{x}\right].\end{aligned}$$

How do these errors compare? Using the Cauchy-Schwarz inequality (Section 2.3):

$$\left(\sum\nolimits_{l=1}^{L} 1\cdot\hat{\varepsilon}(\boldsymbol{x})\right)^2 \le L\sum\nolimits_{l=1}^{L}\hat{\varepsilon}(\boldsymbol{x})^2,$$

so that $\mathcal{E}_{\text{ens}}(\boldsymbol{x}) \le \mathcal{E}_{\text{avg}}(\boldsymbol{x})$. The ensemble error is never worse than the average error picking a single predictor. In the best case, it can be much better. Denote the expected error of each method by $B_l(\boldsymbol{x}) = \mathrm{E}[\hat{\varepsilon}_l(\boldsymbol{x}) \mid \boldsymbol{x}]$. Let us assume that the errors of different methods are uncorrelated: $\mathrm{Cov}[\hat{\varepsilon}_{l_1}(\boldsymbol{x}), \hat{\varepsilon}_{l_2}(\boldsymbol{x}) \mid \boldsymbol{x}] = 0$, $l_1 \ne l_2$. Then,

$$\begin{aligned}\mathcal{E}_{\text{ens}}(\boldsymbol{x}) &= L^{-2}\sum\nolimits_{l_1,l_2}\mathrm{E}\left[\hat{\varepsilon}_{l_1}(\boldsymbol{x})\hat{\varepsilon}_{l_2}(\boldsymbol{x}) \mid \boldsymbol{x}\right] \\ &= L^{-2}\sum\nolimits_{l_1,l_2}\left(\mathrm{Cov}\left[\hat{\varepsilon}_{l_1}(\boldsymbol{x}), \hat{\varepsilon}_{l_2}(\boldsymbol{x}) \mid \boldsymbol{x}\right] + B_{l_1}(\boldsymbol{x})B_{l_2}(\boldsymbol{x})\right) \\ &= \underbrace{L^{-2}\sum\nolimits_{l}\mathrm{Var}\left[\hat{\varepsilon}_l(\boldsymbol{x}) \mid \boldsymbol{x}\right]}_{\text{Variance}} + \underbrace{\left(L^{-1}\sum\nolimits_{l}B_l(\boldsymbol{x})\right)^2}_{\text{Bias}^2},\end{aligned}$$

while

$$\mathcal{E}_{\text{avg}}(\boldsymbol{x}) = \underbrace{L^{-1}\sum\nolimits_{l}\mathrm{Var}\left[\hat{\varepsilon}_l(\boldsymbol{x}) \mid \boldsymbol{x}\right]}_{\text{Avg. Variance}} + \underbrace{L^{-1}\sum\nolimits_{l}B_l(\boldsymbol{x})^2}_{\text{Avg. (Bias)}^2}.$$

The ensemble method reduces the variance by a factor of L in this case! Moreover, the bias is not increased:

$$\left(L^{-1}\sum\nolimits_{l}B_l(\boldsymbol{x})\right)^2 = L^{-2}\left(\sum\nolimits_{l}1\cdot B_l(\boldsymbol{x})\right)^2 \le L^{-1}\sum\nolimits_{l}B_l(\boldsymbol{x})^2,$$

using the Cauchy-Schwarz inequality. In most realistic situations, errors are correlated and gains are somewhat less dramatic. Ensemble methods are widely used in practice in order to reduce variance. Our analysis provides insight into how to choose component methods $\hat{y}_l(\cdot)$. They should be as diverse as possible, so to give rise to weakly correlated errors. Moreover, they should have small bias, since variance is mainly taken care of by the ensemble formation. An obvious drawback of ensemble methods is the increased computational complexity of the final predictor.

Finally, how about bias-variance decompositions for loss functions other than the squared one? There has been quite some work in this direction, but the exact additive decomposition of expected risk into squared bias and variance holds for the squared loss only. Having said that, an increase in bias or variance does in general imply an increase in expected risk.

10.2 Model Selection

Now that we understand what (expected) test risk is composed of, how can we estimate it? The main rationale for doing so in machine learning is model selection. Examples for model selection have been noted in Section 10.1. For simplicity, we will concentrate on selecting the regularization constant ν in Tikhonov-regularized least squares in this section, but the results are transferable. Recall that $\hat{y}_\nu(\boldsymbol{x}) = (\hat{\boldsymbol{w}}_\nu)^T \boldsymbol{\phi}(\boldsymbol{x})$, where

$$\hat{\boldsymbol{w}}_\nu = \underset{\boldsymbol{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n L(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i), t_i) + \nu \|\boldsymbol{w}\|^2, \quad L(y,t) = (y-t)^2,$$

moreover

$$\hat{R}_n(\hat{y}_\nu|\mathcal{D}) = \frac{1}{n} \sum_{i=1}^n L(\hat{y}_\nu(\boldsymbol{x}_i), t_i), \quad R(\hat{y}_\nu|\mathcal{D}) = \mathrm{E}\left[L(\hat{y}_\nu(\boldsymbol{x}), t) \;\middle|\; \mathcal{D} \right].$$

Our goal is to select a value of ν giving rise to small test risk $R(\hat{y}_\nu|\mathcal{D})$ or small expected test risk $\mathrm{E}[R(\hat{y}_\nu|\mathcal{D})]$.

As noted in Section 10.1.1, we can estimate $R(\hat{y}_\nu|\mathcal{D})$ using a validation dataset $\mathcal{D}_{\text{valid}}$ of size n_{valid}, independent of the training dataset $\mathcal{D}$. Since $\hat{y}_\nu(\cdot)$ is independent of $\mathcal{D}_{\text{valid}}$, we have that

$$R(\hat{y}_\nu|\mathcal{D}) = \mathrm{E}\left[L(\hat{y}_\nu(\boldsymbol{x}), t) \;\middle|\; \mathcal{D} \right] \approx \frac{1}{n_{\text{valid}}} \sum_{(\tilde{\boldsymbol{x}}_j, \tilde{t}_j) \in \mathcal{D}_{\text{valid}}} L(\hat{y}_\nu(\tilde{\boldsymbol{x}}_j), \tilde{t}_j)$$

by the law of large numbers. We can now minimize the estimator on the right hand side w.r.t. ν. This is typically done by evaluating the estimator over a candidate set of values for ν, then picking the minimizer. The problem with this approach is that $\mathcal{D}_{\text{valid}}$ cannot be used for training. In the remainder of this section, we discuss a technique for model selection on the training dataset only.

10.2.1 Cross-Validation

Given some dataset $\mathcal{D}$ of size n, we would like to split it into a training and a validation part of $4/5$ and $1/5$ the total size respectively. Let us partition the complete set randomly into 5 equisized parts, $\mathcal{D}_1, \dots, \mathcal{D}_5$, then use parts 2–5,

$$\mathcal{D}_{-1} = \mathcal{D} \setminus \mathcal{D}_1 = \bigcup_{k=2,\dots,5} \mathcal{D}_k,$$

as training dataset and part 1, $\mathcal{D}_1$, as validation dataset. More specifically, we train $\hat{y}_\nu(\cdot)$ on $\mathcal{D}_{-1}$, then estimate the test risk using $\mathcal{D}_1$. Denote this test risk estimator by $\hat{R}^{(-1)}(\mathcal{D})$ (test your understanding by working out its expression). But what is special about $\mathcal{D}_1$ for validation, why not $\mathcal{D}_2$? That results in another estimator $\hat{R}^{(-2)}(\mathcal{D})$. Given how we got here, it is clear that the different estimators $\hat{R}^{(-m)}(\mathcal{D})$, $m = 1, \dots, 5$, have the same distribution. Their expected value is equal to the expected test risk $\mathrm{E}[R(\hat{y}_\nu|\mathcal{D}')]$ for training samples $\mathcal{D}'$ of size $4n/5$. We might as well use all of them in a round-robin fashion:

$$\frac{1}{5} \sum_{m=1}^5 \hat{R}^{(-m)}(\mathcal{D}) \approx \mathrm{E}[R(\hat{y}_\nu|\mathcal{D}')], \quad |\mathcal{D}'| = 4n/5.$$

This is the 5-fold cross-validation (CV) estimator. Note how each case $(\boldsymbol{x}_i, t_i)$ in $\mathcal{D}$ is used once for validation and four times for training. Defining the CV estimator requires some notation, which can be confusing. You should be guided by the idea, which is simple indeed:

- Split the available training dataset $\mathcal{D}$ into M equisized[3] parts (or folds) $\mathcal{D}_m$, which do not overlap. In the previous example, $M = 5$.
- For $m = 1, \dots, M$: Train on $\mathcal{D}_{-m} = \mathcal{D} \setminus \mathcal{D}_m$, then evaluate the validation risk on $\mathcal{D}_m$. Importantly, $\mathcal{D}_m$ was not used for training.
- Average the M validation risk values obtained in this round-robin way.

The *M-fold cross-validation* (CV) estimator is formally defined as follows. For simplicity, assume that $n = |\mathcal{D}|$ is a multiple of $M \geq 2$. Partition the dataset $\mathcal{D}$ into M parts of size n/M each. Define $m(i) \in \{1, \dots, M\}$ so that $(\boldsymbol{x}_i, t_i) \in \mathcal{D}_{m(i)}$. We need to train M predictors, one on each $\mathcal{D}_{-m}$:

$$\begin{aligned}\hat{y}_\nu^{(-m)}(\cdot) &= (\hat{\boldsymbol{w}}_\nu^{(-m)})^T \boldsymbol{\phi}(\cdot), \\ \hat{\boldsymbol{w}}_\nu^{(-m)} &= \operatorname*{argmin}_{\boldsymbol{w}} \frac{M}{(M-1)n} \sum_{i:m(i)\neq m} L(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i), t_i) + \nu \|\boldsymbol{w}\|^2.\end{aligned}$$

Then, the M-fold cross-validation (CV) estimator is

$$\hat{R}_{\mathrm{CV}}^{(M)}(\mathcal{D}) = \frac{1}{n} \sum_{i=1}^n L\left(\hat{y}_\nu^{(-m(i))}, t_i\right) = \frac{1}{M} \sum_{m=1}^M \underbrace{\frac{M}{n} \sum\nolimits_{i:m(i)=m} L\left(\hat{y}_\nu^{(-m)}, t_i\right)}_{=\hat{R}^{(-m)}(\mathcal{D})}.$$

We can choose M between 2 and n. If $M = n$, then $\mathcal{D}_m = \{(\boldsymbol{x}_m, t_m)\}$, so that we leave out single cases for the n validations. This particular case is called *leave-one-out cross-validation.* We have that

$$\mathrm{E}\left[\hat{R}_{\mathrm{CV}}^{(M)}(\mathcal{D})\right] = \mathrm{E}\left[R(\hat{y}_\nu | \mathcal{D}')\right], \quad |\mathcal{D}'| = (M-1)n/M,$$

where $\mathcal{D}'$ is an i.i.d. sample of size $(M-1)n/M$ from the true underlying distribution. For not too small M, $\hat{R}_{\mathrm{CV}}^{(M)}(\mathcal{D})$ can be used as approximate estimator of the expected test risk $\mathrm{E}[R(\hat{y}_\nu|\mathcal{D})]$ for the full sample. In Figure 10.4, we compare the expected tesk risk with both 5-fold and leave-one-out CV estimators on the polynomial curve fitting task used already above (we use a sample size of 50 here, since cross-validation is unrealiable for small datasets). While CV somewhat overestimates the test risk for too small and too large degrees, its minimum points coincide with those of the expected test risk in this example. Notice how it errs on the conservative side, and how its variance (over samples) grows sharply with polynomial degree.

The curious reader may wonder how we computed the leave-one-out cross-validation estimator in Figure 10.4. Do we have to run least squares regression 50 times on samples of size 49? And what if $n = 1000$ or a million? In this light,

[3] In practice, the folds can be of slightly different size, which does not change the general procedure.

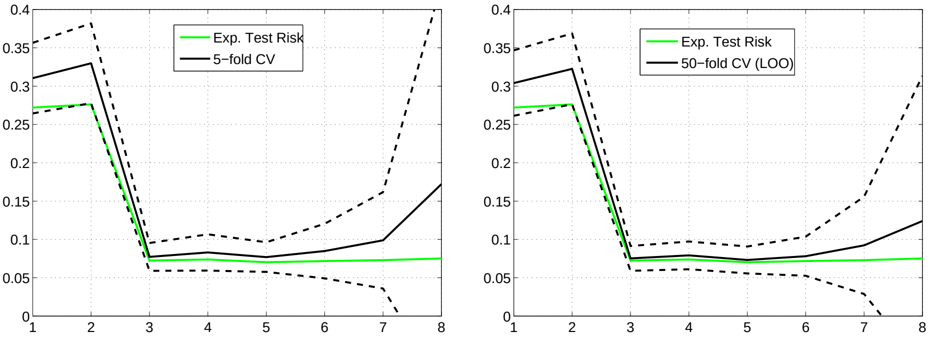


Figure 10.4: Cross-validation estimators versus expected test risk for polynomial curve fitting task. 100 samples of size 50 each were drawn from $y_*(x) = \sin(2\pi x)$ plus $N(0, 1/16)$ noise, where x_i are drawn uniformly from $[0, 1]$. Each sample includes 0 and 1 among the x_i, to avoid high variance due to extrapolation. Shown are expected test risk (green) and its cross-validation estimate (mean (solid) and standard deviation (dashed) over 100 samples), as function of polynomial degree. Left: 5-fold cross-validation (test batches of size 10). Right: 50-fold leave-one-out cross-validation.

leave-one-out CV seems more of an academic exercise. Fortunately, it is possible to evaluate this estimator much more efficiently, at least for (kernelized) linear methods, as is shown in Section 10.2.2.

How shall we choose M? Again, there is no general best recipe. Let us gain insight by applying the bias-variance concept to cross-validation estimators themselves. For the largest choice $M = n$ (leave-one-out), the expected value of $\hat{R}^{(n)}_{\mathrm{CV}}(\mathcal{D})$ is closest to the expected test risk for our sample size: leave-one-out exhibits the smallest bias. The bias of $\hat{R}^{(M)}_{\mathrm{CV}}(\mathcal{D})$ for small M can be problematic with kernelized estimators, where the optimal value of ν is a decreasing function of the training set size n: cross-validation may regularize more than needed in such cases. On the other hand, the variance of leave-one-out CV can be large, since the individual "training subsets" $\mathcal{D}_{-m}$ are strongly overlapping, so that $\hat{y}^{(-m)}_\nu$ are highly dependent. Finally, leave-one-out cross-validation can be problematic for computational[4] reasons, since we have to train n different predictors on samples $\mathcal{D}_{-m}$ as large as $\mathcal{D}$ itself. For kernelized estimators, efficient approximations to leave-one-out have been developed in statistics [21, sect. 7.10.1]. In practice, the choices $M = 5$ and $M = 10$ are frequently used.

We stress again that cross-validation, whatever the choice of M, is not a foolproof method for model selection, but should rather be viewed as a convenient heuristic which often works well and is very widely used in machine learning. The theoretical analysis of cross-validation is somewhat intricate and out of scope of this course. In the remainder of this section, we give some hints about using CV in practice, mainly based on [21, ch. 7]. First, the variance of $\hat{R}^{(M)}_{\mathrm{CV}}(\mathcal{D})$ can be large, in particular for large M, and if the expected curve (as function

[4] It is sometimes implied that leave-one-out CV is superior to other choices of M (you pay more, you get more). This is not true in general. Independent of computational issues, a bias-variance tradeoff has to be faced.

of ν) is rather flat, the CV minimizer for ν may be determined mainly by the variance. We can get an idea about the variance by computing the standard deviation between the different parts $\hat{R}^{(-m)}(\mathcal{D})$, $m = 1, \dots, M$, and it is recommended to always determine this standard error along with CV scores. A rule of thumb for choosing ν is to look at both the CV score curve and standard errors. Suppose the CV score is minimized at ν_*, attaining score $\mathrm{CV}(\nu_*)$ and standard error $\mathrm{std}(\nu_*)$. We pick the largest ν (simplest model) such that $\mathrm{CV}(\nu) \leq \mathrm{CV}(\nu_*) + \mathrm{std}(\nu_*)$.

It is important to use cross-validation correctly in practice. In real-world applications, it is customary (and probably unavoidable) to mix many different machine learning and preprocessing techniques, often in an ad-hoc fashion. A common but erroneous way to use CV goes as follows (a worked-out example is given in [21, sect. 7.10.2]). Suppose our problem is noisy and comes with high-dimensional input points $\boldsymbol{x}$. First, as part of the preprocessing, we screen a large number of potential predictors on the training sample $\mathcal{D}$, retaining only those in the feature map $\boldsymbol{\phi}(\cdot)$ which exhibit a substantial correlation with the target t. Second, we use CV in order to select the regularization parameter ν and to estimate the expected test risk. Used in this way, CV tends to underestimate the test risk dramatically, and we run into over-fitting without a warning. What is wrong? Cross-validation has to applied in the most outer loop of the whole procedure, including the preprocessing if this makes use of the training set labels[5] $\{t_i\}$. The first thing we do is splitting the data into $\mathcal{D}_m$, then we run preprocessing and training on the different $\mathcal{D}_{-m}$ separately.

Other Model Selection Techniques (*)

Cross-validation is maybe the most widely used model selection technique in machine learning, but it is not the only one. First, there is a range of simpler alternatives from statistics. The general idea is to realize that the expected risk $\mathrm{E}[R(\hat{y}_\nu|\mathcal{D})]$ is typically underestimated by the training risk $\hat{R}_n(\hat{y}_\nu)$ by an amount which decreases with the sample size n, but increases with the model complexity (governed by ν in our running example). Under some more or less plausible arguments, this "complexity term" can be represented by a tractable expression. Implementations of this idea are AIC, BIC or minimum description length (MDL). Compared to CV, these are typically more efficient to use, but can be less reliable, in particular for complex nonlinear models. In learning theory, concentration inequalities and Vapnik-Chervonenkis arguments are used to bound $\mathrm{E}[R(\hat{y}_\nu|\mathcal{D})]$ in terms of $\hat{R}_n(\hat{y}_\nu)$ and complexity terms, but these bounds are typically too lose to be useful in practice.

A general problem with hold out and cross-validation estimation is that only a very small number of parameters can be selected. For models with many hyperparameters, *Bayesian model selection* techniques can be far more useful in practice, and they have found widespread use in machine learning. For example, they led to many advances for multi-layer perceptrons [30]. The interested reader is encouraged to study [6, sect. 3.3, sect. 3.4]. Recall that most of the trouble with

[5] Preprocessing involving the input points $\{\boldsymbol{x}_i\}$ only can be done up front, before CV, since this does not reveal information about $\boldsymbol{x} \mapsto t$ which would give preselected predictors an opportunity for over-fitting.

over-fitting comes from the fact that weights $\boldsymbol{w}$ are fitted to data $\mathcal{D}$ in the first place. In maximum likelihood and MAP estimation, we do treat $\boldsymbol{w}$ as a random variable. The sum rule of probability (Chapter 5) tells us to *marginalize* over $\boldsymbol{w}$ in order to robustly estimate hyperparameters such as ν from data. Following this argument, we should maximize the log *marginal* likelihood

$$\log p(\mathcal{D}|\nu) = \log \int p(\mathcal{D}|\boldsymbol{w}, \nu) p(\boldsymbol{w}|\nu)\, d\boldsymbol{w}$$

in order to choose a hyperparameter ν. Marginal likelihood functions play a pivotal role for unsupervised learning (Chapter 12). While their computation is often intractable, it can be approximated in a number of ways [6, ch. 10]. The particularly simple Laplace approximation of the log marginal likelihood gives rise to the evidence framework [6, sect. 3.4], [30], which can be seen as generalization of BIC and MDL to general nonlinear models such as MLPs.

10.2.2 Leave-One-Out Cross-Validation (*)

For a dataset of size n, a naive way to compute the leave-one-out cross-validation (LOO CV) estimator is to recompute $\hat{y}^{(-m)}(\cdot)$ from scratch for each subset, at about n times the cost of a single linear regression. Under such circumstances, LOO CV would not be a tractable option in practice. In this section, we show how the LOO CV can be computed much more efficiently for a linear regression model, effectively at the cost of one linear regression.

Denote the design matrix by $\boldsymbol{\Phi}$, the target vector by $\boldsymbol{t}$. The weights $\boldsymbol{w}$ for the full dataset are given by the normal equations:

$$\boldsymbol{A}\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{t} = \sum_j t_j \boldsymbol{\phi}_j, \quad \boldsymbol{A} = \boldsymbol{\Phi}^T\boldsymbol{\Phi} = \sum_j \boldsymbol{\phi}_j\boldsymbol{\phi}_j^T.$$

Suppose we leave out pattern $(\boldsymbol{\phi}_i, t_i)$. The normal equations for $\hat{y}^{(-i)} = (\boldsymbol{w}_{(-i)})^T\boldsymbol{\phi}(\cdot)$ are

$$\left(\boldsymbol{A} - \boldsymbol{\phi}_i\boldsymbol{\phi}_i^T\right)\boldsymbol{w}_{(-i)} = \boldsymbol{\Phi}^T\boldsymbol{t} - t_i\boldsymbol{\phi}_i = \boldsymbol{A}\boldsymbol{w} - t_i\boldsymbol{\phi}_i.$$

Rearranging this equation, we see that $\boldsymbol{A}(\boldsymbol{w}_{(-i)} - \boldsymbol{w}) = -\alpha_i\boldsymbol{\phi}_i$ for some $\alpha_i \in \mathbb{R}$, or $\boldsymbol{w}_{(-i)} = \boldsymbol{w} - \alpha_i\boldsymbol{A}^{-1}\boldsymbol{\phi}_i$. Plugging this ansatz into the LOO normal equations:

$$\begin{aligned}\left(\boldsymbol{A} - \boldsymbol{\phi}_i\boldsymbol{\phi}_i^T\right)\left(\boldsymbol{w} - \alpha_i\boldsymbol{A}^{-1}\boldsymbol{\phi}_i\right) &= \boldsymbol{\Phi}^T\boldsymbol{t} - \alpha_i\boldsymbol{\phi}_i - (\boldsymbol{\phi}_i^T\boldsymbol{w})\boldsymbol{\phi}_i + \alpha_i(\boldsymbol{\phi}_i^T\boldsymbol{A}^{-1}\boldsymbol{\phi}_i)\boldsymbol{\phi}_i \\ &\overset{!}{=} \boldsymbol{\Phi}^T\boldsymbol{t} - t_i\boldsymbol{\phi}_i.\end{aligned}$$

Rearraging this equation gives

$$\alpha_i\left(1 - \boldsymbol{\phi}_i^T\boldsymbol{A}^{-1}\boldsymbol{\phi}_i\right) = t_i - \boldsymbol{\phi}_i^T\boldsymbol{w} = r_i, \quad \boldsymbol{r} = \boldsymbol{t} - \boldsymbol{\Phi}\boldsymbol{w}.$$

The LOO CV estimator is the sum of squares of the LOO residuals

$$t_i - \boldsymbol{\phi}_i^T\boldsymbol{w}_{(-i)} = r_i + \alpha_i\boldsymbol{\phi}_i^T\boldsymbol{A}^{-1}\boldsymbol{\phi}_i = r_i + \alpha_i\left(\boldsymbol{\phi}_i^T\boldsymbol{A}^{-1}\boldsymbol{\phi}_i - 1 + 1\right) = \alpha_i,$$

so that

$$\hat{R}^{(n)}_{\mathrm{CV}}(\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \alpha_i^2.$$

How do we compute the α_i efficiently? We need the residuals $\boldsymbol{r} = \boldsymbol{t} - \boldsymbol{\Phi}\boldsymbol{w}$, moreover the vector $[\boldsymbol{\phi}_i^T \boldsymbol{A}^{-1}\boldsymbol{\phi}_i]$. The latter is the diagonal of the matrix $\boldsymbol{\Phi}\boldsymbol{A}^{-1}\boldsymbol{\Phi}^T$. Recall how the least squares problem can be solved by the QR decomposition (Section 4.2.3):

$$\boldsymbol{\Phi} = \boldsymbol{Q}\boldsymbol{R} \quad \Rightarrow \quad \boldsymbol{w} = \boldsymbol{A}^{-1}\boldsymbol{\Phi}^T\boldsymbol{t} \overset{!}{=} \boldsymbol{R}^{-1}\boldsymbol{Q}^T\boldsymbol{t}.$$

Since $\boldsymbol{t}$ is arbitrary in this equation, we have that $\boldsymbol{A}^{-1}\boldsymbol{\Phi}^T = \boldsymbol{R}^{-1}\boldsymbol{Q}^T$, therefore

$$\boldsymbol{\Phi}\left(\boldsymbol{A}^{-1}\boldsymbol{\Phi}^T\right) = \boldsymbol{Q}\boldsymbol{R}\boldsymbol{R}^{-1}\boldsymbol{Q}^T = \boldsymbol{Q}\boldsymbol{Q}^T,$$

a result we derived previously in Section 4.2.3. Therefore, if $\boldsymbol{Q} = [\boldsymbol{q}_k]$, then

$$[\boldsymbol{\phi}_i^T \boldsymbol{A}^{-1}\boldsymbol{\phi}_i] = \operatorname{diag}\left(\boldsymbol{\Phi}\boldsymbol{A}^{-1}\boldsymbol{\Phi}^T\right) = \sum_{k=1}^{d} (\boldsymbol{q}_k)^2,$$

where $(\boldsymbol{q}_k)^2$ denotes the pointwise multiplication of $\boldsymbol{q}_k$ with itself. Given that $\boldsymbol{w}$ is determined by a QR decomposition, the additional cost of LOO cross-validation is negligible.

Chapter 11

Dimensionality Reduction

In many, if not most real-world machine learning problems, data cases are most naturally represented in terms of very high-dimensional vectors. Our canonical handwritten digits recognition problem is rather on the low end in this respect. High-resolution images have millions of pixels. In speech recognition, audio waveforms are represented by a large number of windowed Fourier transform features. Text documents are commonly represented as count vectors w.r.t. some dictionary, which for realistic corpora contains hundreds of thousands of words. If learning theory tells us one thing, it is that learning in such huge-dimensional spaces is impossible in general. A practical way out of this dilemma is to reduce the dimensionality of attributes by some form of feature mapping.

In this chapter, we will learn to know some of the most widely used *linear* dimensionality reduction techniques. Principal components analysis is among the most fundamental of all techniques used in machine learning, and we will cover it in some detail. Linear dimensionality reduction techniques are typically based on the eigendecomposition of certain sample covariance matrices, and this foundation will be studied in detail. Other more advanced machine learning techniques share the same mathematical foundations: spectral clustering, manifold learning, multi-dimensional scaling and metric embedding techniques for visualization.

11.1 Principal Components Analysis

In this section, we discuss the most important dimensionality reduction technique in machine learning and statistics: principal components analysis (PCA). In Section 11.1.1, we will arrive at PCA along three seemingly different routes. We gain intuition about PCA by applying it to handwritten digits data. The technique behind PCA, eigendecomposition, is reviewed in Section 11.1.2. In Section 11.1.3, we show how to compute PCA in practice for datasets of moderate size. Companies like your favourite web search engine routinely compute PCA directions for datasets of astronomical size. We give some hints into how this is done in Section 11.1.4.

