Supervised Machine Learning

Lecture notes for the Statistical Machine Learning course

Andreas Lindholm, Niklas Wahlström,
Fredrik Lindsten, Thomas B. Schön

Version: March 12, 2019

Department of Information Technology, Uppsala University

0.1 About these lecture notes

These lecture notes are written for the course Statistical Machine Learning 1RT700, given at the Department of Information Technology, Uppsala University, spring semester 2019. They will eventually be turned into a textbook, and we are very interested in all type of comments from you, our dear reader. Please send your comments to andreas.lindholm@it.uu.se. Everyone who contributes with many useful comments will get a free copy of the book.

During the course, updated versions of these lecture notes will be released. The major changes are noted below in the changelog:

Date	Comments
2019-01-18	Initial version. Chapter 6 missing.
2019-01-23	Typos corrected, mainly in Chapter 2 and 5. Section 2.6.3 added.
2019-01-28	Typos corrected, mainly in Chapter 3.
2019-02-07	Chapter 6 added.
2019-03-04	Typos corrected.
2019-03-11	Typos (incl. eq. (3.25)) corrected.
2019-03-12	Typos corrected.

Contents

1 Introduction

1.1 What is machine learning all about?

Machine learning gives computers the ability to learn without being explicitly programmed for the task at hand. The *learning* happens when data is combined with mathematical models, for example by finding suitable values of unknown variables in the model. The most basic example of learning could be that of fitting a straight line to data, but machine learning usually deals with much more flexible models than straight lines. The point of doing this is that the result can be used to draw conclusions about new data, that was not used in learning the model. If we learn a model from a data set of 1000 puppy images, the model might — if it is wisely chosen — be able to tell whether another image (not among the 1000 used for learning) depicts a puppy or not. That is know as *generalization*.

The science of machine learning is about learning models that generalize well.

These lecture notes are exclusively about *supervised learning*, which refers to the problem where the data is on the form $\{\mathbf{x}_i, y_i\}_{i=1}^n$, where $\mathbf{x}_i$ denotes *inputs*[1] and y_i denotes *outputs*[2]. In other words, in supervised learning we have *labeled data* in the sense that each data point has an input $\mathbf{x}_i$ *and an output* y_i which explicitly explains "what we see in the data". For example, to check for signs of heart disease medical doctors makes use of a so-called electrocardiogram (ECG) which is a test that measures the electrical activity of the heart via electrodes placed on the skin of the patients chest, arms and legs. Based on these readings a skilled medical doctor can then make a diagnosis. In this example the ECG measurements constitutes the input $\mathbf{x}$ and the diagnosis provided by the medical doctor constitutes the output y. If we have access to a large enough pool of *labeled data* of this kind (where we both have the ECG reading $\mathbf{x}$ *and* the diagnosis y) we can use supervised machine learning to learn a model for the relationship between $\mathbf{x}$ and y. Once the model is learned, is can be used to diagnose *new* ECG readings, for which we do not (yet) know the diagnosis y. This is called a prediction, and we use $\widehat{y}$ to denote it. If the model is making good predictions (close to the true y) also for ECGs not in the training data, we have a model which generalizes well.

One of the most challenging problems with supervised learning is that it requires labeled data, i.e. both the inputs and the corresponding outputs $\{\mathbf{x}_i, y_i\}_{i=1}^n$. This is challenging because the process of labeling data is often expensive and sometimes also difficult or even impossible since it requires humans to interpret the input and provide the correct output. The situation is made even worse due to the fact that most of the state-of-the-art methods require a lot of data to perform well. This situation has motivated the development of *unsupervised learning* methods which only require the input data $\{\mathbf{x}_i\}_{i=}^n$, i.e. so-called *unlabeled* data. An important subproblem is that of clustering, where data is automatically organized into different groups based on some notion of similarity. There is also an increasingly important middle ground referred to as *semi-supervised learning*, where we make use of both labeled and unlabeled data. The reason being that we often have access to a lot of unlabeled data, but only a small amount of labeled data. However, this small amount of labeled data might still prove highly valuable when used together with the much larger set of unlabeled data.

In the area of *reinforcement learning*, another branch of machine learning, we do not only want to make use of measured data in order to be able to predict something or understand a given situation, but instead

[1] Some common synonyms used for the input variable include feature, predictor, regressor, covariate, explanatory variable, controlled variable and independent variable.

[2] Some common synonyms used for the output variable include response, regressand, label, explained variable, predicted variable and dependent variable.

we want to develop a system that can *learn how to take actions* in the real world. The most common approach is to learn these actions by trying to maximize some kind of reward encouraging the desired state of the environment. The area of reinforcement learning has very strong ties to control theory. Finally we mention the emerging area of *causal learning* where the aim is to tackle the much harder problem of learning cause and effect relationships. This is very different from the other facets of machine learning briefly introduced above, where it was sufficient to learn associations/correlations between the data. In causal learning the aim is to move beyond learning correlations and instead trying to learn causal relations.

1.2 Regression and classification

A useful categorization of supervised machine learning algorithms is obtained by differentiating with respect to the type—quantitative or a qualitative—of output variable involved in the problem. Let us first have a look at when a variable in general is to be considered as quantitative or qualitative, respectively. See Table 1.1 for a few examples.

Table 1.1: Examples of quantitative and qualitative variables.

Variable type	**Example**	**Handle as**
Numeric (continuous)	32.23 km/h, 12.50 km/h, 42.85 km/h	Quantitative
Numeric (discrete) with natural ordering	0 children, 1 child, 2 children	Quantitative
Numeric (discrete) without natural ordering	1 = Sweden, 2 = Denmark, 3 = Norway	Qualitative
Text (not numeric)	Uppsala University, KTH, Lund University	Qualitative

Depending on whether the output of a problem is quantitative or qualitative , we refer to the problem as either regression or classification.

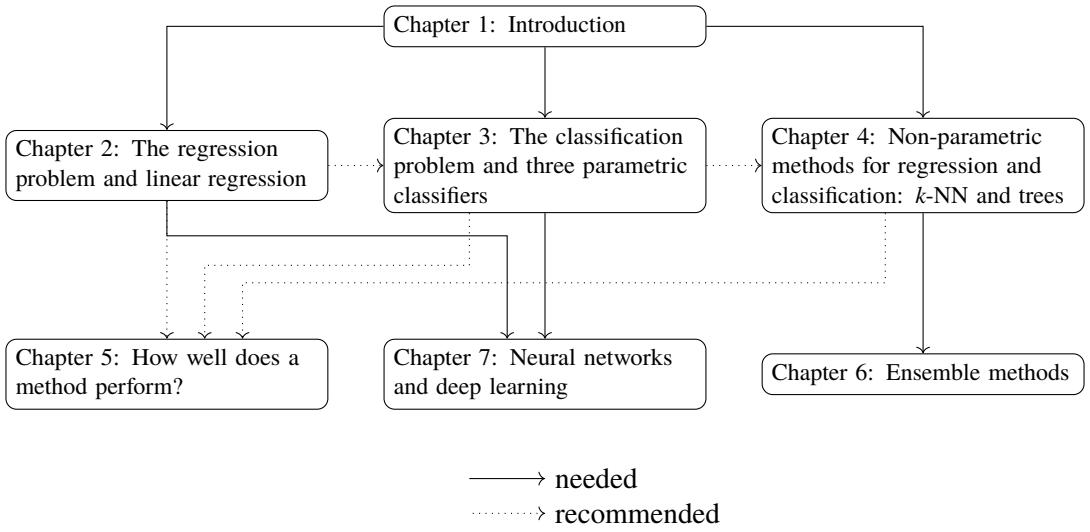
Regression means the output is quantitative, and *classification* means the output is qualitative.

This means that whether a problem is about regression or classification depends only on its output. The input can be either quantitative or qualitative in both cases.

The distinction between quantitative and qualitative, and thereby between regression and classification, is however somewhat arbitrary, and there is not always a clear answer: one could for instance argue that having no children is something qualitatively different than having children, and use the qualitative output “children: yes/no”, instead of “0, 1 or 2 children”, and thereby turn a regression problem into a classification problem, for example.

1.3 Overview of these lecture notes

The following sketch gives an idea on how the chapters are connected.



1.4 Further reading

There are by now quite a few extensive textbooks available on the topic of machine learning which introduce the area in slightly different ways compared to what we do in this book. The book of Hastie, Tibshirani, and Friedman 2009 introduce the area of statistical machine learning in a mathematically solid and accessible manner. A few years later the authors released a new version of their book which is mathematically significantly lighter (James et al. 2013). They still do a very nice work of conveying the main ideas. These books do not venture long into the world of Bayesian methods. However, there are several complementary books doing a good job at covering the Bayesian methods as well, see e.g. (Barber 2012; Bishop 2006; Murphy 2012). MacKay (2003) provided a rather early account drawing interesting and useful connections to information theory. It is still very much worth looking into. Finally, we mention the work of Efron and Hastie 2016, where the authors takes a constructive historical approach to the development of this new area covering the revolution in data analysis that emerged with the computers. A contemporary introduction to the mathematics of machine learning is provided by (Deisenroth, Faisal, and Ong 2019). Two relatively recent papers introducing the area are available here Ghahramani 2015; Jordan and Mitchell 2015.

The scientific field of Machine Learning is extremely vibrant and active at the moment. The two leading conferences within the area are *The International Conference on Machine Learning (ICML)* and the *The Conference on Neural Information Processing Systems (NeurIPS)*. Both are held on a yearly basis and all the new research presented at these two conferences are freely available via their websites (`icml.cc` and `neurips.cc`). Two additional conferences in the area are *The International Conference on Artificial Intelligence and Statistics (AISTATS)* and *The International Conference on Learning Representations (ICLR)*. The leading journals in the area are the *Journal of Machine Learning Research (JMLR)* and the *IEEE Transactions on Pattern Analysis and Machine Intelligence*. There are also quite a lot of relevant work published within statistical journals, in particular within the area of computational statistics.

2 The regression problem and linear regression

The first problem we will study is the *regression problem*. Regression is one of the two main problems that we cover in these notes. (The other one is classification). The first method we will encounter is *linear regression*, which is one (of many) solutions to the regression problem. Even though the relative simplicity of linear regression, it is surprisingly useful and it also constitutes an important building block for more advanced methods (such as deep learning, Chapter 7).

2.1 The regression problem

Regression refers to the problem of learning the relationships between some (qualitative or quantitative[1]) input variables $\mathbf{x} = [x_1\ x_2\ \ldots\ x_p]^\mathsf{T}$ and a quantitative output variable y. In mathematical terms, regression is about learning a model f

$$y = f(\mathbf{x}) + \varepsilon, \tag{2.1}$$

where ε is a noise/error term which describes everything that cannot be captured by the model. With our statistical perspective, we consider ε to be a random variable that is independent of $\mathbf{x}$ and has a mean value of zero.

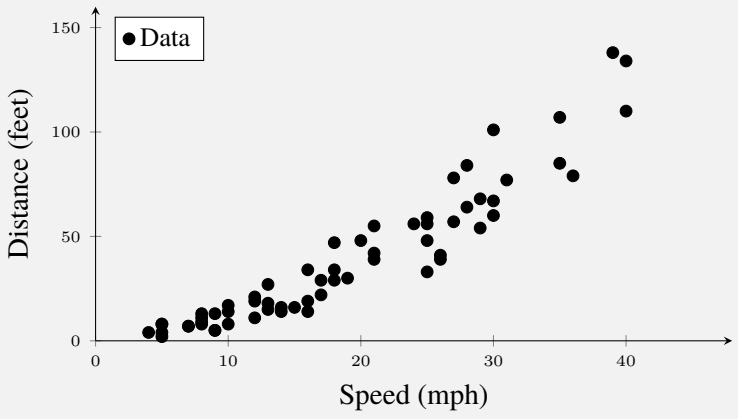
Throughout this chapter, we will use the dataset introduced in Example 2.1 with car stopping distances to illustrate regression. In a sentence, the problem is to learn a regression model which can tell what distance is needed for a car to come to a full stop, given its current speed.

Example 2.1: Car stopping distances

Ezekiel and Fox (1959) presents a dataset with 62 observations of the distance needed for various cars at different initial speeds to break to a full stop.[a] The dataset has the two following variables:

- `Speed`: The speed of the car when the break signal is given.
- `Distance`: The distance traveled after the signal is given until the car has reached a full stop.

We decide to interpret `Speed` as the **input variable** x, and `Distance` as the **output variable** y.



Our goal is to use linear regression to estimate (that is, to *predict*) how long the stopping distance would be if the initial speed would be 33 mph or 45 mph (two speeds at which no data has been recorded).

[a] The dataset is somewhat dated, so the conclusions are perhaps not applicable to modern cars. We believe, however, that the reader is capable of pretending that the data comes from her/his own favorite example instead.

[1] We will start with quantitative input variables, and discuss qualitative input variables later in 2.5.

2.2 The linear regression model

The linear regression model describes the output variable y (a scalar) as an affine combination of the input variables $x_1, x_2, \ldots, x_p$ (each a scalar) plus a noise term ε,

$$y = \underbrace{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p}_{f(\mathbf{x};\boldsymbol{\beta})} + \varepsilon. \tag{2.2}$$

We refer to the coefficients $\beta_0, \beta_1, \ldots \beta_p$ as the *parameters* in the model, and we sometimes refer to β_0 specifically as the intercept term. The noise term ε accounts for non-systematic, i.e., random, errors between the data and the model. The noise is assumed to have mean zero and to be independent of $\mathbf{x}$. Machine learning is about training, or learning, models from data. Hence, the main part of this chapter will be devoted to how to learn the parameters $\beta_0, \beta_1, \ldots, \beta_p$ from some training dataset $\mathcal{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. Before we dig into the details in Section 2.3, let us just briefly start by discussing the purpose of using linear regression. The linear regression model can namely be used for, at least, two different purposes: to *describe* relationships in the data by interpreting the parameters $\boldsymbol{\beta} = [\beta_0\ \beta_1\ \ldots\ \beta_p]^\mathsf{T}$, and to *predict* future outputs for inputs that we have not yet seen.

Remark 2.1 *It is possible to formulate the model also for multiple outputs $y_1, y_2, \ldots$, see the exercises. This is commonly referred to as multivariate linear regression.*

2.2.1 Describe relationships — classical statistics

An often posed question in sciences such as medicine and sociology, is to determine whether there is a correlation between some variables or not ('do you live longer if you only eat sea food?', etc.). Such questions can be addressed by studying the parameters $\boldsymbol{\beta}$ in the linear regression model, after the parameter has been learned from data. The most common question is perhaps whether it can be indicated that *some* correlation between two variables x_1 and y exists, which can be done with the following reasoning: If $\beta_1 = 0$, it would indicate that there is no correlation between y and x_1 (unless the other inputs also depend on x_1). By estimating β_1 together with a confidence interval (describing the uncertainty of the estimate), one can rule out (with a certain significance level) that x_1 and y are uncorrelated if 0 is not contained in the confidence interval for β_1. The conclusion is then instead that *some* correlation is likely to be present between x_1 and y. This type of reasoning is referred to as *hypothesis testing* and it constitutes an important branch of classical statistics. However, we shall mainly be concerned with another purpose of the linear regression model, namely to make predictions.

2.2.2 Predicting future outputs — machine learning

In machine learning, the emphasis is rather on predicting some (not yet seen) output $\widehat{y}_\star$ for some new input $\mathbf{x}_\star = [x_{\star 1}\ x_{\star 2}\ \ldots\ x_{\star p}]^\mathsf{T}$. To make a prediction for a test input $\mathbf{x}_\star$, we insert it into the model (2.2). Since ε (by assumption) has mean value zero, we take the *prediction* as

$$\widehat{y}_\star = \beta_0 + \beta_1 x_{\star 1} + \beta_2 x_{\star 2} + \cdots + \beta_p x_{\star p}. \tag{2.3}$$

We use the symbol $\widehat{\ }$ on $y_\star$ to indicate that it is a prediction, our best guess. If we were able to somehow observe the actual output from $\mathbf{x}_\star$, we would denote it by $y_\star$ (without a hat).

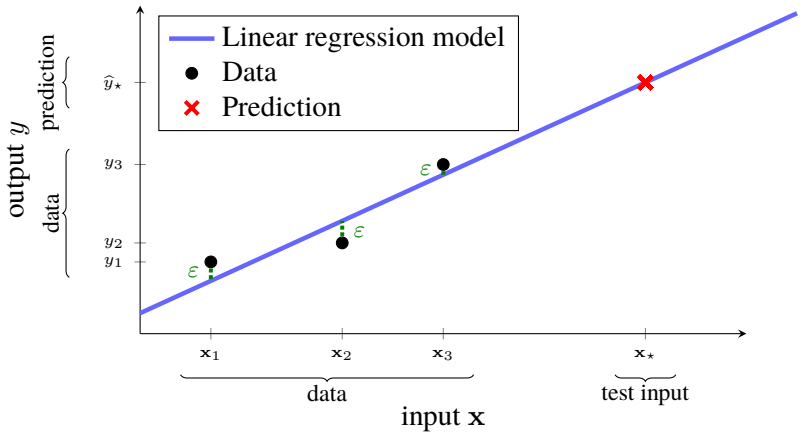


Figure 2.1: Linear regression with $p = 1$: The black dots represent $n = 3$ data points, from which a linear regression model (blue line) is learned. The model does not fit the data perfectly, but there is a remaining error/noise ε (green). The model can be used to *predict* (red cross) the output $\widehat{y}_\star$ for a test input point $\mathbf{x}_\star$.

2.3 Learning the model from training data

To use the linear regression model, we first need to learn the unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$ from a training dataset $\mathcal{T}$. The training data consists of n samples of the output variable y, we call them y_i ($i = 1, \ldots, n$), and the corresponding n samples $\mathbf{x}_i$ ($i = 1, \ldots, n$) (each a column vector). We write the dataset in the matrix form

$$\mathbf{X} = \begin{bmatrix} 1 & -\mathbf{x}_1^\mathsf{T}- \\ 1 & -\mathbf{x}_2^\mathsf{T}- \\ \vdots & \vdots \\ 1 & -\mathbf{x}_n^\mathsf{T}- \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \text{where each } \mathbf{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix}. \tag{2.4}$$

Note that $\mathbf{X}$ is a $n \times (p+1)$ matrix, and $\mathbf{y}$ a n-dimensional vector. The first column of $\mathbf{X}$, with only ones, corresponds to the intercept term β_0 in the linear regression model (2.2). If we also stack the unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$ into a $(p+1)$ vector

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \tag{2.5}$$

we can express the linear regression model as a matrix multiplication

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{2.6}$$

where $\boldsymbol{\epsilon}$ is a vector of errors/noise.

Learning the unknown parameters $\boldsymbol{\beta}$ amounts to finding values such that *the model fits the data well*. There are multiple ways to define what 'well' actually means. We will take a statistical perspective and choose the value of $\boldsymbol{\beta}$ which makes the observed training data $\mathbf{y}$ as likely as possible under the model—the so-called *maximum likelihood* solution.

Example 2.2: Car stopping distances

We will continue Example 2.1, and form the matrices $\mathbf{X}$ and $\mathbf{y}$. Since we only have one input and one output, both x_i and y_i are scalar. We get,

$$\mathbf{X} = \begin{bmatrix} 1 & 4 \\ 1 & 5 \\ 1 & 5 \\ 1 & 5 \\ 1 & 5 \\ 1 & 7 \\ 1 & 7 \\ 1 & 8 \\ \vdots & \vdots \\ 1 & 39 \\ 1 & 39 \\ 1 & 40 \end{bmatrix}, \qquad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4 \\ 2 \\ 4 \\ 8 \\ 8 \\ 7 \\ 7 \\ 8 \\ \vdots \\ 138 \\ 110 \\ 134 \end{bmatrix}. \tag{2.7}$$

2.3.1 Maximum likelihood

Our strategy to learn the unknown parameters $\boldsymbol{\beta}$ from the training data $\mathcal{T}$ will be the *maximum likelihood* method. The word 'likelihood' refers to the statistical concept of the likelihood function, and maximizing the likelihood function amounts to finding the value of $\boldsymbol{\beta}$ that makes observing $\mathbf{y}$ as likely as possible. That is, we want to solve

$$\underset{\boldsymbol{\beta}}{\text{maximize}} \;\; p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\beta}), \tag{2.8}$$

where $p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\beta})$ is the probability density of the data $\mathbf{y}$ given a certain value of the parameters $\boldsymbol{\beta}$. We denote the solution to this problem—the learned parameters—with $\widehat{\boldsymbol{\beta}} = [\widehat{\beta}_0 \; \widehat{\beta}_1 \; \cdots \; \widehat{\beta}_p]^\mathsf{T}$. More compactly, we write this as

$$\widehat{\boldsymbol{\beta}} = \arg\max_{\boldsymbol{\beta}} p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\beta}). \tag{2.9}$$

In order to have a notion of what 'likely' means, and thereby specify $p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\beta})$ mathematically, we need to make assumptions about the noise term ε. A common assumption is that ε follows a Gaussian distribution with zero mean and variance σ_ε^2,

$$\varepsilon \sim \mathcal{N}\left(0, \sigma_\varepsilon^2\right). \tag{2.10}$$

This implies that the conditional probability density function of the output y for a given value of the input $\mathbf{x}$ is given by

$$p(y \,|\, \mathbf{x}, \boldsymbol{\beta}) = \mathcal{N}\left(y \,|\, \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p, \sigma_\varepsilon^2\right). \tag{2.11}$$

Furthermore, the n observed training data points are assumed to be *independent* realizations from this statistical model. This implies that the likelihood of the training data factorizes as

$$p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\beta}) = \prod_{i=1}^{n} p(y_i \,|\, \mathbf{x}_i, \boldsymbol{\beta}). \tag{2.12}$$

Putting (2.11) and (2.12) together we get

$$p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\beta}) = \frac{1}{(2\pi\sigma_\varepsilon^2)^{n/2}} \exp\left(-\frac{1}{2\sigma_\varepsilon^2} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} - y_i)^2\right). \tag{2.13}$$

Recall from (2.8) that we want to maximize the likelihood w.r.t. $\boldsymbol{\beta}$. However, since (2.13) only depends on $\boldsymbol{\beta}$ via the sum in the exponent, and since the exponential is a monotonically increasing function, maximizing (2.13) is equivalent to minimizing

$$\sum_{i=1}^{n} \left(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} - y_i\right)^2. \tag{2.14}$$

This is the sum of the squares of differences between each output data y_i and the model's prediction of that output, $\widehat{y}_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}$. For this reason, minimizing (2.14) is usually referred to as *least squares*.

We will come back on how the values $\widehat{\beta}_0, \widehat{\beta}_1, \ldots, \widehat{\beta}_p$ can be computed. Let us just first mention that it is also possible—and sometimes a very good idea—to assume that the distribution of ε is something else than a Gaussian distribution. One can, for instance, assume that ε instead has a Laplace distribution, which would yield the cost function

$$\sum_{i=1}^{n} |\beta_0 + \beta_1 x_{i1} + \ldots \beta_p x_{ip} - y_i|. \tag{2.15}$$

It contains the sum of the absolute values of all differences (rather than their squares). The major benefit with the Gaussian assumption (2.10) is that there is a closed-form solution available for $\widehat{\beta}_0, \widehat{\beta}_1, \ldots, \widehat{\beta}_p$, whereas other assumptions on ε usually require computationally more expensive methods.

Remark 2.2 *With the terminoloy we will introduce in the next chapter, we could refer to* (2.13) *as the likelihood function, which we will denote by* $\ell(\boldsymbol{\beta})$.

Remark 2.3 *It is not uncommon in the literature to skip the maximum likelihood motivation, and just state* (2.14) *as a (somewhat arbitrary) cost function for optimization.*

2.3.2 Least squares and the normal equations

By assuming that the noise/error ε has a Gaussian distribution as stated in (2.10), the maximum likelihood parameters $\widehat{\boldsymbol{\beta}}$ are the solution to the optimization problem (2.14). We illustrate this by Figure 2.2, and write the least squares problem using the compact matrix and vector notation (2.6) as

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \ \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2, \tag{2.16}$$

where $\|\cdot\|_2$ denotes the usual Euclidean vector norm, and $\|\cdot\|_2^2$ its square. From a linear algebra point of view, this can be seen as the problem of finding the closest (in an Euclidean sense) vector to $\mathbf{y}$ in the subspace of $\mathbb{R}^n$ spanned by the columns of $\mathbf{X}$. The solution to this problem is the orthogonal projection of $\mathbf{y}$ onto this subspace, and the corresponding $\widehat{\boldsymbol{\beta}}$ can be shown (Section 2.A) to fulfill

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{X}^\mathsf{T}\mathbf{y}. \tag{2.17}$$

Equation (2.17) is often referred to as the *normal equations*, and gives the solution to the least squares problem (2.14, 2.16). If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, which often is the case, $\widehat{\boldsymbol{\beta}}$ has the closed form

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{2.18}$$

The fact that this closed-form solution exists is important, and is perhaps the reason why least squares has become very popular and is widely used. As discussed, other assumptions on ε than Gaussianity leads to other problems than least squares, such as (2.15) (where no closed-form solution exists).