Many machine learning problems are characterized by input points living in very high-dimensional spaces. For example, the MNIST handwritten digit bitmaps are of size 28 × 28, so can be seen as vectors in $\mathbb{R}^{784}$. In bioinformatics, gene microarray measurements can easily give rise to input space dimensionalities of many thousands. It is important to understand that the underlying "effective" dimensionality of what is predictively relevant about a signal is typically much smaller. After all, it is impossible to do meaningful statistics on general distributions in hundreds or thousands of dimensions, as there is simply never enough training data to explore the vast majority of dependencies between attributes. This observation is sometimes termed "curse of dimensionality", but let us be precise. The general impossibility of statistics in $\mathbb{R}^{1000}$ is a fact, there is nothing to lament about it. If a problem cannot be represented using much fewer unknowns, it is out of scope and does not deserve[1] our attention. Fortunately, many real-world problems are amenable to useful low-dimensional modelling. The curse is that we have to *find* these adequate low-dimensional representations, and we have to do so in an automated data-driven way. Given we succeed, we will have found a probabilistic low-dimensional representation of the high-dimensional input variable, which conveys enough of the latter's predictive information: this is *dimensionality reduction*. In general, relevant parameters may be nonlinearly related to input points, and this process of "finding needles in haystacks" can be very challenging. In this chapter, we will concentrate on *linear dimensionality reduction*.

Consider our running MNIST 8s versus 9s classification problem from Chapter 2. Suppose we want to apply a maximum likelihood plug-in rule with Gaussian class-conditional distributions $p(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, $t = -1, +1$, and $\boldsymbol{x} \in \mathbb{R}^d$, where $d = 784$. How should we parameterize the covariances $\boldsymbol{\Sigma}_t$? A simple choice would be $\boldsymbol{\Sigma}_t = \boldsymbol{I}$, leaving us only with the means $\boldsymbol{\mu}_t$ to be estimated. However, the variances of pixels are clearly different across the bitmap. Pixels at the boundary almost always have values close to zero, the variance there is much smaller than in the center. Diagonal covariances $\boldsymbol{\Sigma}_t$ would capture different variances at a modest increase in the number of parameters to be estimated. Still, these imply pairwise conditional independence of individual pixels, an assumption which is clearly violated in our data. Digits are composed of smooth pen strokes, which induce substantial correlations between neighbouring pixel values. Full matrices $\boldsymbol{\Sigma}_t$ then? Unfortunately, this needs $d(d+1)/2$ for $\boldsymbol{\Sigma}_t$, so $d(d+3)/2 = 308504$ parameters for each class. We cannot reliably estimate this many parameters from our limited training data.

Covariances between pixels matter, but maybe not *all* of them do. Here is the idea behind linear dimensionality reduction. Instead of modelling the high-dimensional input variable $\boldsymbol{x} \in \mathbb{R}^d$, we map it to $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$ and model $\boldsymbol{z} \in \mathbb{R}^M$ instead. Since M is chosen much smaller than d, we can afford full covariance matrices for the class-conditionals on $\boldsymbol{z}$. This is of course a special case of a linear feature map, $\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{U}^T\boldsymbol{x}$. The matrix $\boldsymbol{U} \in \mathbb{R}^{d\times M}$ determines the dimensionality reduction. It has to be learned from training data for the whole idea to work well. In this section, we concentrate on learning $\boldsymbol{U}$ in order to represent an input point distribution per se, ignoring class label information even if such is available (this is a simple example of unsupervised learning, see

[1] Unless a financial analysis firm pays you lots of money for it.

Chapter 12). We will take several different routes and end up at the same idea: principal components analysis.

We will adopt the following conventions. First, we will consider zero-mean distributions on $\boldsymbol{x}$ only. Given we work with data, we first compute the sample mean $\hat{\boldsymbol{\mu}} = n^{-1}\sum_i \boldsymbol{x}_i$ and subtract it off. Without this preprocessing, the feature map would be $\boldsymbol{U}^T(\boldsymbol{x} - \hat{\boldsymbol{\mu}})$. Second, we will restrict $\boldsymbol{U}$ to have orthonormal columns: $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I} \in \mathbb{R}^{M\times M}$. In other words, $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$ is the "encoding" part of an orthogonal projection (Section 4.2.2). To justify this restriction, notice that we can decompose any full-rank matrix in $\mathbb{R}^{d\times M}$ as product of $\boldsymbol{U}$ with orthonormal columns and an invertible $\mathbb{R}^{M\times M}$ matrix (QR decomposition, Section 4.2.3), and we can absorb the latter into the defition of $\boldsymbol{z}$ at no loss.

What is a good projection $\boldsymbol{U}$ for dimensionality reduction? Let us collect some general criteria of merit.

- Retaining covariance: A distinct property of the distribution of $\boldsymbol{x}$ is its covariance $\mathrm{Cov}[\boldsymbol{x}]$. As data is typically noisy, directions of small covariance are most likely due to random errors, while the signal is often shaped by directions of large covariance. The covariance of $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$ is $\mathrm{Cov}[\boldsymbol{z}] = \boldsymbol{U}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U}$ (Section 5.1.3), and it seems sensible to choose $\boldsymbol{U}$ so as to maximize this covariance.
- Minimizing reconstruction error of orthogonal projection: We can understand dimensionality reduction as an *autoencoding* process. First, we encode $\boldsymbol{x}$ by the coefficients $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$. Second, we reconstruct it as
$$\hat{\boldsymbol{x}} = \boldsymbol{U}\boldsymbol{z} = \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{x}.$$
Recall from Section 4.2.2 that $\boldsymbol{U}\boldsymbol{U}^T$ defines an orthogonal projection $\boldsymbol{x} \mapsto \hat{\boldsymbol{x}}$ onto the M-dimensional linear subspace $\boldsymbol{U}\mathbb{R}^M$ of $\mathbb{R}^d$. In order to retain as much information in $\hat{\boldsymbol{x}}$ about $\boldsymbol{x}$ as possible, we should choose $\boldsymbol{U}$ so to minimize the squared reconstruction error $\mathrm{E}[\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|^2]$.
- Removing linear redundancies: It is often the case with high-dimensional variables $\boldsymbol{x}$ that several of the components are highly correlated or anticorrelated: $\mathrm{Cov}[x_j, x_k] \approx \pm\sqrt{\mathrm{Var}[x_j]\,\mathrm{Var}[x_k]}$ (Section 6.3). This means that x_j and x_k are approximately linear functions of each other. For example, in natural images the intensities of neighbouring pixels are often highly correlated. Modelling such components is wasteful. Given one of them, the other does not convey much additional information. We should aim for a *decorrelating* transformation $\boldsymbol{U}$, in that the components of $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$ should be uncorrelated.

At first sight, these requirements seem to have little to do with each other. However, we will see that they are closely related. We can optimally satisfy all three of them by principal components analysis.

11.1.1 Three Ways to Principal Components Analysis

Let us begin by searching for a single direction $\boldsymbol{u} \in \mathbb{R}^d$ (case $M = 1$, so that $z = \boldsymbol{u}^T\boldsymbol{x} \in \mathbb{R}$), where $\|\boldsymbol{u}\| = 1$. Consider the data in Figure 11.1, and compare

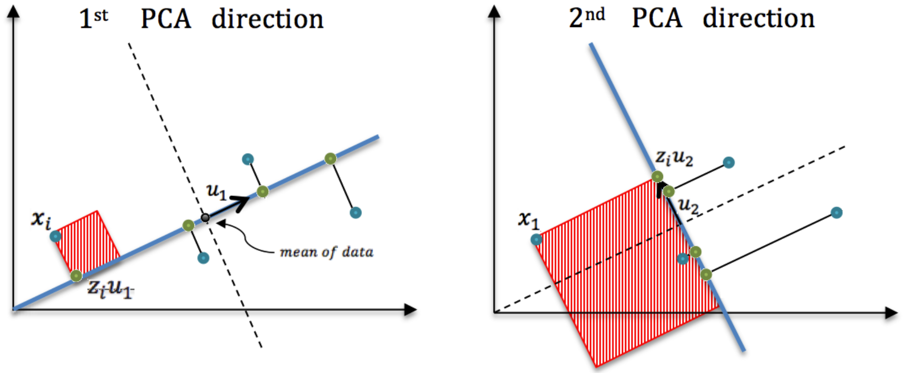


Figure 11.1: Illustration of principal components direction for data (blue dots) in $\mathbb{R}^2$. The first PCA direction minimizes the squared reconstruction error between points and their projections (green dots) onto the PCA subspace. Squared error terms are visualized as area of the red squares.

the two different directions $\boldsymbol{u}$. Since $\|\boldsymbol{u}\| = 1$, $z_i = \boldsymbol{u}^T \boldsymbol{x}_i$ is the signed distance from zero of the projection onto $\boldsymbol{u}\mathbb{R}$. The reconstruction errors are the distances $\|\boldsymbol{x}_i - z_i \boldsymbol{u}\|$. For the direction on the right, values of z_i are larger than on the left, giving rise to a larger sample variance of z. At the same time, the reconstruction errors are smaller on average. The total variance of the data is decomposed into variance along $\boldsymbol{u}\mathbb{R}$ (therefore, variance of z) and variance orthogonal to the direction (squared reconstruction error).

To fully understand this, let us abstract from datasets and assume that we know the distribution of $\boldsymbol{x}$ (recall that we have $\mathrm{E}[\boldsymbol{x}] = \boldsymbol{0}$). The problem of minimizing squared reconstruction error translates into

$$\min_{\boldsymbol{u}\colon \|\boldsymbol{u}\|=1} \mathrm{E}\left[\min_{z} \|\boldsymbol{x} - z\boldsymbol{u}\|^2\right].$$

For each $\boldsymbol{x}$, inside the expectation, we can choose the encoding coefficient z so to minimize $\|\boldsymbol{x} - z\boldsymbol{u}\|^2$. The direction $\boldsymbol{u}$ should work well on average over $\boldsymbol{x}$, so the minimization is outside. We already know that z is chosen by way of orthogonal projection (Section 4.2.2), but let us derive it again:

$$\frac{\partial}{\partial z}\|\boldsymbol{x} - z\boldsymbol{u}\|^2 = 2(z\boldsymbol{u} - \boldsymbol{x})^T \boldsymbol{u} \overset{\boldsymbol{u}^T\boldsymbol{u}=1}{=} 2(z - \boldsymbol{x}^T\boldsymbol{u}) = 0 \quad \Leftrightarrow \quad z = \boldsymbol{u}^T\boldsymbol{x}.$$

Plugging this minimizer in:

$$\begin{aligned} \mathrm{E}\left[\|\boldsymbol{x} - (\boldsymbol{u}^T\boldsymbol{x})\boldsymbol{u}\|^2\right] &= \mathrm{E}\left[\|\boldsymbol{x}\|^2 - 2(\boldsymbol{u}^T\boldsymbol{x})^2 + (\boldsymbol{u}^T\boldsymbol{x})^2\boldsymbol{u}^T\boldsymbol{u}\right] \\ &= \mathrm{E}\left[\|\boldsymbol{x}\|^2 - (\boldsymbol{u}^T\boldsymbol{x})^2\right] = \mathrm{E}\left[\|\boldsymbol{x}\|^2\right] - \boldsymbol{u}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}. \end{aligned}$$

We used that $\boldsymbol{u}^T\boldsymbol{u} = 1$ and

$$\mathrm{E}\left[(\boldsymbol{u}^T\boldsymbol{x})^2\right] = \boldsymbol{u}^T\mathrm{E}[\boldsymbol{x}\boldsymbol{x}^T]\boldsymbol{u} = \boldsymbol{u}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}.$$

As the first term does not depend on $\boldsymbol{u}$, our optimal direction is given by

$$\boldsymbol{u}_* = \operatorname*{argmax}_{\boldsymbol{u}\colon \|\boldsymbol{u}\|=1} \boldsymbol{u}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}.$$

This is also the direction for which $z = \boldsymbol{u}^T \boldsymbol{x}$ has maximum variance, since

$$\mathrm{Var}[z] = \mathrm{E}\left[(\boldsymbol{u}^T \boldsymbol{x})^2\right] = \boldsymbol{u}^T \mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}$$

(recall that $\mathrm{E}[\boldsymbol{x}] = 0$, so that $\mathrm{E}[z] = 0$). The solution $\boldsymbol{u}_*$ is called *first principal components* direction, and the corresponding $z_* = \boldsymbol{u}_*^T \boldsymbol{x}$ is called *first principal component*[2] of the random variable $\boldsymbol{x}$.

Importantly, this definition depends on the covariance of $\boldsymbol{x}$ only. In practice, this means that all we need to estimate of $\boldsymbol{x}$ are first and second order moments (mean and covariance). In fact, our argument amounts to a decomposition of the total amount of covariance:

$$\begin{aligned}\mathrm{E}[\|\boldsymbol{x}\|^2] &= \operatorname{tr}\mathrm{E}[\boldsymbol{x}^T\boldsymbol{x}] = \operatorname{tr}\mathrm{E}[\boldsymbol{x}\boldsymbol{x}^T] = \operatorname{tr}\mathrm{Cov}[\boldsymbol{x}] \\ &= \underbrace{\mathrm{Var}[\boldsymbol{u}^T\boldsymbol{x}]}_{\text{variance explained}} + \underbrace{\mathrm{E}\left[\|\boldsymbol{x} - (\boldsymbol{u}^T\boldsymbol{x})\boldsymbol{u}\|^2\right]}_{\text{squared error}},\end{aligned}$$

where the "amount of covariance" is measured as $\operatorname{tr}\mathrm{Cov}[\boldsymbol{x}]$.

The first principal components direction $\boldsymbol{u}_*$ is a leading eigendirection of the symmetric positive definite matrix $\mathrm{Cov}[\boldsymbol{x}]$, an eigenvector corresponding to the maximum eigenvalue of $\mathrm{Cov}[\boldsymbol{x}]$. Refresh your memory on all things eigen in Section 11.1.2 (no harm to jump there now). Namely, there is some $\lambda_* > 0$ such that

$$\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}_* = \lambda_* \boldsymbol{u}_*.$$

Moreover, λ_* is the largest scalar[3] with this property. We establish this fact at the end of Section 11.1.2.

Multiple Directions

What about more than one direction? Denote the first principal components direction by $\boldsymbol{u}_1$, how shall we choose a second one, $\boldsymbol{u}_2$? This choice has to be constrained, otherwise we would simply choose $\boldsymbol{u}_1$ again. By our convention, $\boldsymbol{u}_2$ is orthogonal to $\boldsymbol{u}_1$. We will see in a moment that this is particularly well suited to principal components. We can now define $\boldsymbol{u}_2$ in the same way as above, subject to $\boldsymbol{u}_2^T \boldsymbol{u}_1 = 0$:

$$\boldsymbol{u}_2 = \operatorname*{argmax}_{\|\boldsymbol{u}_2\|=1, \boldsymbol{u}_2^T\boldsymbol{u}_1=0} \boldsymbol{u}_2^T \mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}_2. \tag{11.1}$$

This is the second principal components direction for $\boldsymbol{x}$, and it once more corresponds to an eigendirection of $\mathrm{Cov}[\boldsymbol{x}]$, corresponding[4] to the second-largest eigenvalue (see Section 11.1.2). Note that eigendirections of a symmetric matrix corresponding to different eigenvalues are necessarily orthogonal (Section 11.1.2), which in hindsight motivates our convention on the columns of $\boldsymbol{U}$.

[2] To be precise, $\boldsymbol{u}_*$ and z_* are *population* principal components, while in practice we estimate them from data through the sample covariance matrix, as is detailed below.

[3] In general, eigenvalues of real-valued matrices can be complex-valued, but this does not happen for symmetric matrices (Section 11.1.2).

[4] An exception is if $\mathrm{Cov}[\boldsymbol{x}]$ has a multiple largest eigenvalue λ_*. In this case, both $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are eigenvectors corresponding to λ_*. For real data, multiple eigenvalues are rarely observed.

More generally, the k-th principal components direction $\boldsymbol{u}_k$ can be defined recursively via a variant of (11.1), where the constraints read $\boldsymbol{u}_k^T \boldsymbol{u}_j = 0$ for all $j < k$. The main point is this. *All principal components (PC) directions are eigendirections of the covariance matrix* $\mathrm{Cov}[\boldsymbol{x}]$. Their ordering follows the ordering of the eigenvalues: first PC for largest eigenvalue, and so on. Selecting the leading M eigendirections, corresponding to the M largest eigenvalues (counting multiplicities), to form a dimensionality reduction matrix $\boldsymbol{U} \in \mathbb{R}^{d \times M}$ is called *principal components analysis* (PCA).

Examples of Principal Components Analysis

PCA is ubiquitous. It is hard to find any work in applied machine learning, statistics or data analysis without stumbling across the technique. Given any sort of multivariate data, the first thing to do is to plot the first few principal components. PCA is the most basic visualization technique for high-dimensional data. In applications, whenever people are interested in X, an eigen-X method has usually been proposed long ago. Examples include eigenfaces, eigendigits (below), or latent semantic analysis for text documents. You will have guessed it by now: if we do not know $\mathrm{Cov}[\boldsymbol{x}]$ exactly, but have access to some data $\mathcal{D} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$, we estimate PCA by the eigendecomposition of the sample covariance matrix

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^T, \quad \sum_{i=1}^{n} \boldsymbol{x}_i = \boldsymbol{0}. \tag{11.2}$$

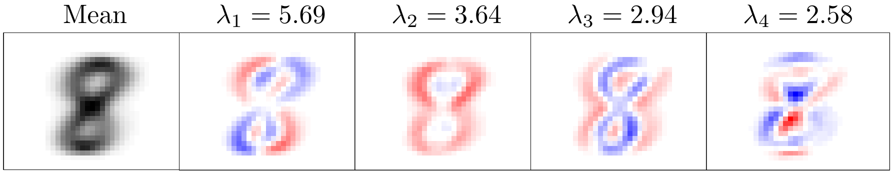
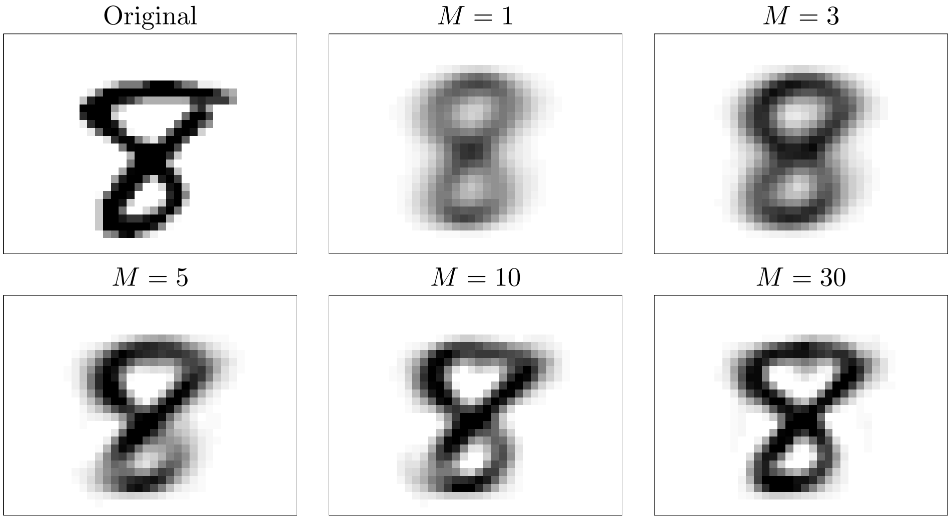


Figure 11.2: Mean and first four principal components directions for MNIST 8s (subset of size 200, drawn at random from the training database). For the latter, positive values are shown in red, negative values in blue. The first direction corresponds to rotations, the second to changes in scale.

In Figure 11.2, we show means and first few principal components directions for a set of 8s from the MNIST database. Using this example, we can test the autoencoder property of PCA. In Figure 11.3, we show reconstructions $\hat{\boldsymbol{x}}$ for digits $\boldsymbol{x}$, using a growing number of PC directions. Notice how scale and orientation are shaped before finer details.

We can arrive at a non-recursive definition of PCA for general $M < d$ and gain additional intuition by developing the minimum reconstruction error and variance decomposition perspective in the general case. The optimal choice for $\min_{\boldsymbol{r} \in \mathbb{R}^M} \|\boldsymbol{x} - \boldsymbol{U}\boldsymbol{r}\|^2$ is $\boldsymbol{r} = \boldsymbol{U}^T \boldsymbol{x}$ (why?), and

$$\mathrm{E}\left[\|\boldsymbol{x} - \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{x}\|^2\right] = \mathrm{E}\left[\|\boldsymbol{x}\|^2\right] - \mathrm{E}\left[\|\boldsymbol{U}^T\boldsymbol{x}\|^2\right],$$



Original $M = 1$ $M = 3$

$M = 5$ $M = 10$ $M = 30$

Figure 11.3: PCA reconstructions of MNIST digit (upper left) from the mean and a growing number M of PCA directions.

where we used $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$. Therefore,

$$\mathrm{E}\left[\|\boldsymbol{x}\|^2\right] = \mathrm{tr}\,\mathrm{Cov}[\boldsymbol{x}] = \underbrace{\mathrm{tr}\,\boldsymbol{U}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U}}_{\text{variance explained}} + \underbrace{\mathrm{E}\left[\|\boldsymbol{x} - \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{x}\|^2\right]}_{\text{squared error}},$$

where $\mathrm{tr}\,\boldsymbol{U}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U} = \mathrm{tr}\,\mathrm{Cov}[\boldsymbol{U}^T\boldsymbol{x}]$. The complete PCA transformation can therefore by described as solution of

$$\boldsymbol{U}_* = \operatorname*{argmax}_{\boldsymbol{U}^T\boldsymbol{U}=\boldsymbol{I}} \mathrm{tr}\,\boldsymbol{U}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U}. \tag{11.3}$$

It is noted in Section 11.1.2 that the columns of $\boldsymbol{U}_*$ are given by the M leading eigendirections[5], and the maximal value of the trace is the sum of the M largest eigenvalues (counting multiplicities).

Whitening and Estimation

What about the third requirement of removing linear redundancies in $\boldsymbol{x}$ by producing decorrelated encoding coefficients $\boldsymbol{z}$? It turns out that using PCA, we get this property for free. Namely, suppose that

$$\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_M], \quad \mathrm{Cov}[\boldsymbol{x}]\boldsymbol{u}_j = \lambda_j\boldsymbol{u}_j, \quad \lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_M.$$

A compact way of writing this is $\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U} = \boldsymbol{U}\boldsymbol{\Lambda}$, where $\boldsymbol{\Lambda} = \mathrm{diag}[\lambda_j] \in \mathbb{R}^{M\times M}$ is a diagonal matrix of eigenvalues. The covariance of $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$ is known from Section 5.1.3 to be

$$\mathrm{Cov}[\boldsymbol{U}^T\boldsymbol{x}] = \boldsymbol{U}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U} = \boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Lambda} = \boldsymbol{\Lambda}.$$

[5] More specifically, $\boldsymbol{U}_*$ is defined only up to a right-multiplication with an orthonormal $\mathbb{R}^{M\times M}$ matrix. It is however customary to order the columns as if they came out of the recursive procedure.

This means that the components z_j of $\boldsymbol{z}$ are uncorrelated: $\mathrm{Cov}[z_j, z_k] = \lambda_j \mathrm{I}_{\{k=j\}}$. This provides us with a second perspective on what PCA is doing.

In fact, PCA preprocessing is often taken a step further in what is known as *whitening*. For moderate d, this may be done with $M = d$, exploiting the decorrelation property of PCA only. Namely, we choose the feature transformation matrix $\boldsymbol{V} = \boldsymbol{U}\boldsymbol{\Lambda}^{-1/2}$, where $\boldsymbol{\Lambda}^{-1/2} = \mathrm{diag}[\sqrt{\lambda_j}]$. Then, the features $\boldsymbol{V}^T\boldsymbol{x}$ are not only uncorrelated, but have unit variances as well:

$$\mathrm{Cov}[\boldsymbol{V}^T\boldsymbol{x}] = \boldsymbol{\Lambda}^{-1/2}\boldsymbol{U}^T\mathrm{Cov}[\boldsymbol{x}]\boldsymbol{U}\boldsymbol{\Lambda}^{-1/2} = \boldsymbol{\Lambda}^{-1/2}\boldsymbol{\Lambda}\boldsymbol{\Lambda}^{-1/2} = \boldsymbol{I}.$$

Why would we whiten our data? It is often the case with multivariate variables that components are scaled differently. For example, they might be measured using different units. Also, as noted already, there might be substantial (anti)correlations between some of them. On the other hand, simple models often assume that components are (conditionally) independent, in order to save parameters and gain robustness. For example, linear classification can be seen as generative model with spherical Gaussian class-conditionals (Section 6.4.1). Whitening ensures that such simplifying assumptions are met at least up to second order (mean, covariance). Moreover, simple optimization methods like gradient descent (Chapter 2) can be slow to converge in the presence of correlated and scaled variables. Whitening is easy to do and tends to pay off in reduced nonlinear optimization time. It is particular useful in the context of multi-layer perceptron training (Section 3.4.1). Some care has to be taken with whitening in practice. Even if our goal is $M = d$ (no dimensionality reduction), we should discard directions corresponding to eigenvalues very close to zero.

A final comment relates to estimating PCA from data, replacing the population covariance $\mathrm{Cov}[\boldsymbol{x}]$ by the sample covariance matrix $\hat{\boldsymbol{\Sigma}}$ ((11.2)). For large d, it might well happen that $d^2 > n$, where n is the dataset size, and we should expect to run into over-fitting problems (Section 7.1). Will we have to regularize our estimator of $\mathrm{Cov}[\boldsymbol{x}]$? The answer is no, at least as long as $M \ll n$. Dimensionality reduction by way of PCA has a special form of regularization built in. We have already seen in Section 7.2 that the smallest eigenvalues of a covariance matrix are most troublesome. First, an eigenvalue close to zero implies a hard constraint on the variable. Second, the smallest eigenvalues are often strongly underestimated in $\hat{\boldsymbol{\Sigma}}$, up to being numerically equal to zero. In the context of least squares estimation, Tikhonov regularization has the effect of lifting small eigenvalues to get them out of the danger zone (Section 7.2.2). In contrast, PCA simply discards the smallest eigenvalue directions: regularization by clipping (or thresholding) rather than by lifting. PCA concentrates on the dominating eigendirections, which are also most reliably[6] estimated from noisy data.

11.1.2 Techniques: Eigendecomposition. Rayleigh-Ritz Characterization

A large number of multivariate statistics and machine learning techniques are based on eigendecompositions of symmetric matrices, or more general singular

[6] One can show that the largest eigenvalues of $\mathrm{Cov}[\boldsymbol{x}]$ are *overestimated* by PCA on $\boldsymbol{\Sigma}$, so the dominating directions are somewhat strengthened by PCA.

value decompositions of arbitrary matrices. Beyond the brief reminder here, make sure to read [45, ch. 6], where you will find examples, applications and geometrical intuition. How does all this relate to PCA? Here is a roadmap through this section:

- Every squared matrix $\boldsymbol{A}$ has a decomposition (11.4) in terms of eigenvectors and eigenvalues.
- For symmetric $\boldsymbol{A}$, eigenvalues are real numbers, and eigenvectors can be chosen orthonormal (11.5). For positive definite $\boldsymbol{A}$, all eigenvalues are positive.
- We can apply the eigendecomposition to the (positive definite) covariance matrix $\mathrm{Cov}[\boldsymbol{x}]$ in order to establish that the PCA directions correspond to orthonormal eigenvectors for the *largest* eigenvalues of $\mathrm{Cov}[\boldsymbol{x}]$.

Recall that a square matrix $\boldsymbol{A} \in \mathbb{R}^{d \times d}$ is a linear transform of $\mathbb{R}^d$ into $\mathbb{R}^d$. Its eigendecomposition characterizes all properties of this transform in a most useful manner. It turns out that there are certain characteristic directions $\boldsymbol{v} \in \mathbb{C}^d$, $\boldsymbol{v} \neq \boldsymbol{0}$, which are mapped back by $\boldsymbol{A}$ onto itself, except for scaling:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}, \quad \lambda \in \mathbb{C}.$$

Both $\boldsymbol{v}$ and λ can be complex-valued in general (we will show below that these quantities are real-valued for symmetric matrices). $\boldsymbol{v}$ is called an *eigenvector*, λ an *eigenvalue* of $\boldsymbol{A}$, $(\boldsymbol{v}, \lambda)$ is an eigenvector-value pair. If $(\boldsymbol{v}, \lambda)$ is such a pair, so is $(\alpha\boldsymbol{v}, \lambda)$ for any $\alpha \in \mathbb{C} \setminus \{0\}$, which is why we refer to $\boldsymbol{v}$ as *eigendirection* as well. Importantly, we can find a basis of $\mathbb{C}^d$, $\boldsymbol{u}_1, \dots, \boldsymbol{u}_d$ linear independent, so that each $\boldsymbol{u}_j$ is an eigenvector:

$$\boldsymbol{A}\boldsymbol{u}_j = \lambda_j \boldsymbol{u}_j, \quad j = 1, \dots, d.$$

Since we know all about $\boldsymbol{A}$ if we know what is does to a complete basis ($\boldsymbol{A}\boldsymbol{u}_j$ for all j), this information determines $\boldsymbol{A}$ entirely. If $\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_d]$, $\boldsymbol{\Lambda} = \mathrm{diag}[\lambda_j]$, then

$$\boldsymbol{A}\boldsymbol{U} = \boldsymbol{U}\boldsymbol{\Lambda} \quad \Rightarrow \quad \boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{-1}. \tag{11.4}$$

This representation of $\boldsymbol{A}$ is called *eigendecomposition.* Why is this useful? For one, we know immediately everything about $\boldsymbol{A}^k$ for any $k \in \mathbb{N}$:

$$\boldsymbol{A}^k = \left(\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{-1}\right)^k = \boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{U}^{-1},$$

where we used $\boldsymbol{U}^{-1}\boldsymbol{U} = \boldsymbol{I}$. Applying $\boldsymbol{A}$ k times does not change its eigenvectors, but raises its eigenvalues to the k-th power. The set of all eigenvalues is called *eigenspectrum.*

Some simple examples. The identity $\boldsymbol{I}$ has only a single eigenvalue $\lambda = 1$ which is counted d times. In general, an eigenvalue λ has *multiplicity* $k \geq 1$ if there are at most k associated linear independent eigenvectors. These span the k-dimensional *eigenspace* corresponding to λ. Note that within an eigenspace, all vectors are alike, any can serve as representative (except for $\boldsymbol{0}$). What about $\lambda = 0$? The corresponding equation is $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{0}$, which means that $\boldsymbol{v} \in \ker \boldsymbol{A}$

is in the kernel of $\boldsymbol{A}$. The kernel is the eigenspace corresponding to $\lambda = 0$. A matrix is nonsingular if and only if it does not have a zero eigenvalue. What is the eigendecomposition of $\boldsymbol{A} = \boldsymbol{v}\boldsymbol{v}^T$, $\boldsymbol{v} \neq \boldsymbol{0}$. First,

$$\boldsymbol{A}\boldsymbol{v} = \boldsymbol{v}\boldsymbol{v}^T\boldsymbol{v} = \|\boldsymbol{v}\|^2\boldsymbol{v},$$

so $\lambda = \|\boldsymbol{v}\|^2$ with multiplicity 1. Second, for any $\boldsymbol{u}$ orthogonal to $\boldsymbol{v}$: $\boldsymbol{A}\boldsymbol{u} = 0$, so $\lambda = 0$ with multiplicity $d-1$. Finally, suppose that $\boldsymbol{A}$ is the orthonormal projection onto a subspace $\mathcal{V}$ of dimension $k < d$ (Section 4.2.2). Then, $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{v}$ for any $\boldsymbol{v} \in \mathcal{V}$ and $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{0}$ for any $\boldsymbol{v} \in \mathcal{V}^\perp$, so $\boldsymbol{A}$ has eigenspace $\mathcal{V}$ for $\lambda = 1$ and kernel $\mathcal{V}^\perp$ ($\lambda = 0$). Picture it geometrically: $\boldsymbol{A}$ leaves $\boldsymbol{v} \in \mathcal{V}$ invariant, while removing all contributions orthogonal to $\mathcal{V}$.

Trace and determinant can be expressed in terms of the eigenspectrum. First,

$$\operatorname{tr}\boldsymbol{A} = \operatorname{tr}\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{-1} = \operatorname{tr}\boldsymbol{\Lambda}\boldsymbol{U}^{-1}\boldsymbol{U} = \operatorname{tr}\boldsymbol{\Lambda} = \sum_{j=1}^{d}\lambda_j.$$

The trace is the sum of eigenvalues. For any matrix, the sum of eigenvalues is the same as the sum of diagonal values. For the determinant:

$$|\boldsymbol{A}| = |\boldsymbol{U}||\boldsymbol{\Lambda}|\left|\boldsymbol{U}^{-1}\right| = |\boldsymbol{U}||\boldsymbol{\Lambda}||\boldsymbol{U}|^{-1} = |\boldsymbol{\Lambda}| = \prod_{j=1}^{d}\lambda_j,$$

the product of the eigenvalues (but *not* the product of the diagonal entries). The last mysteries about determinants should disappear now. For example, the volume interpretation of the determinant given in Section 6.3.1 becomes natural. The parallelepiped spanned by $\{\boldsymbol{u}_j\}$ is mapped to one spanned by $\{\boldsymbol{A}\boldsymbol{u}_j\} = \{\lambda_j\boldsymbol{u}_j\}$, therefore the volume changes by $|\lambda_1 \cdots \lambda_d| = ||\boldsymbol{A}||$.

Eigenvalues are the roots of the characteristic polynomial. Namely,

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v} \quad \Leftrightarrow \quad (\boldsymbol{A} - \lambda\boldsymbol{I})\,\boldsymbol{v} = \boldsymbol{0},$$

so $\boldsymbol{v} \in \ker(\boldsymbol{A} - \lambda\boldsymbol{I})$, $\boldsymbol{v} \neq \boldsymbol{0}$. This happens only if $q(\lambda) = |\boldsymbol{A} - \lambda\boldsymbol{I}| = 0$. Now, a closer look at determinants (Section 6.3.2) reveals that $q(\lambda)$ is nothing but a degree d polynomial in λ with coefficients in $\mathbb{R}$, determined by $\boldsymbol{A}$: $q(\lambda) = \alpha_0 + \alpha_1\lambda + \cdots + (-1)^d\lambda^d$. The spectrum of $\boldsymbol{A}$ is the set of roots of $q(\lambda)$. If you need to determine the eigenvalues of a 2×2 matrix, find the roots of the characteristic polynomial:

$$\left|\begin{bmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{bmatrix}\right| = (a_{11} - \lambda)(a_{22} - \lambda) - a_{12}a_{21}.$$

Symmetric Matrices. Positive Semidefinite Matrices

Things become simpler if we restrict ourselves to symmetric matrices, $\boldsymbol{A}^T = \boldsymbol{A}$. First, eigenvalues are always real-valued then, and eigenvectors can be chosen in $\mathbb{R}^d$. Recall that $(\cdot)^H$ denotes Hermitian transposition, where after transposition each entry is replaced by its complex conjugate. Suppose that $\boldsymbol{v} \in \mathbb{C}^d$, $\lambda \in \mathbb{C}$ is an eigen-pair of $\boldsymbol{A}$. Then,

$$\boldsymbol{v}^H\boldsymbol{A}\boldsymbol{v} = \lambda\|\boldsymbol{v}\|^2, \quad \boldsymbol{v}^H\boldsymbol{A}^T\boldsymbol{v} = \left(\boldsymbol{v}^H\boldsymbol{A}\boldsymbol{v}\right)^H = \bar{\lambda}\|\boldsymbol{v}\|^2.$$

Note that $\boldsymbol{A}^H = \boldsymbol{A}^T$, as $\boldsymbol{A}$ is real-valued. This means that $\lambda = \bar{\lambda}$ (since $\|\boldsymbol{v}\|^2 > 0$), so that $\lambda \in \mathbb{R}$. Moreover, the corresponding eigenspace is $\ker(\boldsymbol{A} - \lambda \boldsymbol{I})$, which has a real-valued basis. Moreover, eigenvectors can be chosen to be orthogonal. For a symmetric matrix $\boldsymbol{A}$, suppose that $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ are eigenvectors corresponding to different eigenvalues $\lambda_1 \neq \lambda_2$. Then, $\boldsymbol{u}_1^T \boldsymbol{u}_2 = 0$. Namely,

$$\boldsymbol{u}_1^T \boldsymbol{A} \boldsymbol{u}_2 = \lambda_2 \boldsymbol{u}_1^T \boldsymbol{u}_2, \quad \boldsymbol{u}_1^T \boldsymbol{A}^T \boldsymbol{u}_2 = \left(\boldsymbol{u}_1^T \boldsymbol{A}^T \boldsymbol{u}_2 \right)^T = \boldsymbol{u}_2^T \boldsymbol{A} \boldsymbol{u}_1 = \lambda_1 \boldsymbol{u}_2^T \boldsymbol{u}_1,$$

so that $(\lambda_1 - \lambda_2)\boldsymbol{u}_1^T \boldsymbol{u}_2 = 0$. This means that the eigendecomposition (11.4) is simpler in the symmetric case. $\boldsymbol{U}$ can be chosen orthonormal, $\boldsymbol{U}^T \boldsymbol{U} = \boldsymbol{I}$. Then, $\boldsymbol{U}^{-1} = \boldsymbol{U}^T$, so that

$$\boldsymbol{A} = \boldsymbol{U} \boldsymbol{\Lambda} \boldsymbol{U}^T, \quad \boldsymbol{\Lambda} = \operatorname{diag}[\lambda_j], \;\; \lambda_j \in \mathbb{R}, \quad \boldsymbol{U}^T \boldsymbol{U} = \boldsymbol{I}. \tag{11.5}$$

Finally, what about positive (semi)definite matrices? They are symmetric, so all eigenvalues are real. Since $\boldsymbol{u}^T \boldsymbol{A} \boldsymbol{u} \geq 0$ for a positive semidefinite $\boldsymbol{A}$, all its eigenvalues are nonnegative. For a positive definite $\boldsymbol{A}$, we cannot have $\lambda = 0$, so that all eigenvalues are strictly positive. The eigendecomposition of a covariance matrix is given in (11.5), where all $\lambda_j > 0$.

Variational Characterization of Eigenvalues

We are one step away now from understanding PCA in terms of the eigendecomposition of the (sample) covariance matrix. More generally, let $\boldsymbol{A}$ be a symmetric matrix. Then, $\boldsymbol{A}$ has a real-valued spectrum and we can choose an orthonormal eigenbasis $\boldsymbol{u}_1, \dots, \boldsymbol{u}_d$ with $\boldsymbol{u}_i^T \boldsymbol{u}_j = \mathrm{I}_{\{i=j\}}$. Suppose that the corresponding eigenvalues are ordered: $\lambda_1 \geq \lambda_2 \geq \dots \lambda_d$. A result due to Rayleigh and Ritz states that

$$\boldsymbol{u}_1 \in \operatorname*{argmax}_{\|\boldsymbol{u}\|=1} \boldsymbol{u}^T \boldsymbol{A} \boldsymbol{u}.$$

If $\lambda_1 > \lambda_2$, this maximizer is unique up to sign flip. To show this, we use the eigendecomposition $\boldsymbol{A} = \boldsymbol{U} \boldsymbol{\Lambda} \boldsymbol{U}^T$, where $\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_d]$ and $\boldsymbol{\Lambda} = \operatorname{diag}[\lambda_j]$. Let $\boldsymbol{u}$ be any unit vector, and let $\boldsymbol{z} = \boldsymbol{U}^T \boldsymbol{u}$. Then,

$$\boldsymbol{u}^T \boldsymbol{A} \boldsymbol{u} = \boldsymbol{u}^T \boldsymbol{U} \boldsymbol{\Lambda} \boldsymbol{U}^T \boldsymbol{u} = \boldsymbol{z}^T \boldsymbol{\Lambda} \boldsymbol{z} = \sum_{j=1}^{d} \lambda_j z_j^2.$$

The numbers z_j^2 are nonnegative and sum to one, they form a distribution over $j \in \{1, \dots, d\}$. Clearly, the expression is maximized by concentrating all probability mass on $j = 1$, since $\lambda_1 = \max\{\lambda_j\}$. This means that $\boldsymbol{u} = \boldsymbol{U}\boldsymbol{z} = \boldsymbol{u}_1$ is a maximizer of the Rayleigh-Ritz ratio $(\boldsymbol{u}^T \boldsymbol{A} \boldsymbol{u})/(\boldsymbol{u}^T \boldsymbol{u})$. The recursive definition of the k-th PC direction is obtained in the same way. Suppose that the unit vector $\boldsymbol{u}$ is orthogonal to $\boldsymbol{u}_j$, $j < k$. Then, $\boldsymbol{z} = \boldsymbol{U}^T \boldsymbol{u}$ must have $z_j = 0$ for $j < k$, and $\boldsymbol{z}^T \boldsymbol{\Lambda} \boldsymbol{z}$ is maximized by placing all probability mass on k, which implies the maximizer $\boldsymbol{u} = \boldsymbol{u}_k$.

Finally, a non-recursive min-max characterization of the eigenvalues is due to Courant and Fisher [25, ch. 4.2]. It directly implies the result (11.3) we used in Section 11.1.1.

11.1.3 Principal Components Analysis in Practice

How to compute PCA directions in practice? Just as with solving linear systems (Section 4.2.3), there are two regimes. If $\min\{n,d\}$ is of moderate size, say up to a few thousand, direct methods should be used, as their state-of-the-art numerical implementations are essentially foolproof to run. We will concentrate on this regime here. For much larger problems, iterative approximate methods have to be used. While these are out of scope of this course, some intuition is provided in Section 11.1.4.

Suppose for now that $n \geq d$ (more datapoints than dimensions). If

$$\boldsymbol{X} = n^{-1/2}\left[\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\right]^T \in \mathbb{R}^{n \times d},$$

then (recalling that the data is centered) the data covariance matrix is

$$\hat{\boldsymbol{\Sigma}} = \boldsymbol{X}^T\boldsymbol{X}.$$

We could compute $\hat{\boldsymbol{\Sigma}}$, then its eigendecomposition. However, for reasons of numerical stability, it is preferable to compute the *singular value decomposition* (SVD) of $\boldsymbol{X}$ instead [45, ch. 6.7]:

$$\boldsymbol{X} = \boldsymbol{V}\boldsymbol{\Psi}\boldsymbol{U}^T, \quad \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}, \quad \boldsymbol{V} \in \mathbb{R}^{n \times k},\ \boldsymbol{U} \in \mathbb{R}^{d \times k}.$$

Here, $k = \mathrm{rk}(\boldsymbol{X}) \leq \min\{d,n\}$, and $\boldsymbol{\Psi} \in \mathbb{R}^{k \times k}$ is diagonal with positive entries. The SVD is a generalization of the eigendecomposition to arbitrary matrices. Now,

$$\hat{\boldsymbol{\Sigma}} = \boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Psi}\underbrace{\boldsymbol{V}^T\boldsymbol{V}}_{=\boldsymbol{I}}\boldsymbol{\Psi}\boldsymbol{U}^T = \boldsymbol{U}\boldsymbol{\Psi}^2\boldsymbol{U}^T.$$

This means that $\boldsymbol{\Lambda} = \boldsymbol{\Psi}^2$ are the eigenvalues, $\boldsymbol{U}$ the eigenvectors of $\hat{\boldsymbol{\Sigma}}$.

What if $n < d$ (less datapoints than dimensions)? For example, gene microarray data can come with thousands of dimensions, yet less than hundred datapoints. In this case, we compute the SVD of $\boldsymbol{X}^T$ instead:

$$\boldsymbol{X}^T = \boldsymbol{U}\boldsymbol{\Psi}\boldsymbol{V}^T, \quad \boldsymbol{U} \in \mathbb{R}^{d \times k},\ \boldsymbol{V} \in \mathbb{R}^{n \times k}.$$

Once more, $\hat{\boldsymbol{\Sigma}} = \boldsymbol{U}\boldsymbol{\Psi}^2\boldsymbol{U}^T$, so that $\boldsymbol{\Lambda} = \boldsymbol{\Psi}^2$ are the eigenvalues.

Representing PCA in terms of the SVD of the data matrix reveals a curious fact which is important with iterative large scale PCA algorithms as well (Section 11.1.4). Since

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Psi}^2\boldsymbol{U}^T, \quad \boldsymbol{X}\boldsymbol{X}^T = \boldsymbol{V}\boldsymbol{\Psi}^2\boldsymbol{V}^T,$$

the matrices $\hat{\boldsymbol{\Sigma}} = \boldsymbol{X}^T\boldsymbol{X}$ and $\boldsymbol{X}\boldsymbol{X}^T$ have the same set of non-zero eigenvalues. Moreover, the corresponding eigenvectors, $\boldsymbol{U}$ and $\boldsymbol{V}$, are closely related. For example, suppose we have determined the eigenvectors $\boldsymbol{V}$ of $\boldsymbol{X}\boldsymbol{X}^T$. Then,

$$\boldsymbol{X}^T\boldsymbol{V} = \boldsymbol{U}\boldsymbol{\Psi}\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{U}\boldsymbol{\Psi} \quad \Rightarrow \quad \boldsymbol{U} = \boldsymbol{X}^T\boldsymbol{V}\boldsymbol{\Psi}^{-1}.$$

This means that leading eigenvectors of $\hat{\boldsymbol{\Sigma}}$ are obtained from corresponding eigenvectors of $\boldsymbol{X}\boldsymbol{X}^T$ by multiplication with $\boldsymbol{X}^T$.

11.1.4 Large Scale Principal Components Analysis (*)

If both d and n are very large, iterative approximate methods have to be used. Two facts allow us to operate in the "astronomical" regime. First, only a small number M of PC directions have to be extracted. Second, the data matrix $\boldsymbol{X}$ is very sparse or otherwise structured, so that matrix-vector multiplications (MVMs) $\boldsymbol{X}\boldsymbol{v}$, $\boldsymbol{X}^T\boldsymbol{w}$ can be computed much faster than $O(dn)$. The method of choice for computing leading eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$ or $\boldsymbol{X}\boldsymbol{X}^T$ is the *Lanczos algorithm* [18], which gives rise to `Matlab eigs`. This algorithm requires one MVM with $\boldsymbol{X}$ and $\boldsymbol{X}^T$ per iteration. Typically, the first few dominating eigendirections are obtained rapidly (the convergence speed depends on the decay rate of the spectrum). In practice, we apply Lanczos to the smaller of $\boldsymbol{X}\boldsymbol{X}^T$ or $\boldsymbol{X}^T\boldsymbol{X}$, then use the equivalence noted in Section 11.1.3. Typically, our data will not be centered, and centering it up front may turn a sparse into a dense matrix. Fortunately, there is a simple remedy, detailed at the end of this section.

First, let us gain some understanding why methods such as Lanczos work. We focus on the *power method*, which is simpler than Lanczos[7]. Given a positive semidefinite $\boldsymbol{A}$ ($\boldsymbol{X}^T\boldsymbol{X}$ or $\boldsymbol{X}\boldsymbol{X}^T$ in the case of PCA), we would like to approximate the leading eigendirection $\boldsymbol{u}_1$. Suppose that $\lambda_1 > \lambda_2$, the largest eigenvalue has multiplicity one. Pick a unit vector $\boldsymbol{v}_0$ uniformly[8] at random. Iterate $\tilde{\boldsymbol{v}}_k = \boldsymbol{A}\boldsymbol{v}_{k-1}$, $\boldsymbol{v}_k = \tilde{\boldsymbol{v}}_k/\|\tilde{\boldsymbol{v}}_k\|$. Then, $\boldsymbol{v}_k$ converges to one of $\pm\boldsymbol{u}_1$. The intuition behind this procedure is as follows. If we multiply $\boldsymbol{v}_0$ with $\boldsymbol{A}$ repeatedly, its contribution along $\boldsymbol{u}_1$ will grow more rapidly than the others, so it will eventually dominate the renormalized vectors. For the proof, we expand $\tilde{\boldsymbol{v}}_k$ into the orthonormal basis given by the eigenvectors $\boldsymbol{u}_j$ of $\boldsymbol{A}$. In fact, we will use a slight modification, defining $\tilde{\boldsymbol{v}}_k = \boldsymbol{A}^k\boldsymbol{v}_0$, which gives rise to the same $\boldsymbol{v}_k$ sequence. The rationale is that we know what $\boldsymbol{A}^k$ is doing on the eigendirections: $\boldsymbol{A}^k\boldsymbol{u}_j = \lambda_j^k\boldsymbol{u}_j$. Therefore, if $\boldsymbol{v}_0 = \sum_j \alpha_j\boldsymbol{u}_j$, then

$$\tilde{\boldsymbol{v}}_k = \sum_{j=1}^{n} \alpha_j\lambda_j^k\boldsymbol{u}_j, \quad \|\tilde{\boldsymbol{v}}_k\|^2 = \sum_{j=1}^{n} \alpha_j^2\lambda_j^{2k} = \lambda_1^{2k}\left(\alpha_1^2 + \sum\nolimits_{j>1}\alpha_j^2(\lambda_j/\lambda_1)^{2k}\right).$$

We may assume that $\alpha_1 \neq 0$. Since $\lambda_j/\lambda_1 \in [0,1)$ for $j > 1$, we know that

$$\frac{\|\tilde{\boldsymbol{v}}_k\|}{\lambda_1^k} = \sqrt{\alpha_1^2 + \sum\nolimits_{j>1}\alpha_j^2(\lambda_j/\lambda_1)^{2k}} \to |\alpha_1| \quad (k \to \infty).$$

Plugging this in,

$$\boldsymbol{v}_k = \frac{\tilde{\boldsymbol{v}}_k}{\|\tilde{\boldsymbol{v}}_k\|} = \sum_{j=1}^{n} \alpha_j(\lambda_j/\lambda_1)^k(\lambda_1^k/\|\tilde{\boldsymbol{v}}_k\|)\boldsymbol{u}_j \to \operatorname{sgn}(\alpha_1)\boldsymbol{u}_1 \quad (k \to \infty).$$

The Lanczos algorithm is slightly more advanced than the power method, but both require one MVM with $\boldsymbol{A}$ per iteration. The Lanczos algorithm is far superior to the power method if it comes to approximating $M > 1$ PCA directions. Details are found in [18].

[7] The power method is widely used in machine learning. Unfortunately, it typically needs many more MVMs than Lanczos, in particular if M is moderately large. We should use Lanczos whenever possible.

[8] This can be done by sampling i.i.d. Gaussians, then normalizing the resulting vector.

Centering

Recall that we assumed so far that our data is centered up front by subtracting off the empirical mean $\hat{\boldsymbol{\mu}} = n^{-1}\sum_i \boldsymbol{x}_i$. However, as noted above, doing so is not always advisable. It turns out that centering can be folded into an iterative method such as Lanczos at no extra cost. Suppose that $\boldsymbol{X} = n^{-1/2}[\boldsymbol{x}_1, \dots, \boldsymbol{x}_n]^T$ is our original data matrix. We can write the empirical mean as

$$\hat{\boldsymbol{\mu}} = n^{-1}\sum_{i=1}^{n} \boldsymbol{x}_i = n^{-1/2}\boldsymbol{X}^T\boldsymbol{1}_n, \quad \boldsymbol{1}_n = [1] \in \mathbb{R}^n.$$

We would like to replace $\boldsymbol{X}$ by the corresponding centered data matrix, which is

$$n^{-1/2}\left[\boldsymbol{x}_1 - \hat{\boldsymbol{\mu}}, \dots, \boldsymbol{x}_n - \hat{\boldsymbol{\mu}}\right]^T = \boldsymbol{X} - n^{-1/2}\boldsymbol{1}_n\hat{\boldsymbol{\mu}}^T.$$

Note that $\boldsymbol{1}_n\hat{\boldsymbol{\mu}}^T = [\hat{\boldsymbol{\mu}}, \dots, \hat{\boldsymbol{\mu}}]^T$ (n rows). Plugging in the expression for $\hat{\boldsymbol{\mu}}$:

$$\boldsymbol{X} - n^{-1}\boldsymbol{1}_n\boldsymbol{1}_n^T\boldsymbol{X} = \boldsymbol{H}\boldsymbol{X}, \quad \boldsymbol{H} = \boldsymbol{I} - n^{-1}\boldsymbol{1}_n\boldsymbol{1}_n^T.$$

The centered data matrix is the product of the uncentered $\boldsymbol{X}$ with the rank one centering matrix $\boldsymbol{H}$. Note that $\boldsymbol{H}$ is the orthogonal projection onto the hyperplane with normal vector $\boldsymbol{1}_n$ (Section 4.2.2), which is the subspace of vectors whose components sum to zero. Implicit centering works by using $\boldsymbol{H}\boldsymbol{X}$ in place of $\boldsymbol{X}$ and $\boldsymbol{X}^T\boldsymbol{H}$ in place of $\boldsymbol{X}^T$. Here, MVMs with $\boldsymbol{H}$ carry essentially no additional cost.

11.2 Linear Discriminant Analysis

Recall our motivation for principal components analysis (PCA) from the beginning of this chapter. For our binary classification problem MNIST 8s versus 9s, we would like to learn a projection $\boldsymbol{U}$ so that $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$, a much lower dimensional vector than $\boldsymbol{x}$ itself, represents the most salient information about $\boldsymbol{x}$. This is a special case of a linear feature map, $\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{U}^T\boldsymbol{x}$: we extract M linear features $z_j = \boldsymbol{u}_j^T\boldsymbol{x}$ from $\boldsymbol{x}$. In other words, the columns of $\boldsymbol{U}$ are directions onto which we project $\boldsymbol{x}$ so to obtain a feature z_j. As we saw above, PCA directions represent $\boldsymbol{x}$ well in terms of squared reconstruction error. Equivalently, they maximize the amount of covariance of $\boldsymbol{x}$ which is retained in $\boldsymbol{z}$. However, our problem is binary classification. Why should the directions of maximum covariance of $\boldsymbol{x}$ also be helpful in classifying $(\boldsymbol{x}, t)$ well?