Time to reflect 2.1: *What does it mean in practice that* $\mathbf{X}^\mathsf{T}\mathbf{X}$ *is not invertible?*

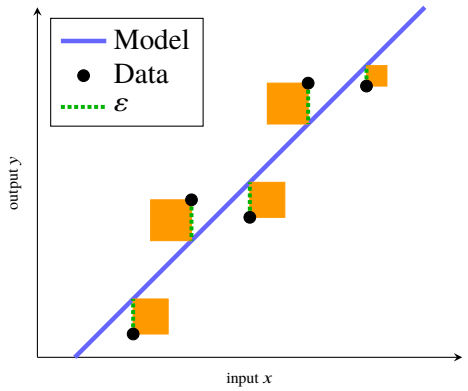
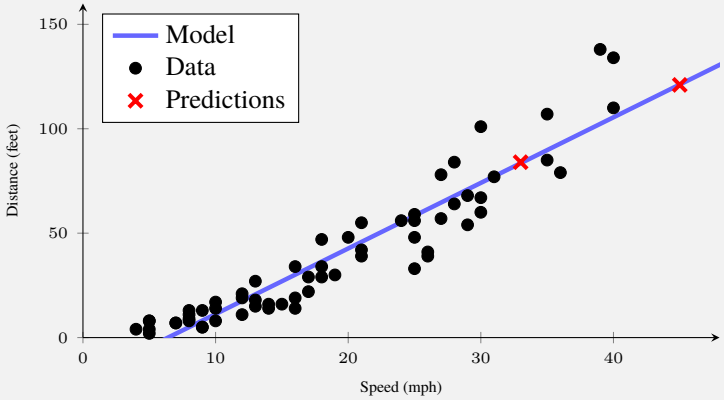


Figure 2.2: A graphical explanation of the least squares criterion: the goal is to choose the model (blue line) such that the sum of the square (orange) of each error ε (green) is minimized. That is, the blue line is to be chosen so that the amount of orange color is minimized. This motivates the name *least squares*.

Time to reflect 2.2: *If the columns of* $\mathbf{X}$ *are linearly independent and* $p = n - 1$, $\mathbf{X}$ *spans the entire* $\mathbb{R}^n$. *That means a unique solution exists such that* $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ *exactly, i.e., the model fits the training data perfectly. If that is the case,* (2.17) *reduces to* $\boldsymbol{\beta} = \mathbf{X}^{-1}\mathbf{y}$, *and the model fits the data perfectly. Why is that not desired?*

Example 2.3: Car stopping distances

By inserting the matrices (2.7) from Example 2.2 into the normal equations (2.6), we obtain $\widehat{\beta}_0 = -20.1$ and $\widehat{\beta}_1 = 3.1$. If we plot the resulting model, it looks like this:

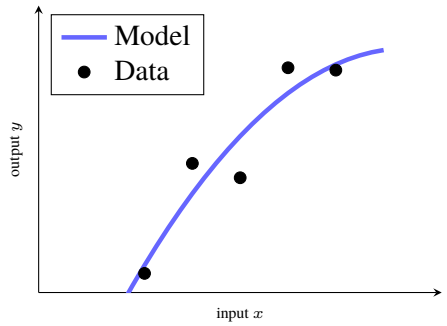


With this model, the predicted stopping distance for $\mathbf{x}_\star = 33$ mph is $\widehat{y}_\star = 84$ feet, and for $\mathbf{x}_\star = 45$ mph it is $\widehat{y}_\star = 121$ feet.

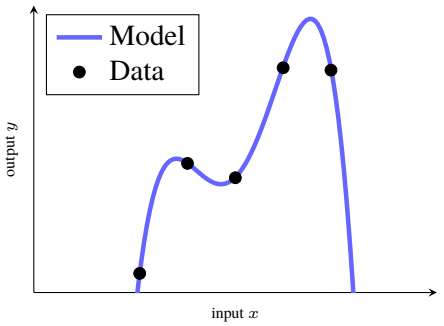
2.4 Nonlinear transformations of the inputs – creating more features

The reason for the word 'linear' in the name 'linear regression' is that the output is modelled as a *linear* combination of the inputs.[2] We have, however, not made a clear definition of what an input is: if the speed is an input, then why could not also the kinetic energy—it's square—be considered as another input? The answer is yes, it can. We can in fact make use of arbitrary *nonlinear* transformations of the "original" input variables as inputs in the linear regression model. If we, for example, only have a one-dimensional

[2] And also the constant 1, corresponding to the offset $\boldsymbol{\beta}_0$. For this reason, affine would perhaps be a better term than linear.



(a) The maximum likelihood solution with a 2nd order polynomial in the linear regression model. As discussed, the line is no longer straight (cf. Figure 2.1). This is, however, merely an artefact of the plot: in a three-dimensional plot with each feature (here, x and x^2) on a separate axis, it would still be an affine set.



(b) The maximum likelihood solution with a 4th order polynomial in the linear regression model. Note that a 4th order polynomial contains 5 unknown coefficients, which roughly means that we can expect the learned model to fit 5 data points exactly (cf. Remark 2.2, $p = n - 1$).

Figure 2.3: A linear regression model with 2nd and 4th order polynomials in the input x, as shown in (2.20).

input x, the vanilla linear regression model is

$$y = \beta_0 + \beta_1 x + \varepsilon. \tag{2.19}$$

However, we can also extend the model with, for instance, $x^2, x^3, \ldots, x^p$ as inputs, and thus obtain a linear regression model which is a polynomial in x,

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_p x^p + \varepsilon. \tag{2.20}$$

Note that this is still a linear regression model *since the unknown parameters appear in a linear fashion* with $x, x^2, \ldots, x^p$ as new inputs. The parameters $\widehat{\boldsymbol{\beta}}$ are still learned the same way, but the matrix $\mathbf{X}$ is different for model (2.19) and (2.20). We will refer to the transformed inputs as *features*. In more complicated settings the distinction between the original input and the transformed features might not be as clear, and the terms feature and input can sometimes be used interchangeably.

Time to reflect 2.3: *Figure 2.3 shows an example of two linear regression models with transformed (polynomial) inputs. When studying the figure one may ask how a* linear *regression model can result in a* curved *line? Are linear regression models not restricted to linear (or affine) straight lines? The answer is that it depends on the plot: Figure 2.3(a) shows a two-dimensional plot with x, y (the 'original' input), but a three-dimensional plot with x, x^2, y (each feature on a separate axis) would still be affine. The same holds true also for Figure 2.3(b) but in that case we would need a 5-dimensional plot.*

Even though the model in Figure 2.3(b) is able to fit all data points exactly, it also suggests that higher order polynomials might not always be very useful: the behavior of the model in-between and outside the data points is rather peculiar, and not very well motivated by the data. Higher-order polynomials are for this reason rarely used in practice in machine learning. An alternative and much more common feature is the so-called radial basis function (RBF) kernel

$$K_{\mathbf{c}}(\mathbf{x}) = \exp\left(-\frac{|\mathbf{x} - \mathbf{c}|_2^2}{\ell}\right), \tag{2.21}$$

i.e., a Gauss bell centered around $\mathbf{c}$. It can be used, instead of polynomials, in the linear regression model as

$$y = \beta_0 + \beta_1 K_{\mathbf{c}_1}(\mathbf{x}) + \beta_2 K_{\mathbf{c}_2}(\mathbf{x}) + \cdots + \beta_p K_{\mathbf{c}_p}(\mathbf{x}) + \varepsilon. \tag{2.22}$$

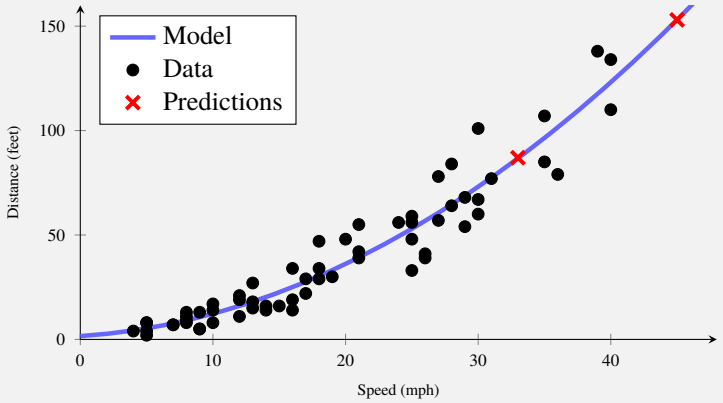
This model is can be seen as p 'bumps' located at $c_1, c_2, \ldots, c_p$, respectively. Note that the locations $c_1, c_2, \ldots, c_p$ as well as the length scale ℓ have to be decided by the user, and only the parameters $\beta_0, \beta_2, \ldots, \beta_p$ are learned from data in linear regression. This is illustrated in Figure 2.4. RBF kernels are in general preferred over polynomials since they have 'local' properties, meaning that a small change in one parameter mostly affects the model only locally around that kernel, whereas a small change in one parameter in a polynomial model affects the model everywhere.

Example 2.4: Car stopping distances

We continue with Example 2.1, but this time we also add the squared speed as a feature, i.e., the features are now x and x^2. This gives the new matrices (cf. (2.7))

$$\mathbf{X} = \begin{bmatrix} 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 5 & 25 \\ \vdots & \vdots & \vdots \\ 1 & 39 & 1521 \\ 1 & 40 & 1600 \end{bmatrix}, \qquad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4 \\ 2 \\ 4 \\ \vdots \\ 110 \\ 134 \end{bmatrix}, \tag{2.23}$$

and when we insert them into the normal equations (2.17), the new parameter estimates are $\widehat{\beta}_0 = 1.58$, $\widehat{\beta}_1 = 0.42$ and $\widehat{\beta}_2 = 0.07$. (Note that $\widehat{\beta}_0$ and $\widehat{\beta}_1$ change, compared to Example 2.3.) This new model looks like



With this model, the predicted stopping distance is now $\widehat{y}_\star = 87$ feet for $\mathbf{x}_\star = 33$ mph, and $\widehat{y}_\star = 153$ for $\mathbf{x}_\star = 45$ mph. This can be compared to Example 2.3, which gives different predictions. Based on the data alone we can not say that this is the "true model", but by visually comparing this model with Example 2.3, this model with more features seems to follow the data slightly better. A systematic method to select between different features (other than just visually comparing plots) is cross-validation, see Chapter 5.

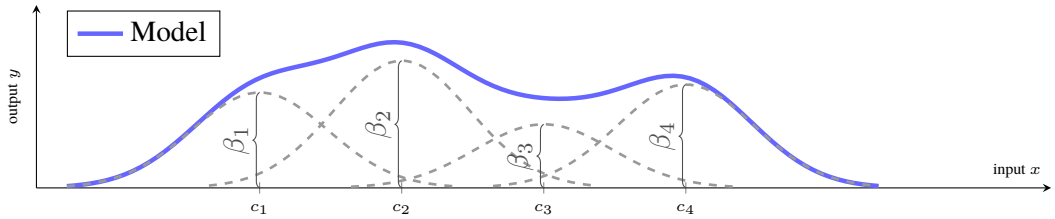


Figure 2.4: A linear regression model using RBF kernels (2.22) as features. Each kernel (dashed gray lines) is located at c_1, c_2, c_3 and c_4, respectively. When the model is learned from data, the parameters $\beta_0, \beta_1, \ldots, \beta_p$ are chosen such that the sum of all kernels (solid blue line) is fitted to the data in, e.g., a least squares sense.

Polynomials and RBF kernels are just two special cases, but we can of course consider any nonlinear transformation of the inputs. To distinguish the 'original' inputs from the 'new' transformed inputs, the term *features* is often used for the latter. To decide which features to use one approach is to compare

competing models (with different features) using cross-validation; see Chapter 5.

2.5 Qualitative input variables

The regression problem is characterized by a quantitative output[3] y, but the nature of the inputs x is arbitrary. We have so far only discussed the case of quantitative inputs x, but qualitative inputs are perfectly possible as well.

Assume that we have a qualitative input variable that only takes two different values (or levels or classes), which we call type A and type B. We can then create a *dummy variable* x as

$$x = \begin{cases} 0 & \text{if type A} \\ 1 & \text{if type B} \end{cases} \tag{2.24}$$

and use this variable in the linear regression model. This effectively gives us a linear regression model which looks like

$$y = \beta_0 + \beta_1 x + \varepsilon = \begin{cases} \beta_0 + \varepsilon & \text{if type A} \\ \beta_0 + \beta_1 + \varepsilon & \text{if type B} \end{cases} \tag{2.25}$$

The choice is somewhat arbitrary, and type A and B can of course be switched. Other choices, such as $x = 1$ or $x = -1$, are also possible. This approach can be generalized to qualitative input variables which take more than two values, let us say type A, B, C and D. With four different values, we create $3 = 4 - 1$ dummy variables as

$$x_1 = \begin{cases} 1 & \text{if type B} \\ 0 & \text{if not type B} \end{cases}, \quad x_2 = \begin{cases} 1 & \text{if type C} \\ 0 & \text{if not type C} \end{cases}, \quad x_3 = \begin{cases} 1 & \text{if type D} \\ 0 & \text{if not type D} \end{cases} \tag{2.26}$$

which, altogether, gives the linear regression model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \varepsilon = \begin{cases} \beta_0 + \varepsilon & \text{if type A} \\ \beta_0 + \beta_1 + \varepsilon & \text{if type B} \\ \beta_0 + \beta_2 + \varepsilon & \text{if type C} \\ \beta_0 + \beta_3 + \varepsilon & \text{if type D} \end{cases} \tag{2.27}$$

Qualitative inputs can be handled similarly in other problems and methods as well, such as logistic regression, k-NN, deep learning, etc.

2.6 Regularization

Even though the linear regression model at a first glance (cf. Figure 2.1) may seem as a fairly rigid and non-flexible model, it is not necessarily so. If more features are obtained by extending the model with nonlinear transformations as in Figures 2.3 or 2.4, or if the number of inputs p is large and the number of data points n is small, one may experience *overfitting*. If considering data as consisting of 'signal' (the actual information) and 'noise' (measurement errors, irrelevant effects, etc.), the term *overfitting* indicates that the model is fitted not only to the 'signal' but also to the 'noise'. An example of overfitting is given in Example 2.5, where a linear regression model with $p = 8$ RBF kernels is learned from $n = 9$ data points. Even though the model follows all data points very well, we can intuitively judge that the model is not particularly useful: neither the interpolation (between the data points) nor the extrapolation (outside the data range) appears sensible. Note that using $p = n - 1$ is an extreme case, but the conceptual problem

[3] If the output variable is qualitative, then we have a classification—and not a regression—problem.

with overfitting is often present also in less extreme situations. Overfitting will be thoroughly discussed later in Chapter 5.

A useful approach to handle overfitting is *regularization*. Regularization can be motivated by 'keeping the parameters $\boldsymbol{\beta}$ small unless the data really convinces us otherwise', or alternatively 'if a model with small values of the parameters $\boldsymbol{\beta}$ fits the data almost as well as a model with larger parameter values, the one with small parameter values should be preferred'. There are several ways to implement this mathematically, which leads to slightly different solutions. We will focus on the *ridge regression* and *LASSO*.

For linear regression, another motivation to use regularization is also when $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, meaning (2.16) has no unique solution $\widehat{\boldsymbol{\beta}}$. In such cases, regularization can be introduced in order to make $\mathbf{X}^\mathsf{T}\mathbf{X}$ invertible and give (2.16) a unique solution. However, the concept of regularization extends well beyond linear regression and can be used also when working with other types of problems and models. For example are regularization-like methods key to obtain a good performance in deep learning, as we will discuss in Section 7.4.

2.6.1 Ridge regression

In *ridge regression* (also known as *Tikhonov regularization*, $L2$ *regularization*, or *weight decay*) the least squares criterion (2.16) is replaced with the modified minimization problem

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \ \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma\|\boldsymbol{\beta}\|_2^2. \tag{2.28}$$

The value $\gamma \geq 0$ is referred to as regularization parameter and has to be chosen by the user. For $\gamma = 0$ we recover the original least squares problem (2.16), whereas if we let $\gamma \to \infty$ we will force all parameters β_j to approach 0. A good choice of γ is in most cases somewhere in between, and depends on the actual problem. It can either be found by manual tuning, or in a more systematic fashion using cross-validation.

It is actually possible to derive a version of the normal equations (2.17) for (2.28), namely

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + \gamma I_{p+1})\widehat{\boldsymbol{\beta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{2.29}$$

where I_{p+1} is the identity matrix of size $(p+1) \times (p+1)$. If $\gamma > 0$, the matrix $\mathbf{X}^\mathsf{T}\mathbf{X} + \gamma I_{p+1}$ is always invertible, and we have the closed form solution

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X} + \gamma I_{p+1})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{2.30}$$

2.6.2 LASSO

With *LASSO* (an abbreviation for Least Absolute Shrinkage and Selection Operator), or equivalently $L1$ *regularization*, the least squares criterion (2.16) is replaced with

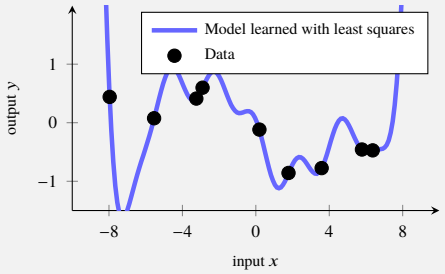
$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \ \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma\|\boldsymbol{\beta}\|_1, \tag{2.31}$$

where $\|\cdot\|_1$ is the Manhattan norm. Contrary to ridge regression, there is no closed-form solution available for (2.31). It is, however, a convex problem which can be solved efficiently by numerical optimization.

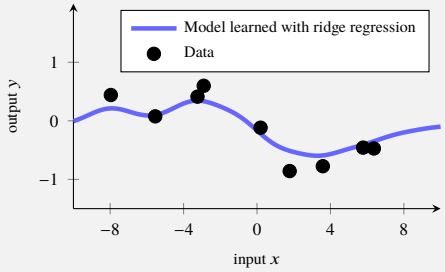
As for ridge regression, the regularization parameter γ has to be chosen by the user also in LASSO: $\gamma = 0$ gives the least squares problem and $\gamma \to \infty$ gives $\boldsymbol{\beta} = 0$. Between these extremes, however, LASSO and ridge regression will result in different solutions: whereas ridge regression pushes all parameters $\beta_0, \beta_1, \ldots, \beta_p$ towards small values, LASSO tends to favor so-called sparse solutions where only a few of the parameters are non-zero, and the rest are exactly zero. Thus, the LASSO solution can effectively 'switch some of the inputs off' by setting the corresponding parameters to zero and it can therefore be used as an input (or feature) selection method.

Example 2.5: Regularization in a linear regression RBF model

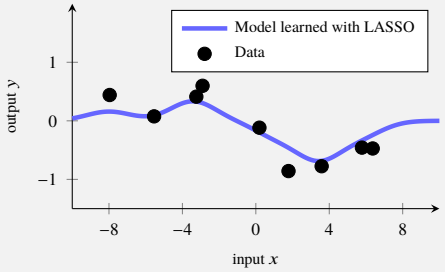
We consider the problem of learning a linear regression model (blue line) with $p = 8$ radial basis function (RBF) kernels as features from $n = 9$ data points (black dots). Since we have $p = n - 1$, we can expect the model to fit the data perfectly. However, as we see in (a) to the right, the model *overfits*, meaning that the model adapts *too* much to the data and has a 'strange' behavior between the data points. As a remedy to this, we can use ridge regression (b) or LASSO (c). Even though the final models with ridge regression and LASSO look rather similar, their parameters $\widehat{\boldsymbol{\beta}}$ are different: the LASSO solution effectively only makes use of 5 (out of 8) radial basis functions. This is referred to as a *sparse* solution. Which approach should be preferred depends, of course, on the specific problem.



(a) The model learned with **least squares** (2.16). Even though the model follows the data exactly, we should typically not be satisfied with this model: neither the behavior between the data points nor outside the range is plausible, but is only an effect of *overfitting*, in that the model is adapted 'too well' to the data. The parameter values $\widehat{\boldsymbol{\beta}}$ are around 30 and -30.



(b) The same model, this time learned with **ridge regression** (2.28) with a certain value of γ. Despite not being perfectly adapted to the training data, this model appears to give a more sensible trade-off between fitting the data and avoiding overfitting than (a), and is probably more useful in most situations. The parameter values $\widehat{\boldsymbol{\beta}}$ are now roughly evenly distributed in the range from -0.5 to 0.5.



(c) The same model again, this time learned with **LASSO** (2.31) with a certain value of γ. Again, this model is not perfectly adapted to the training data, but appears to have a more sensible trade-off between fitting the data and avoiding overfitting than (a), and is probably also more useful than (a) in most situations. In contrast to (b), however, 3 (out of 9) parameters in this model are exactly 0, and the rest are in the range from -1 to 1.

2.6.3 General cost function regularization

Ridge Regression and LASSO are two popular special cases of regularization for linear regression. They both have in common that they modify the cost function, or optimization objective, of (2.16). They can be seen as two instances of a more general regularization scheme

$$\underset{\boldsymbol{\beta}}{\text{minimize}} \ \underbrace{V(\boldsymbol{\beta}, \mathbf{X}, \mathbf{y})}_{\text{data fit}} + \gamma \underbrace{R(\boldsymbol{\beta})}_{\substack{\text{model} \\ \text{flexibility} \\ \text{penalty}}} . \tag{2.32}$$

Note that (2.32) contains three important elements: (i) one term which describes how well the model fits to data, (ii) one term which penalizes model complexity (large parameter values), and (iii) a trade-off parameter γ between them.

2.7 Further reading

Linear regression has now been used for well over 200 years. It was first introduced independently by Adrien-Marie Legendre in 1805 and Carl Friedrich Gauss in 1809 when they discovered the method of least squares. The topic of linear regression is due to its importance described in many textbooks in statistics and machine learning, such as Bishop (2006), Gelman et al. (2013), Hastie, Tibshirani, and Friedman (2009), and Murphy (2012). While the basic least squares technique has been around for a long time, its regularized versions are much younger. Ridge regression was introduced independently in statistics by Hoerl and Kennard (1970) and in numerical analysis under the name of Tikhonov regularization. The LASSO was first introduced by Tibshirani (1996). The recent monograph by Hastie, Tibshirani, and Wainwright (2015) covers the development related to the use of sparse models and the LASSO.

2.A Derivation of the normal equations

The normal equations (2.17)

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{X}^\mathsf{T}\mathbf{y}.$$

can be derived from (2.16)

$$\widehat{\boldsymbol{\beta}} = \operatorname*{argmin}_{\boldsymbol{\beta}} \ \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2,$$

in different ways. We will present one based on (matrix) calculus and one based on geometry and linear algebra.

No matter how (2.17) is derived, if $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, it (uniquely) gives

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y},$$

If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, then (2.17) has infinitely many solutions $\widehat{\boldsymbol{\beta}}$, which all are equally good solutions to the problem (2.16).

2.A.1 A calculus approach

Let

$$V(\boldsymbol{\beta}) = \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 = (\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^\mathsf{T}(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y} - 2\mathbf{y}^\mathsf{T}\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta}, \tag{2.33}$$

and differentiate $V(\boldsymbol{\beta})$ with respect to the vector $\boldsymbol{\beta}$,

$$\frac{\partial}{\partial \boldsymbol{\beta}} V(\boldsymbol{\beta}) = -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta}. \tag{2.34}$$

Since $V(\boldsymbol{\beta})$ is a positive quadratic form, its minimum must be attained at $\frac{\partial}{\partial \boldsymbol{\beta}} V(\boldsymbol{\beta}) = 0$, which characterizes the solution $\widehat{\boldsymbol{\beta}}$ as

$$\frac{\partial}{\partial \boldsymbol{\beta}} V(\widehat{\boldsymbol{\beta}}) = 0 \Leftrightarrow -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta} = 0 \Leftrightarrow \mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{2.35}$$

i.e., the normal equations.

2.A.2 A linear algebra approach

Denote the $p+1$ columns of $\mathbf{X}$ as $c_j, j = 1, \ldots, p+1$. We first show that $\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$ is minimized if $\boldsymbol{\beta}$ is chosen such that $\mathbf{X}\boldsymbol{\beta}$ is the orthogonal projection of $\mathbf{y}$ onto the (sub)space spanned by the columns c_j of $\mathbf{X}$, and then show that the orthogonal projection is found by the normal equations.

Let us decompose $\mathbf{y}$ as $\mathbf{y}_\perp + \mathbf{y}_\parallel$, where $\mathbf{y}_\perp$ is orthogonal to the (sub)space spanned by all columns c_i, and $y_\parallel$ is in the (sub)scpace spanned by all columns c_i. Since $\mathbf{y}_\perp$ is orthogonal to both $\mathbf{y}_\parallel$ and $\mathbf{X}\boldsymbol{\beta}$, it follows that

$$\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\beta} - (\mathbf{y}_\perp + \mathbf{y}_\parallel)\|_2^2 = \|(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}_\parallel) - \mathbf{y}_\perp\|_2^2 \geq \|\mathbf{y}_\perp\|_2^2, \tag{2.36}$$

and the triangle inequality also gives us

$$\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}_\perp - \mathbf{y}_\parallel\|_2^2 \leq \|\mathbf{y}_\perp\|_2^2 + \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}_\parallel\|_2^2. \tag{2.37}$$

This implies that if we choose $\boldsymbol{\beta}$ such that $\mathbf{X}\boldsymbol{\beta} = \mathbf{y}_\parallel$, the criterion $\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$ must have reached its minimum. Thus, our solution $\widehat{\boldsymbol{\beta}}$ must be such that $\mathbf{X}\widehat{\boldsymbol{\beta}} - \mathbf{y}$ is orthogonal to the (sub)space spanned by all columns c_i, i.e.,

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\beta}})^\mathsf{T} c_j = 0, j = 1, \ldots, p+1 \tag{2.38}$$

(remember that two vectors $\mathbf{u}, \mathbf{v}$ are, by definition, orthogonal if their scalar product, $\mathbf{u}^\mathsf{T}\mathbf{v}$, is 0.) Since the columns c_j together form the matrix $\mathbf{X}$, we can write this compactly as

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\beta}})^\mathsf{T}\mathbf{X} = 0, \tag{2.39}$$

where the right hand side is the $p+1$-dimensional zero vector. This can equivalently be written as

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{X}^\mathsf{T}\mathbf{y},$$

i.e., the normal equations.

3 The classification problem and three parametric classifiers

We will now study the *classification* problem. Whereas the regression problem has quantitative outputs, classification is the situation with qualitative outputs. A method that performs classification is referred to as a *classifier*. Our first classifier will be *logistic regression*, and we will in this chapter also introduce the linear and quadratic discriminant analysis classifiers (LDA and QDA, respectively). More advanced classifiers, such as classification trees, boosting and deep learning, will be introduced in the later chapters.