In general, they will not always be helpful, as the simple example in Figure 11.4 demonstrates. This should not come as a surprise. After all, given some training data $\{(\boldsymbol{x}_i, t_i)\}$, PCA does not depend on the labels $\{t_i\}$. For many real-world problems, the directions of maximum variance in the data do not carry much discriminative information. If $\boldsymbol{x}$ is drawn from a distribution over image bitmaps, the largest variance direction often represents differences in global illumination (brightness) across images. If $\boldsymbol{x}$ represents an audio waveform, the first PCA direction may tell us mainly about volume. For natural image patches $\boldsymbol{x}$, the maximum covariance directions are typically low order sinusoids. None of these directions help much with classification.

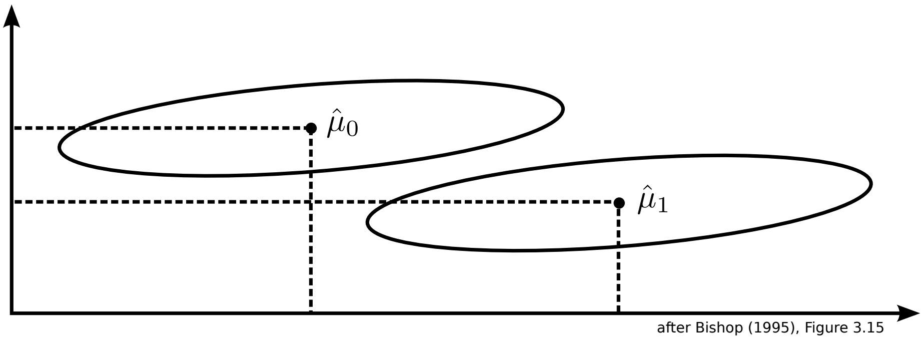


Figure 11.4: PCA directions of largest variance need not be directions which are most useful for classification. In this example, the maximum variance direction is closely aligned with the horizontal axis, yet projecting onto it results in substantial overlap between the classes. We would be better off to project onto the vertical axis.

For binary classification, a single discriminative direction $\boldsymbol{u}$ can be found as *Fisher's linear discriminant* (FLD). To maximize class separation, we need to *maximize* the variance *between* classes, while at the same time *minimizing* the variance *within* classes. The example in Figure 11.4 shows that these requirements can be in conflict with each other. If we project onto the horizontal axis, we attain a larger distance between the class means than if we choose the vertical axis: the between-class variance is larger for the former choice. On the other hand, both classes spread much more along the horizontal axis. In order to minimize the within-class variance, we are better off projecting onto the vertical axis. Fisher's linear discriminant realizes a tradeoff between these two requirements. Define sample means and covariance matrices:

$$\hat{\boldsymbol{\mu}}_k = n_k^{-1} \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} \boldsymbol{x}_i, \quad k = 0, 1,$$

$$\hat{\boldsymbol{\Sigma}}_k = n_k^{-1} \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_k)(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_k)^T = n_k^{-1} \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} \boldsymbol{x}_i \boldsymbol{x}_i^T - \hat{\boldsymbol{\mu}}_k \hat{\boldsymbol{\mu}}_k{}^T.$$

Here, the label space is $\mathcal{T} = \{0, 1\}$ (for our MNIST example, assign $8 \to 0$, $9 \to 1$), and n_k is the number of patterns $\boldsymbol{x}_i$ with $t_i = k$. The total number of patterns is $n = n_0 + n_1$. We can quantify the between-class variance as squared distance between the class means projected onto $\boldsymbol{u}$: $(m_0 - m_1)^2$, where $m_k = \boldsymbol{u}^T \hat{\boldsymbol{\mu}}_k$. On the other hand, the within-class scatter for class k can be measured, in terms of the same units, by summing up the squared distance between $z_i = \boldsymbol{u}^T \boldsymbol{x}_i$ and m_k over all patterns $\boldsymbol{x}_i$ belonging to class k:

$$s_k^2 = n^{-1} \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} (z_i - m_k)^2 = n^{-1} \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} \left(\boldsymbol{u}^T (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i}) \right)^2 .$$

The total amount of within-class variance is $s_0^2 + s_1^2$. One way to maximize between-class scatter while minimizing between-class scatter, the one chosen by

Fisher, is to maximize the ratio

$$J(\boldsymbol{u}) = \frac{(m_1 - m_0)^2}{s_0^2 + s_1^2} = \frac{\left(\boldsymbol{u}^T(\hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0)\right)^2}{n^{-1}\sum_{i=1}^n \left(\boldsymbol{u}^T(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})\right)^2}.$$

over unit norm vectors $\boldsymbol{u}$. Stop for a second to note the subtle form of the denominator $s_0^2 + s_1^2$. It looks like the usual covariance, but note that in $\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i}$ the mean we subtract depends on the label t_i of $\boldsymbol{x}_i$. How do we solve this problem? First, both numerator and denominator are quadratic functions in $\boldsymbol{u}$:

$$J(\boldsymbol{u}) = \frac{\boldsymbol{u}^T \boldsymbol{S}_B \boldsymbol{u}}{\boldsymbol{u}^T \boldsymbol{S}_W \boldsymbol{u}},$$

where

$$\boldsymbol{S}_B = (\hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0)(\hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0)^T$$

is the between-class scatter matrix,

$$\boldsymbol{S}_W = n^{-1}\sum_{i=1}^n (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})^T = (1-\alpha)\hat{\boldsymbol{\Sigma}}_0 + \alpha\hat{\boldsymbol{\Sigma}}_1$$

is the within-class scatter matrix, and $\alpha = n_1/n$. We will assume here and below that the within-class scatter matrix $\boldsymbol{S}_W$ is invertible[9]. Define $\boldsymbol{d} := \hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0$, so that $\boldsymbol{S}_B = \boldsymbol{d}\boldsymbol{d}^T$. The form of $J(\boldsymbol{u})$ looks like a Rayleigh-Ritz ratio we encountered in Section 11.1.2, only that the denominator squared norm $\boldsymbol{u}^T \boldsymbol{S}_W \boldsymbol{u}$ is weighted by $\boldsymbol{S}_W$. We will develop this generalized eigenproblem notion in full generality below, when we generalize FLD to multiple classes. For now, let us just set the gradient to zero. Working out $\nabla_{\boldsymbol{u}} J(\boldsymbol{u})$ is a bit simpler if we apply differentiation to the identity

$$J(\boldsymbol{u})\boldsymbol{u}^T \boldsymbol{S}_W \boldsymbol{u} = \boldsymbol{u}^T \boldsymbol{S}_B \boldsymbol{u},$$

resulting in

$$(dJ(\boldsymbol{u}))\boldsymbol{u}^T \boldsymbol{S}_W \boldsymbol{u} = 2\left(\boldsymbol{S}_B \boldsymbol{u} - J(\boldsymbol{u})\boldsymbol{S}_W \boldsymbol{u}\right)^T (d\boldsymbol{u}).$$

Setting the gradient equal to zero gives

$$J(\boldsymbol{u})\boldsymbol{S}_W \boldsymbol{u} = \boldsymbol{S}_B \boldsymbol{u}. \tag{11.6}$$

Here, $\boldsymbol{S}_B \boldsymbol{u} = (\boldsymbol{d}^T \boldsymbol{u})\boldsymbol{d}$, so the right hand side of (11.6) is a multiple of $\boldsymbol{d}$. Multiplying both sides with $\boldsymbol{S}_W^{-1}$, we obtain $\boldsymbol{u} \propto \boldsymbol{S}_W^{-1}\boldsymbol{d}$. The *Fisher's linear discriminant* direction is given by

$$\hat{\boldsymbol{u}}_{\mathrm{FLD}} = \frac{\boldsymbol{S}_W^{-1}\boldsymbol{d}}{\|\boldsymbol{S}_W^{-1}\boldsymbol{d}\|}, \quad \boldsymbol{d} = \hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0.$$

In order to compute $\hat{\boldsymbol{u}}_{\mathrm{FLD}}$, we determine the within-class scatter matrix $\boldsymbol{S}_W$ and the class means, then solve the linear system

$$\boldsymbol{S}_W \boldsymbol{u} = \hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0,$$

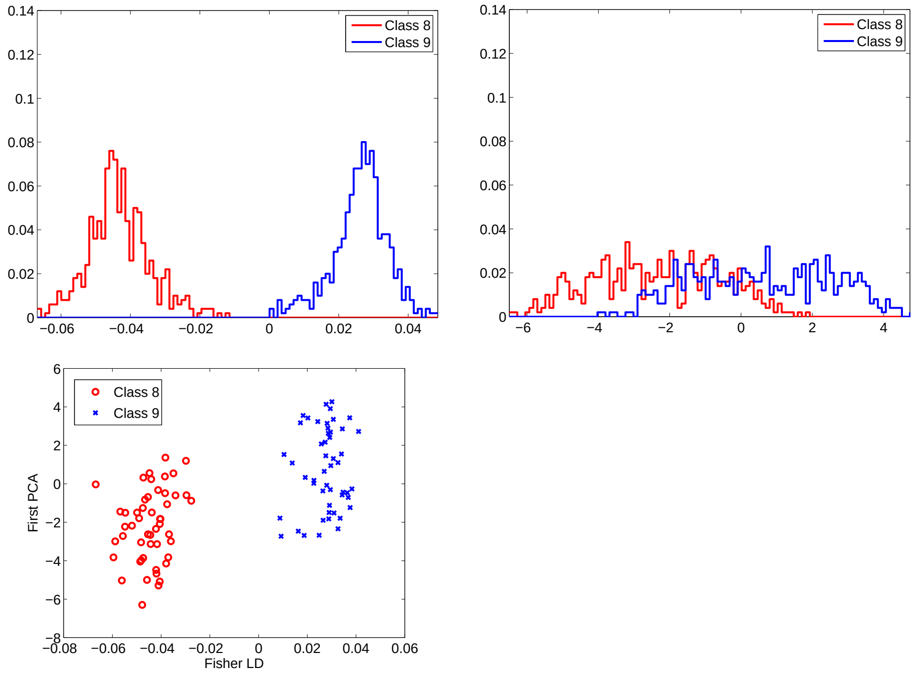


Figure 11.5: Comparison of single linear features on MNIST 8 vs. 9 binary classification problem. The dataset are 500 patterns from each of the two classes, randomly drawn from the MNIST training database. Top left: Fisher's linear discriminant direction. Top right: First principal component direction. Shown are histrograms of feature values over data from each class. Bottom left: Features values FLD vs. PCA. While the FLD histograms are nicely separated, they overlap substantially for the PCA feature (little discriminative power). Bottom left: Data perfectly separable when projected on FLD direction (horizontal axis).

a procedure which is simpler even than PCA.

In Figure 11.5, we compare the FLD direction $\hat{\boldsymbol{u}}_{\text{FLD}}$ against the first principal components direction $\hat{\boldsymbol{u}}_{\text{PCA}}$ (being the leading eigenvector of the total sample covariance matrix $\boldsymbol{S}$, ignoring the class labels). The FLD direction provides an excellent separation of the data, while the PCA direction is not useful in that respect. A few comments are in order. First, does this mean that PCA is not useful as a preprocessing technique for classification? By no means. PCA is widely (and rightly) used in this respect. We do not extract a single PCA direction, but a number of them, feeding these features into a classifier further down the line which makes use of the labels t_i. The weakness of PCA in the context of Figure 11.5, its independence from the labels $\{t_i\}$, can be a strength when it comes to preprocessing. There may be much more unlabeled than labeled data, the former cannot be used by discriminative techniques like FLD. As noted towards the end of Section 10.2.1, feature induction by PCA does not carry the risk of overfitting, while care has to be taken with FLD.

[9] This means that $\boldsymbol{S}_W$ is positive definite, so that the denominator $\boldsymbol{u}^T \boldsymbol{S}_W \boldsymbol{u}$ in $J(\boldsymbol{u})$ is always positive.

If FLD extracts a single discriminative direction, what is the difference to linear classification in general, say by the perceptron algorithm, linear or logistic regression? Strictly speaking, FLD is not a classification technique. Given $\hat{\boldsymbol{u}}_{\text{FLD}}$, we still need to construct a discriminant function. A natural choice is the linear function

$$\hat{y}_{\text{FLD}}(\boldsymbol{x}) = \hat{\boldsymbol{u}}_{\text{FLD}}{}^T \boldsymbol{x} + b_{\text{FLD}},$$

where the bias parameter b_{FLD} is chosen by minimizing the training error. Viewed in this way, FLD is simply an alternative to perceptron learning or logistic regression, which may be simpler to implement. On the other hand, we will see below how to generalize FLD to linear discriminant analysis over $C > 2$ classes, which allows us to extract a discriminative subspace of dimension $C - 1$. These features can be used with any nonlinear classifier.

11.2.1 Decomposition of Total Covariance

We derived FLD based on the intuition of maximizing the between-class scatter matrix, while minimizing the within-class scatter matrix. How do these notions relate to the total sample covariance matrix of the data,

$$\boldsymbol{S} = n^{-1} \sum_{i=1}^{n} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})^T, \quad \hat{\boldsymbol{\mu}} = n^{-1} \sum_{i=1}^{n} \boldsymbol{x}_i,$$

which PCA is based on? In this section, we show that $\boldsymbol{S}$ can be decomposed as the sum of $\boldsymbol{S}_W$ and a multiple of $\boldsymbol{S}_B$. Recall the definitions of $\boldsymbol{S}_W$ and $\boldsymbol{S}_B$ from above, and set $\boldsymbol{d} = \hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0$. We plug

$$\boldsymbol{x}_i - \hat{\boldsymbol{\mu}} = \left(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i}\right) + \left(\hat{\boldsymbol{\mu}}_{t_i} - \hat{\boldsymbol{\mu}}\right)$$

into the equation for $\boldsymbol{S}$. Expanding the squares, we see that the "cross-talk" vanishes:

$$\sum_{i=1}^{n} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})(\hat{\boldsymbol{\mu}}_{t_i} - \hat{\boldsymbol{\mu}})^T = \sum_{k=0,1} \sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_k)(\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}})^T = \boldsymbol{0},$$

since

$$\sum_{i=1}^{n} \mathrm{I}_{\{t_i=k\}} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_k) = n_k (\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}}_k) = \boldsymbol{0}.$$

Therefore,

$$\boldsymbol{S} = n^{-1} \sum_{i=1}^{n} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})^T + n^{-1} \sum_{k=0,1} n_k (\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}})(\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}})^T.$$

The first part of this equation is just $\boldsymbol{S}_W$. Now, $n\hat{\boldsymbol{\mu}} = n_0 \hat{\boldsymbol{\mu}}_0 + n_1 \hat{\boldsymbol{\mu}}_1$, so that

$$n\left(\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}}\right) = (n_0 + n_1)\hat{\boldsymbol{\mu}}_k - n_0 \hat{\boldsymbol{\mu}}_0 - n_1 \hat{\boldsymbol{\mu}}_1 = (2k-1) n_{1-k} \boldsymbol{d},$$

therefore

$$n^{-1} \sum_{k=0,1} n_k (\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}})(\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}})^T = n^{-1} \sum_{k=0,1} \frac{n_k (n_{1-k})^2}{n^2} \boldsymbol{d}\boldsymbol{d}^T = \alpha(1-\alpha) \boldsymbol{d}\boldsymbol{d}^T,$$

where $\alpha = n_1/n$. All in all,

$$\boldsymbol{S} = \boldsymbol{S}_W + \alpha(1-\alpha)\boldsymbol{d}\boldsymbol{d}^T, \quad \boldsymbol{d} = \hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0.$$

This simple decomposition provides insight into the relationship between PCA and FLD. Maximizing total variance is a good idea for data without class labels, but once we know the assignment of training pattern to classes, we recognize that a part of this variance helps with classification (variance between patterns of different classes), while the remaining part hurts (variance of patterns within each class) by creating overlap. When maximizing the former part, we have to control the size of the latter in order to obtain a discriminative direction. Note that the two parts of $\boldsymbol{S}$ are different in nature. The within-class scatter matrix $\boldsymbol{S}_W$ is a positive definite matrix just like $\boldsymbol{S}$, while the between-class scatter matrix $\boldsymbol{S}_B$ is of rank one only. We will understand the significance of this observation in Section 11.2.3.

11.2.2 Relationship to Optimal Classification (*)

The attentive reader may spot a similarity between the FLD direction $\hat{\boldsymbol{u}}_{\text{FLD}} \propto \boldsymbol{S}_W^{-1}(\hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_0)$ and the optimal discriminant for two Gaussian classes with equal covariance matrices, which we derived in Section 6.4.1. In order to understand this relationship, we will analyze the following question in this section. If data comes from two Gaussian classes $P(\boldsymbol{x}|t) = N(\boldsymbol{\mu}_t, \boldsymbol{\Sigma})$, $t = 0, 1$, moreover $P(t = 1) = \alpha \in (0, 1)$, and we determine the population FLD direction, how does it relate to the Bayes optimal weight vector $\boldsymbol{w}_* = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$? Do they give rise to the same discriminant functions, or is FLD suboptimal even if we compute it based on true means and covariances?

The population FLD direction is proportional to $\boldsymbol{w}_{\text{FLD}} = \boldsymbol{\Sigma}_W^{-1}\boldsymbol{d}$, where $\boldsymbol{d} = \boldsymbol{\mu}_1 - \boldsymbol{\mu}_0$ (distance between true means) and

$$\boldsymbol{\Sigma}_W = \mathrm{E}\left[(\boldsymbol{x} - \boldsymbol{\mu}_t)(\boldsymbol{x} - \boldsymbol{\mu}_t)^T\right]$$

is the true within-class covariance matrix. Note the subtlety in this definition: this is *not* the covariance of $\boldsymbol{x}$, which would be

$$\boldsymbol{\Sigma} = \mathrm{E}\left[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T\right],$$

since the expectation is over $(\boldsymbol{x}, t)$, not just $\boldsymbol{x}$. The Bayes optimal rule was determined in Section 6.4.1, it comes with the weight vector $\boldsymbol{w}_* = \boldsymbol{\Sigma}^{-1}\boldsymbol{d}$. Since $\boldsymbol{\Sigma} \neq \boldsymbol{\Sigma}_W$, we have that $\boldsymbol{w}_* \neq \boldsymbol{w}_{\text{FLD}}$. However, we will see that $\boldsymbol{w}_*$ and $\boldsymbol{w}_{\text{FLD}}$ are in fact proportional to each other. Since the length of the weight vector does not matter, we end up with the same optimal discriminant.

From Section 11.2.1, we know that $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}_W + \alpha(1-\alpha)\boldsymbol{d}\boldsymbol{d}^T$, where $\alpha(1-\alpha) > 0$. Therefore,

$$\boldsymbol{\Sigma}\boldsymbol{w}_* = \boldsymbol{d} = \left(\boldsymbol{\Sigma} - \alpha(1-\alpha)\boldsymbol{d}\boldsymbol{d}^T\right)\boldsymbol{w}_{\text{FLD}},$$

so that

$$\begin{aligned}\boldsymbol{\Sigma}\boldsymbol{w}_{\text{FLD}} &= \boldsymbol{d} + \alpha(1-\alpha)(\boldsymbol{w}_{\text{FLD}}^T\boldsymbol{d})\boldsymbol{d} = \left(1 + \alpha(1-\alpha)\boldsymbol{d}^T\boldsymbol{\Sigma}_W^{-1}\boldsymbol{d}\right)\boldsymbol{d} \\ &= \left(1 + \alpha(1-\alpha)\boldsymbol{d}^T\boldsymbol{\Sigma}_W^{-1}\boldsymbol{d}\right)\boldsymbol{\Sigma}\boldsymbol{w}_*.\end{aligned}$$

Here, we used that $\boldsymbol{w}_{\mathrm{FLD}}^T \boldsymbol{d} = \boldsymbol{d}^T \boldsymbol{\Sigma}_W^{-1} \boldsymbol{d} > 0$. This means that $\boldsymbol{w}_{\mathrm{FLD}}$ is proportional to $\boldsymbol{w}_*$, and FLD recovers the Bayes optimal discriminant in this case.

11.2.3 Multiple Classes

In this section, we generalize Fisher's linear discriminant to multi-way classification with $C \geq 2$ classes. This general procedure is known as *linear discriminant analysis* (LDA). On the way, we will learn about generalized eigenproblems and simultaneous diagonalization, techniques which many other machine learning methods are based on. We will also understand why the basic idea between LDA for C classes is limited to extracting no more than $C-1$ independent linear features.

A natural way to derive LDA is to generalize the decomposition of the total data covariance matrix $\boldsymbol{S}$ from Section 11.2.1. Assume that $\mathcal{T} = \{0, \dots, C-1\}$. In the remainder of this section, we write $\sum_k$ short for $\sum_{k \in \mathcal{T}}$. We assume that $C < \min\{d, n\}$. Our aim is to extract M linearly independent directions: $\boldsymbol{U} \in \mathbb{R}^{d \times M}$, where $\mathrm{rk}(\boldsymbol{U}) = M$. We will specify $M \geq 1$ below. The within-class scatter matrix is generalized easily:

$$\boldsymbol{S}_W = n^{-1} \sum_k n_k \hat{\boldsymbol{\Sigma}}_k = n^{-1} \sum_{i=1}^{n} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{t_i})^T.$$

At this point, you should go through the derivation in Section 11.2.1 and confirm for yourself that

$$\boldsymbol{S} = \boldsymbol{S}_W + \boldsymbol{S}_B, \quad \boldsymbol{S}_B = n^{-1} \sum_k n_k \boldsymbol{d}_k \boldsymbol{d}_k^T, \quad \boldsymbol{d}_k = \hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}}.$$

Here, $\hat{\boldsymbol{\mu}}_k$ is the sample mean over the patterns from class k, and $\hat{\boldsymbol{\mu}}$ is the overall sample mean. What is the rank of $\boldsymbol{S}_B$? Obviously, $\mathrm{rk}(\boldsymbol{S}_B) \leq C$. We even have $\mathrm{rk}(\boldsymbol{S}_B) \leq C-1$. Define

$$\boldsymbol{M} = \left[\sqrt{n_k/n}\, \boldsymbol{d}_k \right] \in \mathbb{R}^{d \times C}.$$

Then, $\boldsymbol{S}_B = \boldsymbol{M}\boldsymbol{M}^T$. Now, $\mathrm{rk}(\boldsymbol{M}) \leq C-1$, since

$$\boldsymbol{M} \left[\sqrt{n_k/n} \right] = n^{-1} \sum_k n_k \left(\hat{\boldsymbol{\mu}}_k - \hat{\boldsymbol{\mu}} \right) = \hat{\boldsymbol{\mu}} - \hat{\boldsymbol{\mu}} = \boldsymbol{0},$$

so the C columns of $\boldsymbol{M}$ are not linearly independent. We may assume that the class means $\hat{\boldsymbol{\mu}}_k$ are linearly independent, so that $\mathrm{rk}(\boldsymbol{M}) = C-1$ and $\mathrm{rk}(\boldsymbol{S}_B) = C-1$. Note that for the binary case $C = 2$, we recover the FLD situation of a rank one between-class scatter matrix.

We need to choose $\boldsymbol{U}$ so that the covariance $\boldsymbol{U}^T \boldsymbol{S}_B \boldsymbol{U}$ is maximum, while the covariance $\boldsymbol{U}^T \boldsymbol{S}_W \boldsymbol{U}$ is minimum. There are several ways to measure the size of covariance, for example its trace (sum of eigenvalues) or its determinant (product of eigenvalues), see Section 11.1.2. For the derivation of LDA, both give the same result, so let us go with the trace. The situation for LDA is a bit more complicated than for PCA, as we need to simultaneously deal with two covariance matrices $\boldsymbol{S}_B$ and $\boldsymbol{S}_W$ here, not just with one. At this point, it is important to understand that what we are after is not the matrix $\boldsymbol{U}$, but rather the M-dimensional subspace spanned by its columns. We can replace any $\boldsymbol{U}$ by $\boldsymbol{U}\boldsymbol{R}$ for an invertible $\boldsymbol{R} \in \mathbb{R}^{M\times M}$. Also, the scale of $\boldsymbol{U}$ is arbitrary, so we can fix it without loss of generality. In order to maximize the between-class scatter $\operatorname{tr}\boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U}$ while controlling the within-class scatter, we can just fix the size of the latter by imposing the constraint $\boldsymbol{U}^T\boldsymbol{S}_W\boldsymbol{U} = \boldsymbol{I}$. The *linear discriminant analysis* (LDA) problem is:

$$\max_{\boldsymbol{U}\in\mathbb{R}^{d\times M}} \operatorname{tr}\boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U} \quad \text{s.t.} \quad \boldsymbol{U}^T\boldsymbol{S}_W\boldsymbol{U} = \boldsymbol{I}. \tag{11.7}$$

This is a *generalized eigenproblem.* If we replaced the invertible matrix $\boldsymbol{S}_W$ by $\boldsymbol{I}$, we would obtain the standard Rayleigh-Ritz characterization of the eigendecomposition of $\boldsymbol{S}_B$. We will see how to solve the LDA problem in Section 11.2.4.

Let us look into properties of the LDA problem and its solution. First, what do we mean by talking about "the solution"? For some solution $\boldsymbol{U}$ and some orthonormal $\boldsymbol{Q} \in \mathbb{R}^{M\times M}$, the matrix $\boldsymbol{U}\boldsymbol{Q}$ is a solution as well. Namely,

$$\begin{aligned}&(\boldsymbol{U}\boldsymbol{Q})^T\boldsymbol{S}_W\boldsymbol{U}\boldsymbol{Q} = \boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I},\\ &\operatorname{tr}(\boldsymbol{U}\boldsymbol{Q})^T\boldsymbol{S}_B\boldsymbol{U}\boldsymbol{Q} = \operatorname{tr}\boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U}\boldsymbol{Q}\boldsymbol{Q}^T = \operatorname{tr}\boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U}.\end{aligned}$$

The same indeterminacy holds for PCA as well: $\boldsymbol{U}$ and $\boldsymbol{U}\boldsymbol{Q}$ span the same subspace. Second, note that a solution of (11.7) is not in general orthonormal. In fact, $\boldsymbol{U}$ *whitens* the within-class scatter $\boldsymbol{S}_W$ (Section 11.1.1). As we will see in Section 11.2.4, more is true. For a solution $\boldsymbol{U}$:

$$\boldsymbol{U}^T\boldsymbol{S}_W\boldsymbol{U} = \boldsymbol{I}, \quad \boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U} \text{ diagonal.}$$

The transformation $\boldsymbol{U}$ diagonalizes both $\boldsymbol{S}_W$ and $\boldsymbol{S}_B$ at the same time: it performs a *simultaneous diagonalization.* The geometric picture provides insight into the relationship between PCA and LDA. The former focusses on a single matrix $\boldsymbol{S}$, which we can diagonalize by an *orthonormal* transform $\boldsymbol{U}$, consisting of its eigenvectors. Alternatively, we can choose a non-orthogonal transform to whiten $\boldsymbol{S}$, say by rescaling the eigenvectors by the square root of eigenvalues. In LDA, we have to deal with *two* matrices $\boldsymbol{S}_W$ and $\boldsymbol{S}_B$, so are more restricted in what we can do. Obviously, we cannot whiten both of them with one transform in general. Also, it is not in general possible to diagonalize both of them with a single orthonormal transform $\boldsymbol{U}$. What we *can* do is to whiten $\boldsymbol{S}_W$ and diagonalize $\boldsymbol{S}_B$ with a single general transform, and solutions to LDA are found among such simultaneously diagonalizing transforms.

Finally, how to choose M, the number of independent LDA features? Ultimately this is a model selection problem, but what is the largest M we can possibly choose? Recall that $\boldsymbol{S}_B = \boldsymbol{M}\boldsymbol{M}^T$, where $\boldsymbol{M} \in \mathbb{R}^{d\times C}$, therefore

$$\operatorname{tr}\boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U} = \operatorname{tr}(\boldsymbol{U}^T\boldsymbol{M})^T\boldsymbol{U}^T\boldsymbol{M}.$$

Since $\operatorname{rk}(\boldsymbol{M}) = C - 1$, the matrix $\boldsymbol{U}^T\boldsymbol{M}$ has at most rank $C - 1$. This means that maximizing the criterion $\operatorname{tr}\boldsymbol{U}^T\boldsymbol{S}_B\boldsymbol{U}$ cannot determine more than $C - 1$ independent directions (columns of $\boldsymbol{U}$). If $M > C-1$, there are always solutions

$\boldsymbol{U}$ to (11.7) with $M - (C-1)$ zero columns (a proof of this fact is given in Section 11.2.4). This limitation of LDA is most clearly seen for $C = 2$. There, $\boldsymbol{S}_B \propto \boldsymbol{d}\boldsymbol{d}^T$ is a rank one matrix, and the criterion $\operatorname{tr} \boldsymbol{U}^T \boldsymbol{S}_B \boldsymbol{U}$ clearly only determines a single direction.

11.2.4 Techniques: Generalized Eigenproblems. Simultaneous Diagonalization (*)

The LDA problem (11.7) is an example of a generalized eigenproblem, defined by two symmetric matrices $\boldsymbol{S}_B$ and $\boldsymbol{S}_W$, where $\boldsymbol{S}_W$ is positive definite. Such problems are ubiquitous in machine learning and applied statistics. In this section, we discuss generalized eigenproblems in the context of simultaneous diagonalization, before showing how they can be solved efficiently.

The equality-constrained form of the LDA problem (11.7) suggests the following procedure:

- Whitening of $\boldsymbol{S}_W$: Determine *some* $\boldsymbol{V} \in \mathbb{R}^{d\times d}$ such that $\boldsymbol{V}^T \boldsymbol{S}_W \boldsymbol{V} = \boldsymbol{I}$. Note that *all* whitening transforms are given by $\boldsymbol{V}\boldsymbol{Q}$, where $\boldsymbol{Q} \in \mathbb{R}^{d\times d}$ is orthonormal.
- Diagonalization of $\boldsymbol{S}_B$: Find orthonormal $\boldsymbol{Q} \in \mathbb{R}^{d\times d}$ such that $(\boldsymbol{V}\boldsymbol{Q})^T \boldsymbol{S}_B \boldsymbol{V}\boldsymbol{Q}$ is diagonal. Output M columns of $\boldsymbol{V}\boldsymbol{Q}$ corresponding to the largest diagonal entries.

For the whitening step, we use the eigendecomposition $\boldsymbol{S}_W = \boldsymbol{R}\boldsymbol{\Lambda}\boldsymbol{R}^T$, where $\boldsymbol{R}$ are the eigenvectors. Then, $\boldsymbol{V} = \boldsymbol{R}\boldsymbol{\Lambda}^{1/2}$ is a whitening transform. Now,

$$(\boldsymbol{V}\boldsymbol{Q})^T \boldsymbol{S}_B \boldsymbol{V}\boldsymbol{Q} = \boldsymbol{Q}^T \left(\boldsymbol{V}^T \boldsymbol{S}_B \boldsymbol{V} \right) \boldsymbol{Q},$$

so $\boldsymbol{Q}$ can be determined by the eigendecomposition $\boldsymbol{V}^T \boldsymbol{S}_B \boldsymbol{V} = \boldsymbol{Q}\tilde{\boldsymbol{\Lambda}}\boldsymbol{Q}^T$. We manage to simultaneously diagonalize $\boldsymbol{S}_W$ and $\boldsymbol{S}_B$ by way of two standard eigendecompositions. While we will shortly explore a more efficient method for simultaneous diagonalization, the current approach has a simple geometrical interpretation. First, we rotate $\boldsymbol{S}_W$ into its eigenbasis, where it becomes diagonal. Second, we scale it to become white. This step is crucial, since a further orthogonal transformation of $\boldsymbol{V}^T \boldsymbol{S}_B \boldsymbol{V}$ into its eigenbasis (diagonalization) leaves $\boldsymbol{S}_W$ whitened.

We noted in Section 11.2.3 that we are restricted to $M \le C - 1$ in the LDA problem (11.7). We will see in a moment that a more efficient procedure can take this reduced size into account. But first, we prove this fact. Suppose that $M \ge C - 1$. We show that we can always find a solution $\boldsymbol{U}$ of Section 11.2.3 whose trailing $M - (C-1)$ columns are zero. This means that only $C - 1$ columns can sensibly be determined by the data. Suppose that $\boldsymbol{U}$ is any solution of (11.7). Since $\operatorname{rk}(\boldsymbol{U}^T \boldsymbol{M}) = r \le C - 1$, its eigendecomposition is $\boldsymbol{U}^T \boldsymbol{M} = \boldsymbol{E}\boldsymbol{\Lambda}\boldsymbol{F}^T$, where $\boldsymbol{\Lambda} \in \mathbb{R}^{r\times r}$ is diagonal and $\boldsymbol{E} \in \mathbb{R}^{M\times r}$ has orthonormal columns. Extend $\boldsymbol{Q} = [\boldsymbol{V}, *] \in \mathbb{R}^{M\times M}$ to be orthonormal. Then, the last $M - r$ rows of $(\boldsymbol{U}\boldsymbol{Q})^T \boldsymbol{M} = \boldsymbol{Q}^T \boldsymbol{U}^T \boldsymbol{M}$ are zero, so we might as well blank out the trailing $M - r$ columns of $\boldsymbol{U}\boldsymbol{Q}$ to attain a solution of (11.7) with trailing zeros.

Solving LDA in Practice (*)

In the remainder of this section, we show how to solve LDA efficiently in practice. We will see that by combining the lessons learned in this chapter in a clever way, we can get away with a small eigenproblem of size $C \times C$, along with having to solve C linear systems with $\boldsymbol{S}_W$, problems which can be solved much more efficiently than a $d \times d$ eigendecomposition. First, since $\boldsymbol{S}_W$ is positive definite, it has a Cholesky decomposition $\boldsymbol{S}_W = \boldsymbol{V}\boldsymbol{V}^T$ (Section 4.2.2). Substituting $\boldsymbol{W} = \boldsymbol{V}^T\boldsymbol{U}$, the LDA problem becomes

$$\max \operatorname{tr} \boldsymbol{W}^T\boldsymbol{V}^{-1}\boldsymbol{M}\boldsymbol{M}^T\boldsymbol{V}^{-T}\boldsymbol{W} \quad \text{s.t.} \quad \boldsymbol{W}^T\boldsymbol{W} = \boldsymbol{I}.$$

This is the usual Rayleigh-Ritz characterization (Section 11.1.2), so the solution is the $(C-1)$-dimensional leading eigenspace of $\boldsymbol{V}^{-1}\boldsymbol{M}\boldsymbol{M}^T\boldsymbol{V}^{-T} = \boldsymbol{V}^{-1}\boldsymbol{M}(\boldsymbol{V}^{-1}\boldsymbol{M})^T$.

At this point, we employ the trick discussed in Section 11.1.4. If $\boldsymbol{X} = \boldsymbol{V}^{-1}\boldsymbol{M}$, then $\boldsymbol{X}\boldsymbol{X}^T$ has essentially the same eigendecomposition as

$$\boldsymbol{X}^T\boldsymbol{X} = (\boldsymbol{V}^{-1}\boldsymbol{M})^T(\boldsymbol{V}^{-1}\boldsymbol{M}) = \boldsymbol{M}^T(\boldsymbol{S}_W)^{-1}\boldsymbol{M} \in \mathbb{R}^{C\times C}.$$

Since $\mathrm{rk}(\boldsymbol{M}) = C - 1$, this matrix has $C - 1$ positive eigenvalues. We have derived the following procedure for solving LDA.

- Compute $\boldsymbol{M} \in \mathbb{R}^{d\times C}$ and solve for $(\boldsymbol{S}_W)^{-1}\boldsymbol{M}$ (C linear systems).
- Compute the eigendecomposition

  $$\boldsymbol{M}^T(\boldsymbol{S}_W)^{-1}\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T, \quad \boldsymbol{Q} \in \mathbb{R}^{C\times(C-1)}, \ \ \boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}.$$

  According to Section 11.1.4, $\hat{\boldsymbol{W}} = (\boldsymbol{V}^{-1}\boldsymbol{M})\boldsymbol{Q}\boldsymbol{\Lambda}^{-1/2}$ are the eigenvectors for $\boldsymbol{X}\boldsymbol{X}^T$, and

  $$\hat{\boldsymbol{U}} = \boldsymbol{V}^{-T}\hat{\boldsymbol{W}} = \left((\boldsymbol{S}_W)^{-1}\boldsymbol{M}\right)\boldsymbol{Q}\boldsymbol{\Lambda}^{-1/2}$$

  solves the LDA problem. Here, we can reuse $(\boldsymbol{S}_W)^{-1}\boldsymbol{M}$, which was computed above.

With a bit more work, we can even get by with solving $C - 1$ linear systems and a $(C-1)\times(C-1)$ eigendecomposition. For $C = 2$, this modified procedure exactly recovers the FLD method derived at the beginning of this section.

Chapter 12

Unsupervised Learning

Machine learning is about inducing robust statistical relationships between variables from data in an automatic fashion. Variables can be of different kind and structure and be dependent in many different ways. Many prominent machine learning problems are of the *supervised learning* type, where a *function* from some input point $\boldsymbol{x}$ to some target variable t is to be learned. The latter lives in $\mathbb{R}$ (regression estimation), in $\{-1,+1\}$ (binary classification), or in some finite set (multi-way classification, ordinal regression, ranking). Datasets are *labeled*, meaning that pairs $(\boldsymbol{x}_i, t_i)$ are observed. In supervised learning, the machine learning method plays the role of a student in a rote learning session: for $\boldsymbol{x}_1$ you say t_1, for $\boldsymbol{x}_2$ you say t_2, and so on.

There is more to machine learning, and in this chapter we will begin to get an idea about some fundamental concepts we have not touched so far, and which in machine learning lingo are called *unsupervised learning.* They apply in situations where rote learning on carefully preprocessed and hand-labeled data is not possible or would be too costly, or where fitting a function or classifier is not the aim in the first place. For example, we may want to discover structure in data $\{\boldsymbol{x}_i\}$ per se, without obtaining any teaching signals. Or our goal may be classification, but we have to deal with "raw" data, partly unlabeled, riddled with missing attribute values, outliers and other distortions. In general, supervised learning cannot deal with such data, and it is up to us to clean it up. Unsupervised learning techniques can help us in that respect.

In this chapter, we will mainly concentrate on clustering and mixture density estimation, important instances of unsupervised learning, yet no more than the tip of the iceberg. We will understand why it makes sense to augment probabilistic models by variables which we can never observe. We will find that maximum likelihood estimation has more to it than computing sample means, covariances and counting words, is in fact powerful enough to drive general unsupervised learning. Dimensionality reduction techniques such as principal components analysis (Section 11.1) are instances of unsupervised learning as well, even though their modern interpretation in terms of density estimation [46] is out of scope of this course. In general, the foundation of statistical machine learning on probabilistic modelling (Chapter 5,Chapter 6) gathers full stream only with probabilistic unsupervised learning.

12.1 Clustering. K-Means Algorithm

Pretty much from the day you were born, you are faced with having to process a multitude of sensory input data, which you need to make sense off to survive and to thrive. You do get some teaching signals from parents, friends, and school teachers later on, but most of the time you do not. How do you do it? While human learning is imperfectly understood, a key element to it must be conceptual grouping. We organize our world view by putting patterns into distinct or overlapping bins, the personal labels of which we typically invent ourselves. If some grouping scheme works well, it can be adopted as consensus, but that does not mean there has to be any physical "reality" to it. Examples include colours, spoken words, or names of biological species. A first step to understanding and organizing data is to *cluster* it.

For simplicity, we restrict ourselves to non-overlapping clustering. Given a set of points $\{\boldsymbol{x}_i \mid i = 1, \dots, n\}$, a clustering is an assignment of datapoints to $K > 1$ groups, a partition of the dataset into K clusters. As we will see below, it makes perfect sense to allow the clustering to be probabilistic, in the sense that each $\boldsymbol{x}_i$ belongs to cluster k with a certain probability which can be different from 0 and 1. However, in the current section, we restrict our attention to *deterministic clustering*, where each $\boldsymbol{x}_i$ belongs to precisely one of the K clusters. We can formalize the assignment by introducing additional variables $t_i \in 1, \dots, K$, one for each pattern $\boldsymbol{x}_i$. A clustering of $\{\boldsymbol{x}_i\}$ is defined by an instantiation of $\{t_i\}$, in the sense that pattern $\boldsymbol{x}_i$ belongs to cluster t_i under the assignment. We have deliberately used the same notation as for K-way classification: clustering is classification, with the twist that we never get to see any of the labels. Before we get into any details, a central point to understand about clustering should be clear from the analogies above. Unlike classification with label data provided, clustering is a fundamentally ill-posed problem. There is no "best" solution independent of any assumptions. In order to comprehend the results you get with a certain clustering scheme, you need to understand the assumptions it is based on. It is good practice to run several different clustering schemes on your data in order to get a balanced picture. Notwithstanding such conceptual issues, the following questions have to be addressed by a clustering scheme.

- How to score the "quality" of cluster assignment $\{t_i\}$, given our assumptions?
- How to find an assignment of maximum score among the combinatorial set of all possible clusterings?
- How to choose the number K of clusters?

A large number of answers to these questions have been proposed, all coming with strengths and weaknesses. Among the more basic concepts, two are most widely used: agglomerative and divisive clustering. Both are hierarchical clustering principles, in that not only a single K-clustering is produced, but a tree of nested clusterings at different granularities (number of groups). They start with a user-supplied distance function $d(\boldsymbol{x}, \boldsymbol{x}')$ and the general assumption that distance values between two patterns in the same cluster should be smaller than distance values between two patterns in different clusters.

Another degree of freedom is the extension of distance between two points to distance between two sets of points $\mathcal{A}$ and $\mathcal{B}$. Frequently used extensions include $d(\mathcal{A},\mathcal{B}) = \min_{\boldsymbol{x}\in\mathcal{A},\boldsymbol{x}'\in\mathcal{B}} d(\boldsymbol{x},\boldsymbol{x}')$, $d(\mathcal{A},\mathcal{B}) = \max_{\boldsymbol{x}\in\mathcal{A},\boldsymbol{x}'\in\mathcal{B}} d(\boldsymbol{x},\boldsymbol{x}')$, or $d(\mathcal{A},\mathcal{B}) = d(\boldsymbol{\mu}_{\mathcal{A}},\boldsymbol{\mu}_{\mathcal{B}})$, where $\boldsymbol{\mu}_{\mathcal{S}}$ is the empirical mean of points in $\mathcal{S}$. *Agglomerative clustering* works bottom up. It starts with $K = n$ and each $\boldsymbol{x}_i$ forming its own group. In successive rounds, the two groups with the smallest distance are combined to form a new group. In contrast, *divisive clustering* is top down, starting with $K = 1$ and all $\boldsymbol{x}_i$ in a single group. In each round, one of the larger remaining groups is split along a boundary of largest pairwise distances. It is typically somewhat simpler to implement agglomerative schemes. On the other hand, divisive clustering can be reduced to combinatorial problems such as minimum cut, which can be solved efficiently. If only a few large clusters are sought, it can be more efficient than a bottom up scheme.

K-Means Clustering

In the remainder of this section, we concentrate on a widely used clustering scheme known as *K-means clustering* or vector quantization. K-means does not produce hierarchical groupings, the number of clusters K has to be specified up front. During the algorithm, we do not only maintain an assignment $\{t_i\}$ between datapoints $\boldsymbol{x}_i$ and clusters, but also a set of prototype vectors $\boldsymbol{\mu}_k$, one for each group. Intuitively, $\boldsymbol{\mu}_k$ represents the k-th group as its center of mass. K-means is typically based on the Euclidean distance between vectors (after preprocessing), and we will concentrate on this case. The method is driven by the following two requirements on $\{t_i\}$ and $\{\boldsymbol{\mu}_k\}$:

- Each prototype vector $\boldsymbol{\mu}_k$ should be the mean of the datapoints $\boldsymbol{x}_i$ assigned to class k:
$$\boldsymbol{\mu}_k = n_k^{-1}\sum_{i=1}^n \mathrm{I}_{\{t_i=k\}}\boldsymbol{x}_i, \quad n_k = \sum_{i=1}^n \mathrm{I}_{\{t_i=k\}}.$$
For this reason, $\boldsymbol{\mu}_k$ is also called *cluster center*.
- Each datapoint $\boldsymbol{x}_i$ should be assigned to the group whose prototype vector $\boldsymbol{\mu}_k$ is closest to $\boldsymbol{x}_i$ in Euclidean distance:
$$\|\boldsymbol{x}_i - \boldsymbol{\mu}_{t_i}\| = \min_{k=1,\dots,K}\|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|$$

Notice the "chicken-and-egg" structure of these requirements: what we want for one group of variables depends on what the other group is doing. If we fix the assignment $\{t_i\}$, the cluster centers $\boldsymbol{\mu}_k$ are obtained by maximum likelihood estimation, independently for each group. On the other hand, if we fix the cluster centers, the t_i are assigned by nearest neighbour classification. As we are not given any of these variables up front, our first attempt should be an iterative strategy. We initialize the $\boldsymbol{\mu}_k$ at random, say by placing them on top of K randomly selected datapoints. Then, we iterate over rounds of updating $\{t_i\}$, then updating $\{\boldsymbol{\mu}_k\}$ according to the requirements. We stop until the assignment does not change anymore. This is the *K-means algorithm.*

In Figure 12.1, we apply K-means to some data. Empirically, the algorithm always seems to converge to an assignment which fulfils the nearest neighbour

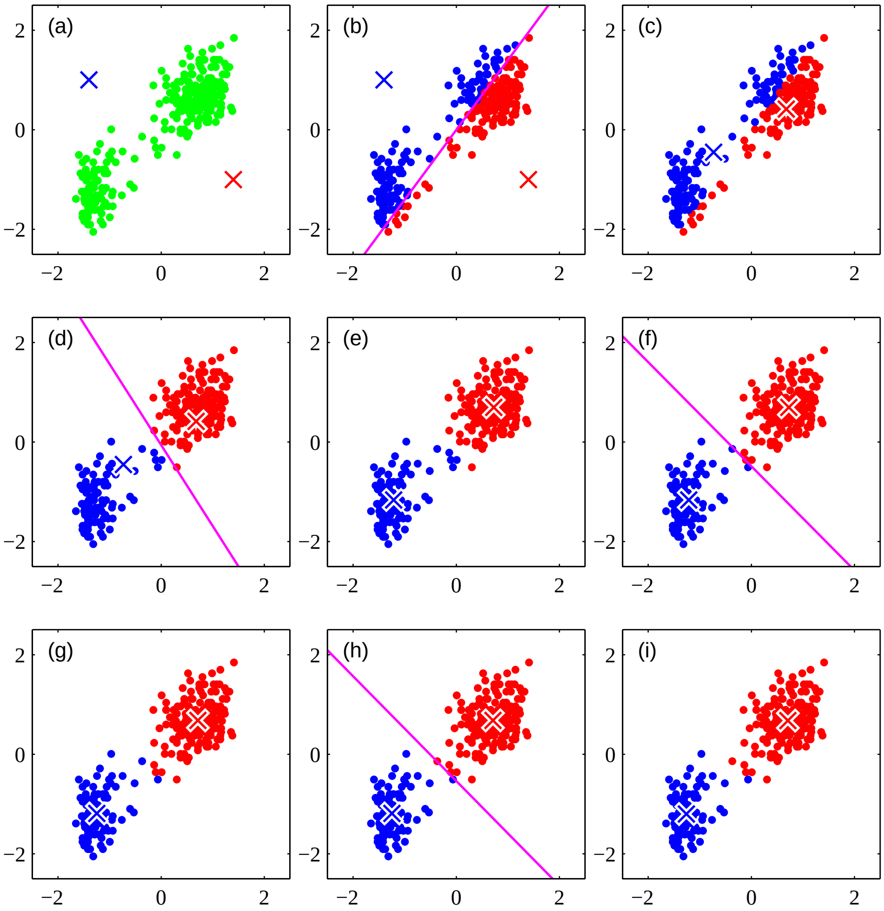


Figure 12.1: Illustration of K-means algorithm on data shown in (a), $K = 2$. The top row shows the first iteration, starting from the initial cluster centers in (a). (b): Each datapoint is assigned to the nearest cluster center. (c): The centers are re-estimated as empirical means of their data. (d)–(i): Further iterations until convergence. Figure from [6], used with permission.

criterion. However, when we run it multiple times, we can end up with different solutions, depending on how the initialization was done. Since each assignment we find satisfies our assumption, we would have to be more specific about how to compare two different outcomes of K-means. In Section 12.1.1, we will devise an energy function which scores assignments $\{t_i\}$, and which K-means descends on.

All in all, K-means is a simple and efficient algorithm, which is widely used for data analysis and within preprocessing pipelines of machine learning applications. Its main attraction is that it reduces to nearest neighbour search, a computational primitive which is very well studied, and for which highly efficient algorithms and data structures are known. On the other hand, the non-uniqueness of solutions can be a significant problem in practice, necessitating restarts with different initial conditions. Also, K-means is not a robust algorithm in general: small changes in the data $\{\boldsymbol{x}_i\}$ can imply large changes in the clustering $\{t_i\}$. Some of these problems are due to the hard cluster assignments we are forced to make in every single iteration. Finally, a good value for K is hard to select automatically from data. We will address some of these issues in the remainder of this chapter, when we devise a probabilistic foundation for K-means with "soft" assignments.

12.1.1 Analysis of the K-Means Algorithm

In this section, we obtain a more detailed understanding about the K-means clustering algorithm. We will postulate an *energy function* over assignments $\{t_i\}$ and prototype vectors $\{\boldsymbol{\mu}_k\}$ and prove that each iteration of K-means decreases this function. In fact, we will show that K-means terminates in finitely many iterations. Viewed as a function of $\{\boldsymbol{\mu}_k\}$ only, this energy is not convex, which provides a rationale for the non-uniqueness of K-means in practice.

For conciseness, denote $\boldsymbol{t} = [t_i]$ and $\boldsymbol{\mu} = [\boldsymbol{\mu}_k]$. An energy function for K-means is

$$\phi(\boldsymbol{t}, \boldsymbol{\mu}) = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathrm{I}_{\{t_i = k\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2.$$

We will show that $\phi(\boldsymbol{t}, \boldsymbol{\mu})$ cannot increase with an iteration of K-means. Moreover, whenever an iteration does not produce a decrease, a target assignment is reached. This means that the algorithm is finitely convergent. Namely, both $\boldsymbol{t}$ and $\boldsymbol{\mu}$ can only ever attain a finite number of different values. This is clear for $\boldsymbol{t}$. Moreover, each $\boldsymbol{\mu}_k$ is the empirical average over a subset of the datapoints $\boldsymbol{x}_i$, which are fixed up front.

Suppose we are at $(\boldsymbol{t}, \boldsymbol{\mu})$. We first update $\boldsymbol{t} \to \boldsymbol{t}'$, then $\boldsymbol{\mu} \to \boldsymbol{\mu}'$, so that at the end of an iteration, each $\boldsymbol{\mu}_k$ is the sample average over the patterns assigned to class k. Denote $\phi_0 = \phi(\boldsymbol{t}, \boldsymbol{\mu})$, $\phi_1 = \phi(\boldsymbol{t}', \boldsymbol{\mu})$, $\phi_2 = \phi(\boldsymbol{t}', \boldsymbol{\mu}')$. We also assume that if for any i,

$$\|\boldsymbol{x}_i - \boldsymbol{\mu}_{t_i}\|^2 = \min_k \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2,$$

then $t'_i = t_i$ (no change). First,

$$\begin{aligned}\phi_0 - \phi_1 &= \sum_{i=1}^{n} \left(\|\boldsymbol{x}_i - \boldsymbol{\mu}_{t_i}\|^2 - \|\boldsymbol{x}_i - \boldsymbol{\mu}_{t'_i}\|^2 \right) \\ &= \sum_{i=1}^{n} \left(\|\boldsymbol{x}_i - \boldsymbol{\mu}_{t_i}\|^2 - \min_k \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2 \right) \ge 0,\end{aligned}$$

and $\phi_1 = \phi_0$ if and only if $\boldsymbol{t} = \boldsymbol{t}'$. Second,

$$\boldsymbol{\mu}'_k = \frac{1}{n'_k} \sum_{i=1}^{n} \mathrm{I}_{\{t'_i = k\}} \boldsymbol{x}_i, \quad n'_k = \sum_{i=1}^{n} \mathrm{I}_{\{t'_i = k\}}.$$

As so often before, we expand the quadratic

$$\|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2 = \|(\boldsymbol{x}_i - \boldsymbol{\mu}'_k) + (\boldsymbol{\mu}'_k - \boldsymbol{\mu}_k)\|^2,$$

making use of vanishing cross-talk $\sum_i \mathrm{I}_{\{t'_i=k\}}(\boldsymbol{x}_i - \boldsymbol{\mu}'_k) = 0$, so that

$$\phi_1 - \phi_2 = \sum_{k=1}^{K}\sum_{i=1}^{n} \mathrm{I}_{\{t'_i=k\}} \left(\|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2 - \|\boldsymbol{x}_i - \boldsymbol{\mu}'_k\|^2\right) = \sum_{k=1}^{K} n'_k \|\boldsymbol{\mu}'_k - \boldsymbol{\mu}_k\|^2,$$

and $\phi_2 = \phi_1$ if and only if $\boldsymbol{\mu}'_k = \boldsymbol{\mu}_k$ for all k. All in all, $\phi_2 \le \phi_0$. Moreover, if $\phi_2 = \phi_0$, then $\boldsymbol{t}' = \boldsymbol{t}$ and K-means stops.