3.1 The classification problem

Classification is about predicting a *qualitative output* from p inputs of arbitrary types (quantitative and/or qualitative). Since the output is qualitative, it can only take values from a finite set. We use K to denote the number of elements in the set of possible output values. The set of possible output values can, for instance, be $\{\text{false, true}\}$ ($K = 2$) or $\{\text{Sweden, Norway, Finland, Denmark}\}$ ($K = 4$). We will refer to these elements as *classes* or *labels*. The number of classes K is assumed to be known throughout this text. To prepare for a concise mathematical notation, we generically use integers $1, 2, \ldots, K$ to denote the output classes. The integer labeling of the classes is arbitrary, and we use it only for notational convenience. The use of integers does *not* mean there is any inherent ordering of the classes.

When there are only $K = 2$ classes, we have the important special case of *binary* classification. In binary classification, we often use the labels[1] 0 and 1 (instead of 1 and 2). Occasionally, we will also use the terms *positive* (class $k = 1$) and *negative* (class $k = 0$) as well. The reason for using different choices for the two labels in binary classification is purely for mathematical convenience.

Classification amounts to predicting the output from the input. In our statistical approach, we understand classification as the problem of predicting class probabilities

$$p(y \mid \mathbf{x}), \tag{3.1}$$

where y is the output ($1, 2, \ldots,$ or K) and $\mathbf{x}$ is the input. Note that we use $p(y \mid \mathbf{x})$ to denote probability masses (y qualitative) as well as probability densities (y quantitative). In words, $p(y \mid \mathbf{x})$ describes *the probability for the output y (a class label) given that we know the input* $\mathbf{x}$. This probability will be a cornerstone from now on, so we will first spend some effort to understand it well. Talking about $p(y \mid \mathbf{x})$ implies that we think about the class label y as a random variable. Why? Because we choose to model the real world, from where the data originates, as involving a certain amount of randomness (cf. ε in regression). Let us illustrate with an example:

[1] In Chapter 6 we will use $k = 1$ and $k = -1$ instead.

Example 3.1: Modeling voting behavior—randomness in the class label y

If we are to describe voting preferences ($= y$, the qualitative output) among different population groups ($= \mathbf{x}$, the input), we have to face that all people in a certain population group will not vote for the same political party. To describe this mathematically, we can think of y as a random variable which follows a certain probability distribution. If we knew that the vote count in the group of 45 year old women ($= \mathbf{x}$) is 13% for the cerise party, 39% for the turquoise party and 48% for the purple party, we could describe it as

$$\begin{aligned} p(y = \text{cerise party} \,|\, \text{45 year old women}) &= 0.13, \\ p(y = \text{turqoise party} \,|\, \text{45 year old women}) &= 0.39, \\ p(y = \text{purple party} \,|\, \text{45 year old women}) &= 0.48. \end{aligned}$$

In this way, we use probabilities $p(y \,|\, \mathbf{x})$ to describe the non-trivial fact that

(a) not all 45 year old women vote for the same party, but

(b) the choice of party does not appear to be completely random among 45 year old women either; the purple party is the most popular, and the cerise party is the least popular.

The number of output classes in this example is $K = 3$.

3.2 Logistic regression

For binary classification (y is either 0 or 1), learning a classifier amounts to learning a model of $p(y = 1 \,|\, \mathbf{x})$ and $p(y = 0 \,|\, \mathbf{x})$. Since, by the laws of probability, we have that $p(y = 1 \,|\, \mathbf{x}) + p(y = 0 \,|\, \mathbf{x}) = 1$, it is sufficient to learn a model for only one of the class probabilities, say $p(y = 1 \,|\, \mathbf{x})$, from which $p(y = 0 \,|\, \mathbf{x})$ will follow.

From the previous chapter, we have the linear regression model. If we slightly change the notation from Chapter 2 and let $\mathbf{x} = [1 \; x_1 \; x_2 \; \dots \; x_p]^\mathsf{T}$ (i.e., include 1 in the first position), we can write

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p = \boldsymbol{\beta}^\mathsf{T} \mathbf{x}. \tag{3.2}$$

The input to this function is $\mathbf{x}$, and the output, here denoted by z, takes values on the entire real line. Note that we have skipped the noise ε. In classification we are interested in $p(y = 1 \,|\, \mathbf{x})$, which however is a function of $\mathbf{x}$ which takes values only within the interval $[0, 1]$ (since p, in this context, is a probability mass function). The key idea underlying logistic regression is thus to 'squeeze' the output from linear regression z into the interval $[0, 1]$ by using the logistic function (Figure 3.1)

$$h(z) = \frac{e^z}{1 + e^z}. \tag{3.3}$$

Since the logistic function is limited to take values between 0 and 1, we obtain altogether a function from $\mathbf{x}$ to $[0, 1]$, which we can use as a model for $p(y = 1 \,|\, \mathbf{x})$,

$$p(y = 1 \,|\, \mathbf{x}) = \frac{e^{\boldsymbol{\beta}^\mathsf{T} \mathbf{x}}}{1 + e^{\boldsymbol{\beta}^\mathsf{T} \mathbf{x}}}. \tag{3.4a}$$

Note that this implicitly also gives us that

$$p(y = 0 \,|\, \mathbf{x}) = 1 - \frac{e^{\boldsymbol{\beta}^\mathsf{T} \mathbf{x}}}{1 + e^{\boldsymbol{\beta}^\mathsf{T} \mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\beta}^\mathsf{T} \mathbf{x}}}. \tag{3.4b}$$

We now have a model for $p(y = 1 \,|\, \mathbf{x})$ and $p(y = 0 \,|\, \mathbf{x})$, which contains unknown parameters $\boldsymbol{\beta}$ that can be learned from training data. That is, we have constructed a binary classifier, which is called *logistic regression*.

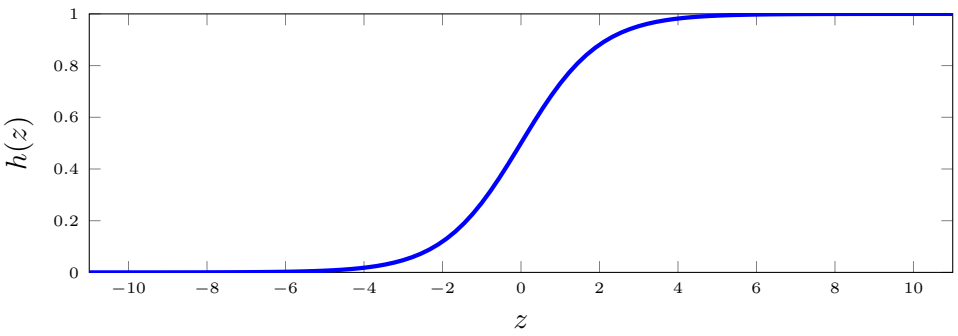


Figure 3.1: The logistic function $h(z) = \frac{e^z}{1+e^z}$.

Remark 3.1 *Despite its name, logistic regression is a method for classification, not regression! The (confusing) name is due only to historical reasons.*

3.2.1 Learning the logistic regression model from training data

By using the logistic function, we have transformed linear regression, a regression method, into logistic regression, a classification method. The price to pay is that we will not be able to use the handy normal equations for learning $\boldsymbol{\beta}$ in logistic regression (as we could for linear regression). Just as in linear regression, we want to learn $\boldsymbol{\beta}$ from training data $\mathcal{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ using the maximum likelihood approach, that is, solving

$$\widehat{\boldsymbol{\beta}} = \arg\max_{\boldsymbol{\beta}} \ell\left(\boldsymbol{\beta}\right), \tag{3.5}$$

where $\ell\left(\boldsymbol{\beta}\right)$ is the *likelihood function*

$$\ell\left(\boldsymbol{\beta}\right) \triangleq p(\mathbf{y} \,|\, \mathbf{X}; \boldsymbol{\beta}). \tag{3.6}$$

Let us now work out a detailed expression for the likelihood function[2],

$$\begin{aligned}\ell\left(\boldsymbol{\beta}\right) = p(\mathbf{y} \,|\, \mathbf{X}; \boldsymbol{\beta}) = \prod_{i=1}^{n} p(y_i \,|\, \mathbf{x}_i; \boldsymbol{\beta}) = \prod_{i:y_i=1} p(y=1 \,|\, \mathbf{x}_i; \boldsymbol{\beta}) \prod_{i:y_i=0} p(y=0 \,|\, \mathbf{x}_i; \boldsymbol{\beta}) = \\ = \prod_{i:y_i=1} \frac{e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}}{1+e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}} \prod_{i:y_i=0} \frac{1}{1+e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}}.\end{aligned} \tag{3.7}$$

This is the function which we would like to optimize with respect to $\boldsymbol{\beta}$, cf. (3.5). For numerical reasons, it is often better to optimize the logarithm of $\ell\left(\boldsymbol{\beta}\right)$ (since the logarithm is a monotone function, the maximizing argument is the same),

$$\begin{aligned}\log \ell\left(\boldsymbol{\beta}\right) &= \sum_{i:y_i=1} \left(\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i - \log\left(1+e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}\right)\right) - \sum_{i:y_i=0} \log\left(1+e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}\right) \\ &= \sum_{i=1}^{n} y_i \boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i - \log\left(1+e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}\right).\end{aligned} \tag{3.8}$$

The simplification in the second equality relies on the chosen labeling, that $y_i = 0$ or $y_i = 1$, which is indeed the reason for why this labeling is convenient.

A necessary condition for the maximum of $\log \ell\left(\boldsymbol{\beta}\right)$ is that its gradient is zero,

$$\nabla_{\boldsymbol{\beta}} \log \ell\left(\boldsymbol{\beta}\right) = \sum_{i=1}^{n} \mathbf{x}_i \left(y_i - \frac{e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}}{1+e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i}}\right) = 0. \tag{3.9}$$

[2]We now add $\boldsymbol{\beta}$ to the expression $p(y \,|\, \mathbf{x})$, to explicitly show its dependence also on $\boldsymbol{\beta}$.

Note that this equation is vector-valued, i.e., we have a system of $p+1$ equations to solve (with $p+1$ unknown elements of the vector $\boldsymbol{\beta}$). Contrary to the linear regression model (with Gaussian noise) in Section 2.3.1, this maximum likelihood problem results in a *nonlinear* system of equations, lacking a general closed form solution. Instead, we are forced to use a numerical solver, as discussed in Appendix B. The standard choice is to use the Newton–Raphson algorithm (equivalent to the so-called iteratively reweighted least squares algorithm), see e.g. Hastie, Tibshirani, and Friedman 2009, Chapter 4.4.

Algorithm 1: Logistic regression for binary classification

Data: Training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ (with output classes $y = 0, 1$ and test input $\mathbf{x}_\star$
Result: Predicted test output $\widehat{y}_\star$

Learn

1 Compute $\widehat{\beta}$ by solving (3.9) numerically.

Predict

2 Compute $p(y = 1 \,|\, \mathbf{x}_\star)$ (3.4a) and $p(y = 0 \,|\, \mathbf{x}_\star)$ (3.4b).
3 If $p(y = 1 \,|\, \mathbf{x}_\star) > p(y = 0 \,|\, \mathbf{x}_\star)$ set $\widehat{y}_\star \leftarrow 1$, otherwise set $\widehat{y}_\star \leftarrow 0$.

3.2.2 Decision boundaries for logistic regression

So far, we have presented logistic regression as a method for modeling the class probabilities $p(y = 0 \,|\, \mathbf{x})$ and $p(y = 1 \,|\, \mathbf{x})$. However, if we want to use logistic regression for actually making a prediction for the test input $\mathbf{x}_\star$, i.e., deciding whether we believe $y_\star = 0$ or $y_\star = 1$, we have to add a final step. First, we learn $\boldsymbol{\beta}$ from training data, thereafter compute $p(y = 0 \,|\, \mathbf{x}_\star)$ and $p(y = 1 \,|\, \mathbf{x}_\star)$, and finally we let the prediction $\widehat{y}_\star$ be the most probable class (we will give a motivation for this later in Section 3.4.1),

$$\widehat{y}_\star = \arg\max_{k=0,1} p(y = k \,|\, \mathbf{x}_\star). \tag{3.10}$$

This is illustrated in Figure 3.2 for a one-dimensional input $\mathbf{x}$.

A classifier associates all points in the space of possible test inputs $\mathbf{x}_\star$ to a prediction $y_\star$. Most often, the classifier forms certain regions which belong to the same prediction. The boundary between those regions, that is, the curve which separates different class predictions from each other, is called the *decision boundary*. A decision boundary for logistic is illustrated by Figure 3.2 for a one-dimensional input case, and in Figure 3.3 for two two-dimensional input cases.

We can find the decision boundary by solving the equation

$$p(y = 1 \,|\, \mathbf{x}) = p(y = 0 \,|\, \mathbf{x}) \tag{3.11}$$

which with logistic regression gives

$$\frac{e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}}} \Leftrightarrow e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{x}} = 1 \Leftrightarrow \boldsymbol{\beta}^\mathsf{T}\mathbf{x} = 0. \tag{3.12}$$

The equation $\boldsymbol{\beta}^\mathsf{T}\mathbf{x} = 0$ parameterizes a (linear) hyperplane. Hence, the decision boundaries in logistic regression always have the shape of a (linear) hyperplane.

We distinguish between different types of classifiers by the shape of their decision boundary: Since logistic regression only has *linear* decision boundaries, it is consequently called a *linear classifier*.

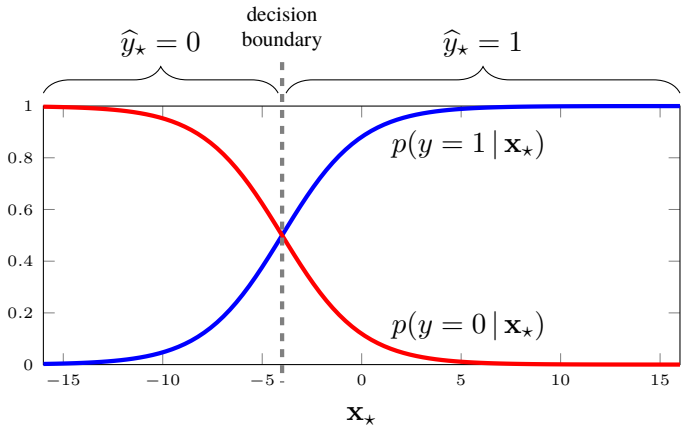
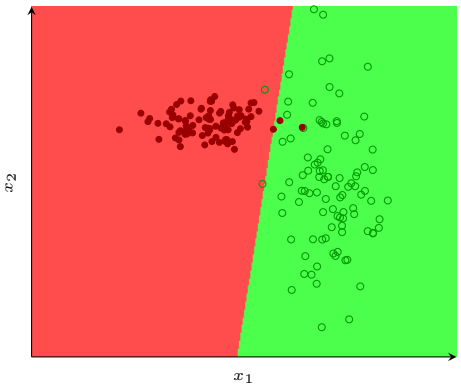
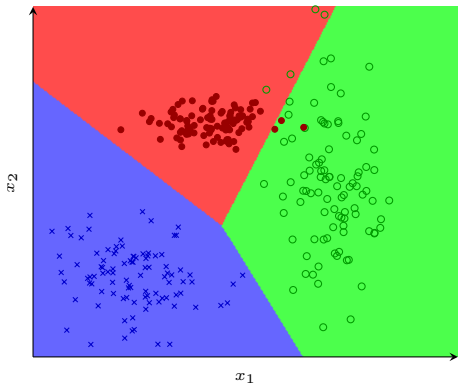


Figure 3.2: Consider binary classification ($y = 0$ or 1) when the input $\mathbf{x}$ is scalar (horizontal axis). Once $\boldsymbol{\beta}$ is learned from training data (not shown), logistic regression gives us a model for $p(y = 1 \mid \mathbf{x}_\star)$ (blue) and $p(y = 0 \mid \mathbf{x}_\star)$ (red) for any test input $\mathbf{x}_\star$. To turn these modeled probabilities into actual class predictions ($\widehat{y}_\star$ is either 0 or 1), the class which is modeled to have the highest probability is taken as the prediction. The point(s) where the prediction changes from from one class to another is called the *decision boundary* (dashed vertical line).



(a) Logistic regression for $K = 2$ classes always gives a linear decision boundary. The red dots and green circles are training data from different classes, and the intersection between the red and green fields is the decision boundary obtained for the logistic regression classifier learned from the training data.



(b) Logistic regression for $K = 3$ classes. We have now introduced training data from a third class, marked with blue crosses. The decision boundary between any pair of two classes is still linear.

Figure 3.3: Examples of decision boundaries for logistic regression.

3.2.3 Logistic regression for more than two classes

Logistic regression can be used also if there are more than two classes, $K > 2$. There are several ways of generalizing logistic regression to the multi-class problem, and we will follow one path which also will be useful later in deep learning (Chapter 7). This generalization requires two steps: first, we will introduce the one-hot encoding, and second, we will replace the logistic function with a softmax function.

Let us start with introducing the one-hot encoding. Instead of letting y be an integer value in the range $\{1, \ldots, K\}$ ("vanilla encoding"), we replace the output y_i with a K-dimensional vector $\mathbf{y}_i$. If the original output is k, then the k:th entry of $\mathbf{y}_i$ is 1, and the remaining entries zero. For $K = 3$, it would look like

Vanilla encoding	One-hot encoding
$y_i = 1$	$\mathbf{y}_i = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^\mathsf{T}$
$y_i = 2$	$\mathbf{y}_i = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\mathsf{T}$
$y_i = 3$	$\mathbf{y}_i = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^\mathsf{T}$.

Since we now have a vector-valued output $\mathbf{y}$, we also need a vector-valued alternative to the logistic function. To this end, we introduce the vector-valued softmax function

$$\mathrm{softmax}(\mathbf{z}) \triangleq \frac{1}{\sum_{j=1}^{K} e^{z_j}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_K} \end{bmatrix}, \tag{3.13}$$

to which $\mathbf{z}$ is a K-dimensional input vector $\mathbf{z} = [z_1 \; z_2 \; \ldots \; z_K]$. The softmax function has the following properties: The sum of its output vector is always 1, and each element is always in the interval $[0, 1]$. Similarly to how we combined linear regression and the logistic function for the binary classification problem (3.4), we now combine linear regression and the softmax function to model the class probabilities,

$$\begin{bmatrix} p(1 \,|\, \mathbf{x}_i) \\ p(2 \,|\, \mathbf{x}_i) \\ \vdots \\ p(K \,|\, \mathbf{x}_i) \end{bmatrix} = \mathrm{softmax}(\mathbf{z}), \text{ where } \mathbf{z} = \begin{bmatrix} \boldsymbol{\beta}_1^\mathsf{T} \mathbf{x}_i \\ \boldsymbol{\beta}_2^\mathsf{T} \mathbf{x}_i \\ \vdots \\ \boldsymbol{\beta}_K^\mathsf{T} \mathbf{x}_i \end{bmatrix}, \tag{3.14}$$

or equivalently

$$p(k \,|\, \mathbf{x}_i) = \frac{e^{\boldsymbol{\beta}_k^\mathsf{T} \mathbf{x}_i}}{\sum_{l=1}^{K} e^{\boldsymbol{\beta}_l^\mathsf{T} \mathbf{x}_i}}. \tag{3.15}$$

This is our multi-class logistic regression. Note that we in this construction use K vectors $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_K$ (one for each class), meaning that the number of parameters to learn grows with K. As for binary logistic regression, we can learn those parameters using the maximum likelihood idea. We use $\boldsymbol{\theta}$ to denote *all* parameters in $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_K$. With the one-hot encoding, the likelihood function takes the particular form

$$\begin{aligned} \log \ell(\boldsymbol{\theta}) &= \log p(\mathbf{y} \,|\, \mathbf{X}; \boldsymbol{\theta}) = \sum_{i=1}^{n} \log p(y_i \,|\, \mathbf{x}_i; \boldsymbol{\theta}) = \\ &= \sum_{i:y_i=1} \log p(1 \,|\, \mathbf{x}_i; \boldsymbol{\theta}) + \sum_{i:y_i=2} \log p(2 \,|\, \mathbf{x}_i; \boldsymbol{\theta}) + \cdots + \sum_{i:y_i=K} \log p(K \,|\, \mathbf{x}_i; \boldsymbol{\theta}) = \\ &= \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log p(k \,|\, \mathbf{x}_i; \boldsymbol{\theta}). \end{aligned} \tag{3.16}$$

where y_{ik} are the elements of the one-hot encoding vectors. We will not pursue any more details here, but similarly to the binary case this likelihood function can also be used as objective function in numerical optimization. The particular form of (3.16) will appear every time we use the one-hot encoding, and is sometimes referred to as *cross-entropy*.

Time to reflect 3.1: *The softmax solution is actually slightly over-parameterized, compared to binary logistic regression* (3.4)*. That is not a problem in practice, but consider the case* $K = 2$ *and see if you can spot it!*

Remark 3.2 *The one-hot encoding will later be useful in deep learning. We will, however, not use it for all multiclass methods. LDA, QDA and* k*-NN, for example, all use the vanilla encoding also for the multi-class problem.*

3.3 Linear and quadratic discriminant analysis (LDA & QDA)

We will now introduce two other classifiers, namely linear and quadratic discriminant analysis (LDA[3] and QDA, respectively). In logistic regression, we used linear regression and the logistic function to model $p(y \mid \mathbf{x})$. In LDA and QDA, we instead make the assumption that $p(\mathbf{x} \mid y)$ is a Gaussian distribution. As it turns out, this will give us a classifier which is easy to learn (requires no numerical optimization, in contrast to logistic regression), and is useful in practice also when the Gaussian assumption about $p(\mathbf{x} \mid y)$ is not met.

3.3.1 Using Gaussian approximations in Bayes' theorem

From probability theory, Bayes' theorem might be familiar, which says that

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid y)p(y)}{\sum_{k=1}^{K} p(\mathbf{x} \mid k)p(k)}. \tag{3.17}$$

The left hand side, $p(y \mid \mathbf{x})$, is our core interest in classification. In a practical machine learning problem, neither the left nor the right hand side of (3.17) is known to us; we only have training data, and no one provides us with any equations. In logistic regression, we went straight to the left hand side and modeled that as (3.4). In LDA and QDA, instead, we focus on the right hand side, by *assuming* that $p(\mathbf{x} \mid y)$ has a Gaussian distribution (no matter what the data actually looks like). Since this is now a distribution over the input[4] $\mathbf{x}$, and $\mathbf{x}$ usually has more than one dimension, $p(\mathbf{x} \mid y)$ has to be a multivariate Gaussian distribution with a mean vector and a covariance matrix.

Of course, we want a classifier which learns something from the training data. That is done by learning the parameters of the Gaussian distribution, the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$, from the training data. In LDA the mean vector $\boldsymbol{\mu}$ is assumed to be different for each class, but the covariance matrix $\boldsymbol{\Sigma}$ is assumed to be the same for all classes. In QDA, on the other hand, are both the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ assumed to be different for each class. Since the mean vectors and the covariance matrices will be learned from data, we will append them with a hat symbol, $\widehat{\boldsymbol{\mu}}$ and $\widehat{\boldsymbol{\Sigma}}$.

The right hand side of Bayes' theorem (3.17) also includes the factor $p(y)$, which usually is unknown as well. The meaning of this term is the probability that a completely random data point has label y (without knowing its inputs $\mathbf{x}$). As an approximation of $p(y)$, the occurrence of class k in the training data, denoted $\widehat{\pi}_k$, is used $p(y) \approx p(k)$. For example, if 22% of the training data has label 1, we approximate $p(1)$ as $\widehat{\pi}_1 = 0.22$, etc.

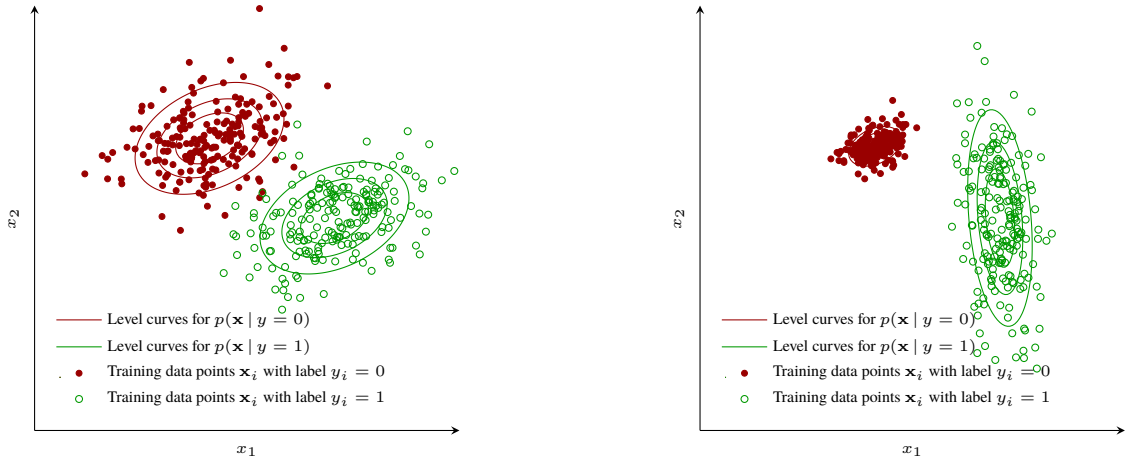
Thus, in LDA $p(y \mid \mathbf{x})$ is modeled as

$$p(y = k \mid \mathbf{x}_\star) = \frac{\widehat{\pi}_k \mathcal{N}\left(\mathbf{x}_\star \mid \widehat{\boldsymbol{\mu}}_k, \widehat{\boldsymbol{\Sigma}}\right)}{\sum_{j=1}^{K} \widehat{\pi}_j \mathcal{N}\left(\mathbf{x}_\star \mid \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}\right)}, \tag{3.18}$$

for $k = 1, 2, \ldots, K$. This is what is shown in Figure 3.4.

[3] Note to be confused with Latent Dirichlet Allocation, which is a completely different machine learning method.

[4] TODO: This actually assumes that $\mathbf{x}$ is quantiative. How are qualitative inputs handled in LDA/QDA?



(a) In LDA, it is assumed that the input $\mathbf{x}$, for a given output y, is distributed as a Gaussian distribution. The mean is different for each class y, but the covariance is the same for all classes. This plot shows how we assume that the training data looks like, when we derive LDA.

(b) Also in QDA, it is assumed that the input $\mathbf{x}$, for a given output y, is distributed as a Gaussian distribution, but for QDA both the mean and the covariance can differ between the classes. This plot shows how we assume that the training data looks like, when we derive QDA.

Figure 3.4: LDA and QDA is derived by assuming that $p(\mathbf{x} \mid y)$ has a Gaussian distribution. This means that we think about the input variables as random and *assume* that they have a certain distribution. In LDA (left panel, a), we assume that the covariance of the input distribution (shape of the level curves) is the same for all classes, and they only differ in locations. QDA (right panel, b) assumes that the covariance can be different for different classes. In fact, when using LDA and QDA in practice, these assumptions on how the inputs $\mathbf{x}$ are distributed are rarely satisfied, but this is nevertheless the way we motivate the methods.

In full analogy, for QDA we have (the only difference is the covariance matrix $\widehat{\boldsymbol{\Sigma}}$)

$$p(y = k \mid \mathbf{x}_\star) = \frac{\widehat{\pi}_k \mathcal{N}\left(\mathbf{x}_\star \mid \widehat{\boldsymbol{\mu}}_k, \widehat{\boldsymbol{\Sigma}}_k\right)}{\sum_{j=1}^{K} \widehat{\pi}_j \mathcal{N}\left(\mathbf{x}_\star \mid \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}_j\right)}. \tag{3.19}$$

Note that we have not made any restriction on K, but we can use LDA and QDA for binary as well as multi-class classification. We will now discuss some aspects in more detail.

3.3.2 Using LDA and QDA in practice

We have derived LDA and QDA by studying Bayes' theorem (3.17). We are ultimately interested in the left hand side of (3.17), and we went there by making an assumption about the right hand side, namely that $p(\mathbf{x} \mid y)$ has a Gaussian distribution. In most practical cases that assumption does not hold in reality (or, at least, it is hard for us to verify whether it holds or not), but LDA as well as QDA turns out to be useful classifiers even when that assumption does not hold.

How do we go about in practice, if we want to learn an LDA or QDA classifier from training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ (without knowing something about the *real* distribution $p(\mathbf{x} \mid y)$) and use it to make a prediction?

Learning the parameters

First, the parameters $\widehat{\pi}_k$, $\widehat{\boldsymbol{\mu}}_k$ and $\widehat{\boldsymbol{\Sigma}}$ (for LDA) or $\widehat{\boldsymbol{\Sigma}}_k$ (for QDA) have, for each $k = 1, \ldots, K$, to be learned, or estimated, from the training data. The perhaps most straightforward parameter to learn is $\widehat{\pi}_k$, the relative occurrence of class k in the training data,

$$\widehat{\pi}_k = \frac{n_k}{n}, \tag{3.20a}$$

where n_k is the number of training data samples in class k. Consequently, all n_k must sum to n, and thereby $\sum_k \widehat{\pi}_k = 1$. Further, the mean vector $\boldsymbol{\mu}_k$ of each class is learned as

$$\widehat{\boldsymbol{\mu}}_k = \frac{1}{n_k} \sum_{i:y_i=k} \mathbf{x}_i, \tag{3.20b}$$

the empirical mean among all training samples of class k. For LDA, the common covariance matrix $\boldsymbol{\Sigma}$ for all classes is usually learned as

$$\widehat{\boldsymbol{\Sigma}} = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_k)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_k)^\mathsf{T} \tag{3.20c}$$

which can be shown to be an unbiased estimate of the covariance matrix[5]. For QDA, one covariance matrix $\boldsymbol{\Sigma}_k$ has to be learned for each class $k = 1, \ldots, K$, usually as

$$\widehat{\boldsymbol{\Sigma}}_k = \frac{1}{n_k - 1} \sum_{i:y_i=k} (\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_k)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_k)^\mathsf{T}, \tag{3.20d}$$

which similarly also can be shown to be an unbiased estimate.

Remark 3.3 *To derive the learning of LDA and QDA, we did not make use of the maximum likelihood idea, in contrast to linear and logistic regression. Furthermore, learning LDA and QDA amounts to inserting the training data into the closed-form expressions* (3.20)*, similar to linear regression (the normal equations), but different from logistic regression (which requires numerical optimization).*

Making predictions

Once we have learned the parameters $\widehat{\pi}_k$, $\widehat{\boldsymbol{\mu}}_k$ and $\widehat{\boldsymbol{\Sigma}}$ or $\widehat{\boldsymbol{\Sigma}}_k$ for all classes $k = 1, \ldots, K$, we have a model for $p(y \,|\, \mathbf{x})$ (3.18 and 3.19) we can use it for making predictions for a test input $\mathbf{x}_\star$. As for logistic regression, we turn $p(y \,|\, \mathbf{x}_\star)$ into actual predictions $\widehat{y}_\star$ by taking the most probable class as the prediction,

$$\widehat{y}_\star = \arg\max_k p(y = k \,|\, \mathbf{x}_\star). \tag{3.21}$$

We summarize this by algorithm 2 and 3, and illustrate by Figure 3.5 and 3.6.

Algorithm 2: Linear Discriminant Analysis, LDA

Data: Training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ (with output classes $k = 1, \ldots, K$) and test input $\mathbf{x}_\star$
Result: Predicted test output $\widehat{y}$

Learn
for $k = 1, \ldots, K$ **do**
  Compute $\widehat{\pi}_k$ (3.20a) and $\widehat{\boldsymbol{\mu}}_k$ (3.20b)
end
Compute $\widehat{\boldsymbol{\Sigma}}$ (3.20c)

Predict
for $k = 1, \ldots, K$ **do**
  Compute $p(y = k \,|\, \mathbf{x}_\star)$ (3.18)
end
Find largest $p(y = k \,|\, \mathbf{x}_\star)$ and set $\widehat{y}_\star$ to that k

[5] This means that the if we estimate $\widehat{\boldsymbol{\Sigma}}$ like this for new training data over and over again, the average would be the true covariance matrix of $p(\mathbf{x})$.

Algorithm 3: Quadratic Discriminant Analysis, QDA

Data: Training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ (with output classes $k = 1, \ldots, K$) and test input $\mathbf{x}_\star$
Result: Predicted test output $\widehat{y}_\star$

Learn

for $k = 1, \ldots, K$ **do**
  Compute $\widehat{\pi}_k$ (3.20a), $\widehat{\boldsymbol{\mu}}_k$ (3.20b) and $\widehat{\boldsymbol{\Sigma}}_k$ (3.20d)
end

Predict

for $k = 1, \ldots, K$ **do**
  Compute $p(y = k \,|\, \mathbf{x}_\star)$ (3.19)
end
Find largest $p(y = k \,|\, \mathbf{x}_\star)$ and set $\widehat{y}_\star$ to that k

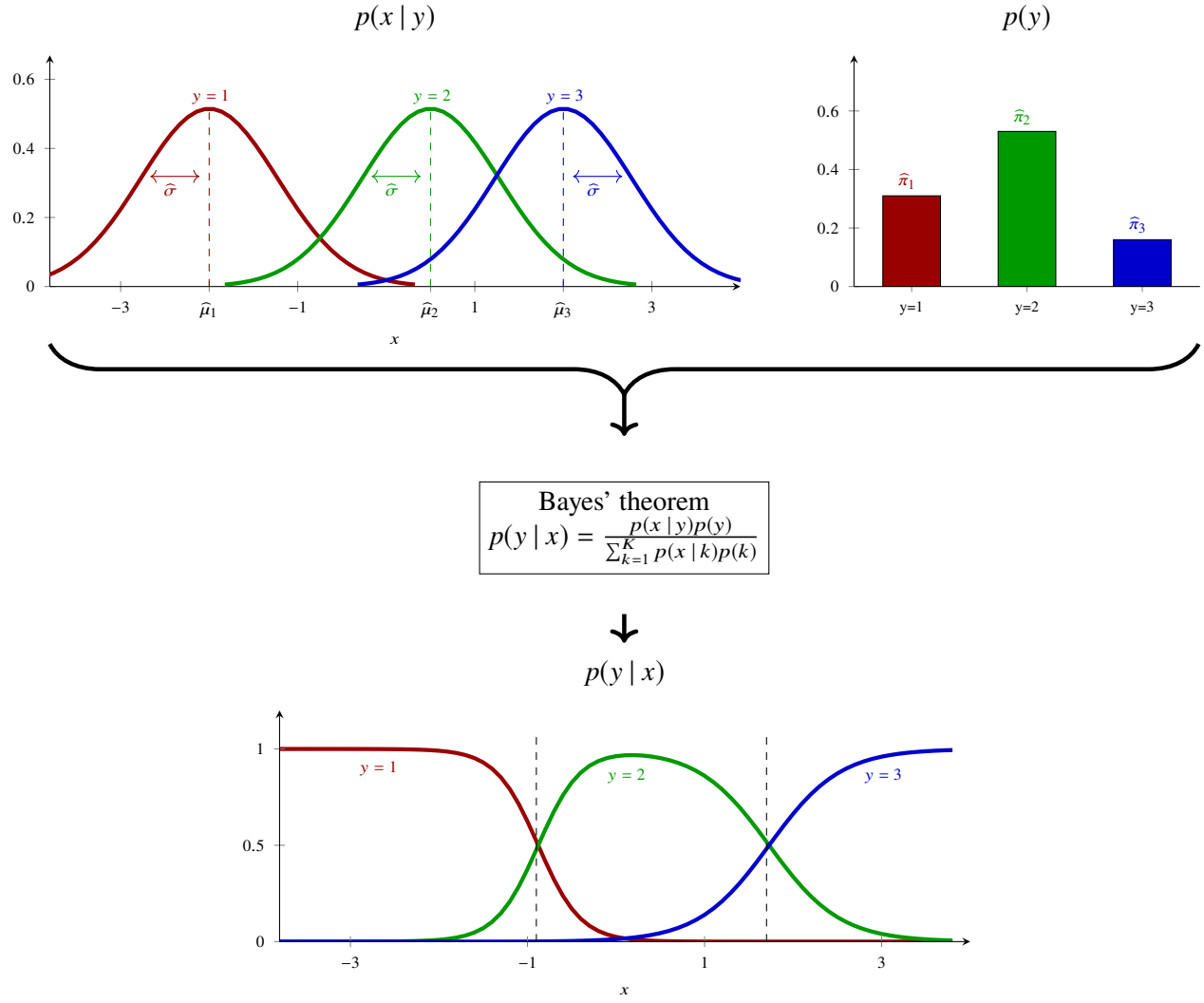


Figure 3.5: An illustration of LDA for $K = 3$ classes, with dimension $p = 1$ of the input $\mathbf{x}$. In the upper left panel is the Gaussian model of $p(\mathbf{x} \,|\, k)$ shown, parameterized by $\widehat{\boldsymbol{\mu}}_k$ and $\widehat{\boldsymbol{\Sigma}}$. The parameters $\widehat{\boldsymbol{\mu}}_k$ and $\widehat{\boldsymbol{\Sigma}}$, as well as $\widehat{\pi}_k$, are learned from training data, not shown in the figure. (Since $p = 1$, we only have a scalar variance $\boldsymbol{\Sigma}^2$, instead of a covaraince matrix $\boldsymbol{\Sigma}$). In the upper right panel is $\widehat{\pi}_k$, an approximation of $p(k)$, shown. These are used in Bayes' theorem to compute $P(k \,|\, x)$, shown in the bottom panel. We take the final prediction as the class which is modeled to have the highest probability, which means the topmost solid colored line in the bottom plot (e.g., the prediction for $\mathbf{x}_\star = 0.7$ would be $\widehat{y} = 2$ (green)). The decision boundaries (vertical dotted lines in the bottom plot) are hence found where the solid colored lines are intersecting.

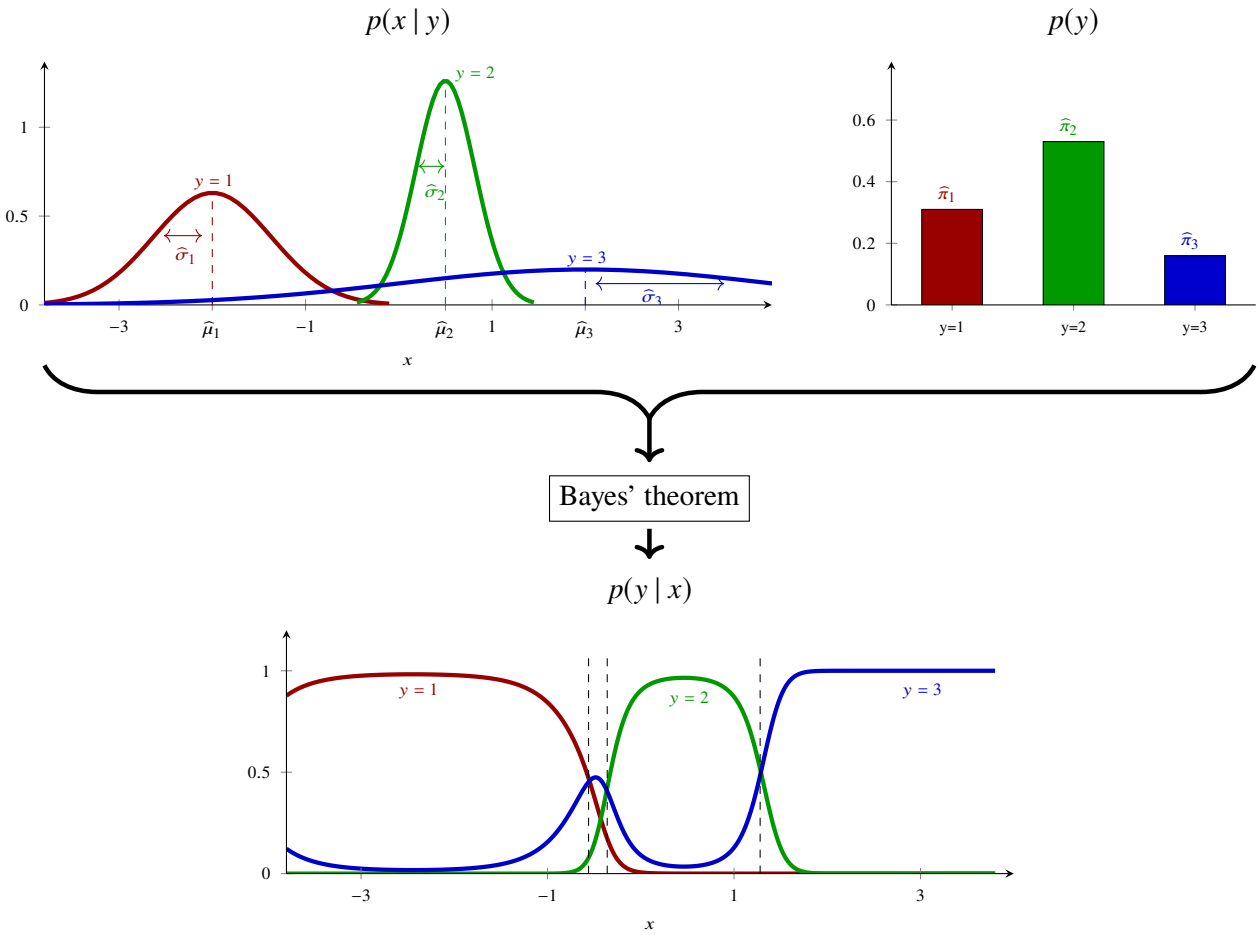


Figure 3.6: An illustration of QDA for $K = 3$ classes, in the same fashion as Figure 3.5. However, in contrast to LDA in Figure 3.5, is the learned variance $\widehat{\mathbf{\Sigma}}_k$ of $p(\mathbf{x} \mid k)$ different for different k (upper left panel). For this reason can the resulting decision boundaries (bottom panel) be more complicated than for LDA, note for instance the small slice of $\widehat{y} = 3$ (blue) inbetween $\widehat{y} = 1$ (red) and $\widehat{y} = 2$ (green) around -0.5.

Decision boundaries for LDA and QDA

Once we have learned the parameters from training data, we can compute $\widehat{y}_\star$ for a test input $\mathbf{x}_\star$ by inserting everything into (3.18) for each class k, and take the prediction as the class which is predicted to have the highest probability $p(y \mid \mathbf{x})$. As it turns out, the equations (3.18) and (3.19) are simple enough so that we can, by only using pen and paper, say something about the *decision boundary*, i.e., the boundary (in the input space) where the predictions shift between different classes.

If we note that neither the logarithm nor terms independent of k change the location of the maximizing argument ($\arg\max_k$), we can for LDA write

$$\begin{aligned}
\widehat{y}_{\text{LDA}} &= \arg\max_k p(y=k\,|\,\mathbf{x}) = \\
&= \arg\max_k \log p(y=k\,|\,\mathbf{x}) = \\
&= \arg\max_k \log\pi_k + \log\mathcal{N}\left(\mathbf{x}\,|\,\widehat{\boldsymbol{\mu}}_k, \widehat{\boldsymbol{\Sigma}}\right) - \log\left(\sum_{j=1}^{K}\pi_j\mathcal{N}\left(\mathbf{x}\,|\,\widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}\right)\right) = \\
&= \arg\max_k \log\pi_k + \log\mathcal{N}\left(\mathbf{x}\,|\,\widehat{\boldsymbol{\mu}}_k, \widehat{\boldsymbol{\Sigma}}\right) = \\
&= \arg\max_k \log\pi_k - \frac{1}{2}\log\det 2\pi\widehat{\boldsymbol{\Sigma}} - \frac{1}{2}\left(\mathbf{x}-\widehat{\boldsymbol{\mu}}_k\right)^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\left(\mathbf{x}-\widehat{\boldsymbol{\mu}}_k\right) = \\
&= \arg\max_k \underbrace{\log\pi_k - \frac{1}{2}\widehat{\boldsymbol{\mu}}_k^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\widehat{\boldsymbol{\mu}}_k + \mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\widehat{\boldsymbol{\mu}}_k}_{\triangleq\delta_k^{\text{LDA}}(\mathbf{x})}. \qquad (3.22)
\end{aligned}$$

The function $\delta_k^{\text{LDA}}(x)$ on the last row is sometimes referred to as the *discriminant function*. The points $\mathbf{x}$ on the *boundary* between two class predictions, say $k=0$ and $k=1$, is characterized by $\delta_0^{\text{LDA}}(\mathbf{x}) = \delta_1^{\text{LDA}}(\mathbf{x})$, i.e., the decision boundary between two classes 0 and 1 can be written as the set of points $\mathbf{x}$ which fulfills

$$\begin{aligned}
\delta_0^{\text{LDA}}(\mathbf{x}) &= \delta_1^{\text{LDA}}(\mathbf{x}) \Leftrightarrow \qquad (3.23)\\
\log\pi_0 - \frac{1}{2}\boldsymbol{\mu}_0^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\boldsymbol{\mu}_0 + \mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\boldsymbol{\mu}_0 &= \log\pi_1 - \frac{1}{2}\widehat{\boldsymbol{\mu}}_1^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\widehat{\boldsymbol{\mu}}_1 + \mathbf{x}^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}\widehat{\boldsymbol{\mu}}_1 \Leftrightarrow \\
\mathbf{x}^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\widehat{\boldsymbol{\mu}}_0 - \widehat{\boldsymbol{\mu}}_1) &= \underbrace{\log\widehat{\pi}_1 - \log\widehat{\pi}_0 - \frac{1}{2}\left(\boldsymbol{\mu}_1^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}\widehat{\boldsymbol{\mu}}_1 - \widehat{\boldsymbol{\mu}}_0^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}^{-1}\widehat{\boldsymbol{\mu}}_0\right)}_{\text{constant (independent of }\mathbf{x})}. \qquad (3.24)
\end{aligned}$$

From linear algebra, we know that $\{\mathbf{x} : \mathbf{x}^{\mathsf{T}}\mathbf{A} = \mathbf{c}\}$, for some matrix A and some constant c, defines a hyperplane in the $\mathbf{x}$-space. Thus, the decision boundary for LDA is always *linear*, and hence its name, *linear* discriminant analysis.

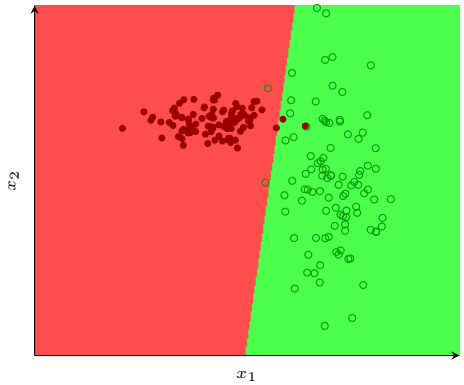
For QDA we can do a similar derivation

$$\widehat{y}_{\text{QDA}} = \arg\max_k \underbrace{\log\pi_k - \frac{1}{2}\log\det\widehat{\boldsymbol{\Sigma}}_k - \frac{1}{2}\boldsymbol{\mu}_k^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_k^{-1}\widehat{\boldsymbol{\mu}}_k + \mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_k^{-1}\widehat{\boldsymbol{\mu}}_k - \frac{1}{2}\mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_k^{-1}\mathbf{x}}_{\triangleq\delta_k^{\text{QDA}}(\mathbf{x})}. \qquad (3.25)$$

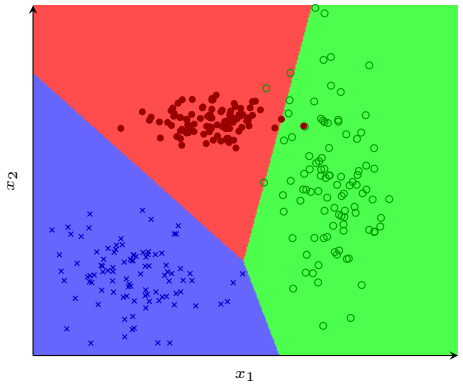
and set $\delta_0^{\text{QDA}}(\mathbf{x}) = \delta_1^{\text{QDA}}(\mathbf{x})$ to find the decision boundary as the set of points $\mathbf{x}$ for which

$$\begin{aligned}
&\log\widehat{\pi}_0 - \frac{1}{2}\log\det\widehat{\boldsymbol{\Sigma}}_0 - \frac{1}{2}\widehat{\boldsymbol{\mu}}_0^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_0^{-1}\widehat{\boldsymbol{\mu}}_0 + \mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_0^{-1}\widehat{\boldsymbol{\mu}}_0 - \frac{1}{2}\mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_0^{-1}\mathbf{x} \\
&= \log\pi_1 - \frac{1}{2}\log\det\widehat{\boldsymbol{\Sigma}}_1 - \frac{1}{2}\boldsymbol{\mu}_1^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_1^{-1}\widehat{\boldsymbol{\mu}}_1 + \mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_1^{-1}\widehat{\boldsymbol{\mu}}_1 - \frac{1}{2}\mathbf{x}^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_1^{-1}\mathbf{x} \\
&\Leftrightarrow \mathbf{x}^{\mathsf{T}}(\widehat{\boldsymbol{\Sigma}}_0^{-1}\widehat{\boldsymbol{\mu}}_0 - \widehat{\boldsymbol{\Sigma}}_1^{-1}\widehat{\boldsymbol{\mu}}_1) - \frac{1}{2}\mathbf{x}^{\mathsf{T}}(\widehat{\boldsymbol{\Sigma}}_0^{-1} + \widehat{\boldsymbol{\Sigma}}_1^{-1})\mathbf{x} \\
&= \underbrace{\log\widehat{\pi}_1 - \log\widehat{\pi}_0 - \frac{1}{2}\log\det\widehat{\boldsymbol{\Sigma}}_1 + \frac{1}{2}\log\det\widehat{\boldsymbol{\Sigma}}_0 - \frac{1}{2}\left(\widehat{\boldsymbol{\mu}}_1^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_1^{-1}\widehat{\boldsymbol{\mu}}_1 - \widehat{\boldsymbol{\mu}}_0^{\mathsf{T}}\widehat{\boldsymbol{\Sigma}}_0^{-1}\widehat{\boldsymbol{\mu}}_0\right)}_{\text{constant (independent of }\mathbf{x})}, \qquad (3.26)
\end{aligned}$$

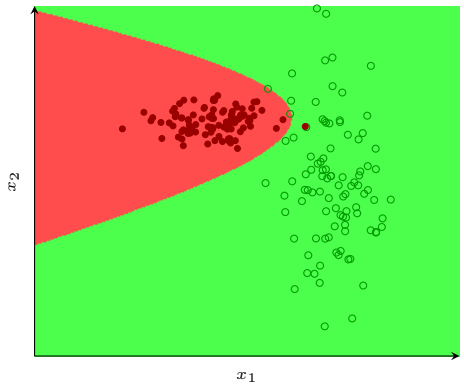
This is now on the format $\{\mathbf{x} : \mathbf{x}^{\mathsf{T}}\mathbf{A} + \mathbf{x}^{\mathsf{T}}\mathbf{B}\mathbf{x} = \mathbf{c}\}$, a *quadratic form*, and the decision boundary for QDA is thus always *quadratic* (and thereby also nonlinear!), which is the reason for its name *quadratic* discriminant analysis.



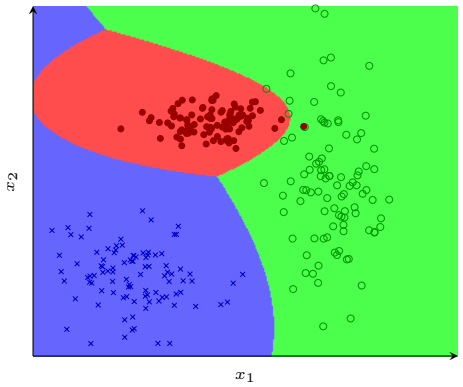
(a) LDA for $K = 2$ classes always gives a linear decision boundary. The red dots and green circles are training data from different classes, and the intersection between the red and green fields is the decision boundary obtained for an LDA classifier learned from the training data.