In practice, K-means tends to converge in few rounds. However, it is not necessarily a well-behaved algorithm. Unfortunately, it does not always converge to a global minimum solution of $\phi(\boldsymbol{t}, \boldsymbol{\mu})$. Finding such a global minimum can be a hard problem.

12.2 Density Estimation. Mixture Models

Clustering methods such as K-means do not have an immediate interpretation in terms of probabilistic modelling. They are formulated in terms of distances rather than likelihood functions, and they use hard assignments rather than posterior probabilities during optimization. However, already our notation in terms of $(\boldsymbol{x}_i, t_i)$ and $\boldsymbol{\mu}_k$, as well as the use of Euclidean squared distances $\|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2$ suggests some link to generative models with Gaussian class-conditional densities we learned about in Section 6.4. Consider a generative model defined in terms of class-conditionals $p(\boldsymbol{x}|t) = N(\boldsymbol{x}|\boldsymbol{\mu}_t, \boldsymbol{I})$ and $P(t = k) = 1/K$, where $k \in \{1, \dots, K\}$. It encodes the assumption that a datapoint $\boldsymbol{x}_i$ is sampled by first drawing its group label t_i uniformly from $\{1, \dots, K\}$, then the pattern from the spherical Gaussian $N(\boldsymbol{x}_i|\boldsymbol{\mu}_{t_i}, \boldsymbol{I})$. Given this model, we can understand the single steps of K-means as operations we already know. First, the update of $\boldsymbol{\mu}_k$ is the maximum likelihood estimator for a Gaussian mean, restricted to the points $\boldsymbol{x}_i$ assigned to the k-th group. Second, the update of t_i is a maximum a posteriori assignment for fixed model parameters $\{\boldsymbol{\mu}_k\}$:

$$t_i = \operatorname*{argmin}_k \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2 = \operatorname*{argmax}_k N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I})P(t = k) = \operatorname*{argmax}_k P(t_i = k|\boldsymbol{x}_i).$$

There is a marked difference between our clustering situation and generative classification in Section 6.4: we do not know the label values t_i here. Maximum likelihood estimation does not work here, because we are missing half of the data! Does it? How about the likelihood function for $\{\boldsymbol{x}_i\}$ only?

$$\prod_{i=1}^{n} p(\boldsymbol{x}_i) = \prod_{i=1}^{n} \left(\sum_{k=1}^{K} p(\boldsymbol{x}_i|t_i = k)P(t_i = k) \right).$$

Maybe we should adjust the parameters $\boldsymbol{\mu}_k$ and $P(t = k)$ by maximizing *this* likelihood function. Ironically, it is made up of these normalization constants $p(\boldsymbol{x}_i)$ which we happily dropped in all developments up to now.

In order to analyze and understand data beyond estimating some classification or regression function, we can try to build a model of the data-generating density itself. We can then fit model parameters by maximizing the likelihood. This is called *density estimation*, the basic concept behind “unsupervised learning”.

Did we not do all that before in Chapter 6, under the umbrella of generative modelling? True, but we will apply this principle to more complex models in this chapter, thereby unleashing its power. For the models in Chapter 6, variables were either observed or could be estimated by simple formulae such as empirical mean, empirical covariance, or count ratios. Here, we augment models by additional *latent variables*, such as our group indicators t_i, with the aim of making the model more expressive and realistic.

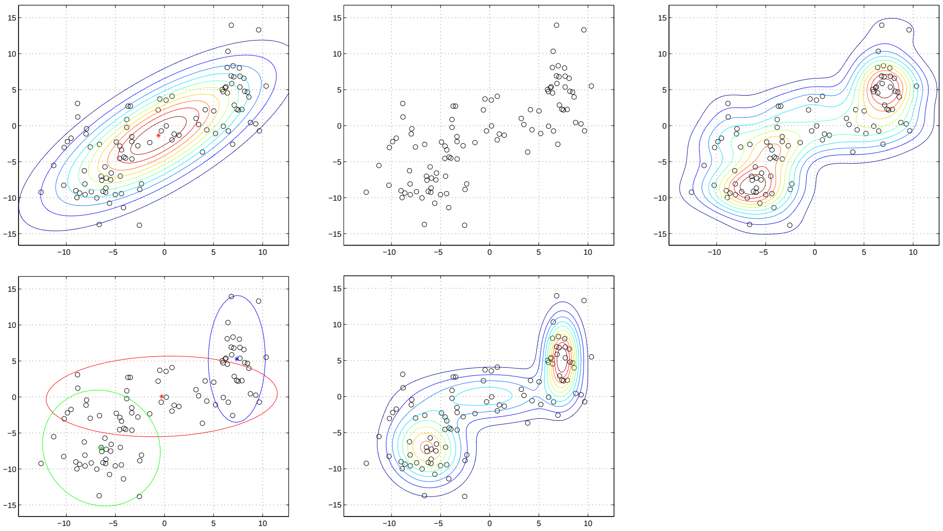


Figure 12.2: Illustration of different density estimation techniques, applied to the dataset shown in the top middle panel. Top left: A single Gaussian does not provide a good fit of the shape of the data. Top right: Kernel density estimator with kernel width $h = 2$ (see text). Good fit which comes at a high cost (one Gaussian kernel is placed on each of the 100 datapoints). Bottom middle: Gaussian mixture model with 3 components, fitted to the data by the EM algorithm (result shown bottom left). Excellent fit at moderate cost.

12.2.1 Mixture Models

Given the data in Figure 12.2 (top middle), what type of model should we use for density estimation? The simplest choice would be a single Gaussian $N(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. However, the maximum likelihood fit is not good in this case: it is unlikely that this data comes from a single Gaussian (Figure 12.2, top left). Another simple idea is known as *kernel density estimation*. We place a Gaussian function $N(\boldsymbol{x}|\boldsymbol{x}_i, h^2\boldsymbol{I})$ on each datapoint, then estimate the density as

$$\hat{p}(\boldsymbol{x}|h) = \frac{1}{n}\sum_{i=1}^{n} N(\boldsymbol{x}|\boldsymbol{x}_i, h^2\boldsymbol{I}).$$

The kernel width $h > 0$ is the only free parameter, its choice is a model selection problem (Chapter 10). The kernel density estimator is an instance of a nonparametric[1] statistical technique. As seen in Figure 12.2 (top right), this density estimator can provide a good fit, even though adjusting the kernel width h can be difficult. For example, if h is overly small, $\hat{p}(\boldsymbol{x}|h)$ is a set of peaks at the datapoints, the equivalent of over-fitting for density estimation. In contrast, if h is too large, fine but significant details are smoothed away. A more serious problem is the very high cost of evaluating $\hat{p}(\boldsymbol{x}|h)$ on future data. In fact, we have to store the whole dataset to represent $\hat{p}(\boldsymbol{x}|h)$, and each evaluation requires the evaluation of n Euclidean distances.

A compromise between a single Gaussian and one Gaussian on each datapoint is to use $K \ll n$ Gaussians, yet allow their parameters to be adjusted at will. In particular, their means $\boldsymbol{\mu}_k$ do not have to coincide with datapoints, and they may have different covariance matrices $\boldsymbol{\Sigma}_k$. This is a *Gaussian mixture model* (GMM)

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} p(\boldsymbol{x}|t=k)P(t=k) = \sum_{k=1}^{K} N(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)P(t=k).$$

Here, the $p(\boldsymbol{x}|t)$ are called *mixture components* and $P(t)$ is called *prior distribution*. The free parameters of this model are $\{(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\}$ and $\{P(t=k)\}$. If $\boldsymbol{x}$ is high-dimensional, it is common to use spherical ($\boldsymbol{\Sigma}_k = \sigma_k^2 \boldsymbol{I}$) or diagonal covariance matrices. Our probabilistic viewpoint of K-means developed above corresponds to a GMM with $\boldsymbol{\Sigma}_k = \boldsymbol{I}$. As seen in Figure 12.2 (bottom middle), a GMM with three components provides an excellent[2] fit for our data. On the other hand, it can be evaluated cheaply on future data.

Mixture models are ubiquitous in machine learning and applied statistics. For data in $\mathbb{R}^d$, whenever a single Gaussian does not look like an optimal choice, the next choice must be a GMM with few components. Much of their popularity stems from the simple and intuitive expectation maximization algorithm for parameter fitting, which we will learn about shortly. For discrete data, mixtures of multinomials or of independent Bernoulli (binary) distributions are widely used. There are many variations of the theme, such as hierarchical mixtures, parameter tying, and many more out of scope of this course. The `AutoClass` code[3] can be used to setup mixture models over data of many different attribute types and to run expectation maximization. Mixture models have profound impact on many applications. For example, modern large vocabulary continuous speech recognition systems are based on Gaussian mixture densities.

Given a mixture model, we can fit its parameters to data $\{\boldsymbol{x}_i\}$ by maximum likelihood estimation. For the GMM introduced at the beginning of this section, the log likelihood is

$$L(\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{i=1}^{n} \log \underbrace{\sum_{k=1}^{K} \pi_k N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I})}_{=p(\boldsymbol{x}_i)},$$

[1] Other examples for nonparametric techniques are nearest neighbour classification (Section 2.1) and kernel methods such as the support vector machine (Chapter 9).

[2] Not too surprisingly, as the true data generating distribution was a Gaussian mixture with three components in this case (not shown).

[3] `ti.arc.nasa.gov/tech/rse/synthesis-projects-applications/autoclass/autoclass-c/`

where $\pi_k = P(t = k)$ and $\boldsymbol{\mu}_k$ is the mean of the k-th mixture component. This log likelihood is different from what we encountered so far. The sum over k is inside the logarithm, and we cannot solve for $\boldsymbol{\mu}$ and $\boldsymbol{\pi}$ directly anymore. Such functions are called (log) *marginal likelihoods*. Still, let us try to take the gradient w.r.t. $\boldsymbol{\mu}_k$ and see how far we get. Recall the definition of the posterior:

$$P(t_i = k|\boldsymbol{x}_i) = \frac{p(\boldsymbol{x}_i|t_i = k)P(t_i = k)}{p(\boldsymbol{x}_i)} = \frac{\pi_k N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I})}{p(\boldsymbol{x}_i)}.$$

Let us concentrate on the i-th term:

$$\begin{aligned}\nabla_{\boldsymbol{\mu}_k} \log p(\boldsymbol{x}_i) &= \frac{\nabla_{\boldsymbol{\mu}_k} p(\boldsymbol{x}_i)}{p(\boldsymbol{x}_i)} = \frac{\nabla_{\boldsymbol{\mu}_k} p(\boldsymbol{x}_i, t_i = k)}{p(\boldsymbol{x}_i)} = \frac{\nabla_{\boldsymbol{\mu}_k} e^{\log\{\pi_k N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I})\}}}{p(\boldsymbol{x}_i)} \\ &= \frac{\pi_k N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I}) \nabla_{\boldsymbol{\mu}_k} \log N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I})}{p(\boldsymbol{x}_i)} \\ &= P(t_i = k|\boldsymbol{x}_i) \nabla_{\boldsymbol{\mu}_k} \log N(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{I}) = P(t_i = k|\boldsymbol{x}_i)(\boldsymbol{x}_i - \boldsymbol{\mu}_k).\end{aligned}$$

Setting this equal to zero and solving for $\boldsymbol{\mu}_k$, we obtain

$$\boldsymbol{\mu}_k = \frac{1}{n_k} \sum_{i=1}^{n} P(t_i = k|\boldsymbol{x}_i)\boldsymbol{x}_i, \quad n_k = \sum_{i=1}^{n} P(t_i = k|\boldsymbol{x}_i). \tag{12.1}$$

The update equation for the mean $\boldsymbol{\mu}_k$ is an empirical average over the datapoints $\boldsymbol{x}_i$, as usual in ML estimation. However, each datapoint is weighted by its posterior probability of belonging to the k-th class. If $\boldsymbol{x}_i$ lies halfway between class 1 and 2, then half of it contributes to $\boldsymbol{\mu}_1$ *and* $\boldsymbol{\mu}_2$ respectively. This is the soft group assignment we are after. A similar derivation provides us with the update for π_k. However, we need to take into account that $\boldsymbol{\pi}$ is a distribution, therefore make use of the technique detailed in Section 6.5.3. You should confirm for yourself that the update is

$$\pi_k = \frac{n_k}{n}, \quad n_k = \sum_{i=1}^{n} P(t_i = k|\boldsymbol{x}_i). \tag{12.2}$$

We sum up the contributions of each point $\boldsymbol{x}_i$ to group k.

There is something not quite right here. When we "solved" the gradient equation for $\boldsymbol{\mu}_k$, we ignored the fact that the posterior $P(t_i = k|\boldsymbol{x}_i)$ depends on $\boldsymbol{\mu}_k$ as well. What we have really done is to derive a set of *coupled* equations, where parameters we are after appear free on the left side and hidden in the posteriors on the right side. Nevertheless, it seems sensible to iterate these equations as follows:

- Compute all posterior distributions $[P(t_i = k|\boldsymbol{x}_i)]_k$, $i = 1, \dots, n$.
- Update $\{\boldsymbol{\mu}_k\}$ and $\boldsymbol{\pi}$ as posterior-weighted averages, according to (12.1) and (12.2).

This is an instance of the *expectation maximization* (EM) algorithm for Gaussian mixture models. We will establish convergence results for EM in Section 12.3.3. It provably converges to a local maximum of the log marginal likelihood function

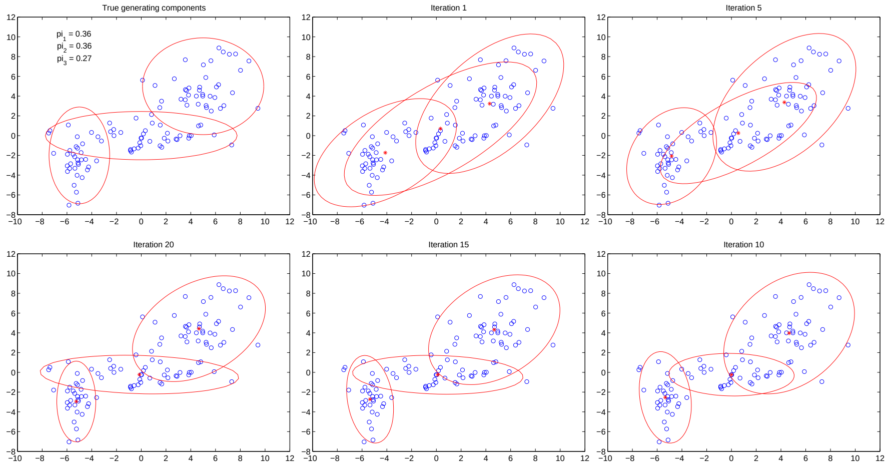


Figure 12.3: Expectation maximization algorithm applied to Gaussian mixture model with three components and general covariance matrices.

$L(\boldsymbol{\mu}, \boldsymbol{\pi})$. Since L is not in general a convex function, this is all we can hope for. Iterations of EM applied to a Gaussian mixture model are shown in Figure 12.3.

Properties and generalization of EM are discussed in the next section. Let us close by making the link between EM and K-means precise. Recall that $\boldsymbol{\Sigma}_k = \boldsymbol{I}$ and $\pi_k = 1/K$, and the group means $\boldsymbol{\mu}_k$ are the sole parameters to learn.

K-means Clustering	Expectation Maximization
$\boldsymbol{\mu}_k \leftarrow n_k^{-1} \sum_i \mathrm{I}_{\{t_i=k\}} \boldsymbol{x}_i$	$\boldsymbol{\mu}_k \leftarrow n_k^{-1} \sum_i Q_i(t_i = k) \boldsymbol{x}_i$
$n_k \leftarrow \sum_i \mathrm{I}_{\{t_i=k\}}$	$n_k \leftarrow \sum_i Q_i(t_i = k)$
$t_i \leftarrow \operatorname{argmax}_k P(t_i = k\vert\boldsymbol{x}_i)$	$Q_i(t_i = k) \leftarrow P(t_i = k\vert\boldsymbol{x}_i)$

K-means can be seen as a hard version of EM, or EM as a soft version of K-means. The main difference is that the posterior probabilities in EM are replaced by the indicator distributions $[\mathrm{I}_{\{t_i=k\}}]_k$, where t_i is the maximizer of the posterior $P(t_i = k|\boldsymbol{x}_i)$. K-means follows a "winner takes all" approach, in that $\boldsymbol{x}_i$ is counted fully towards $\boldsymbol{\mu}_{t_i}$ instead of being shared between components according to the posterior probability.

12.3 Latent Variable Models. Expectation Maximization

A mixture model is an instance of a *latent variable model*. For every observed variable $\boldsymbol{x}_i$, we augment our model by a latent (or hidden) variable t_i. Why do we bother with variables whose value we can never observe? Because they allow us, in an economic and intuitive way, to construct complex models from simple well-understood ingredients. Gaussians have many interesting properties (Section 6.3), but in terms of expressiveness they are no match to Gaussian mixtures. Mixture models are only the tip of the latent variable model iceberg.

We can create hierarchical mixtures in order to represent sharing of information at different levels. We can represent rotations, scaling, translations and other distortions which may affect our data by latent variables, then use the EM algorithm in order to learn classifiers which are invariant to these transformations. We can learn from partially incomplete data by assigning latent variables to missing entries. While advanced latent variable models are out of scope of this course, they are founded on no more than the general principles laid out here. We begin with deriving the EM algorithm for general latent variables models, then give an example involving missing data before analysing the convergence properties of EM.

12.3.1 The Expectation Maximization Algorithm

In this section, we derive the expectation maximization (EM) algorithm in full generality. Our latent variable model is given by $P(\boldsymbol{x},\boldsymbol{h}|\boldsymbol{\theta})$. Here, $\boldsymbol{x}$ collects observed variables, $\boldsymbol{h}$ hidden variables, and $\boldsymbol{\theta}$ are the parameters to be learned. We frequently encounter the particular model structure

$$P(\boldsymbol{x},\boldsymbol{h}|\boldsymbol{\theta}) = P(\boldsymbol{x}|\boldsymbol{h},\boldsymbol{\theta})P(\boldsymbol{h}|\boldsymbol{\theta}),$$

but this does not have to be the case. In our GMM example above, $\boldsymbol{x} \leftarrow \boldsymbol{x}$, $\boldsymbol{h} \leftarrow t$, and $\boldsymbol{\theta} \leftarrow (\{\boldsymbol{\mu}_k\},\boldsymbol{\pi})$. Moreover,

$$P(\boldsymbol{x}|\boldsymbol{h},\boldsymbol{\theta}) = P(\boldsymbol{x}|t,\{\boldsymbol{\mu}_k\}) = N(\boldsymbol{x}|\boldsymbol{\mu}_t,\boldsymbol{I}), \quad P(\boldsymbol{h}|\boldsymbol{\theta}) = P(t|\boldsymbol{\pi}) = \pi_t.$$

Given some training data $\{\boldsymbol{x}_i\}$, we postulate latent variables $\boldsymbol{h}_i$, one paired with each $\boldsymbol{x}_i$. The log marginal likelihood function is

$$L(\boldsymbol{\theta}) = \log\prod_{i=1}^{n}\sum_{\boldsymbol{h}_i} P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta}) = \sum_{i=1}^{n}\log\sum_{\boldsymbol{h}_i} P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta}).$$

Here, $\sum_{\boldsymbol{h}_i}$ denotes a sum over all possible values of the discrete variable $\boldsymbol{h}_i$. Our derivation goes through just as well for continuous $\boldsymbol{h}_i$, if we replace the sum by an integral $\int \dots d\boldsymbol{h}_i$.

We can derive the EM criterion just as in Section 12.2.1, by taking the gradient of L w.r.t. $\boldsymbol{\theta}$. First, since $L(\boldsymbol{\theta})$ is a sum of terms of equal form,

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n}\log P(\boldsymbol{x}_i|\boldsymbol{\theta}),$$

we can concentrate our derivation on a single pattern $\boldsymbol{x}_i$ and sum up results at the end. For fixed i:

$$\nabla_{\boldsymbol{\theta}}\log P(\boldsymbol{x}_i|\boldsymbol{\theta}) = \frac{\sum_{\boldsymbol{h}_i}\nabla_{\boldsymbol{\theta}}P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta})}{P(\boldsymbol{x}_i|\boldsymbol{\theta})} \overset{!}{=} \sum_{\boldsymbol{h}_i} P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}}\log P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta}).$$

The gradient is the expected value of $\nabla_{\boldsymbol{\theta}}\log P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta})$, where $\boldsymbol{h}_i \sim P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})$. The stationary equations are complicated by the fact that the posterior distribution $P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})$ itself depends on $\boldsymbol{\theta}$. We can solve these equations by iterative decoupling. In each iteration, we compute the posteriors:

$$Q_i(\boldsymbol{h}_i) \leftarrow P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta}).$$

This is called the *E step* ("E" for "expectation"). The $Q_i(\cdot)$ are distribution parameters, en par with $\boldsymbol{\theta}$. In particular, they do not depend on $\boldsymbol{\theta}$ during the second part of the EM iteration, the *M step* ("M" for "maximization"). The gradient we consider there is

$$\mathrm{E}_{Q_i}\left[\nabla_{\boldsymbol{\theta}} \log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right] = \nabla_{\boldsymbol{\theta}} \left\{E_i(\boldsymbol{\theta}; Q_i) := \mathrm{E}_{Q_i}\left[\log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right]\right\}.$$

Solving the stationary equation for $\boldsymbol{\theta}$ amounts to maximizing the surrogate function

$$E(\boldsymbol{\theta}; \{Q_i\}) = \sum_{i=1}^{n} E_i(\boldsymbol{\theta}; Q_i) = \sum_{i=1}^{n} \mathrm{E}_{Q_i}\left[\log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right]$$

w.r.t. $\boldsymbol{\theta}$, for the $Q_i(\cdot)$ determined in the E step. To sum up, we initialize[4] $\boldsymbol{\theta}$ to some sensible values, then iterate over the following two steps:

- E step: Compute posterior distributions

  $$Q_i(\boldsymbol{h}_i) \leftarrow P(\boldsymbol{h}_i|\boldsymbol{x}_i, \boldsymbol{\theta}), \quad i = 1, \dots, n.$$

  In practice, this amounts to accumulating posterior *expectations* in terms of which we can represent the $E_i(\boldsymbol{\theta}; Q_i)$ functions.

- M step: *Maximize* the surrogate criterion

  $$E(\boldsymbol{\theta}; \{Q_i\}) = \sum_{i=1}^{n} E_i(\boldsymbol{\theta}; Q_i) = \sum_{i=1}^{n} \mathrm{E}_{Q_i}\left[\log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right]$$

  w.r.t. $\boldsymbol{\theta}$. Given the E step statistics, this step is typically not more difficult to do than maximizing the complete data log likelihood.

It is a valid question to ask why the M step computations should be simpler to do than optimizing $L(\boldsymbol{\theta})$ directly. For *some* latent variable models, the M step optimization can be complicated, and in such cases using EM is not recommended (see end of Section 12.3.3). But for many latent variable models, the M step maximization can be done in closed form. The Gaussian mixture model case can serve as an example, which is a typical rather than a special case. Recall the comparison between K-means and EM in Section 12.2.1. Suppose we knew all the group assignments t_i. Then, the update for $\boldsymbol{\theta}$ consists of summing up the datapoints $\boldsymbol{x}_i$ according to the corresponding indicators. Since we don't know the t_i, we use EM instead. Importantly, *the M step updates look exactly the same as the updates for known t_i*, except that we sum over posterior probabilities instead of indicator values.

Some questions are still open with our derivation of EM. Does it always converge? If so, what does it optimize? We will give answers to these questions in Section 12.3.3, after looking at an application of EM to missing data.

[4] It *does* matter how you initialize EM, but you need to pick a domain-dependent heuristic. For mixture models, it is often a good idea to assign component means $\boldsymbol{\mu}_k$ to randomly chosen datapoints and to set (co)variance parameters to large values (say, to the total (co)variance for all the dataset).

12.3.2 Missing Data

Real-world datasets are seldomly complete. With high-dimensional input points, some attribute values may be missing for many, if not most of the cases. Faced with such data, we can throw out every incomplete case, which leaves us with far less data. We can use heuristics to fill in the missing values, risking that this skews learning and predictions. Alternatively, we can use latent variable models. Before we start, we should note that the latent variable treatment of missing data we advocate here is based on the assumption that the pattern of which attribute values are missing does not carry information: values are *missing at random*. This assumption is not always justifiable. If your data is a set of medical records of patients, missing entries may be due to a doctor deciding against certain tests based on insight about the patient. If so, then these entries are not missing at random. Or a measurement devise may fail to output certain attribute values, because they lie out of range or saturate a sensor. If the pattern of missing values is not random, we may want to take into account its structure in a more complex model.

Recall the naive Bayes document classifier from Section 6.5. Let us try to improve upon the bag-of-words assumption by using a *bigram* model for text. Based on a conditional distribution $[p_{a|b}]$, $a, b \in \{1, \dots, M\}$ words in the dictionary,

$$p_{a|b} \geq 0, \quad \sum_{a=1}^{M} p_{a|b} = 1,$$

the likelihood for a document $\boldsymbol{x}$ is given by

$$P(\boldsymbol{x}) = \prod_{j=1}^{N} p_{x_j|x_{j-1}}, \quad \boldsymbol{x} = [x_1, \dots, x_N].$$

Here, $x_0 = \emptyset$ and $[p_{a|\emptyset}]$ is an additional distribution for the first word, which we assume to be known. For complete data, we would write

$$P(\boldsymbol{x}) = \prod_{a=1}^{M} \left((p_{a|\emptyset})^{\mathrm{I}_{\{x_1=a\}}} \prod_{b=1}^{M} (p_{a|b})^{\phi_{a|b}(\boldsymbol{x})} \right), \quad \phi_{a|b}(\boldsymbol{x}) = \sum_{j=2}^{N} \mathrm{I}_{\{x_j=a, x_{j-1}=b\}}, \tag{12.3}$$

then estimate $p_{a|b}$ by maximum likelihood, accumulating counts over the documents assigned to each class.

However, what do we do if some words x_j are missing in our training documents? A simple idea would be to break up documents at missing word locations, to treat all completely observed parts as independent documents per se. This would probably be a good working solution in this case, even though it would wreak havoc with a structured[5] document model. But let us work out a principled solution based on latent variables and EM, which in the case of a substantial fraction of missing words may well make a difference even in this simple setup.