(b) LDA for $K = 3$ classes. We have now introduced training data from a third class, marked with blue crosses. The decision boundary between any two pair of classes is still linear.

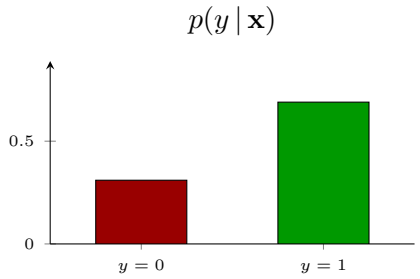


(c) QDA has quadratic (i.e., nonlinear) decision boundaries, as in this example where a QDA classifier is learned from the shown training data.

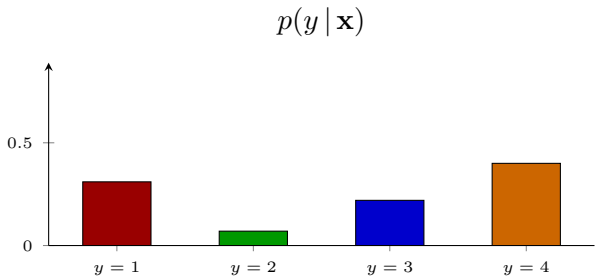


(d) With $K = 3$ classes are the decision boundaries for QDA possibly more complex than with LDA, as in this case (cf. (b)).

Figure 3.7: Examples of decision boundaries for LDA and QDA, respectively. This can be compared to Figure 3.3, where the decision boundary for logistic regression (with the same training data) is shown. LDA and logistic regression both have linear decision boundaries, but they are not identical.



(a) With $K = 2$ classes, Bayes' classifier tell us to take the class which has probability > 0.5 as the prediction $\widehat{y}$. Here, the prediction would therefore be $\widehat{y} = 1$.



(b) For $K = 4$ classes, Bayes' classifier tells us to take the prediction $\widehat{y}$ as the highest bar, which means $\widehat{y} = 4$ here. (In contrast to $K = 2$ classes in (a), it can happen that no class has probability > 0.5.)

Figure 3.8: Bayes' classifier: The probabilities $p(y \mid \mathbf{x})$ are shown as the height of the bars. Bayes' classifier says that if we want to make as few misclassificiations as possible, on the average, we should predict $\widehat{y}$ as the class which has highest probability.

3.4 Bayes' classifier — a theoretical justification for turning $p(y \mid \mathbf{x})$ into $\widehat{y}$

We have introduced logistic regression, LDA and QDA as three different classifiers. You have probably noted that they are all constructed as different ways of modeling $p(y \mid \mathbf{x})$, and we will now reason about an optimal (but only hypothetical!) classifier, referred to as the Bayes' classifier. In most cases, Bayes' classifier can *not* be implemented (we usually do not have enough information), but it tells us how we should convert $p(y \mid \mathbf{x})$ into actual predictions, and gives a coherent framework for the classifiers.

3.4.1 Bayes' classifier

Assume that we want to design a classifier which, on the average, makes *as few misclassification errors as possible*. That means that the predicted output label $\widehat{y}$ should equal the true output label y for as many test data points as possible. If we knew the probabilities $p(y \mid \mathbf{x})$ exactly (in logistic regression, LDA, QDA, and all other classifiers, we only have a model—a guess—for $p(y \mid \mathbf{x})$, we never know it exactly), then the optimal classifier is given by

$$\widehat{y} = \arg\max_k p(y = k \mid \mathbf{x}_\star). \tag{3.27}$$

Or, in words, the optimal classifier predicts $\widehat{y}$ as the label which has the highest probability given the input $\mathbf{x}$. The optimal classifier, equation (3.27), is the *Bayes' classifier*. This is illustrated by Figure 3.8. Let us first show why this is optimal, and then discuss how it connects to the other classifiers.

3.4.2 Optimality of Bayes' classifier

Making, on the average, as many correct predictions as possible means that we want $\widehat{y}$ (which is not random, but we can choose it ourselves) to be as likely as possible to equal y (which is random). How likely $\widehat{y}$ is to equal y can be expressed mathematically using the expected value over the distribution for the random variable y, which we write as $\mathbb{E}_{y \sim p(y \mid \mathbf{x})}$. Using the indicator function $\mathbb{I}\{\}$ (one when its argument is true, otherwise zero) we can write

$$\mathbb{E}_{y \sim p(y \mid \mathbf{x})} \left[\mathbb{I}\{\widehat{y} = y\} \right] = \sum_{k=1}^{K} \mathbb{I}\{\widehat{y} = k\} p(y = k \mid \mathbf{x}) = p(\widehat{y} \mid \mathbf{x}), \tag{3.28}$$

where we used the definition of expected value in the first step, and ignored all terms equal to zero in the second step. In order to make this quantity as large as possible, we can see that we should select $\widehat{y}$ such that $p(\widehat{y} \mid \mathbf{x})$ is as large as possible, which also is what we claimed in (3.27).

3.4.3 Bayes' classifier in practice: useless, but a source of inspiration

Bayes' classifier (3.27) makes use of the probability distribution $p(y \mid \mathbf{x})$, and thereby also assume that we *know* it. If we happen to have a problem where we actually know $p(y \mid \mathbf{x})$, we can of course use it, and we should look no further for other methods, since Bayes' classifier is optimal. In most machine learning problems, however, we do not know $p(y \mid \mathbf{x})$. In fact, *the entire point of machine learning is that we know very little about how* y *depends on* $\mathbf{x}$, other than what the training data tells us!

This does, however, not mean that Bayes' classifier is a useless concept. In fact, most of the classifiers in these notes can be understood as various *approximations* of Bayes' classifier, or put differently, that we have methods to learn (or rather estimate) $p(y \mid \mathbf{x})$ from the training data. Even though we have not introduced all methods yet, we give a brief overview of how some classifiers relate to (3.27):

- In binary logistic regression $p(y \mid \mathbf{x})$ is modeled as

$$\begin{cases} p(y = 1 \mid \mathbf{x}) = \frac{\exp(\boldsymbol{\beta}^T \mathbf{x})}{1+\exp(\boldsymbol{\beta}^T \mathbf{x})} \\ p(y = 0 \mid \mathbf{x}) = \frac{1}{1+\exp(\boldsymbol{\beta}^T \mathbf{x})} \end{cases} \tag{3.29}$$

- In linear and quadratic discriminant analysis (LDA and QDA), $p(y \mid \mathbf{x})$ is computed by Bayes' theorem (3.17), in which $p(\mathbf{x} \mid y)$ is assumed to be a Gaussian distribution (with mean and variance learned from the training data) and $p(y)$ is taken as the empirical distribution of the training data.
- In k-nearest neighbor (k-NN), $p(y \mid \mathbf{x})$ is modeled as the empirical distribution in the k-nearest samples in the training data.
- In tree-based methods, $p(y \mid \mathbf{x})$ is modeled as the empirical distribution among the training data samples in the same leaf node.
- In deep learning, $p(y \mid \mathbf{x})$ is modeled using a deep neural network and a softmax function.

All these classifiers use different ways to model/learn/approximate $p(y \mid \mathbf{x})$, and the default choice[6] is thereafter to predict $\widehat{y}$ according to Bayes' classifier (3.27), that is, pick the prediction as the class y which is modeled to have the highest probability $p(y \mid \mathbf{x})$. With only two classes this means that the prediction is taken as the class which is modeled to have probability > 0.5.

3.4.4 Is it always good to predict according to Bayes' classifier?

Even though Bayes' classifier usually is not available to us in practice (we only have the data, and it does not tell us what $p(y \mid \mathbf{x})$ looks like), it can still be used as an argument when turning a modeled/learned approximation of $p(y \mid \mathbf{x})$ into a prediction $\widehat{y}$. In fact, we have already done so throughout this chapter without questioning it further. Does Bayes' classifier mean that we should *always* choose the prediction $\widehat{y}$ as the class which our model assigns the highest probability? *No.* Even though (3.27) is a sensible default option to explore, we should be aware of the following facts

- Bayes' classifier is optimal only if the goal is to make as few misclassifications as possible. It might sound as an obvious goal, but that is not always the case! If we are to predict the health status of a patient, falsely predicting $\widehat{y} =$'well' might be much more severe than falsely predicting $\widehat{y} =$'bad' (or, perhaps, vice versa?). Sometimes the goal is therefore asymmetric, and Bayes' classifier (3.27) is not optimal for such situations.
- Bayes' classifier is guaranteed to be optimal only if we know $p(y \mid \mathbf{x})$ exactly. However, when we only have an approximation of $p(y \mid \mathbf{x})$, it is not guaranteed that (3.27) is the best thing to do anymore.

[6] Sometimes this is not very explicit in the method, but if you look carefully, you will find it.

3.5 More on classification and classifiers

3.5.1 Linear and nonlinear classifiers

A regression model which is linear in its parameters is called linear regression (Chapter 2). For the classification problem, the term "linear" is used differently; a *linear classifier* is a classifier whose decision boundary (for the problem with $K = 2$ classes) is linear, and a *nonlinear classifier* is a classifier which can have a nonlinear decision boundary. Among the classifiers introduced in this chapter, logistic regression and LDA are linear classifiers, whereas QDA is a nonlinear classifier, cf. Figure 3.3 and 3.7. Note that even though logistic regression and LDA both are linear classifiers, their decision boundaries are not identical. All classifiers that will follow in the subsequent chapters, except for decision stumps (Chapter 6), will be nonlinear.

As for linear regression (Section 2.4), it is possible to include nonlinear transformation of the inputs to create more features. With such transformations can the (seemingly inflexible?) linear classifier obtain rather complicated decision boundaries. It requires, however, the manual crafting and selection of nonlinear transformations. Instead, a more often used (and importantly more automatic) approach to build a complicated classifier from a simple one is boosting, which is introduced in Chapter 6.

3.5.2 Regularization

As with linear regression (Section 2.6), overfit might be a problem if n (the number of training data samples) is not much bigger than p (the number of inputs). We will define and discuss overfitting in more detail in Chapter 5. However, regularization can be useful also in classification to avoid overfit. A common regularization approach for logistic regression is a Ridge Regression-like penalty for $\boldsymbol{\beta}$, cf. (2.28). For LDA and QDA, it can be useful to regularize the covariance matrix estimation ((3.20d) and (3.20c)).

3.5.3 Evaluating binary classifiers

An important use of binary classification, i.e. $K = 2$, is to detect the presence of something, such as a disease, an object on the radar, etc. The convention is to let $y = 1$ ("positive") denote presence, and $y = 0$ ("negative") denote absence. Such applications have the important characteristics that

(i) Most of the data is usually $y = 0$, meaning that a classifier which always predicts $\widehat{y} = 0$ might score well if we only care about the number of correct classifications (accuracy). Indeed, a medical support system which always predicts "healthy" is probably correct most of the time, but nevertheless useless.

(i) A missed detection (predicting $\widehat{y} = 0$, when in fact $y = 1$) might have much more sever consequences than a false detection (predicting $\widehat{y} = 1$, when in fact $y = 0$).

For such classification problems, there is a set of analysis tools and terminology which we will introduce now.

Ratio	Name
FP/N	False positive rate, Fall-out, Probability of false alarm
TN/N	True negative rate, Specificity, Selectivity
TP/P	True positive rate, Sensitivity, Power, Recall, Probability of detection
FN/P	False negative rate, Miss rate
TP/P*	Positive predictive value, Precision
FP/P*	False discovery rate
TN/N*	Negative predictive value
FN/N*	False omission rate
P/n	Prevalence
(TN+TP)/n	Accuracy

Table 3.1: Common terminology related to the quantities (TN, FN, FP, TP) in the confusion matrix.

Confusion matrix

If one learns a binary classifier and evaluates it on a test dataset, a simple yet useful way to visualize the result is a *confusion matrix*. By separating the test data in four groups depending on y (the actual output) and $\widehat{y}$ (the output predicted by the classifier), we can make the following table

	$y = 0$	$y = 1$	*total*
$\widehat{y} = 0$	True neg (TN)	False neg (FN)	N*
$\widehat{y} = 1$	False pos (FP)	True pos (TP)	P*
total	N	P	n

Of course, TN, FN, FP, TP (and also N*, P*, N, P and n) should be replaced by the actual numbers, as will be seen in the next example. There is also a wide body of terminology related to the confusion matrix, which is summarized in Table 3.1.

The confusion matrix provides a quick and informative overview of the characteristics of a classifier. Depending on the application, it might be important to distinguish between false positive (FP, also called *type I error*) and false negative (FN, also called *type II error*). Ideally they both should be 0, but that is rarely the case in practice.

With the Bayes' classifier as a motivation, our default choice has been to convert $p(y = 1 \,|\, \mathbf{x})$ into predictions as

$$\begin{cases} \text{if } p(y = 1 \,|\, \mathbf{x}) \geq t & \text{let } \widehat{y} = 1 \\ \text{if } p(y = 1 \,|\, \mathbf{x}) < t & \text{let } \widehat{y} = 0 \end{cases} \tag{3.30}$$

with $t = 0.5$ as a threshold. If we, however, are interested in decreasing the false positive rate (at the expense of an increased false negative rate), we may consider to raise the threshold t, and vice versa.

ROC curve

As suggested by the example above, the tuning of the threshold t in (3.30) can be crucial for the performance in binary classification. If we want to compare different classifiers (say, logistic regression and QDA) for a certain problem beyond the specific choice of t, the *ROC curve* can be useful. The abbreviation ROC means "receiver operating characteristics", and is due to its history from communications theory.

To plot an ROC curve, the true positive rate (TP/P) is drawn against the false positive rate (FP/N) for all values of $t \in [0, 1]$. The curve typically looks as shown in Figure 3.9. An ROC curve for a perfect classifier (always predicting the correct value with full certainty) touches the upper left corner, whereas a classifier which only assigns random guesses gives a straight diagonal line.

A compact summary of the ROC curve is the *area under the ROC curve, AUC*. From Figure 3.9, we conclude that a perfect classifier has AUC 1, whereas a classifier which only assigns random guesses has AUC 0.5.

Example 3.2: Confusion matrix in thyroid disease detection

The thyroid is an endocrine gland in the human body. The hormones it produces influences the metabolic rate and the protein synthesis, and thyroid disorders may have serious implications. We consider the problem of detecting thyroid diseases, using the dataset provided by UCI Machine Learning Repository (Dheeru and Karra Taniskidou 2017). The dataset contains 7200 data points, each with 21 medical indicators as inputs (both qualitative and quantitative). It also contains the qualitative diagnosis $\{\texttt{normal}, \texttt{hyperthyroid}, \texttt{hypothyroid}\}$, which we convert into the binary problem with only $\{\texttt{normal}, \texttt{not normal}\}$ as outputs. The dataset is split into a training and test part, with 3772 and 3428 samples respectively. We train a logistic regression classifier on the training dataset, and use it for predicting the test dataset (using the default $t = 0.5$), and obtain the following confusion matrix:

	$y = \texttt{normal}$	$y = \texttt{not normal}$
$\widehat{y} = \texttt{normal}$	3177	237
$\widehat{y} = \texttt{not normal}$	1	13

Most test data points are correctly predicted as `normal`, but a large part of the `not normal` data is also falsely predicted as normal. This might indeed be undesired in the application.

To change the picture, we change the threshold to $t = 0.15$, and obtain new predictions with the following confusion matrix instead:

	$y = \texttt{normal}$	$y = \texttt{not normal}$
$\widehat{y} = \texttt{normal}$	3067	165
$\widehat{y} = \texttt{not normal}$	111	85

This change gives a significantly better true positive rate (85 instead of 13 patients are correctly predicted as `not normal`), but this happens at the expense of a worse false positive rate (111, instead of 1, patients are now falsely predicted as `not normal`). Whether it is a good trade-off depends, of course, on the specifics of the application: which type of error has the most severe consequences?

For this problem, only considering the total accuracy (misclassification rate) would not be very informative. In fact, the useless predictor of always predicting `normal` would give an accuracy of almost 93%, whereas the second confusion matrix above corresponds to an accuracy of 92%, even though it probably would probably be much more useful in practice.

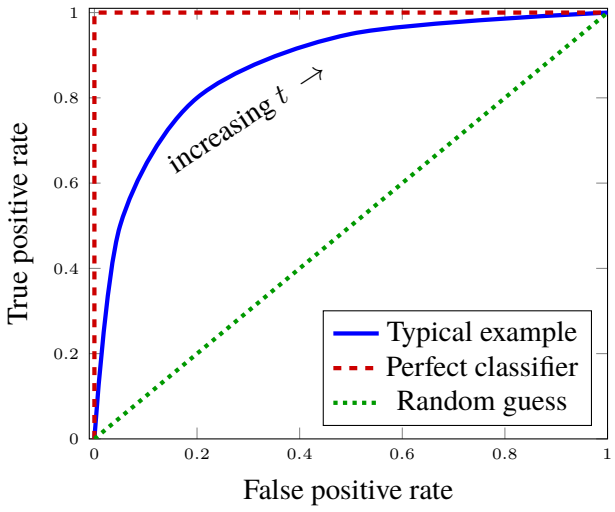


Figure 3.9: ROC curve

4 Non-parametric methods for regression and classification: *k*-NN and trees

The methods (linear regression, logistic regression, LDA and QDA) we have encountered so far all have a fixed set of parameters. The parameters are learned from the training data, and once the parameters are learned and stored, the training data is not used anymore and could be discarded. Furthermore, all those methods have had a fix structure; if the amount of training data increases the parameters can be estimated more accurately, with smaller variance, but the flexibility or expressiveness of the model does not increase; logistic regression can only describe linear decision boundaries, no matter how much training data that is available.

There exists another class of methods, not relying on a fixed structure and set of parameters, but which adapts more to the training data. Two methods in this class, which we will encounter now, are k-nearest neighbors (k-NN) and tree-methods. They can both be used for classification as well as regression, but we will focus our presentation on the classification problem.

4.1 *k*-NN

The name k-nearest neighbors (k-NN) is almost self-explanatory. To approximate $p(y \mid \mathbf{x}_\star)$, the proportion of the different classes among the k number of training data points closest to $\mathbf{x}_\star$ is used. The value k is a user-chosen integer ≥ 1 which controls the properties of the classifier, as discussed below. Formally, we define the set $R_\star = \{i : \mathbf{x}_i \text{ is one of the } k \text{ training data points closest to } \mathbf{x}_\star\}$ and can then express the classifier as

$$p(y = j|\mathbf{x}_\star) = \frac{1}{k} \sum_{i \in R_\star} \mathbb{I}\{y_i = j\} \tag{4.1}$$

for $j = 1, 2, \ldots, K$. Following Bayes' classifier and always predict the class which has the largest probability, k-NN simply amounts to a majority vote among the k nearest training data points. We explain this by the example on the next page.

Note that in contrast to the previous methods that we have discussed there is no parametric model to be trained, with which we then compute the prediction $\widehat{y}$. Therefore, we talk about k-NN as a non-parametric method. Instead, $\widehat{y}$ depends on the training data in a more direct fashion. The k-NN method can be summarized in the following algorithm.

Algorithm 4: k-nearest neighbor, k-NN

Data: Training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ (with output classes $1, \ldots, K$) and test input $\mathbf{x}_\star$
Result: Predicted test output $\widehat{y}$

1 Find the k training data point(s) $\mathbf{x}_i$ which has the shortest Euclidian distance $\|\mathbf{x}_i - \mathbf{x}_\star\|$ to $\mathbf{x}_\star$
2 Decide $\widehat{y}$ with a majority vote among those k nearest neighbors

Example 4.1: Predicting colors with k-NN

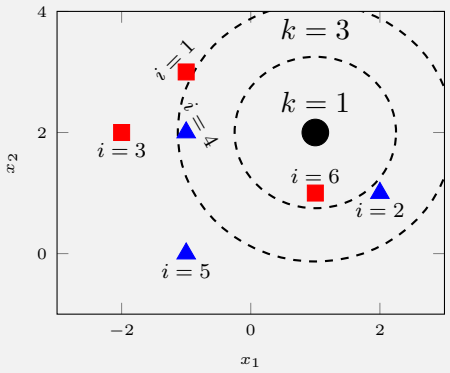
We are given a training data set with $n = 6$ observations of $p = 2$ input variables x_1, x_2 and one (qualitative) output y, the color Red or Blue,

i	x_1	x_2	y
1	-1	3	Red
2	2	1	Blue
3	-2	2	Red
4	-1	2	Blue
5	-1	0	Blue
6	1	1	Red

and we are interested in predicting the output for $\mathbf{x}_\star = [1\ 2]^\mathsf{T}$. For this purpose, we will explore two different k-NN classifiers, one using $k = 1$ and one using $k = 3$.

First, we compute the Euclidian distance $\|\mathbf{x}_i - \mathbf{x}_\star\|$ between each training data point $\mathbf{x}_i$ and the test data point $\mathbf{x}_\star$, and then sort them in descending order.

i	$\|\mathbf{x}_i - \mathbf{x}_\star\|$	y_i
6	$\sqrt{1}$	Red
2	$\sqrt{2}$	Blue
4	$\sqrt{4}$	Blue
1	$\sqrt{5}$	Red
5	$\sqrt{8}$	Blue
3	$\sqrt{9}$	Red



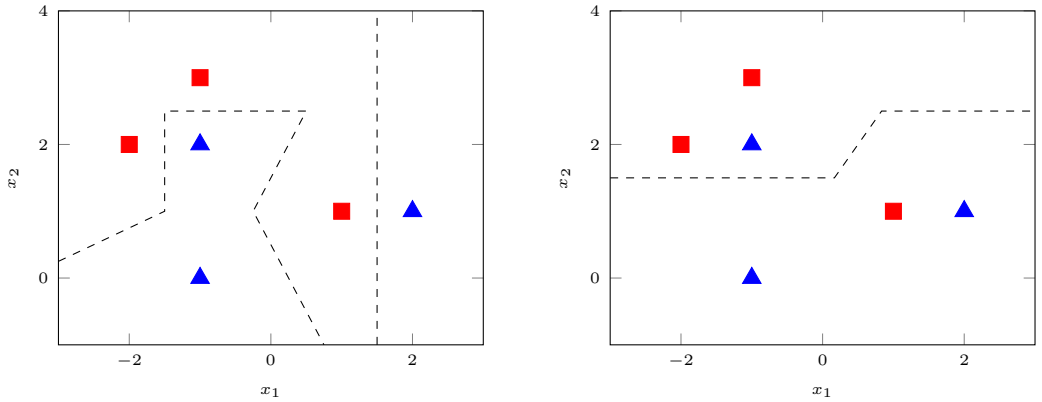
Since the closest training data point to $\mathbf{x}_\star$ is the data point $i = 6$ (Red), it means that for k-NN with $k = 1$, we get the model $p(\text{Red} \mid \mathbf{x}_\star) = 1$ and $p(\text{Blue} \mid \mathbf{x}_\star) = 0$. This gives the prediction $\widehat{y}_\star = \text{Red}$.

Further, for $k = 3$, the 3 nearest neighbors are $i = 6$ (Red), $i = 2$ (Blue), and $i = 4$ (Blue), which gives the model $p(\text{Red} \mid \mathbf{x}_\star) = \frac{1}{3}$ and $p(\text{Blue} \mid \mathbf{x}_\star) = \frac{2}{3}$. The prediction, which also can be seen as a majority vote among those 3 training data points, thus becomes $\widehat{y}_\star = \text{Blue}$.

This is also illustrated by the figure above where the training data points $\mathbf{x}_i$ are represented with red squares and the blue triangles depending on which class they belong to. The test data point $\mathbf{x}_\star$ is represented with a black filled circle. For $k = 1$ the closest training data point is identified by the inner circle and for $k = 3$ the three closest points are identified by the outer circle.

4.1.1 Decision boundaries for *k*-NN

In Example 4.1 we only computed a prediction for one single test data point $\mathbf{x}_\star$. If we would shift that test point by one step to the left at $\mathbf{x}_\star^{\text{alt}} = [0\ 2]^\mathsf{T}$ the three closest training data points would still include $i = 6$ and $i = 2$ but now $i = 2$ is exchanged for $i = 1$. For $k = 3$ this would give the approximation $p(\text{Red}|\mathbf{x}_\star) = \frac{2}{3}$ and we would predict $\widehat{y} = \text{Red}$. In between these two test data points $\mathbf{x}_\star$ and $\mathbf{x}_\star^{\text{alt}}$ at $[0.5\ 2]^\mathsf{T}$ it is equally far to $i = 1$ as to $i = 2$ and this point would consequently be located at the decision boundary between the two classes. Continuing this way of reasoning we can sketch the full decision boundaries in Example 4.1 which are displayed in Figure 4.1. Obviously, k-NN is not restricted to linear decision boundaries and is therefore a nonlinear classification method.



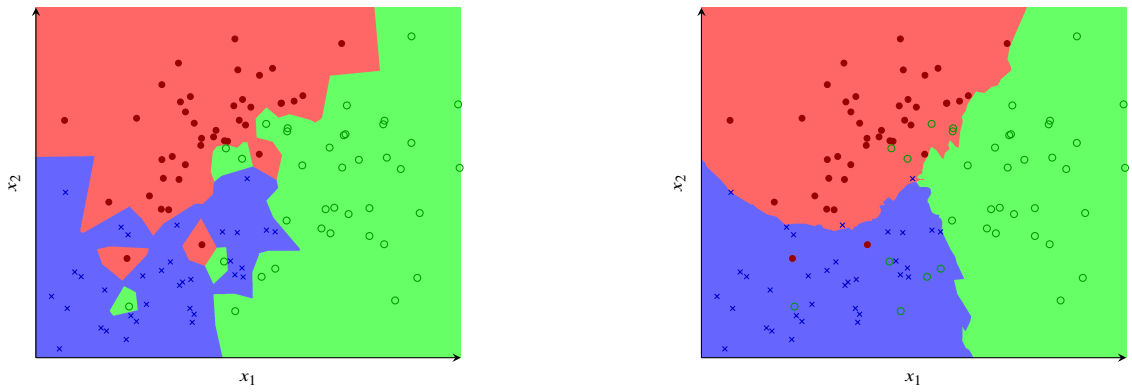
(a) $k = 1$ **(b)** $k = 3$

Figure 4.1: Decision boundaries for the problem in Example 4.1 for the two choices of the parameter k.