Our training corpus consists of documents $\boldsymbol{x}_i = [x_{ij}] \in \{1, \dots, M\}^{N_i}$. For notational simplicity, we drop the document index i and concentrate on a single

[5] For example, on top of the bigram probabilities, we may impose a higher order structure (title, abstract, introduction, main, references), or we may want to model document length.

document $\boldsymbol{x} = [x_j]$ of length N. The "missing at random" assumption is implicit in our derivation: the fact that some word is missing in some document does not depend on the document identity within the corpus, neither on the position within the document. Purely for simplicity, we will also assume[6] that no two consecutive words can be missing: if x_j is missing, then x_{j-1} and x_{j+1} are observed. To avoid trivialities, we also assume that the first word x_1 is observed. If the last word x_N is missing, we simply strip it off, so it is no restriction to assume that x_N is observed. We declare all existing words in $\boldsymbol{x}$ as observed variables and all missing words as latent variables. In the notation of Section 12.3.1, these would be $\boldsymbol{x}$ (observed) and $\boldsymbol{h}$ (latent), but it is easier to stick with $\boldsymbol{x}$ for the document, identifying missing spots by $\mathcal{H} \subset \{1, \dots, N\}$. Therefore, entries x_j with $j \in \mathcal{H}$ are missing, while entries x_j with $j \notin \mathcal{H}$ are observed. Note that $\mathcal{H}$ for different documents need not be the same, and we allow for $\mathcal{H} = \emptyset$. Denote the observed index by $\mathcal{O} = \{1, \dots, N\} \setminus \mathcal{H}$, moreover $\boldsymbol{x}_{\mathcal{O}} = [x_j]_{j \in \mathcal{O}}$ for the observed, $\boldsymbol{x}_{\mathcal{H}} = [x_j]_{j \in \mathcal{H}}$ for the missing part of the document. The *marginal* likelihood $P(\boldsymbol{x}_{\mathcal{O}})$ over the observed data is obtained by starting from the *complete* likelihood $P(\boldsymbol{x})$ as in (12.3), then summing over $\boldsymbol{x}_{\mathcal{H}}$: $P(\boldsymbol{x}_{\mathcal{O}}) = \sum_{\boldsymbol{x}_{\mathcal{H}}} P(\boldsymbol{x})$. Our log marginal likelihood criterion is $L(\boldsymbol{\theta}) = \log P(\boldsymbol{x}_{\mathcal{O}})$, where $\boldsymbol{\theta} = [p_{a|b}]$ consists of M conditional distributions over $\{1, \dots, M\}$.

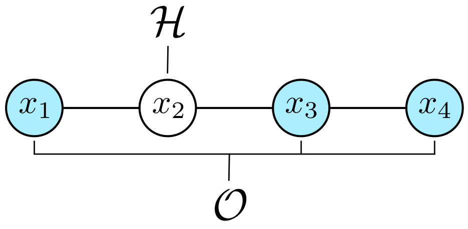


Figure 12.4: Graphical model for four consecutive words x_1, x_2, x_3, x_4. Here, $\mathcal{O} = \{x_1, x_3, x_4\}$ are observed, while $\mathcal{H} = \{x_2\}$ are latent. We can use the EM algorithm in order to learn from the observed data only.

How does the EM look like for this missing data example? In the E step, we need to compute the posterior distributions

$$Q(\boldsymbol{x}_{\mathcal{H}}) \leftarrow P(\boldsymbol{x}_{\mathcal{H}} | \boldsymbol{x}_{\mathcal{O}}, \boldsymbol{\theta}),$$

one for each document. To fix ideas, consider the four-word example in Figure 12.4. Here, $\mathcal{H} = \{2\}$, while $\mathcal{O} = \{1, 3, 4\}$. The posterior distribution is

$$P(x_2 | x_1, x_3, x_4) = \frac{P(x_1, x_2, x_3, x_4)}{\sum_{x_2'} P(x_1, x_2', x_3, x_4)} = \frac{p_{x_4|x_3} p_{x_3|x_2} p_{x_2|x_1} p_{x_1|\emptyset}}{\sum_{x_2'} p_{x_4|x_3} p_{x_3|x_2'} p_{x_2'|x_1} p_{x_1|\emptyset}}.$$

The first point to note is that all terms in the numerator which do not depend on x_2, appear in the denominator as well, and they cancel each other out:

$$P(x_2 | x_1, x_3, x_4) = \frac{p_{x_3|x_2} p_{x_2|x_1}}{\sum_{x_2'} p_{x_3|x_2'} p_{x_2'|x_1}}.$$

[6] It is straightforward to remove this assumption, but the derivation becomes a bit more tedious without revealing new ideas.

What happens in the general case? No matter how many other observed words occur elsewhere: if only a single word x_j is missing, cancellation happens all the same:

$$P(x_j|\boldsymbol{x}_{\mathcal{O}}) = \frac{p_{x_{j+1}|x_j} p_{x_j|x_{j-1}}}{\sum_{x'_j} p_{x_{j+1}|x'_j} p_{x'_j|x_{j-1}}}. \tag{12.4}$$

The same formula holds even if other words are missing as well. To understand the following general argument, it helps to write down some small examples (do that!). Namely,

$$P(x_j|\boldsymbol{x}_{\mathcal{O}}) = \frac{P(x_j, \boldsymbol{x}_{\mathcal{O}})}{P(\boldsymbol{x}_{\mathcal{O}})}, \quad P(x_j, \boldsymbol{x}_{\mathcal{O}}) = \sum\nolimits_{\boldsymbol{x}_{\mathcal{H}}\setminus x_j} P(\boldsymbol{x}).$$

By our assumptions, if $j \in \mathcal{H}$, then $x_{j\pm 1}$ are observed. If we denote $\mathcal{H}_{<j} = \{k \in \mathcal{H} \mid k < j\}$ and $\mathcal{H}_{>j} = \{k \in \mathcal{H} \mid k > j\}$, then

$$\begin{aligned} &P(x_j, \boldsymbol{x}_{\mathcal{O}}) = (C_{>j}) p_{x_{j+1}|x_j} p_{x_j|x_{j-1}} (C_{<j}), \\ &C_{<j} = \sum\nolimits_{\boldsymbol{x}_{\mathcal{H}_{<j}}} P(x_1, \dots, x_{j-1}), \quad C_{>j} = \sum\nolimits_{\boldsymbol{x}_{\mathcal{H}_{>j}}} P(x_{j+1}, \dots, x_N). \end{aligned}$$

Since $C_{<j}$ and $C_{>j}$ do not depend on x_j, they cancel out in (12.4). We have shown that

$$Q(\boldsymbol{x}_{\mathcal{H}}) \leftarrow P(\boldsymbol{x}_{\mathcal{H}}|\boldsymbol{x}_{\mathcal{O}}, \boldsymbol{\theta}) = \prod_{j\in\mathcal{H}} P(x_j|\boldsymbol{x}_{\mathcal{O}}, \boldsymbol{\theta}),$$

where $P(x_j|\boldsymbol{x}_{\mathcal{O}}, \boldsymbol{\theta})$ is computed as (12.4). A naive[7] way to do the E step computations costs $O(|\mathcal{H}|\, M)$ per document.

For the M step, we follow the mantra laid out in Section 12.3.1. The surrogate criterion $E(\boldsymbol{\theta}; Q)$ for our document is obtained by averaging

$$\log P(\boldsymbol{x}) \doteq \sum_{a=1}^{M} \sum_{b=1}^{M} \phi_{a|b}(\boldsymbol{x}) \log p_{a|b}$$

over $Q(\boldsymbol{x}_{\mathcal{H}})$ (here, $\doteq$ denotes equality up to a constant independent of $\boldsymbol{\theta}$). All we have to do is to average the indicators $\phi_{a|b}(\boldsymbol{x})$, then update the $p_{a|b}$ in the same way as before, based on these average statistics. A convenient way to write the average statistics is to extend the Q distribution over the whole document $\boldsymbol{x}$. To this end, we deviate from our convention above and denote the observed values by $\tilde{\boldsymbol{x}}_{\mathcal{O}}$ instead of $\boldsymbol{x}_{\mathcal{O}}$. Then,

$$Q(\boldsymbol{x}) = Q(\boldsymbol{x}_{\mathcal{H}}) \prod_{j\in\mathcal{O}} \mathrm{I}_{\{x_j = \tilde{x}_j\}}.$$

Taking an expectation w.r.t. $Q(\boldsymbol{x})$ means plugging in $\tilde{\boldsymbol{x}}_{\mathcal{O}}$ for $\boldsymbol{x}_{\mathcal{O}}$, then taking the expectation over $Q(\boldsymbol{x}_{\mathcal{H}})$. Given this definition, the M step surrogate is

$$\begin{aligned} &E(\boldsymbol{\theta}; Q) \doteq \sum_{a=1}^{M} \sum_{b=1}^{M} Q_{a|b} \log p_{a|b}, \\ &Q_{a|b} = \mathrm{E}_Q\left[\phi_{a|b}(\boldsymbol{x})\right] = \sum\nolimits_{j=2}^{N} Q(x_j = a) Q(x_{j-1} = b). \end{aligned}$$

[7] In practice, for any given b, $p_{a|b} \approx 0$ for most $a \in \{1, \dots, M\}$. This approximate sparsity of the distributions can be used to speed up accumulations dramatically.

Please check for yourself how this computation would be done efficiently in practice. For example, at least one of the factors in $Q(x_j = a)Q(x_{j-1} = b)$ is an indicator. If most of the words are observed, it may be fastest to first do the usual accumulation over observed pairs (x_j, x_{j-1}), followed by adding terms for the missing x_j. Finally, according to Section 6.5.3, the M step updates are

$$p_{a|b} = \frac{Q_{a|b}}{\sum_{a'} Q_{a'|b}}, \quad a, b \in \{1, \dots, M\}.$$

12.3.3 Convergence of Expectation Maximization

In this section, we show that under mild assumptions, the EM algorithm converges to a local maximum of the log marginal likelihood. We also comment on the relationship between EM and alternative nonlinear optimization techniques. Recall that the log marginal likelihood for a general latent variable model is

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log P(\boldsymbol{x}_i|\boldsymbol{\theta}), \quad P(\boldsymbol{x}_i|\boldsymbol{\theta}) = \sum_{\boldsymbol{h}_i} P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta}).$$

Here, the latent variable $\boldsymbol{h}$ is discrete. Our derivation applies without changes for continuous (or mixed) latent variables as well, we only have to replace sums by integrals. During EM, we maintain distributions $Q_i(\boldsymbol{h}_i)$ over $\boldsymbol{h}_i$, one for each datapoint, writing Q for $\{Q_i\}$. Our argument is based on the auxiliary criterion

$$\phi(\boldsymbol{\theta}; Q) = \sum_{i=1}^{n} \left(\mathrm{E}_{Q_i}[\log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})] + \mathrm{E}_{Q_i}[-\log Q_i(\boldsymbol{h}_i)]\right).$$

Here,

$$\mathrm{H}[Q_i(\boldsymbol{h}_i)] = \mathrm{E}_{Q_i}[-\log Q_i(\boldsymbol{h}_i)] = \sum_{\boldsymbol{h}_i} Q_i(\boldsymbol{h}_i)(-\log Q_i(\boldsymbol{h}_i))$$

is the *entropy* of Q_i, a measure of the amount of uncertainty in $\boldsymbol{h}_i \sim Q_i$ [11]. Note that

$$\phi(\boldsymbol{\theta}; Q) = E(\boldsymbol{\theta}; Q) + \sum_{i=1}^{n} \mathrm{H}[Q_i(\boldsymbol{h}_i)],$$

where $E(\boldsymbol{\theta}; Q)$ is the surrogate criterion defined in Section 12.3.1. The entropic part is added to the energy E for technical reasons. It does not depend on $\boldsymbol{\theta}$, so the M step is not influenced.

In order to establish EM convergence, we show that $L(\boldsymbol{\theta})$ cannot decrease with an EM iteration. Moreover, if it does not increase, we must be at a local maximum. A key step will be the bound

$$\phi(\boldsymbol{\theta}; Q) \le L(\boldsymbol{\theta}) \quad \text{for all } \boldsymbol{\theta}, Q_i(\boldsymbol{h}_i), \tag{12.5}$$

where ϕ is maximized w.r.t. the Q_i in the E step and increased sufficiently w.r.t. $\boldsymbol{\theta}$ in the M step. Our arguments are illustrated in Figure 12.5. A general requirement for EM convergence is that the log marginal likelihood $L(\boldsymbol{\theta})$ is upper bounded, which may require additional assumptions on the parameters. For example, the log likelihood for a Gaussian mixture model with different

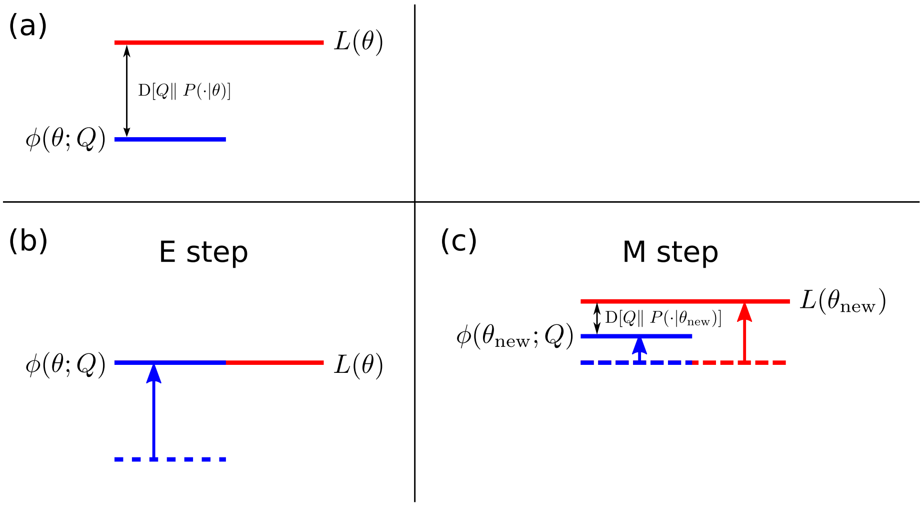

(a)
$L(\theta)$
$\mathrm{D}[Q\,\|\, P(\cdot|\theta)]$
$\phi(\theta;Q)$
(b)
E step
$\phi(\theta;Q)$
$L(\theta)$
(c)
M step
$L(\theta_{\text{new}})$
$\mathrm{D}[Q\,\|\, P(\cdot|\theta_{\text{new}})]$
$\phi(\theta_{\text{new}};Q)$

Figure 12.5: Illustration of one iteration of the EM algorithm. (a) The EM criterion $\phi(\boldsymbol{\theta};Q)$ is a lower bound on the log marginal likelihood $L(\boldsymbol{\theta})$. (b) In the E step, we update Q so to equate $\phi(\boldsymbol{\theta};Q)$ and $L(\boldsymbol{\theta})$ for the current parameters $\boldsymbol{\theta}$. (c) In the M step, we update $\boldsymbol{\theta} \to \boldsymbol{\theta}_{\text{new}}$ so to maximize $\phi(\boldsymbol{\theta};Q)$. Since $L(\boldsymbol{\theta}_{\text{new}}) \geq \phi(\boldsymbol{\theta}_{\text{new}};Q)$, this update increases the marginal likelihood as well.
Figure inspired by [6], figures 9.11 until 9.13.

covariances for each component is *not* upper bounded per se. We can place one component on top of a datapoint and shrink the variance to zero in order to obtain infinite likelihood! Such degenerate solutions are avoided by constraining the variances by some positive lower bound. In the remainder of this section, we assume that $L(\boldsymbol{\theta})$ is upper bounded.

We begin by relating $L(\boldsymbol{\theta})$ and $\phi(\boldsymbol{\theta};Q)$ for arbitrary distributions $Q_i(\boldsymbol{h}_i)$. Pick any $i \in \{1,\dots,n\}$. For the following, recall the definition of the posterior $P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})$:

$$\begin{aligned}
&\log P(\boldsymbol{x}_i|\boldsymbol{\theta}) = \mathrm{E}_{Q_i}\left[\log P(\boldsymbol{x}_i|\boldsymbol{\theta})\right] = \mathrm{E}_{Q_i}\left[\log \frac{P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta})}{P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})}\right] \\
&= \mathrm{E}_{Q_i}\left[\log \frac{P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta})Q_i(\boldsymbol{h}_i)}{P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})Q_i(\boldsymbol{h}_i)}\right] = \mathrm{E}_{Q_i}\left[\log \frac{P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta})}{Q_i(\boldsymbol{h}_i)} + \log \frac{Q_i(\boldsymbol{h}_i)}{P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})}\right] \\
&= \mathrm{E}_{Q_i}[\log P(\boldsymbol{x}_i,\boldsymbol{h}_i|\boldsymbol{\theta})] + \mathrm{H}[Q_i(\boldsymbol{h}_i)] + \mathrm{D}[Q_i(\boldsymbol{h}_i)\,\|\,P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})].
\end{aligned}$$

Here, $\mathrm{D}[Q_i(\boldsymbol{h}_i)\,\|\,P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})]$ is the relative entropy from Section 6.5.3 (make sure you recall the derivations in that section, we will need them here). This means that

$$\phi(\boldsymbol{\theta};Q) = L(\boldsymbol{\theta}) - \sum_{i=1}^{n} \mathrm{D}[Q_i(\boldsymbol{h}_i)\,\|\,P(\boldsymbol{h}_i|\boldsymbol{x}_i,\boldsymbol{\theta})] \qquad (12.6)$$

for any $\boldsymbol{\theta}$ and any distributions $Q_i(\boldsymbol{h}_i)$. But we know that the relative entropy between two distributions is nonnegative, and zero if and only if the distributions are the same, so that (12.6) implies (12.5), with equality if and only if $Q_i(\boldsymbol{h}_i) =$

$P(\boldsymbol{h}_i|\boldsymbol{x}_i, \boldsymbol{\theta})$ for all $i = 1, \dots, n$. As this is what we do in the E step, it can only increase $\phi(\boldsymbol{\theta}; Q)$. Moreover, the M step increases $E(\boldsymbol{\theta}; Q)$, and therefore $\phi(\boldsymbol{\theta}; Q)$, by definition. Since $\phi(\boldsymbol{\theta}; Q) \leq L(\boldsymbol{\theta})$, it is upper bounded as well, which completes our convergence proof.

Moreover, if an iteration of EM leaves $\phi(\boldsymbol{\theta}; Q)$ the same, we have reached a stationary point of the log marginal likelihood $L(\boldsymbol{\theta})$. Specifically, if $Q_i(\boldsymbol{h}_i) = P(\boldsymbol{h}_i|\boldsymbol{x}_i, \boldsymbol{\theta})$ for all $i = 1, \dots, n$, then

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \partial_{\boldsymbol{\theta}} \phi(\boldsymbol{\theta}; Q).$$

The right hand side are partial derivatives, the Q_i do not depend on $\boldsymbol{\theta}$. Therefore, a stationary point of the EM algorithm must be a stationary point of $L(\boldsymbol{\theta})$ as well. We already derived the gradient of $L(\boldsymbol{\theta})$ in Section 12.3.1:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \log \sum_{\boldsymbol{h}_i} P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta}) &= \frac{1}{P(\boldsymbol{x}_i|\boldsymbol{\theta})} \sum_{\boldsymbol{h}_i} P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta}) \left(\nabla_{\boldsymbol{\theta}} \log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right) \\ &= \mathrm{E}_{Q_i(\boldsymbol{h}_i)} \left[\nabla_{\boldsymbol{\theta}} \log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right] = \nabla_{\boldsymbol{\theta}} \left(\mathrm{E}_{Q_i} \left[\nabla_{\boldsymbol{\theta}} \log P(\boldsymbol{x}_i, \boldsymbol{h}_i|\boldsymbol{\theta})\right] + \mathrm{H}[Q_i(\boldsymbol{h}_i)]\right).\end{aligned}$$

Summing over $i = 1, \dots, n$, we obtain the gradient identity.

To EM or not to EM

This concludes our discussion of the EM algorithm, a simple and often surprisingly efficient method for finding a stationary point (local maximum) of the log likelihood of observed data. As discussed in Section 12.3.1, EM is particularly attractive if the M step computations can be done in closed form, by accumulating sufficient statistics over E step posteriors rather than counts. However, it is important to note that EM is not always the best algorithm to solve $\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ locally. We briefly discussed nonlinear gradient-based optimizers in Section 3.4.2. As seen above, computing the gradient $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ comes at exactly the same cost as a single E step in EM. When comparing EM with alternatives, we should focus on two points:

- Can we solve the M step in closed form, or do we need iterative techniques there as well?
- How many EM iterations do we require until convergence, versus how many iterations for an alternative optimizer?

To address the second point first, EM is generally reported to converge rapidly to a solution of low accuracy, but can be very slow to attain medium to high accuracy. If it is important to attain high accuracy, EM is not a good choice. However, in the author's opinion, it is the first point which decides for or against EM. If the M step can be done in closed form, it is hard to argue against the simplicity of EM, in particular since a low accuracy solution is typically reached in few iterations. On the other hand, if the M step requires iterative optimization as well, or even worse, necessitates some additional bounding, EM should not be used, since proper nonlinear optimization codes are almost always faster. An infamous example is maximum likelihood learning of conditional random fields, or log-linear undirected models in general. Most of the early work used variants of EM, such as iterative proportional fitting. In these algorithms, we obtain closed form M step updates only after additional crude bounding. These approaches have vanished entirely today, as modern optimizers such as scaled conjugate gradients, limited memory quasi-Newton or truncated Newton run several orders of magnitude faster. Another poor idea is to use EM in order to compute principal components analysis (PCA) directions (Section 11.1): the Lanczos algorithm is much faster in practice (Section 11.1.4), and good code is publicly available.

A final comment concerns the E step computations, which are required in order to compute $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ just as well. For the Gaussian mixture models with a moderate number K of components, the E step posteriors are cheap to compute. But in general, this computation can be hard to do if $\boldsymbol{h}$ is large and has dependent components. If you find yourself in such a situation, you need to consider *approximate* posterior computations. It is possible to extend the EM algorithm to allow for such. The resulting *variational EM* algorithm will still be convergent, but it does not in general maximize the exact log likelihood anymore.

Appendix A

Lagrange Multipliers and Lagrangian Duality

We haved derived the dual formulation of the soft margin SVM problem in Section 9.3. In this section, we provide a much more general picture on the underlying principle of Lagrange duality, which plays a central role in modern machine learning, far beyond support vector machines. We will introduce the framework in two stages, first seeking to generalize the first-order stationary condition to constrained problems by way of a Lagrangian function, then exposing the duality embedded in this function, leading to a dual optimization problem. Finally, we will rederive the soft margin SVM dual from this more general perspective. This section is slightly more advanced and can be skipped in a first reading.

Lagrange duality is a powerful framework which helps solving constrained optimization problems with continuously differentiable objective. We will restrict ourselves to linear constraints, as this is all we need in this course, but the framework is more generally applicable. There are two aspects to the Lagrangian technique. The first is a generalization of the famous first order optimality condition $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \mathbf{0}$ to the constrained case, by introducing new variables (multipliers) and the Lagrangian function. This is in fact what Lagrange did. It often helps to fence in optimal solutions by exploring stationary points of the Lagrangian, but does not provide a method to find them. The second aspect is Lagrange duality. The nontrivial stationary points of the Lagrangian are saddlepoints, which can be narrowed down naturally via two optimization problems (primal and dual). At least for convex optimization problems, Lagrange duality directly leads to algorithms for finding optimal solutions. Modern optimization textbooks often start with the duality and present the optimality condition in passing, but we will thread with Lagrange and develop the Lagrangian from first principles.

The optimization problem we will be interested in here is

$$\tilde{p}_* = \min_{\boldsymbol{x}} f(\boldsymbol{x}), \qquad \text{subj. to } \; g_j(\boldsymbol{x}) = 0 \;\; \forall j, \quad h_k(\boldsymbol{x}) \le 0 \;\; \forall k, \tag{A.1}$$

where $g_j(\boldsymbol{x}) = (\boldsymbol{a}_j)^T \boldsymbol{x} - b_j$, $h_k(\boldsymbol{x}) = (\boldsymbol{c}_k)^T \boldsymbol{x} - e_k$, and $\boldsymbol{x} \in \mathbb{R}^p$. It will be called the *primal problem* below. Each $g_j(\boldsymbol{x}) = 0$ is called linear equality constraint, each $h_k(\boldsymbol{x}) \leq 0$ is a linear inequality constraint. The set of those $\boldsymbol{x}$ which fulfil all constraints is called the *feasible set*, and its members $\boldsymbol{x}$ are called *feasible.* For an unconstrained problem, all $\boldsymbol{x} \in \mathbb{R}^p$ are feasible. The feasible set in our case is a convex polytope (Section 9.1.1). The number $\tilde{p}_*$ is called the *value* of the primal problem. We will assume that the feasible set is not empty, and that $\tilde{p}_* \in \mathbb{R}$, in particular $\tilde{p}_* > -\infty$.

Optimality Conditions

Consider an unconstrained optimization problem $\min_{\boldsymbol{x}} f(\boldsymbol{x})$, where $f(\boldsymbol{x})$ is continuously differentiable. If $\boldsymbol{x}_*$ is a local minimum point, then $\nabla_{\boldsymbol{x}} f = \boldsymbol{0}$ (Section 2.4.1). We have used this first order[1] necessary condition many times in this course already. But suppose we add linear constraints on $\boldsymbol{x}$. Then, this optimality condition does not work anymore. For example, $\boldsymbol{x} \in \mathbb{R}^2$,

$$\min_{\boldsymbol{x}} \{f(\boldsymbol{x}) = x_1\}, \quad \text{subj. to } x_1 = 1$$

has optimal solutions $\boldsymbol{x}_* = [1, \alpha]^T$, $\alpha \in \mathbb{R}$, but $\nabla_{\boldsymbol{x}_*} f = \boldsymbol{\delta}_1 \neq \boldsymbol{0}$. In order to derive the Lagrangian generalization of $\nabla_{\boldsymbol{x}} f = \boldsymbol{0}$, we proceed as in Section 2.4.1. Imagine you are a mountaineer who needs to get down, as it gets dark. But there are straight paths you cannot stray from (equality constraints), or fences and rivers you cannot cross (inequality constraints). Are there directions along which you can descend without violating any constraints? Let us start with a single equality constraint:

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}), \quad \text{subj. to } g(\boldsymbol{x}) = \boldsymbol{a}^T\boldsymbol{x} - b = 0.$$

Suppose we are at the feasible point $\boldsymbol{x}$, so that $g(\boldsymbol{x}) = 0$. Let us first determine along which directions $\boldsymbol{d}$ we may move at all, without violating the constraint:

$$g(\boldsymbol{x} + \varepsilon\boldsymbol{d}) = \boldsymbol{a}^T(\boldsymbol{x} + \varepsilon\boldsymbol{d}) - b = g(\boldsymbol{x}) + \varepsilon\boldsymbol{a}^T\boldsymbol{d} = \varepsilon\boldsymbol{a}^T\boldsymbol{d}.$$

A legal direction $\boldsymbol{d}$ to move along must be orthogonal to $\boldsymbol{a}$: $\boldsymbol{a}^T\boldsymbol{d} = 0$. Since $\nabla_{\boldsymbol{x}} g = \boldsymbol{a}$, we must have $(\nabla_{\boldsymbol{x}} g)^T\boldsymbol{d} = 0$. Next, we use the Taylor expansion of f at $\boldsymbol{x}$:

$$f(\boldsymbol{x} + \varepsilon\boldsymbol{d}) = f(\boldsymbol{x}) + \varepsilon(\nabla_{\boldsymbol{x}} f)^T\boldsymbol{d} + O(\varepsilon^2).$$

If there is some direction $\boldsymbol{d}$ such that $(\nabla_{\boldsymbol{x}} g)^T\boldsymbol{d} = 0$ and $(\nabla_{\boldsymbol{x}} f)^T\boldsymbol{d} < 0$, we can descend in $f(\boldsymbol{x})$ while staying feasible. A useful optimality condition must imply

$$(\nabla_{\boldsymbol{x}} g)^T\boldsymbol{d} = 0 \quad \Rightarrow \quad (\nabla_{\boldsymbol{x}} f)^T\boldsymbol{d} \geq 0.$$

Now, if $(\nabla_{\boldsymbol{x}} g)^T\boldsymbol{d} = 0$ and $(\nabla_{\boldsymbol{x}} f)^T\boldsymbol{d} > 0$, we can descend along $-\boldsymbol{d}$, so we need

$$(\nabla_{\boldsymbol{x}} g)^T\boldsymbol{d} = 0 \quad \Rightarrow \quad (\nabla_{\boldsymbol{x}} f)^T\boldsymbol{d} = 0.$$

This condition is fulfilled only if $\nabla_{\boldsymbol{x}} f$ is parallel to $\nabla_{\boldsymbol{x}} g$ or zero:

$$\nabla_{\boldsymbol{x}} f = -\lambda\nabla_{\boldsymbol{x}} g, \quad \nabla_{\boldsymbol{x}} f + \lambda\nabla_{\boldsymbol{x}} g = \boldsymbol{0}, \quad \lambda \in \mathbb{R}.$$

[1] First order in this context means that only the gradient (first derivatives) are used. In contrast, second order conditions look at the Hessian as well (Section 3.4).