4.1.2 Choosing *k*

The user has to decide on which k to use in k-NN and this decision has a big impact on the final classifier. In Figure 4.2 another scenario is illustrated with $p = 2$ input variables, $K = 3$ classes and significantly more training data samples. In the two subfigures the decision boundaries for a k-NN classifier with $k = 1$ and $k = 11$ are illustrated.

By definition, with $k = 1$ all training data points are classified correctly and the boundaries are more adapted to what the training data exactly looks like (including its 'noise' and other random effects). With the averaging procedure that takes place for the k-NN classifier with $k = 11$, some training data points end up in the wrong region and the decision boundaries are less adapted to this specific realization of the training data. Even though k-NN with $k = 1$ fits all training data points perfectly, the one with $k = 11$ might be preferred, since it is less prone to *overfit* to the training data meaning there are good reasons to believe this model would perform better on test data. A systematic way of choosing k is to use cross-validation, and we will discuss these aspects more in Chapter 5.



(a) Decision boundary for k-NN with $k = 1$ for a 3-class problem. A complex, but possibly also overfitted, decision boundary.

(b) Decision boundary for k-NN with $k = 11$. A more rigid and less flexible decision boundary.

Figure 4.2: Decision boundaries for k-NN.

4.1.3 Normalization

Finally, one practical aspects crucial for the k-NN worth mentioning is the importance of normalization of the input data. Since k-NN is based on the Euclidean distances between points, it is important that these distances are a valid measure of the closeness between two data points. Imagine a training data

set with two input variables $\mathbf{x}_i = [x_{i1},\ x_{i2}]^\mathsf{T}$ where all values of x_{i1} are in the range $[0,\ 100]$ and the values for x_{i2} in the much smaller range $[0,\ 1]$. This could for example be the case if x_{i1} and x_{i2} represent different physical quantities (where the values can be quite different, depending on which unit is used). In this case the Euclidean distance between a test point $\mathbf{x}_\star$ and a training data point $\|\mathbf{x}_i - \mathbf{x}_\star\| = \sqrt{(x_{i1} - x_{\star 1})^2 + (x_{i2} - x_{\star 2})^2}$ would almost only depend on the first term $(x_{i1} - x_{\star 1})^2$ and the values of the second component x_{i2} would have a small impact.

One way to overcome this problem is to divide the first component with 100 and create $x_{i1}^{\text{new}} = x_{i1}/100$ such that both components are in the range $[0,\ 1]$. More generally, this normalization procedure for the input data can be written as

$$x_{ij}^{\text{new}} = \frac{x_{pi} - \min(x_{ij})}{\max(x_{ij}) - \min(x_{ij})}, \qquad \forall j = 1, \ldots, p, \quad i = 1, \ldots, n. \tag{4.2}$$

Another popular way of normalizing is by using the mean and standard deviation in the training data:

$$x_{ij}^{\text{new}} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}, \qquad \forall j = 1, \ldots, p, \quad i = 1, \ldots, n, \tag{4.3}$$

where $\bar{x}_j$ and σ_j are the mean and standard deviation for each input variable, respectively.

4.2 Trees

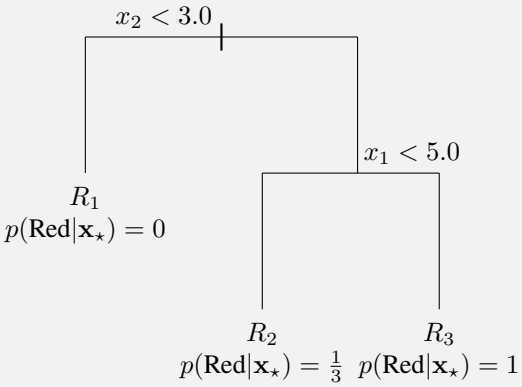
Tree-based methods divides the input space into different regions. Within each region, $p(y \mid \mathbf{x})$ is modeled as the empirical distribution among the training data samples in the same region. The rules to divide the input space can be summarized in a tree, and hence these methods are known as decision trees. Trees can be used for both regression and classification problems. Here, our description will focus on classification trees.

4.2.1 Basics

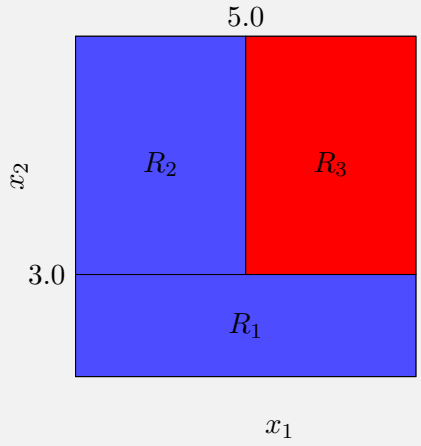
In a classification tree the function $p(y \mid \mathbf{x})$ is modeled with a series of rules on the input variables $x_1, \ldots, x_p$. These rules can be represented by a binary tree. This tree effectively divides the input space into multiple regions and in each region a constant value for the predicted class probability $p(y \mid \mathbf{x})$ is assigned. We illustrate this with an example.

Example 4.2: Predicting colors with a classification tree

Consider a problem with two input variables x_1 and x_2 and one quantitative output y, the color red or blue. A classification tree for this problem can look like the one below. To use this tree to classify a new point $\mathbf{x}_\star = [x_{\star 1},\, x_{\star 2}]^\mathsf{T}$ we will start at the top and work the way down until we reach the end of a branch. Each such final branch corresponds to a constant predicted class probability $p(\text{Red}|\mathbf{x}_\star)$.



A classification tree. At each internal node a rule on the form $x_j < s_k$ indicates the left branch coming from that split and the right branch then consequently corresponds to $x_j \geq s_k$. This tree as two internal nodes and three leaf nodes.



A region partition of the classification tree. Each region corresponds to a leaf node in the classification tree to the left, and each border between regions correspond to a split in the tree. Each region is marked with the color representing the highest predicted class probability.

A pseudo code for classifying a test input with the tree above would look like

```
if x_2 < 3.0 then
    return p(Red|x)=0
else
    if x_1 < 5.0 then
       return p(Red|x)=1/3
    else
        return p(Red|x)=1
    end
end
```

As an example if we have $\mathbf{x}_\star = [2.5,\, 3.5]^\mathsf{T}$, in the first split we would take the right branch since $x_{\star 2} = 3.5 \geq 3.0$ and in the second split we would take the left branch since $x_{\star 1} = 2.5 < 5.0$. Consequently, the for this test point we would get $p(\text{Red}|\mathbf{x}_\star) = 1/3$ and hence $p(\text{Blue}|\mathbf{x}_\star) = 2/3$. This classification tree can also be represented as splitting the input space into multiple rectacle-shaped regions. This is illustrated in the left figure above.

To set the terminology, the endpoint of each branch R_1, R_2 and R_3 in the example are called *leaf nodes* and the internal splits, $x_2 < 3.0$ and $x_1 < 5.0$ are known as *internal nodes*. The lines that connect the nodes are referred to as *branches*. Note that in an example with more than two input variables, the region partition (right figure in the example) is difficult to draw, but the tree will work out in the same way with exactly two branches coming out of each internal node.

This example illustrates how a classification tree can be used to make predictions, but how do we learn the tree from training data? This will be explained in the next section.

4.2.2 Training a classification tree

More mathematically, the classification tree models the class probability $p(k|\mathbf{x})$ as a constant c_{mk} in each region R_m and for each class $k = 1, 2, \ldots, K$:

$$p(y = k|\mathbf{x}) = \sum_{m=1}^{M} c_{mk} \mathbb{I}\{\mathbf{x} \in R_m\}, \tag{4.4}$$

where M is the total number of regions (leaf nodes) in the tree and where $\mathbb{I}\{\mathbf{x} \in R_m\} = 1$ if $\mathbf{x} \in R_m$ and 0 otherwise. Since the probabilities should sum up to 1 in each region we also have the constraint $\sum_{k=1}^{K} c_{mk} = 1$.

The overall goal in constructing a classification tree based on training data $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$ is to find a tree that makes the observed training data as likely as possible. This approach is known as the maximum likelihood method, which we also used to derive a solution to the logistic regression problem previously.

Maximizing the likelihood is equivalent of minimizing the negative logarithm of the likelihood. Therefore, we want to find a tree T which minimizes the following expression:

$$\begin{aligned} -\log \ell(T) &= -\log p(\mathbf{y}|\mathbf{X}, T) \\ &= -\sum_{i=1}^{n} \log p(y_i|\mathbf{x}_i) \\ &= -\sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{I}\{y_i = k\} \log p(k|\mathbf{x}_i). \end{aligned} \tag{4.5}$$

By inserting the model stated in (4.4) into (4.5), we get

$$\begin{aligned} -\log \ell(T) &= -\sum_{i=1}^{n} \sum_{k=1}^{K} \sum_{m=1}^{M} \log c_{mk} \mathbb{I}\{y_i = k\} \mathbb{I}\{\mathbf{x}_i \in R_m\} \\ &= -\sum_{m=1}^{M} n_m \sum_{k=1}^{K} \log c_{mk} \underbrace{\frac{1}{n_m} \sum_{i:\mathbf{x}_i \in R_m} \mathbb{I}\{y_i = k\}}_{\widehat{\pi}_{mk}} \\ &= -\sum_{m=1}^{M} n_m \sum_{k=1}^{K} \widehat{\pi}_{mk} \log c_{mk}. \end{aligned} \tag{4.6}$$

Here $\widehat{\pi}_{mk}$ is the proportion of training data points in region R_m that are from class k with n_m being the total number of training data points in region m. We can show[1] that

$$-\sum_{k=1}^{K} \widehat{\pi}_{mk} \log c_{mk} = \underbrace{\sum_{k=1}^{K} \widehat{\pi}_{mk} \log \frac{\pi_{mk}}{c_{mk}}}_{\geq 0} - \sum_{k=1}^{K} \widehat{\pi}_{mk} \log \widehat{\pi}_{mk} \geq -\sum_{k=1}^{K} \widehat{\pi}_{mk} \log \widehat{\pi}_{mk}. \tag{4.7}$$

which is fulfilled with equality if $c_{mk} = \widehat{\pi}_{mk}$. Hence, minimizing (4.6) with respect to c_{mk} gives $c_{mk} = \widehat{\pi}_{mk}$. It remains to find the regions R_m, and based on the discussion above we want to select the regions in order to minimize,

$$\min \sum_{m=1}^{M} n_m Q_m(T), \qquad \text{where} \qquad Q_m(T) = -\sum_{k=1}^{K} \widehat{\pi}_{mk} \log \widehat{\pi}_{mk} \tag{4.8}$$

[1] We use the so called log sum inequality and the two constraints $\sum_{k=1}^{K} c_{mk} = 1$ and $\sum_{k=1}^{K} \widehat{\pi}_{mk} = 1$ for all $m = 1, \ldots, M$.

is known as the *entropy* for region m.[2]

Finding the best tree T that minimizes (4.8) is, unfortunately, a combinatorial problem and hence computationally infeasible. Instead, we choose a greedy algorithm known as *recursive binary splitting* which does the minimization for each node split separately (instead of optimizing the entire tree at the same time). This approach starts in the top of the tree and successively splits the input where each split divides one branch into two new branches, and builds a tree similar to what we saw in the example above. This approach is greedy since it builds the tree by introducing only one split at a time, without having the full tree 'in mind'.

Consider the setting when we are about to do our first split into a pair of half-planes

$$R_1(j,s) = \{\mathbf{x} | x_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{\mathbf{x} | x_j > s\}.$$

The split depends on the index j of the input variable at which the split is performed and the cutpoint s. The corresponding proportions $\widehat{\pi}_{mk}$ will also depend on j and s.

$$\widehat{\pi}_{1k}(j,s) = \frac{1}{n_1} \sum_{i:\mathbf{x}_i \in R_1(j,s)} \mathbb{I}\{y_i = k\}, \qquad \widehat{\pi}_{2k}(j,s) = \frac{1}{n_2} \sum_{i:\mathbf{x}_i \in R_2(j,s)} \mathbb{I}\{y_i = k\}.$$

We seek the splitting variable j and cutpoint s that solve

$$\min_{j,s} \left[n_1 \left(-\sum_{k=1}^{K} \widehat{\pi}_{1k}(j,s) \log \widehat{\pi}_{1k}(j,s) \right) + n_2 \left(-\sum_{k=1}^{K} \widehat{\pi}_{2k}(j,s) \log \widehat{\pi}_{2k}(j,s) \right) \right]. \tag{4.9}$$

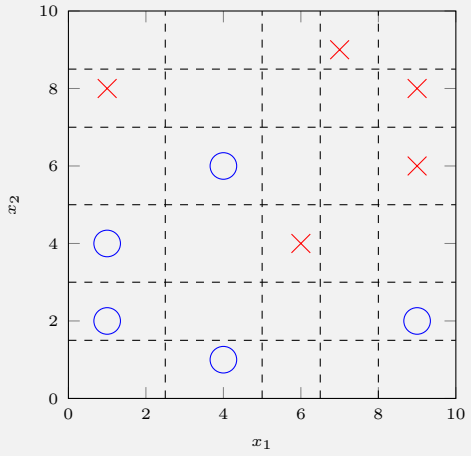
For each input variable we can scan through the finite number of possible splits and pick the pair $(j,\ s)$ which minimizes (4.9). After that, we repeat the process to create new splits by finding the best values $(j,\ s)$ for each of the new branches. We continue the process until some stopping criteria is reached, for example until no region contains more than five training data points.

The tree in Example 4.2 has been constructed based on the methodology outlined above, which we will illustrate in the example below.

Example 4.3: Learning a classification tree (continuation of Example 4.2)

We consider the same setup as in Example 4.2 with the following dataset

x_1	x_2	y
9.0	2.0	Blue
1.0	4.0	Blue
4.0	6.0	Blue
4.0	1.0	Blue
1.0	2.0	Blue
1.0	8.0	Red
6.0	4.0	Red
7.0	9.0	Red
9.0	8.0	Red
9.0	6.0	Red



We want to learn a classification tree, by using the entropy criteria in (4.8) and growing the tree until there are no regions with more than five data points left.

First split: There are infinitely many possible splits we can make, but all splits which gives the same partition of the data points will be the same. Hence, in practice we only have nine different

[2] If any $\widehat{\pi}_{mk}$ is equal to 0, the term $0 \log 0$ is taken to be zero, which is also consistent with the limit $\lim_{r \to 0^+} r \log r = 0$.

splits to consider in this dataset. The data and these splits (dashed lines) are visualized in the figure above.

We consider all nine splits in turn. We start with the split at $x_1 = 2.5$ which splits the input space into the two regions $R_1 = x_1 < 2.5$ and $R_2 = x_1 \geq 2.5$. In region R_1 we have two blue data points and one red, in total $n_1 = 3$ data points. The proportion of the two classes in region R_1 will therefore be $\widehat{\pi}_{1\text{B}} = 2/3$ and $\widehat{\pi}_{1\text{R}} = 1/3$. The entropy is calculated as

$$Q_1(T) = -\widehat{\pi}_{1\text{B}} \log(\widehat{\pi}_{1\text{B}}) - \widehat{\pi}_{1\text{R}} \log(\widehat{\pi}_{1\text{R}}) = -\frac{2}{3}\log(\frac{2}{3}) - \frac{1}{3}\log(\frac{1}{3}) = 0.64. \tag{4.10}$$

In region R_2 we have $n_2 = 7$ data points with the proportions $\widehat{\pi}_{2\text{B}} = 3/7$ and $\widehat{\pi}_{2\text{R}} = 4/7$. The entropy for this regions will be

$$Q_2(T) = -\widehat{\pi}_{2\text{B}} \log(\widehat{\pi}_{2\text{B}}) - \widehat{\pi}_{2\text{R}} \log(\widehat{\pi}_{2\text{R}}) = -\frac{3}{7}\log(\frac{3}{7}) - \frac{4}{7}\log(\frac{4}{7}) = 0.68 \tag{4.11}$$

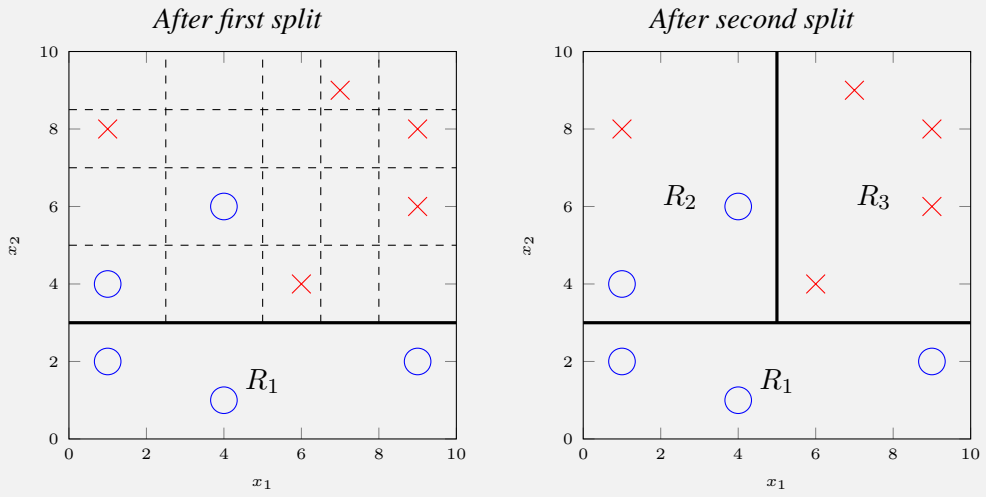
and the total weighted entropy for this split becomes

$$n_1 Q_1(T) + n_2 Q_2(T) = 3 \cdot 0.64 + 7 \cdot 0.68 = 6.69. \tag{4.12}$$

We compute the cost for all other splits in the same manner, and summarize it in the table below.

Split (R_1)	n_1	$\widehat{\pi}_{1\text{B}}$	$\widehat{\pi}_{1\text{R}}$	$Q_1(T)$	n_2	$\widehat{\pi}_{2\text{B}}$	$\widehat{\pi}_{2\text{R}}$	$Q_2(T)$	$n_1 Q_1(T) + n_2 Q_2(T)$
$x_1 < 2.5$	3	2/3	1/3	0.64	7	3/7	4/7	0.68	6.69
$x_1 < 5.0$	5	4/5	1/5	0.50	5	1/5	4/5	0.50	5.00
$x_1 < 6.5$	6	4/6	2/6	0.64	4	1/4	3/4	0.56	6.07
$x_1 < 8.0$	7	4/7	3/7	0.68	3	1/3	2/3	0.64	6.69
$x_2 < 1.5$	1	1/1	0/1	0.00	9	4/9	5/9	0.69	6.18
$x_2 < 3.0$	3	3/3	0/3	0.00	7	2/7	5/7	0.60	**4.18**
$x_2 < 5.0$	5	4/5	1/5	0.50	5	1/5	4/5	0.06	5.00
$x_2 < 7.0$	7	5/7	2/7	0.60	3	0/3	3/3	0.00	**4.18**
$x_2 < 8.5$	9	5/9	4/9	0.69	1	0/1	1/1	0.00	6.18

From the table we can read that the two splits at $x_2 < 3.0$ and $x_2 < 7.0$ are both equally good. We choose to continue with $x_2 < 3.0$.



Second split: We notice that only R_2 has more than five data points. Also there is no point splitting region R_1 further since it only contains data points from the same class. In the next step we therefore split the second region into two new regions R_2 and R_3. All possible splits are displayed above to the left (dashed lines) and we compute their cost in the same manner as before.

Splits (R_1)	n_2	$\widehat{\pi}_{2\text{B}}$	$\widehat{\pi}_{2\text{R}}$	$Q_2(T)$	n_3	$\widehat{\pi}_{3\text{B}}$	$\widehat{\pi}_{3\text{R}}$	$Q_3(T)$	$n_2Q_2(T) + n_3Q_3(T)$
$x_1 < 2.5$	2	1/2	1/2	0.69	5	1/5	4/5	0.50	3.89
$x_1 < 5.0$	3	2/3	1/3	0.63	4	0/4	4/4	0.00	**1.91**
$x_1 < 6.5$	4	2/4	2/4	0.69	3	0/3	3/3	0.00	2.77
$x_1 < 8.0$	5	2/5	3/5	0.67	2	0/2	2/2	0.00	3.37
$x_2 < 5.0$	2	1/2	1/2	0.69	5	1/5	4/5	0.50	3.88
$x_2 < 7.0$	4	2/4	2/4	0.69	3	0/3	3/3	0.00	2.77
$x_2 < 8.5$	6	2/6	4/6	0.64	1	0/1	1/1	0.00	3.82

The best split is the one at $x_1 < 5.0$ visualized above to the right. The final tree and partition were displayed in Example 4.2. None of the three regions has more than five data points. Therefore, we terminate the training.

If we want to use the tree for prediction, we get $p(\text{Red}|\mathbf{x}_\star) = \widehat{\pi}_{1\text{R}} = 0$ if $\mathbf{x}_\star$ falls into region R_1, $p(\text{Red}|\mathbf{x}_\star) = \widehat{\pi}_{2\text{R}} = 1/3$ if falls into region R_2 or $p(\text{Red}|\mathbf{x}_\star) = \widehat{\pi}_{3\text{R}} = 1$ if falls into region R_3, in the same manner as displayed in Example 4.2.

4.2.3 Other splitting criteria

There are other splitting criteria that can be considered instead of the entropy (4.8). One simple alternative is the *misclassification rate*

$$Q_m(T) = 1 - \max_k \widehat{\pi}_{mk}, \tag{4.13}$$

which is simply the proportion of data points in region R_m which do not belong to the most common class. This sounds like a reasonable choice since it is often the misclassification rate that we use to evaluate the final classifier. However, one drawback with is that it does not favor pure nodes in the same extent as the entropy criteria does. With pure nodes we mean nodes where most data points belong to a certain class. It is usually an advantage to favor pure nodes in the greedy procedure that we use to grow the tree, since it can lead to a total of fewer splits.

For example, consider the first split in Example 4.3. If we would use the misclassification rate as splitting criteria, both the split $x_2 < 5.0$ as well as $x_2 < 3.0$ would provide a total misclassification rate of 0.2. However, the split at $x_2 < 3.0$, which the entropy criteria favored, provides a pure node R_1. If we would go with the split $x_2 < 5.0$ the misclassification after the second split would still be 0.2. If we would continue to grow the tree until no data points are misclassified we would need three splits if we used the entropy criteria whereas we would need five splits if we would use the misclassification criteria and started with the split at $x_2 < 5.0$.

Another common splitting criteria is the *Gini index*

$$Q_m(T) = \sum_{k=1}^{K} \widehat{\pi}_{mk}(1 - \widehat{\pi}_{mk}). \tag{4.14}$$

Similar to the entropy criteria, Gini index favors node purity more than misclassification rate does.

If we consider two classes where r is the proportion in the second class, the three criteria are

$$\begin{aligned} \text{Misclassification rate:} \quad & Q_m(T) = 1 - \max(r, 1-r) \\ \text{Gini index:} \quad & Q_m(T) = 2r(1-r) \\ \text{Entropy/deviance:} \quad & Q_m(T) = -r\log r - (1-r)\log(1-r) \end{aligned}$$

These functions are shown in Figure 4.3. We can se that the entropy and Gini index are quite similar.

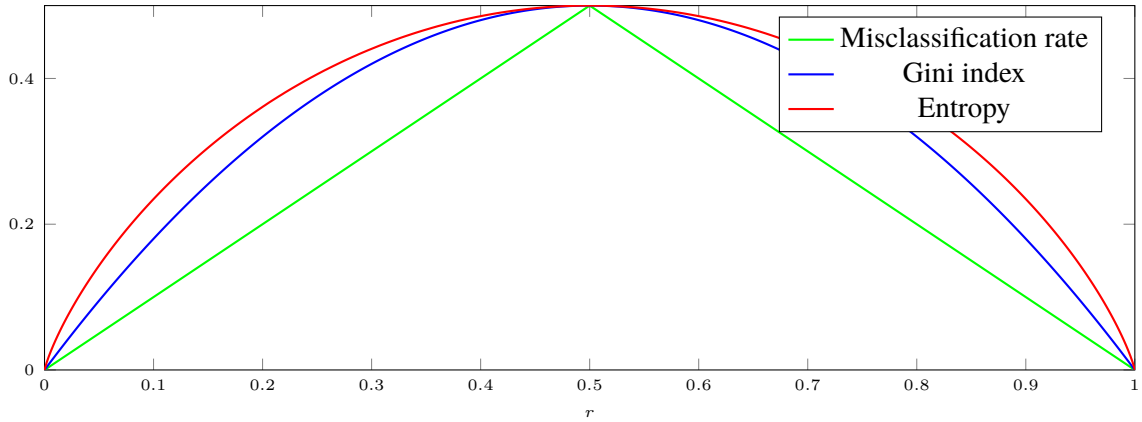


Figure 4.3: Three splitting criteria for classification trees as a function of the proportion in class 2. The entropy criteria has been scaled such that it passes through (0.5,0.5).

4.2.4 Regression trees

Trees can also be used for regression. Instead of the aforementioned splitting criteria we typically use the mean squared error as splitting criteria

$$Q_m(T) = \frac{1}{n_m} \sum_{j \in R_m} (y_i - \widehat{y}_m)^2, \qquad \text{where} \qquad \widehat{y}_m = \frac{1}{n_m} \sum_{i \in R_m} y_i. \tag{4.15}$$

This splitting criteria can also be motivated from a maximum likelihood point of view as we did for the classification trees. At prediction time the mean $\widehat{y}_m$ in each region is used for prediction. In all other aspects the procedure to train a regression tree is the same as training a classification tree as explained above.

5 How well does a method perform?

So far we have studied different methods of how to learn models by adapting them to training data. We hope that the models thereby will give us good predictions also when faced with new, previously unseen, data. But can we really expect that to work? This may first sound like a trivial question, but on second thought it is perhaps not (so) obvious anymore, and we will give it the attention it deserves in this chapter. By doing so, we will find some interesting concepts, which will give us practical tools for evaluating and understanding supervised machine learning methods better.

5.1 Expected new data error E_{new}: performance in production

We start by introducing some concepts and notation. First, we define an error function $E(\widehat{y}, y)$ encoding the purpose of classification or regression. The error function compares a prediction $\widehat{y}$ to a measured data point y, and returns a small value (possibly zero) if $\widehat{y}$ is a good prediction of y, and a larger value otherwise: The worse the prediction, the larger the value. There are many different error functions that could be considered, but our choices in this chapter will be:

$$E(\widehat{y}, y) \triangleq \begin{cases} 0 & \text{if } \widehat{y} = y \\ 1 & \text{if } \widehat{y} \neq y \end{cases} \qquad \text{(classification)} \tag{5.1a}$$

$$E(\widehat{y}, y) \triangleq (\widehat{y} - y)^2 \qquad \text{(regression)} \tag{5.1b}$$

The error function $E(\widehat{y}, y)$ has similarities to a loss function. However, they are used differently: A loss function is used to *train* (or learn) a model, whereas the error function is used to *analyze performance* of an already trained model.

In the end every machine learning mostly cares about how a method performs when faced with an endless stream of new, unseen data. Imagine for example all real-time recordings of street views that have to be processed by a vision system in a self-driving car once it is sold to a customer, or all new patients that have to be classified by a medical diagnosis system. The performance on fresh unseen data can in mathematical terms be understood as the average of the error function—how often the classifier is right, or how good does the regression method predict. To be able to mathematically describe the endless stream of new data, we introduce a *distribution over data* $p(\mathbf{x}, y)$. In the previous chapters, we have mostly considered the output y as a random variable whereas the inputs $\mathbf{x}$ have been thought of as fixed. Now, we have to think of also the input $\mathbf{x}$ as a random variable with a certain probability distribution. In any real-world machine learning scenario $p(\mathbf{x}, y)$ can be extremely complicated and really hard or even impossible to model. That is, however, not a problem since we will only use $p(\mathbf{x}, y)$ to *reason* about machine learning methods, and the bare notion of $p(\mathbf{x}, y)$ will be helpful for that.

Remark 5.1 *In Section 3.4 about Bayes' classifier, we made a hypothetical argument about the optimal classifier,* if *we had access to $p(y \mid \mathbf{x})$ (which we usually do not have). The arguments in this section are made from an even more hypothetical point of view, assuming that we do know not only $p(y \mid \mathbf{x})$, but also $p(\mathbf{x})$, since $p(\mathbf{x}, y) = p(y \mid \mathbf{x})p(\mathbf{x})$. While this is often an unrealistic assumption, the reasoning will lead us to useful insights about when and how we can expect machine learning to work.*

Irrespective of which classification or regression method we are considering, once the model has been trained on training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, it will provide us with predictions $\widehat{y}_\star$ for any new input $\mathbf{x}_\star$ we give to it. We will in this chapter write $\widehat{y}(\mathbf{x}; \mathcal{T})$ as a function of $\mathbf{x}$ and $\mathcal{T}$, like $\widehat{y}_\star \triangleq \widehat{y}(\mathbf{x}_\star; \mathcal{T})$, to indicate that the prediction (via the model) depends both on the value of the test input $\mathbf{x}_\star$ and on the training data $\mathcal{T}$ used to train the model.

In the previous chapters, we have mostly discussed how a model predicts one, or a few, test inputs $\mathbf{x}_\star$. Let us now take that to the next level, by *integrating the error function* (5.1) *over all possible test data points* (rather than considering only one or a few) with respect to the distribution $p(\mathbf{x}, y)$. We refer to this as the *expected new data error*

$$\boxed{E_{\text{new}} \triangleq \mathbb{E}_\star \left[E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) \right],} \tag{5.2}$$

where the expectation $\mathbb{E}_\star$ is the expectation over all possible test data points with respect to the distribution $(\mathbf{x}_\star, y_\star) \sim p(\mathbf{x}, y)$, that is,

$$\mathbb{E}_\star \left[E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) \right] = \int E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) p(\mathbf{x}_\star, y_\star) \, d\mathbf{x}_\star dy_\star. \tag{5.3}$$

We emphasize that the model (no matter whether it is linear regression, a decision tree, a neural network or something else) is trained on a given training data set $\mathcal{T}$ and represented by $\widehat{y}(\cdot; \mathcal{T})$. What is happening in equation (5.2) is an averaging over possible test data points $(\mathbf{x}_\star, y_\star)$. Thus, E_{new} describes how well the model *generalizes* from the training data $\mathcal{T}$ to new situations.

The expected new data error E_{new} tells us how well a method performs when we put it into production; what proportions of predictions a classifier will get right, and how well a regression method will predict in terms of average squared error. Or, in a more applied setting, what rate of false and missed detections of pedestrians we can expect a vision system in a self-driving car to make, or how big a proportion of all future patients a medical diagnosis system will get wrong.

The overall goal in supervised machine learning is to achieve as small E_{new} as possible.

Unfortunately, in practical cases we can never compute E_{new} to assess how well we are doing. The reason is that $p(\mathbf{x}, y)$—which we do not know in practice—is part of the definition of E_{new}. It seems, however, to be a too important construction to be abandoned, just because we cannot compute it. We will instead spend the remaining parts of this chapter trying to *estimate* E_{new} (essentially by replacing the integral with a sum), and also understanding how E_{new} behaves, to better understand how we can decrease it.

Remark 5.2 *Note that E_{new} is a property of a trained model* and *a specific machine learning problem. Thus, we cannot talk about "E_{new} for QDA" in general, but instead we have to make more specific statements, like "E_{new} for QDA on handwritten digit recognition, when QDA is trained with the MNIST data[1]".*

[1] `http://yann.lecun.com/exdb/mnist/`

5.2 Estimating E_{new}

There are multiple reasons for a machine learning engineer to be interested in E_{new}, such as:

- judging if the performance is satisfying (whether E_{new} is small enough), or if more work should be put into the solution and/or more training data should be collected
- choosing between different methods
- choosing hyperparameters (such as k in k-NN, the regularization parameter in ridge regression or the number of hidden layers in deep learning)
- reporting the expected performance to the customer

As discussed above, we can unfortunately not compute E_{new} in any practical situation. We will therefore explore some possibilities to *estimate* E_{new}, which eventually will lead us to a very useful concept known as cross-validation.

5.2.1 $E_{\text{train}} \not\approx E_{\text{new}}$: We cannot estimate E_{new} from training data

Let us start with defining the *average training error*,

$$\boxed{E_{\text{train}} \triangleq \frac{1}{n}\sum_{i=1}^{n} E(\widehat{y}(\mathbf{x}_i; \mathcal{T}), y_i)}, \tag{5.4}$$

where $\{\mathbf{x}_i, y_i\}_{i=1}^n$ is the training data $\mathcal{T}$. E_{train} simply describes how well a method performs on the training data on which it was trained. In contrast to E_{new}, we can always compute E_{train}.

We usually assume that $\mathcal{T}$ consists of samples from $p(\mathbf{x}, y)$. This assumption means that the training data is collected under similar circumstances as the ones the learned model will be used under, which seems reasonable. (If it was not, we would have very little reasons to believe the training data would tell us anything useful.) When an integral is hard to compute, it can be numerically approximated with a sum (see details in Appendix A.2). Now, the question is if the integral in E_{new} can be well approximated by the sum in E_{train}, like

$$E_{\text{new}} = \int E(\widehat{y}(\mathbf{x}; \mathcal{T}), y))p(\mathbf{x}, y)d\mathbf{x}dy \overset{??}{\approx} \frac{1}{n}\sum_{i=1}^{n} E(\widehat{y}(\mathbf{x}_i; \mathcal{T}), y_i) = E_{\text{train}}. \tag{5.5}$$

Or, put differently: Can we expect a method to perform equally well (or badly) when faced with new, previously unseen, data, as it did on the training data?

The answer is, unfortunately, **no**.

> ***Time to reflect 5.1:*** *Why can we not expect the performance on training data (E_{train}) to be a good approximation for how a method will perform on new, previously unseen data (E_{new})?*

Equation (5.5) does *not* hold, and the reason is that the training data are not just any data points, but $\widehat{y}$ depends on them since they are used for training the model. We cannot therefore expect (5.5) to hold. (Technically, the conditions in Appendix A.2 are not fulfilled, since $\widehat{y}$ depends on $\mathcal{T}$.)

As we will discuss more thoroughly later in Section 5.3.1, the average behavior of E_{train} and E_{new} is, in fact, typically $E_{\text{train}} < E_{\text{new}}$. That means that a method usually performs worse on new, unseen data, than on training data. *The performance on training data is therefore not a good measure of E_{new}.*

5.2.2 $E_{\text{test}} \approx E_{\text{new}}$: We can estimate E_{new} from test data

We could not use the "replace-the-integral-with-a-finite-sum" trick to estimate E_{new} by E_{train}, due to the fact that it effectively meant using the training data twice: first, to train the model ($\widehat{y}$ in (5.4)) and second,

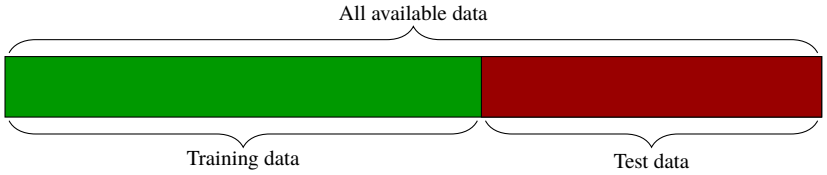


Figure 5.1: The test data set approach: If we split the available data in two sets and train the model on the training set, we can compute E_{test} using the test set. The more data that are in the test data set, the less variance (better estimate) in E_{test}, but the less data left for training the model. The split here is only pictorial, in practice one should always split the data randomly.

to evaluate the error function (the sum in (5.4)). A remedy is to set aside some *test data* $\{\mathbf{x}_j, y_j\}_{j=1}^{m}$, which are not used for training, and then use the test data only for estimating the model performance

$$\boxed{E_{\text{test}} \triangleq \frac{1}{m}\sum_{j=1}^{m} E(\widehat{y}(\mathbf{x}_j; \mathcal{T}), y_j)}. \tag{5.6}$$

In this way, not all data will be used for training, but some data points (the test data) will be saved and used only for computing E_{test}. This is illustrated by Figure 5.1.

Be aware! *If you are splitting your data into a training and test set, always do it randomly! Someone might—intentionally or unintentionally—have sorted the data set for you. If you do not split randomly, you might end up having only one class in your training data, and another class in your test data . . .*

As long as $m \geq 1$, it can be shown that E_{test} is an unbiased estimate of E_{new} (meaning that if the entire procedure is repeated multiple times, the average value of E_{test} would be E_{new}). That is reassuring, but it does not tell us how close E_{test} will be to E_{new} in a single experiment. However, the variance of E_{test} decreases when the size of test data m increases; a small variance of E_{test} means that we can expect it to be close to E_{new}. Thus, if we take the test data set big enough, E_{test} will be close to E_{new}. On the other side the amount of data available is usually limited in real machine learning problems, meaning the more data points we put into the test set, the fewer data points are left for training. Typically the more training data, the smaller E_{new} (which we will discuss later in Section 5.3). Achieving a small E_{new} is our ultimate goal. We are therefore faced with the following dilemma: *the better we want to know* E_{new} (more test data gives less variance in E_{test}), *the worse we have to make it* (less training data increases E_{new}). That is not very satisfying.

One could suggest the following two-step procedure, to circumvent the situation:

(i) Split the available data in one training and one test set, train the model on the training data and compute E_{test} using test data (as in Figure 5.1).

(ii) Train the model again, this time using the entire data set.

By such a procedure, we get both a value of E_{test} and a model trained using the entire data set. That is not bad, but not perfect either. Why? To achieve small variance in E_{test}, we have to put lots of data in the test data set. That means the model trained in step (i) will quite possibly be very different from the model trained in step (ii). And E_{test} from step (i) is an estimate of E_{new} for the model from step (i), not the model in step (ii). Consequently, if we use E_{test} from step (i) to, e.g., select a hyperparameter in step (i), we can not be sure it was a wise choice for the model trained in step (ii). This procedure is, however, not very far from cross-validation that we will present next.

5.2.3 Cross-validation: $E_{\text{val}} \approx E_{\text{new}}$ without setting aside test data

We would like to use all available data to train a model, and at the same time have a good estimate of E_{new} for that model. After reading the previous section, that might appear as an impossible request. There is, however, a clever solution called *cross-validation*.

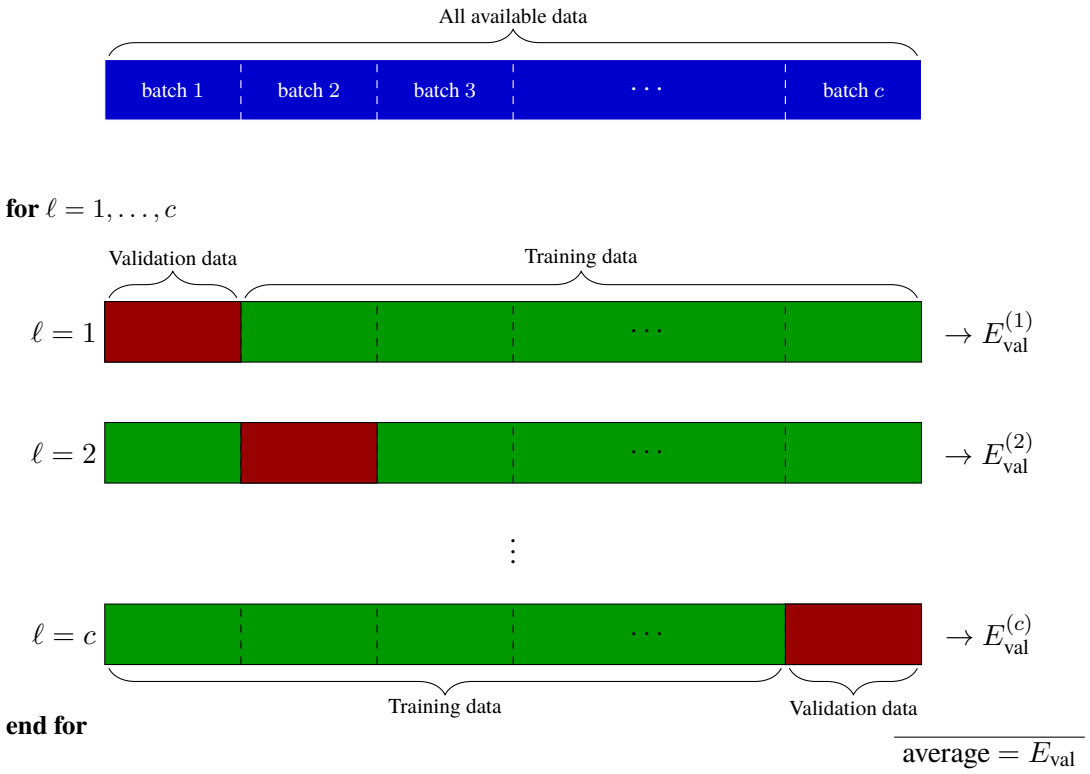


Figure 5.2: Illustration of c-fold cross-validation. The data is split in c batches of similar sizes. When looping over $\ell = 1, 2, \ldots, c$, batch ℓ is held out as validation data, and the model is trained on the remaining $c - 1$ data batches. Each time, the trained model is used to compute the average error $E_{\text{val}}^{(\ell)}$ for the validation data. The final model is trained using all available data, and the estimate of E_{new} for that model is E_{val}, the average of all $E_{\text{val}}^{(\ell)}$.

The idea of cross-validation is simply to repeat the test data set approach (using a small test data set) multiple times with a *different* test data set each time, in the following way:

(i) split the data set in c batches of similar size (see Figure 5.2), and let $\ell = 1$

(ii) take batch ℓ as validation data, and the remaining batches as training data

(iii) train the model on the training data, and compute $E_{\text{val}}^{(\ell)}$ as the average error on the validation data (analogously to (5.6))

(iv) if $\ell < c$, set $\ell \leftarrow \ell + 1$ and return to (ii). If $\ell = c$, compute

$$\boxed{E_{\text{val}} \triangleq \frac{1}{c} \sum_{\ell=1}^{c} E_{\text{val}}^{(\ell)}} \tag{5.7}$$

(v) train the model again, this time using all available data points

More precisely, this procedure is known as *c-fold cross-validation*, and illustrated in Figure 5.2.

With c-fold cross-validation, we get a model which is trained on all data, as well as an approximation of E_{new} for that model, namely E_{val}. Whereas E_{test} (Section 5.2.2) was an unbiased estimate of E_{new} (to the cost of setting aside test data), E_{val} is only approximately unbiased. However, with c large enough, it turns out to often be a sufficiently good approximation, and is commonly used in practice. Let us try to understand how c-fold cross-validation works.

First, we have to distinguish between the final model, which is trained on all data in step (v), and the intermediate models which are trained on all except a $1/c$ fraction of the data in step (iii). The key in

c-fold cross-validation is that if c is large enough, the intermediate models are quite similar to the final model (since they are trained on almost the same data set, only a fraction $1/c$ of the data is missing). Furthermore, each intermediate $E_{\text{val}}^{(\ell)}$ is by construction an unbiased estimate of E_{new} for its corresponding intermediate model. Since the intermediate and final models are similar, $E_{\text{val}}^{(\ell)}$ is *approximately* also an unbiased estimate of E_{new} for the final model. Since the validation sets are small (only $1/c$ of all available data), the variance of $E_{\text{val}}^{(\ell)}$ is high, but when averaging over multiple high-variance estimates $E_{\text{val}}^{(1)}, E_{\text{val}}^{(2)}, \ldots, E_{\text{val}}^{(c)}$, the final estimate E_{val} (5.7) does not suffer from high variance.

We usually talk about training (or learning) as something that is done once. However, in c-fold cross-validation the training is repeated c (or even $c+1$) times. A common value for c is 10, but you may of course try different values. For methods such as linear regression, the actual training (solving the normal equations) is usually done within milliseconds on modern computers, and doing it an extra c times is usually not really a problem. If one is working with computationally heavy methods, such as certain deep neural networks, it is perhaps less appealing to increase the computational load by a factor of $c+1$. However, if one wants to get a good estimate of E_{new}, some version of cross-validation is most often needed.

We now have a method for estimating E_{new} for a model trained on all available training data. A typical use of cross-validation is to select different types of hyperparameters, such as k in k-NN or a regularization parameter.

Be aware! *For the same reason as with the test data approach, it is important to always split the data randomly for cross-validation to work! A simple solution is to first randomly permute the entire data set, and thereafter split it into batches.*

5.3 Understanding E_{new}

Cross-validation is an important and powerful tool for *estimating* E_{new}. Designing a method with small E_{new} is the goal in supervised machine learning, and it is therefore useful to also have a tool to estimate how we are doing. However, more can be said to also understand what affects E_{new}. To be able to reason about E_{new}, we have to introduce another abstraction level, namely the *training-data averaged* versions of E_{new} and E_{train},

$$\bar{E}_{\text{new}} \triangleq \mathbb{E}_{\mathcal{T}}\left[E_{\text{new}}\right], \tag{5.8a}$$

$$\bar{E}_{\text{train}} \triangleq \mathbb{E}_{\mathcal{T}}\left[E_{\text{train}}\right]. \tag{5.8b}$$

Here, $\mathbb{E}_{\mathcal{T}}$ denotes the expected value when the training data set $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ (of a fixed size n) is drawn from $p(\mathbf{x}, y)$. Thus $\bar{E}_{\text{new}}$ *is the average E_{new} if we would train the model multiple times on different training data sets*, and similarly for $\bar{E}_{\text{train}}$. The point of introducing these, as it turns out, is that we can say more about the *average* behavior of E_{new} and E_{train}, than we can say about E_{new} and E_{train} when the model is trained on one specific training data set $\mathcal{T}$. Even though we in practical problems very seldom encounter $\bar{E}_{\text{new}}$ (the training data is usually fixed), the insights we gain from studying $\bar{E}_{\text{new}}$ are still useful.

5.3.1 $E_{\text{new}} = E_{\text{train}} +$ generalization error

We have already discussed the fact that E_{train} cannot be used in estimating E_{new}. In fact, it usually holds that

$$\bar{E}_{\text{train}} < \bar{E}_{\text{new}}, \tag{5.9}$$

Put in words, this means that on average, a method usually performs worse on new, unseen data, than on training data. A methods ability to perform well on unseen data after being trained on training data, can be understood as the method's ability to *generalize* from training data. The difference between E_{new} and E_{train} is accordingly called the *generalization error*[2], as

$$\text{generalization error} \triangleq E_{\text{new}} - E_{\text{train}}. \tag{5.10}$$

The generalization error thereby gives a connection between the performance on training data and the performance 'in production' on new, previously unseen data. It can therefore be interesting to understand how big (or small) the generalization error is.

Generalization error and model complexity

The size of the generalization error depends on the method and the problem. Concerning the method, one can typically say that *the more the model has adopted to the training data, the larger the generalization error*. A theoretical study of how much a model adopts to training data can be done using the so-called VC dimension, eventually leading to probabilistic bounds on the generalization error. Unfortunately those bounds are usually rather conservative, and we will not pursue that formal approach any further.[3] Instead, we only use the vague term *model complexity*, by which we mean the ability of a method to adopt to complicated patterns in the training data, and reason about what we see in practice. A model with high complexity (such as a neural network) can describe very complicated relationships, whereas a model with low complexity (such as LDA) is less flexible in what functions it can describe. For parametric methods, the model complexity is related to the number of parameters that are trained. Flexible non-parametric methods (such as trees with many leaf nodes or k-NN with small k) have higher model complexity than parametric methods with few parameters, etc. Techniques such as regularization, early stopping and dropout (for neural networks) effectively decrease the model complexity.

[2] Sometimes E_{new} is called generalization error; not in this text. In our terminology we do not distinguish between the generalization error for a model trained on a certain training data set, and its training-data averaged counterpart.

[3] If you are interested, a good book is Abu-Mostafa, Magdon-Ismail, and Lin 2012.

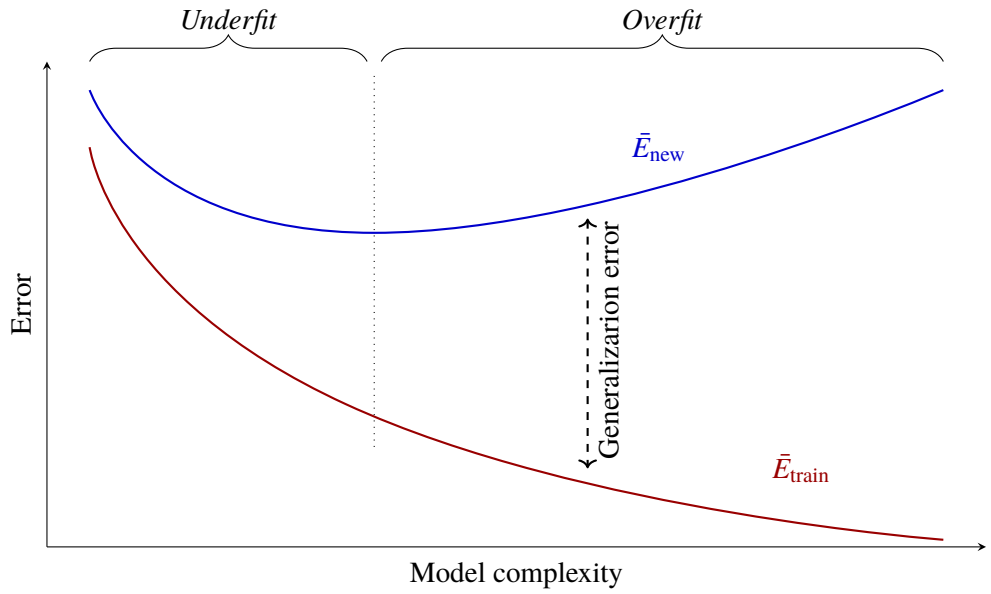


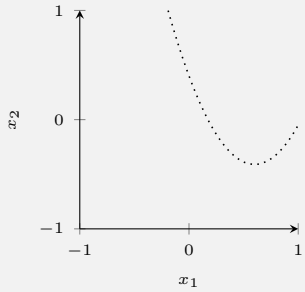
Figure 5.3: Behavior of $\bar{E}_{\text{train}}$ and $\bar{E}_{\text{new}}$ for many supervised machine learning methods, as a function of model complexity. We have not made a formal definition of complexity, but a rough proxy is the number of parameters that are learned from the data. The difference between the two curves is the generalization error. In general, one can expect $\bar{E}_{\text{train}}$ to decrease as the model complexity increases, whereas $\bar{E}_{\text{new}}$ typically has a U-shape. If the model is so complex that $\bar{E}_{\text{new}}$ is larger than it had been with a less complex model, the term *overfit* is commonly used. Somewhat less common is the term *underfit* used for the opposite situation. The level of model complexity which gives the minimum $\bar{E}_{\text{new}}$ (at the dotted line) would in a consistent terminology perhaps be called a balanced fit. A method with a balanced fit is usually desirable, but often hard to find since we do not know neither $\bar{E}_{\text{new}}$ nor E_{new} in practice.