Let us see whether this works for our example above. $f(\boldsymbol{x}) = x_1$, $\boldsymbol{a} = \boldsymbol{\delta}_1$, $b = 1$, so that $\nabla_{\boldsymbol{x}} f = \boldsymbol{\delta}_1 = \nabla_{\boldsymbol{x}} g$, and the condition holds for $\lambda = -1$. It is not very useful in this example, since it holds for any $\boldsymbol{x} \in \mathbb{R}^2$, but it certainly is a necessary condition for optimality. We illustrate Lagrange's condition in Figure A.1, top.

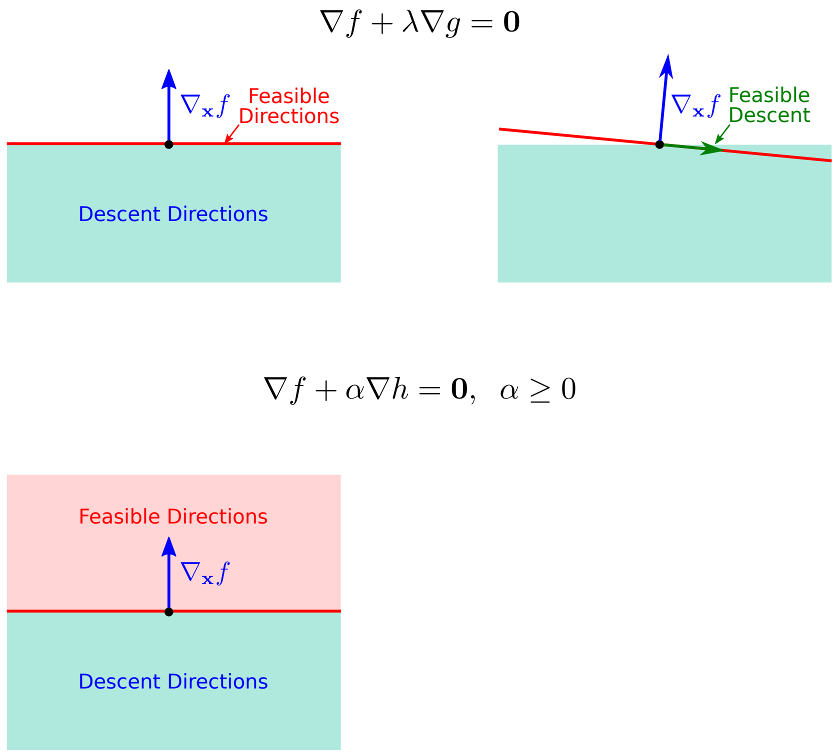


Figure A.1: To understand Lagrange's first-order conditions, we can visualize the set of descent directions $\mathcal{A} = \{\boldsymbol{d} \mid \boldsymbol{d}^T(\nabla f) < 0\}$ (open halfspace; blue) and the set $\mathcal{B}$ of feasible directions (red). The conditions ensure that the intersection $\mathcal{A} \cap \mathcal{B}$ is empty: there is no feasible descent direction.
Top: For a linear equality constraint $g(\boldsymbol{x}) = 0$, $\mathcal{B}$ is a line with normal vector ∇g. $\mathcal{A} \cap \mathcal{B} = \emptyset$ if ∇f and ∇g are parallel. Bottom: For an active linear inequality constraint $h(\boldsymbol{x}) \leq 0$, $\mathcal{B}$ is a closed halfspace. In this case, it is not enough for ∇f and ∇h to be parallel, they must also point in opposite directions.

Next, consider a single inequality constraint:

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}), \quad \text{subj. to } \; h(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} - e \leq 0.$$

Two things can happen now at $\boldsymbol{x}$. Either, $h(\boldsymbol{x}) < 0$, so that $h(\boldsymbol{x} + \varepsilon \boldsymbol{d}) < 0$ for small ε, no matter what $\boldsymbol{d}$. The constraint is *inactivate*, and we can simply pretend it is not there. The optimality condition is $\nabla_{\boldsymbol{x}} f = \mathbf{0}$ then. Or, $h(\boldsymbol{x}) = 0$: the constraint is *active*. Then,

$$h(\boldsymbol{x} + \varepsilon \boldsymbol{d}) = h(\boldsymbol{x}) + \varepsilon \boldsymbol{c}^T \boldsymbol{d} = \varepsilon \boldsymbol{c}^T \boldsymbol{d},$$

which is nonpositive if and only if $\boldsymbol{c}^T\boldsymbol{d} \le 0$. For an active constraint, we must have

$$(\nabla_{\boldsymbol{x}} h)^T \boldsymbol{d} \le 0 \quad \Rightarrow \quad (\nabla_{\boldsymbol{x}} f)^T \boldsymbol{d} \ge 0,$$

This implies that $\nabla_{\boldsymbol{x}} f = -\alpha \nabla_{\boldsymbol{x}} h$, or $\nabla_{\boldsymbol{x}} f + \alpha \nabla_{\boldsymbol{x}} h = \boldsymbol{0}$, where $\alpha \ge 0$. The condition is visualized in Figure A.1, bottom. It works for inactive constraints as well if $\alpha = 0$. A succinct way to write the optimality condition is

$$\nabla_{\boldsymbol{x}} f + \alpha \nabla_{\boldsymbol{x}} h = \boldsymbol{0}, \quad \alpha \ge 0, \quad \alpha h(\boldsymbol{x}) = 0.$$

Our primal problem (A.1) has several equality and inequality constraints. What do we do then? Our conditions all have the same form, so why not add them up and divide by the number of constraints:

$$\nabla_{\boldsymbol{x}} f + \sum_j \lambda_j \nabla_{\boldsymbol{x}} g_j + \sum_k \alpha_k \nabla_{\boldsymbol{x}} h_k = \boldsymbol{0}, \quad \alpha_k \ge 0, \;\; \alpha_k h_k(\boldsymbol{x}) = 0 \;\; \forall k.$$

These conditions read as follows. If $\boldsymbol{x}$ is a local minimum point of the constrained problem, then there are some values for $\boldsymbol{\lambda}$, $\boldsymbol{\alpha}$ such that $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ fulfil the conditions. But did we not lose something by just adding the single conditions? Let us check. Suppose our optimality conditions hold for $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$. Sort the inequality constraints into inactive ones ($h_k(\boldsymbol{x}) < 0$, $\alpha_k = 0$) and active ones ($h_k(\boldsymbol{x}) = 0$), and let k_a run over the latter only. Let $\boldsymbol{d} \ne \boldsymbol{0}$ be any feasible direction, meaning that $\boldsymbol{d}^T(\nabla_{\boldsymbol{x}} g_j) = 0$ for all j and $\boldsymbol{d}^T(\nabla_{\boldsymbol{x}} h_{k_a}) \le 0$ for all active k_a. Then,

$$\begin{aligned} 0 &= \boldsymbol{d}^T \left(\nabla_{\boldsymbol{x}} f + \sum\nolimits_j \lambda_j \nabla_{\boldsymbol{x}} g_j + \sum\nolimits_k \alpha_k \nabla_{\boldsymbol{x}} h_k \right) \\ &= \boldsymbol{d}^T \left(\nabla_{\boldsymbol{x}} f + \sum\nolimits_j \lambda_j \nabla_{\boldsymbol{x}} g_j + \sum\nolimits_{k_a} \alpha_{k_a} \nabla_{\boldsymbol{x}} h_{k_a} \right) \\ &= \boldsymbol{d}^T(\nabla_{\boldsymbol{x}} f) + \sum\nolimits_{k_a} \underbrace{\alpha_{k_a}}_{\ge 0} \underbrace{\boldsymbol{d}^T(\nabla_{\boldsymbol{x}} h_{k_a})}_{\le 0} \quad \Rightarrow \quad \boldsymbol{d}^T(\nabla_{\boldsymbol{x}} f) \ge 0. \end{aligned}$$

It works! If $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ fulfils our conditions and $\boldsymbol{d}$ is a feasible direction, moving along $\boldsymbol{d}$ does not lead to descent at least to first order. We can write the condition in a nicer way by pulling $\nabla_{\boldsymbol{x}}$ outside. Let us introduce the *Lagrangian*

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_j \lambda_j g_j(\boldsymbol{x}) + \sum_k \alpha_k h_k(\boldsymbol{x}), \quad \boldsymbol{\alpha} \succeq \boldsymbol{0}.$$

Here, $\boldsymbol{\alpha} \succeq \boldsymbol{0}$ is short for $\alpha_k \ge 0$ for all k. The Lagrangian is a function not only of $\boldsymbol{x}$, but also of $\boldsymbol{\lambda}$, $\boldsymbol{\alpha}$. These additional variables are called *Lagrange multipliers*. The complete optimality conditions for $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ read

$$\begin{aligned} &\frac{\partial L}{\partial \boldsymbol{x}} = \boldsymbol{0}, \qquad \frac{\partial L}{\partial \lambda_j} = 0 \quad \forall j, \\ &\alpha_k \ge 0, \;\; \frac{\partial L}{\partial \alpha_k} \le 0, \;\; \alpha_k \frac{\partial L}{\partial \alpha_k} = 0 \quad \forall k. \end{aligned} \tag{A.2}$$

Here, we used that $\partial L / \partial \lambda_j = g_j(\boldsymbol{x})$ and $\partial L / \partial \alpha_k = h_k(\boldsymbol{x})$. Note how all conditions are expressed in terms of derivatives of the Lagrangian. A worthy replacement for $\nabla_{\boldsymbol{x}} f = \boldsymbol{0}$ indeed. A brief way of describing (A.2) is that $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ constitutes a stationary point of the Lagrangian.

Lagrange Duality

We found that in order to lift the optimality condition $\nabla_{\boldsymbol{x}} f = \mathbf{0}$ to the constrained case, we require additional Lagrange multipliers $\boldsymbol{\lambda}$, $\boldsymbol{\alpha}$ as well as an extended criterion $L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$. Can this Lagrangian do more for us? Recall the primal problem (A.1). We can express it in terms of the Lagrangian:

$$\tilde{p}_* = \min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}, (\boldsymbol{\alpha} \succeq \mathbf{0})} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}). \tag{A.3}$$

Note that the minimization over $\boldsymbol{x}$ is unconstrained, much in contrast to the constrained minimization in (A.1). Let us do the inner maximization for a $\boldsymbol{x}$. Either $\boldsymbol{x}$ is feasible or not. In the former case, $g_j(\boldsymbol{x}) = 0$ for all j, and we can pick any $\boldsymbol{\lambda}$. Also, $h_k(\boldsymbol{x}) \le 0$, so that $\alpha_k h_k(\boldsymbol{x}) \le 0$, since $\alpha_k \ge 0$. The best we can do is to set $\alpha_k = 0$. For a feasible $\boldsymbol{x}$, the inner maximization gives $f(\boldsymbol{x})$. Now, suppose $\boldsymbol{x}$ is not feasible. Then, at least one constraint must be violated. We show that the inner maximum[2] is $+\infty$ in this case. If $g_j(\boldsymbol{x}) \neq 0$, we can send $\lambda_j \to \operatorname{sgn}(g_j(\boldsymbol{x}))\infty$. If $h_k(\boldsymbol{x}) > 0$, we can send $\alpha_k \to \infty$. Therefore,

$$\max_{\boldsymbol{\lambda}, (\boldsymbol{\alpha} \succeq \mathbf{0})} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \left\{ \begin{array}{ll} f(\boldsymbol{x}) & \mid \boldsymbol{x} \text{ feasible} \\ +\infty & \mid \boldsymbol{x} \text{ not feasible} \end{array} \right\},$$

whose minimum over $\boldsymbol{x}$ is precisely the primal problem (A.1). What does that mean? The Lagrangian may have stationary points of many kinds, but as our goal is to solve the primal problem, we should be only interested in its *saddlepoints* of the type (A.3).

These are not the only saddlepoints, we might just as well look at

$$\tilde{d}_* = \max_{\boldsymbol{\lambda}, (\boldsymbol{\alpha} \succeq \mathbf{0})} \min_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}). \tag{A.4}$$

Here, the unconstrained minimization over $\boldsymbol{x}$ is inside, the maximization over Lagrange multipliers is outside. This is called the *dual problem.* How does it relate to the primal problem (A.3)? First of all, we always have

$$\tilde{d}_* \le \tilde{p}_*.$$

Namely, for each fixed $\boldsymbol{\lambda}$, $\boldsymbol{\alpha} \succeq \mathbf{0}$:

$$\min_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \le \min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}', (\boldsymbol{\alpha}' \succeq \mathbf{0})} L(\boldsymbol{x}, \boldsymbol{\lambda}', \boldsymbol{\alpha}').$$

The dual value $\tilde{d}_*$ bounds the primal value $\tilde{p}_*$ from below, a fact which is called *weak duality.* Why is this useful? It turns out that in many situations, the dual problem is easier to solve than the primal problem. For example, it is easily confirmed that (A.4) is a concave[3] maximization problem, no matter what the primal problem is. Even for very hard primal problems, the dual (A.4) can often be solved easily and provides a lower bound $\tilde{d}_*$ on the primal value $\tilde{p}_*$. This technique is used frequently in theoretical computer science. Moreover,

[2] We explicitly allow minima to be $-\infty$ and maxima to be $+\infty$, with the understanding that $-\infty < v < +\infty$ for all $v \in \mathbb{R}$.

[3] Namely, $\Phi_D(\boldsymbol{\lambda}, \boldsymbol{\alpha}) = \min_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ is a minimum of affine linear functions, therefore concave.

there may be far fewer dual variables $(\boldsymbol{\lambda}, \boldsymbol{\alpha})$ than primal ones $(\boldsymbol{x})$, which is what happens for the soft margin SVM problem (Section 9.3).

Let us look at an example, illustrated in Figure A.2. The primal problem is

$$\min_x \frac{1}{2}x^2, \quad \text{subj. to } x \leq -1.$$

The primal value is $\tilde{p}_* = 1/2$, attained at $x_* = -1$. The Lagrangian is

$$L(x, \alpha) = \frac{1}{2}x^2 + \alpha(x+1), \quad \alpha \geq 0.$$

For any $\alpha \geq 0$:

$$\frac{\partial L}{\partial x} = x + \alpha,$$

so that $x_*(\alpha) = -\alpha$ minimizes $L(x, \alpha)$. The dual function is

$$g(\alpha) = L(x_*(\alpha), \alpha) = \alpha - \frac{1}{2}\alpha^2.$$

Its maximum is $\tilde{d}_* = 1/2$, attained at $\alpha_* = 1$, and $x_*(-1) = -1$ is the solution to the primal problem.

We are almost there now, but need one more step. It is fine to know that the dual may be easier to solve, but after all we want to solve the primal. Now, under additional conditions on the optimization problem (A.1), we can ensure that

$$\tilde{d}_* = \tilde{p}_*,$$

meaning that primal and dual are simply two different formulations of the same problem. This property is called *strong duality.* The condition we need for (A.1) is that $f(\boldsymbol{x})$ is a convex function. In short, for any primal problem (A.1) with linear constraints and convex objective, strong duality holds. The primal and dual values are the same, and their respective optimal points agree:

$$\max_{\boldsymbol{\lambda}, (\boldsymbol{\alpha} \succeq \mathbf{0})} L(\boldsymbol{x}_*, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = L(\boldsymbol{x}_*, \boldsymbol{\lambda}_*, \boldsymbol{\alpha}_*) = \min_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}_*, \boldsymbol{\alpha}_*).$$

A proof of this statement can be found in [3, ch. 3.4]. The practical meaning of this exercise is as follows. If we know that strong duality holds, we can solve our primal problem as follows. First, we solve the dual problem, giving rise to $\boldsymbol{\lambda}_*$, $\boldsymbol{\alpha}_* \succeq \mathbf{0}$, and the value $\tilde{d}_*$. Then, $\boldsymbol{x}_* = \operatorname{argmin}_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}_*, \boldsymbol{\alpha}_*)$ is an optimal point for the primal problem, whose value is $\tilde{p}_* = \tilde{d}_*$. Even better, some modern primal-dual optimizers use primal and dual problems in an interleaved fashion, the gap between their current values is used to monitor progress.

A.1 Soft Margin SVM Revisited

In this section, we illustrate Lagrange duality by rederiving the soft margin SVM dual problem previously obtained in Section 9.3 by a specific route. Recall

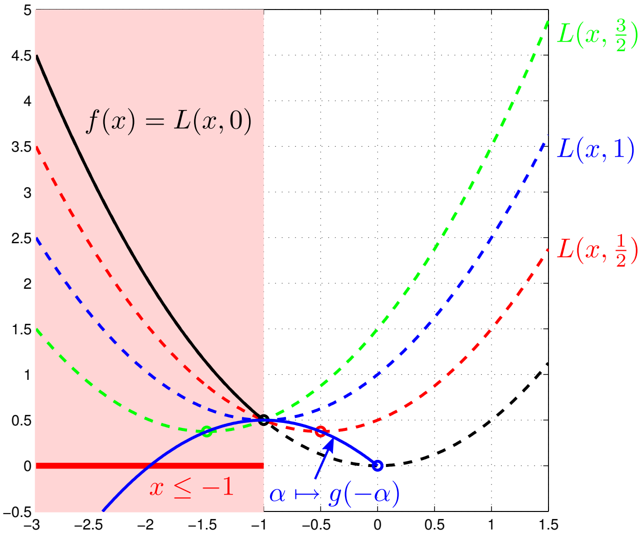


Figure A.2: Example for primal and dual function for a simple QP. The primal criterion is $f(x) = x^2/2$, subject to $x \leq -1$. Shown are Lagrangian curves $L(x, \alpha)$ for various values of α (dashed), as well as their minimum points $x_*(\alpha)$ (circles). Note that $L(x, 0)$ coincides with $f(x)$. The largest $x_*(\alpha)$ is attained at $\alpha_* = 1$.
Since $x_*(\alpha) = -\alpha$, we can plot the "inversion" of the dual $g(\alpha)$, namely $\alpha \mapsto g(-\alpha)$, in the same figure. Note how $g(\alpha) \leq f(x)$ in respective feasible regions $x \leq -1$ and $\alpha \geq 0$, and that they coincide at the saddle point $x_* = -1$, $\alpha_* = 1$.

the notation from there. The Lagrangian is

$$L = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_j \xi_j + \sum_i \alpha_i(1 - t_i y_i - \xi_i) - \sum_j \nu_j \xi_j$$
$$= \frac{1}{2}\|\boldsymbol{w}\|^2 + \boldsymbol{\alpha}^T(\mathbf{1} - \boldsymbol{T}\boldsymbol{y}) + \boldsymbol{\xi}^T(C\mathbf{1} - \boldsymbol{\alpha} - \boldsymbol{\nu}).$$

Make sure to understand the vectorization before moving on. The primal variables are $(\boldsymbol{w}, b, \boldsymbol{\xi})$, the dual variables are $\boldsymbol{\alpha} \succeq \mathbf{0}$ and $\boldsymbol{\nu} \succeq \mathbf{0}$, both in $\mathbb{R}^n$. The criterion of the dual problem is

$$\Phi_D(\boldsymbol{\alpha}, \boldsymbol{\nu}) = \min_{\boldsymbol{w}, b, \boldsymbol{\xi}} L.$$

The inner minimization w.r.t. $\boldsymbol{w}$ and b works in the same way as in Section 9.3. Setting the gradient equal to zero results in

$$\boldsymbol{w} = \sum_{i=1}^n \alpha_i t_i \boldsymbol{\phi}(\boldsymbol{x}_i), \quad \boldsymbol{\alpha}^T\boldsymbol{t} = \sum_{i=1}^n \alpha_i t_i = 0$$

Finally,

$$\nabla_{\boldsymbol{\xi}} L = C\mathbf{1} - \boldsymbol{\alpha} - \boldsymbol{\nu} = \mathbf{0} \quad \Rightarrow \quad \boldsymbol{\nu} = C\mathbf{1} - \boldsymbol{\alpha}.$$

This allows us to eliminate the dual variables $\boldsymbol{\nu}$. But careful, $\boldsymbol{\nu} \succeq \mathbf{0}$ implies additional constraints $\boldsymbol{\alpha} \preceq C\mathbf{1}$. Plugging all of this in, we obtain the dual problem (9.7), as well as the kernel expansion (9.8).

Next, we can use the Lagrange optimality conditions (also sometimes called Karush-Kuhn-Tucker conditions) in order to reproduce the classification of patterns $(\boldsymbol{x}_i, t_i)$ derived in Section 9.3. Suppose we are at a saddlepoint, dropping the "*" subscripts. The optimality conditions are $\alpha_i \in [0, C]$, $\nu_i \geq 0$, $\xi_i \geq 0$, and

$$(1 - t_i y_i - \xi_i)\alpha_i = 0, \qquad \nu_i \xi_i = 0 = (C - \alpha_i)\xi_i.$$

Three different things can happen:

- $\alpha_i = 0$, so that $\xi_i = 0$ (no slack) and $1 - t_i y_i \leq 0$. Not a support vector.
- $\alpha_i \in (0, C)$, so that $\xi_i = 0$ and $1 - t_i y_i = 0$. An essential support vector.
- $\alpha_i = C$, so that $\xi_i \geq 0$ and $1 - t_i y_i - \xi_i = 0$, which implies $1 - t_i y_i \geq 0$. A bound support vector.

Bibliography

[1] D. Barber. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 1st edition, 2012.

[2] P. Bartlett and A. Tewari. Sparseness vs estimating conditional probabilities: Some asymptotic results. In *Conference on Computational Learning Theory 17*, pages 564–578. Springer, 2004.

[3] D. Bertsekas. *Nonlinear Programming.* Athena Scientific, 2nd edition, 1999.

[4] P. Billingsley. *Probability and Measure.* John Wiley & Sons, 3rd edition, 1995.

[5] C. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1st edition, 1995.

[6] C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 1st edition, 2006.

[7] L. Bottou. Online learning and stochastic approximations. In D. Saad, editor, *On-Line Learning in Neural Networks.* Cambridge University Press, 1998.

[8] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[9] K. L. Chung. *A Course in Probability Theory.* Academic Press, 2nd edition, 1974.

[10] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[11] Thomas Cover and Joy Thomas. *Elements of Information Theory.* John Wiley & Sons, 1st edition, 1991.

[12] L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition.* Applications of Mathematics: Stochastic Modelling and Applied Probability. Springer, 1st edition, 1996.

[13] R. Duda, P. Hart, and D. Stork. *Pattern Classification.* John Wiley & Sons, 2nd edition, 2000.

[14] William H. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, 3rd edition, 1968.

[15] William H. Feller. *An Introduction to Probability Theory and its Applications*, volume 2. John Wiley & Sons, 2nd edition, 1971.

[16] Roger Fletcher. *Practical Methods of Optimization: Unconstrained Optimization*, volume 1. John Wiley & Sons, 1980.

[17] P. Gill, W. Murray, and M. Wright. *Practical Optimization.* Academic Press, 1981.

[18] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3rd edition, 1996.

[19] G. Grimmett and D. Stirzaker. *Probability and Random Processes.* Oxford University Press, 3rd edition, 2001.

[20] C. Grinstead and J. Snell. *Introduction to Probability.* American Mathematical Society, 2nd edition, 1997.

[21] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2nd edition, 2009.

[22] David Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, July 1999. See `http://www.cse.ucsc.edu/~haussler/pubs.html`.

[23] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.

[24] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:1771–1800, 2002.

[25] R. Horn and C. Johnson. *Matrix Analysis.* Cambridge University Press, 1st edition, 1985.

[26] Tommi Jaakkola, Marina Meila, and Tony Jebara. Maximum entropy discrimination. In Solla et al. [43], pages 470–476.

[27] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques.* MIT Press, 1st edition, 2009.

[28] N. D. Lawrence, M. Seeger, and R. Herbrich. Fast sparse Gaussian process methods: The informative vector machine. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 609–616. MIT Press, 2003.

[29] D. Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley, 2nd edition, 1984.

[30] D. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

[31] K. Murphy. *Machine Learning: A Probabilistic Perspective.* Adaptive Computation and Machine Learning. MIT Press, 1st edition, 2012.

[32] I. Nabney. *Netlab: Algorithms for Pattern Recognition.* Advances in Pattern Recognition. Springer, 1st edition, 2001.

[33] C. Paige and M. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, 1982.

[34] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, 1988.

[35] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods: Support Vector Learning*, pages 185–208. MIT Press, 1998.

[36] J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In A. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers.* MIT Press, 1999.

[37] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning.* MIT Press, 2006.

[38] B. D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, 1996.

[39] B. Schölkopf and A. Smola. *Learning with Kernels.* MIT Press, 1st edition, 2002.

[40] M. Seeger. Bayesian model selection for support vector machines, Gaussian processes and other kernel classifiers. In Solla et al. [43], pages 603–609.

[41] M. Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems*, 14(2):69–106, 2004.

[42] G. Simmons. *Calculus with Analytic Geometry.* McGraw-Hill, 2nd edition, 1996.

[43] S. Solla, T. Leen, and K.-R. Müller, editors. *Advances in Neural Information Processing Systems 12.* MIT Press, 2000.

[44] I. Steinwart and A. Christmann. *Support Vector Machines.* Springer, 1st edition, 2008.

[45] G. Strang. *Introduction to Linear Algebra.* Wellesley – Cambridge Press, 4th edition, 2009.

[46] M. Tipping and C. Bishop. Probabilistic principal component analysis. *Journal of Roy. Stat. Soc. B*, 61(3):611–622, 1999.

[47] C. Williams. Computation with infinite neural networks. *Neural Computation*, 10(5):1203–1216, 1998.

[48] Christopher K. I. Williams. Prediction with Gaussian processes: From linear regression to linear prediction and beyond. In M. I. Jordan, editor, *Learning in Graphical Models.* Kluwer, 1997.