Typically, *higher model complexity implies larger generalization error*. Furthermore, $\bar{E}_{\text{train}}$ usually decreases as the model complexity increases, whereas $\bar{E}_{\text{new}}$ attains a minimum for some intermediate model complexity value: too small *and* too high model complexity both raises $\bar{E}_{\text{new}}$. This is illustrated in Figure 5.3. The region where $\bar{E}_{\text{new}}$ is larger than its minimum due to too high model complexity is commonly referred to as *overfit*. The other region (where $\bar{E}_{\text{new}}$ is larger than its minimum due to too small model complexity) is sometimes referred to as *underfit*. In a consistent terminology, the point where $\bar{E}_{\text{new}}$ attains it minimum could be referred to as a balanced fit. Since the goal is to minimize $\bar{E}_{\text{new}}$, we are interested in finding this point. We also illustrate this by Example 5.1.

Remark 5.3 *This and the next section discuss the usual behavior of $\bar{E}_{new}$, $\bar{E}_{train}$ and the generalization error. We use the term 'usually' because there are so many supervised machine learning methods and problems that it is almost impossible to make any claim that is* always *true for all possible situations. Pathological counter-examples may exist. One should also keep in mind that claims about $\bar{E}_{train}$ and $\bar{E}_{new}$ are about the* average *behavior, which hopefully is clear in Example 5.1.*

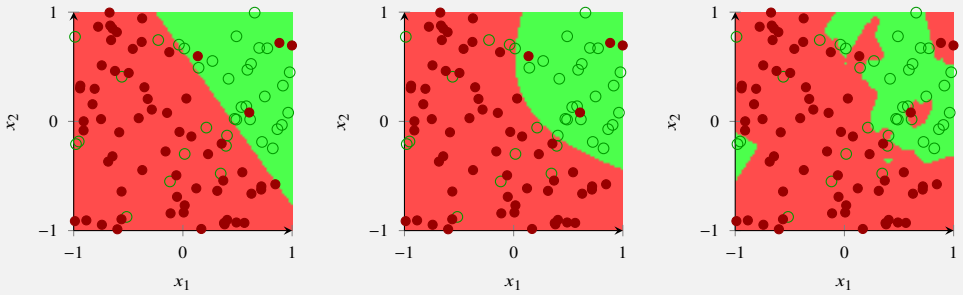
Example 5.1: E_{test} and E_{new} in a simulated example

We consider a simulated binary classification example with two-dimensional inputs $\mathbf{x}$. On the contrary to all real world machine learning problems, in a simulated problem like this we can actually compute E_{new}, since we do know $p(\mathbf{x}, y)$ (otherwise we could not make the simulation). In this example, $p(\mathbf{x})$ is a uniform distribution on the square $[-1, 1]^2$, and $p(y \mid \mathbf{x})$ is defined as follows: all points above the dotted curve in the figure below are green with probability 0.9, and points below the curve are red with probability 0.9. *(The dotted line is also the decision boundary for Bayes' classifier. Why? And what would E_{new} be for Bayes' classifier?)*



We generate $n = 100$ samples as training data, and learn three classifiers: a logistic regression classifier, a QDA classifier and a k-NN classifier with $k = 2$. If we are to rank these methods in model complexity order, logistic regression is simpler than QDA (logistic regression is a linear classifier, whereas QDA is more general), and QDA is simpler than k-NN (since k-NN is non-parametric and can have rather complicated decision boundaries). We plot their decision boundaries, together with the training data:

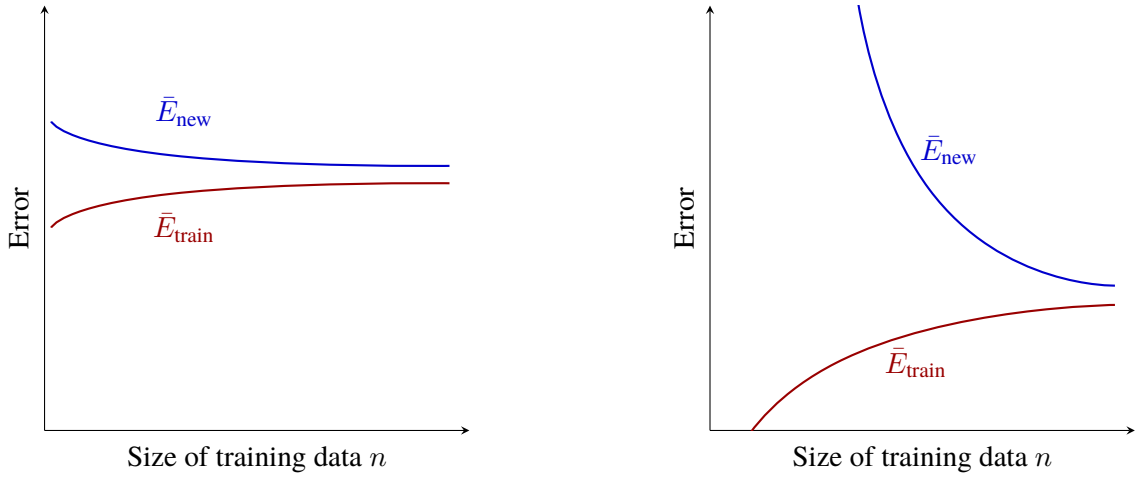
Logistic regression QDA k-NN, $k = 2$



For each of these three classifiers, we can compute E_{train} by simply counting the fraction of training data points that are on the wrong side of the decision boundary. From left to right, we get $E_{\text{train}} = 0.17, 0.16, 0.11$. Since we are in a simulated example, we can also access E_{new} (or rather estimate it numerically by simulating a lot of test data), and from left to right we get $E_{\text{new}} = 0.22, 0.15, 0.24$. This pattern resembles Figure 5.3, except for the fact that E_{new} is smaller than E_{train} for QDA. Is this unexpected? Not really, what we have discussed in the main text is the *average* $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$, *not* the situation with E_{new} and E_{train} for one particular set of training data. We therefore repeat this experiment 100 times, and compute the average $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$ over those 100 experiments:

	Logistic regression	QDA	k-NN with $k = 2$
$\bar{E}_{\text{train}}$	0.17	0.14	0.10
$\bar{E}_{\text{new}}$	0.18	0.15	0.19

This follows Figure 5.3 well: The generalization error (difference between $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$) is positive and increases with model complexity, $\bar{E}_{\text{train}}$ decreases with model complexity, and $\bar{E}_{\text{new}}$ has its minimum for QDA. This suggests that k-NN with $k = 2$ suffers from overfitting for this problem, whereas logistic regression is a case of underfitting.



(a) Simple model

(b) Complex model

Figure 5.4: Typical relationship between $\bar{E}_{\text{new}}$, $\bar{E}_{\text{train}}$ and the number of data points n in the training data set. The generalization error (difference between $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$) decreases, at the same time as $\bar{E}_{\text{train}}$ increases. Typically, a more complex model (right panel) will for large enough n attain a smaller $\bar{E}_{\text{new}}$ than a simpler model (left panel) would on the same problem (the axes of the figures are comparable). However, the generalization error is typically larger for a more complex model, in particular when there is little training data n.

Generalization error and size n of training data

The previous section and Figure 5.3 are concerned about the relationship between $\bar{E}_{\text{new}}$, $\bar{E}_{\text{train}}$, the generalization error (their difference) and the model complexity. Yet another important aspect is the size of the training data set, n. Intuitively, one may expect that the more training data, the better the possibilities to learn how to generalize. Yet again, we do not make a formal derivation, but we can in general expect that *the more training data, the smaller the generalization error*. On the other hand, $\bar{E}_{\text{train}}$ typically increases as n increases, since most models are not able to fit all training data perfectly if there are too many of them. A typical behavior of $\bar{E}_{\text{train}}$ and $\bar{E}_{\text{new}}$ is sketched in Figure 5.4.

5.3.2 E_{new} = bias^2 + variance + irreducible error

We will now introduce another decomposition of E_{new} into the terms known as *bias* and *variance* (which we can affect by our choice of method) as well as an unavoidable component of irreducible noise. This decomposition is most natural in the regression setting, but the intuition carries over also to classification.

We first make the assumption that the true relationship between input and output can be described as some (possibly very complicated) function $f(\mathbf{x})$ plus independent noise ε,

$$y = f(\mathbf{x}) + \varepsilon, \quad \text{with } \mathbb{E}[\varepsilon] = 0 \text{ and } \text{var}(\varepsilon) = \sigma^2. \tag{5.11}$$

Since we have made no restriction on f, this is not a very restrictive assumption, but we can expect it to describe the reality well.

In our notation, $\widehat{y}(\mathbf{x}; \mathcal{T})$ represents the model when it is trained on training data $\mathcal{T}$. We now also introduce the *average trained model*

$$g(\mathbf{x}) \triangleq \mathbb{E}_{\mathcal{T}}\left[\widehat{y}(\mathbf{x}; \mathcal{T})\right]. \tag{5.12}$$

As before, $\mathbb{E}_{\mathcal{T}}$ denotes the expected value over training data drawn from $p(\mathbf{x}, y)$. Thus, $g(\mathbf{x})$ is the (hypothetical) average model we would achieve, if we could marginalize all random effects associated with the training data.

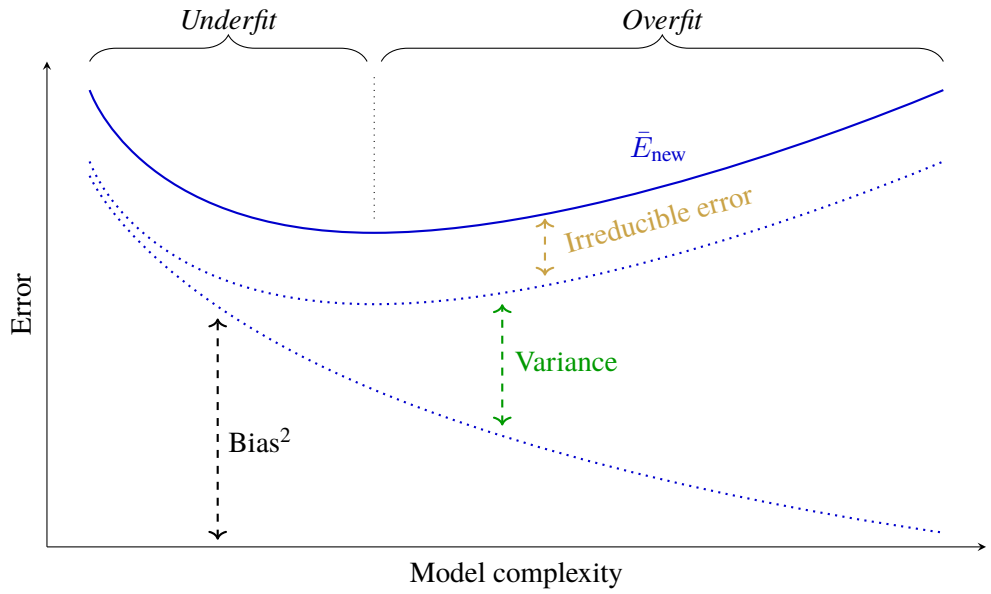


Figure 5.5: The bias-variance decomposition of $\bar{E}_{\text{new}}$ (cf. Figure 5.3). The bias typically decreases with model complexity; the more complicated the model is, the less systematic errors in the predictions. The variance, on the other hand, typically increases as the model complexity grows; the more complex the model is, the more it will adapt to peculiarities that by chance happened to occur in the particular training data set that was used. The irreducible error is always constant. In order to achieve a small E_{new}, one has to trade between bias and variance (for example by using another model, or using regularization as in Example 5.2). in order to avoid over- and underfit.

We are now ready to rewrite $\bar{E}_{\text{new}}$, the average expected new data error, as

$$\begin{aligned}\bar{E}_{\text{new}} &= \mathbb{E}_{\mathcal{T}}\left[\mathbb{E}_{\star}\left[\left(\widehat{y}(\mathbf{x}_\star;\mathcal{T})-y_\star\right)^2\right]\right] = \mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}\left[\left(\widehat{y}(\mathbf{x}_\star;\mathcal{T})-f(\mathbf{x}_\star)-\varepsilon\right)^2\right]\right] \\ &= \mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}\left[\left(\widehat{y}(\mathbf{x}_\star;\mathcal{T})\right)^2\right] - 2\mathbb{E}_{\mathcal{T}}\left[\widehat{y}(\mathbf{x}_\star;\mathcal{T})\right] f(\mathbf{x}_\star) + f(\mathbf{x}_\star)^2\right] + \sigma^2 \\ &= \mathbb{E}_{\star}\Big[\underbrace{\mathbb{E}_{\mathcal{T}}[(\widehat{y}(\mathbf{x}_\star;\mathcal{T}))^2] - g(\mathbf{x}_\star)^2}_{\mathbb{E}_{\mathcal{T}}\left[(\widehat{y}(\mathbf{x}_\star;\mathcal{T})-g(\mathbf{x}_\star))^2\right]} + \underbrace{g(\mathbf{x}_\star)^2 - 2g(\mathbf{x}_\star)f(\mathbf{x}_\star) + f(\mathbf{x}_\star)^2}_{(g(\mathbf{x}_\star)-f(\mathbf{x}_\star))^2}\Big] + \sigma^2. \end{aligned} \tag{5.13}$$

Here, we used the fact that ε is independent of $\mathbf{x}$, as well as $\mathbb{E}_{\mathcal{T}}\left[g(\mathbf{x})\right] = g(\mathbf{x})$.

The term $(g(\mathbf{x}_\star) - f(\mathbf{x}_\star))^2$ describes in a sense how much the model—if it could be 'perfectly trained' with an infinite amount of training data—differs from the true $f(\mathbf{x}_\star)$. Hence, we will refer to this term *bias*^{2} (reads "squared bias").

The other term, $\mathbb{E}_{\mathcal{T}}[(\widehat{y}(\mathbf{x}_\star;\mathcal{T}) - g(\mathbf{x}_\star))^2]$, captures how much the model $\widehat{y}(\mathbf{x};\mathcal{T})$ varies each time it is trained on a new training data set. If this term is small, the trained model is not very sensitive to exactly which data points happened to be in the training data, and vice versa. We return to (5.13),

$$\bar{E}_{\text{new}} = \underbrace{\mathbb{E}_{\star}\left[\left(g(\mathbf{x}_\star) - f(\mathbf{x}_\star)\right)^2\right]}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}\left[\left(\widehat{y}(\mathbf{x}_\star;\mathcal{T}) - g(\mathbf{x}_\star)\right)^2\right]\right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\substack{\text{Irreducible} \\ \text{error}}}. \tag{5.14}$$

The irreducible error is simply an effect of the assumed intrinsic stochasticity of the problem – it is not possible to predict ε since it is truly random. We will hence leave the irreducible error as it is, and focus on the bias and variance terms to further understand how $\bar{E}_{\text{new}}$ is affected by our choice of method; there are interesting situations where one can decrease $\bar{E}_{\text{new}}$ by trading bias for variance or vice versa.

The bias-variance trade-off and its relation to model complexity

We continue with the (never properly defined) notion of model complexity. High model complexity means that the model is able to express more complicated functions, implying that the bias term is small. On the other hand, the more complex a model is, the more it will adapt to training data $\mathcal{T}$—not only to the interesting patterns, but also to the actual data points and noise that happened to be in that realization of the training data. Had there been another realization of $\mathcal{T}$, the trained model could have looked (very) differently. Exactly such 'sensitivity' to training data is described by the variance term. In summary, if everything else is fixed and only the model complexity increases, the variance also increases, but the bias decreases. The optimal model complexity (smallest E_{new}) is therefore usually 'somewhere in the middle', where the model has a good trade-off between bias and variance. This is illustrated by Figure 5.5.

One should remember that the model complexity is not simply the number of parameters in the model, but rather a measure of how much the model adapts to complicated patterns in the training data. We introduced regularization in Section 2.6 as a method to counteract overfit, or effectively decreasing the model complexity, without changing the number of parameters in the model. Regularization therefore gives a tool for changing the model complexity in a continuous fashion, which opens up for fine-tuning of this bias-variance trade-off. This is further explored in Example 5.2

Example 5.2: Regularization—trading bias for variance

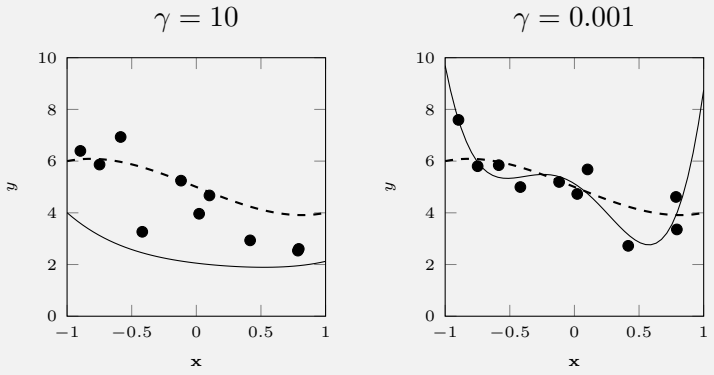
Let us consider a simulated regression example. We let $p(\mathbf{x})$ and $p(y \mid \mathbf{x})$ be defined as $\mathbf{x} \sim \mathcal{U}[0, 1]$ and

$$y = 5 - 2\mathbf{x} + \mathbf{x}^3 + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1).$$

We let the training data consist of only $n = 10$ samples. We now try to model the data using linear regression with a 4th order polynomial, as

$$y = \beta_0 + \beta_1 \mathbf{x} + \beta_2 \mathbf{x}^2 + \beta_3 \mathbf{x}^3 + \beta_4 \mathbf{x}^4 + \varepsilon,$$

where we assume ε to have a Gaussian distribution (which happens to be true in this example), so that we will end up with the normal equations. Since the model contains the true model and least squares would not introduce any systematic errors, the bias term (5.14) would be exactly zero. However, learning 5 parameters from only 10 data points will almost inevitably lead to very high variance and overfit, so we decide to train the model with a regularized method, namely Ridge Regression. Using regularization means that we will trade the unbiasedness (regularization introduces systematic bias in how the model is trained) for smaller variance. Two examples on what it could look like, for different regularization parameters, are shown:

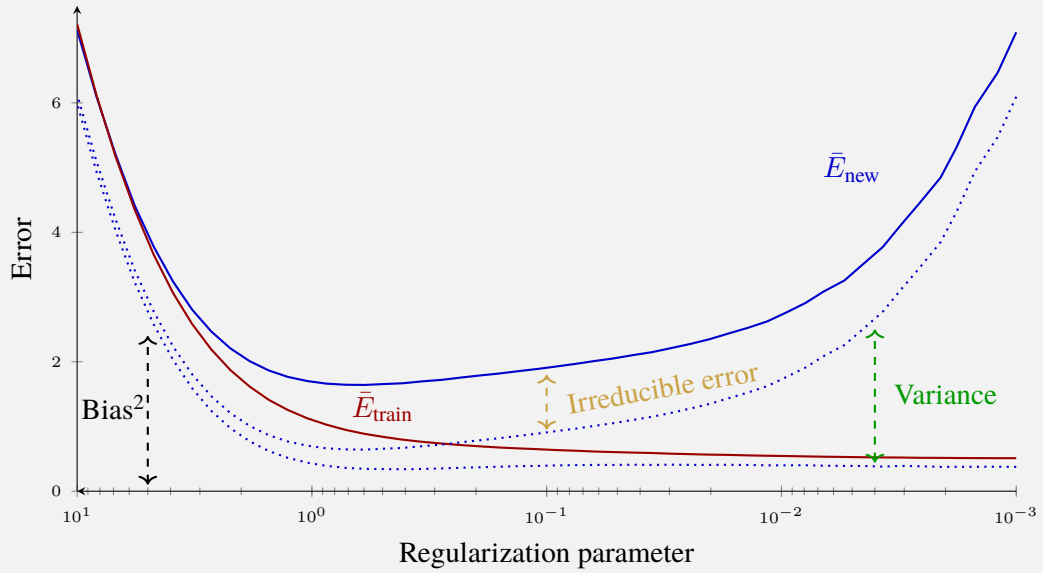


The dots are the $n = 10$ data points, the solid line is the trained model and the dashed line is the true model. In the case with $\gamma = 0.001$, the plot suggests overfit, whereas $\gamma = 10$ seems to be a case of underfit. It is clear how regularization affects the model complexity: with a small regularization (in this case $\gamma = 0.001$), the model is prone to adapt to the noise in the training data.

The effect would be even more severe with no regularization ($\gamma = 0$). Heavier regularization (in this case $\gamma = 10$) effectively prevents the model from adapting well to the training data (it pushes the parameters, including β_0, towards 0).

Let us understand this in terms of bias and variance. In the low-regularized case, the trained model (solid line) will look very different each time, depending on what $\mathbf{x}$-values and noise happen to be in the training data: a high variance. However, if one would repeat the experiment many times with different training data, the average model will probably be relatively close to the true model: a low bias. The completely opposite situation is found in the highly regularized case: the variance is low (the model will be quite similar each time, no matter what realization of the training data it is trained on), and the bias is high (the predictions from the model will systematically be closer to zero than the true model).

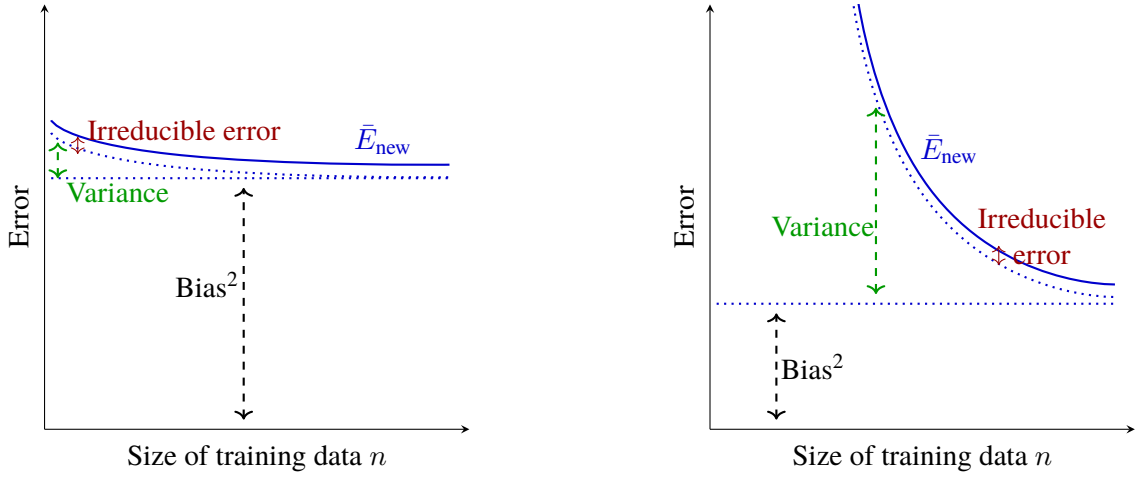
Since we are in a simulated environment, we can repeat the experiment multiple times, and thereby compute the bias and variance terms (or rather numerically estimate them, since we can simulate as much training and test data as we want). We plot them in the very same style as Figures 5.3 and 5.5 (note the reversed x-axis: a smaller regularization parameter corresponds to a higher model complexity). For this problem, the optimal value of γ would have been about 0.7 (since $\bar{E}_{\text{new}}$ attains its minimum there).



If this had been a real problem with a fixed data set, we could of course not have made this plot. Instead, one would have to rely on cross-validation for estimating E_{new} for that particular data set (and not its average $\bar{E}_{\text{new}}$).

The bias-variance trade-off and its relation to the size n of training data

In the first place, the bias term is a property of the model rather than of the training data set, and we may think[4] of the bias term as independent of the number of data points n in the training data. The variance term, on the other hand, varies highly with n. As we know, $\bar{E}_{\text{new}}$ typically decreases as n increases, and essentially the entire decline in $\bar{E}_{\text{new}}$ is because of the decline in the variance. Intuitively, the more data, the more information about the parameters, meaning less variance. This is summarized by Figure 5.6.



(a) Simple model

(b) Complex model

Figure 5.6: The typical relationship between bias, variance and the size n of the training data set (cf. Figure 5.4). The bias is (approximately) constant, whereas the variance decreases as the size of the training data set increases.

[4] Indeed, the average model g might be different if we are averaging over an infinite number of models each trained with $n = 2$ or $n = 100\,000$ data points. That effect is, however, conceptually not very interesting here, and we will not treat it further.

6 Ensemble methods

In the previous chapters we have introduced some fundamental methods for machine learning. In this chapter we will introduce techniques of a slightly different flavor, referred to as *ensemble methods*. These methods are based on the idea of combining the predictions from many so called *base models*. They can therefore be seen as a type of meta-algorithms, in the sense that they are methods composed of other methods.

We start in Section 6.1 by introducing a general technique referred to as bootstrap aggregating, or *bagging* for short. The idea behind bagging is to train multiple models of the same type in parallel, but on slightly different "versions" of the training data. By averaging the predictions of the resulting ensemble of models it is possible to reduce the variance compared to using only a single model. This idea is extended in Section 6.2, resulting in a powerful off-the-shelf method called random forests. Random forests make use of classification or regression trees as base models. Each tree is randomly perturbed in a certain way which opens up for additional variance reduction. Finally, in Section 6.3 we derive an alternative ensemble method known as *boosting*. Boosting is different from bagging and random forests, since its base models are learned sequentially, one after the other, so that each model tries to correct for the mistakes made by the previous ones. By taking a weighted average of the predictions made by the base models, it is possible to turn the ensemble of "weak" models into one "strong" model.

6.1 Bagging

As discussed in the previous chapter, a central concept in machine learning is the bias–variance trade-off. Roughly speaking, the more flexible a model is, the lower its bias will be. That is, a flexible model is capable of representing complicated input–output relationships. This is of course beneficial if the actual relationship between inputs and outputs is complicated, as is often the case in machine learning applications. We have seen examples of highly flexible models, not least in Chapter 4 when we discussed non-parametric models. For instance, k-NN with a small value of k, or a classification tree that is grown deep, are capable of expressing very complicated decision boundaries. However, as previously discussed there is a price to pay when using a very flexible model: fitting it based on a finite number of training data points can result in over-fitting[1], or equivalently, high model variance.

Motivated by this issue, we can ask the following question:

> *Given access to a low-bias, high-variance method, is there a way to reduce the variance while keeping the bias low?*

In this section we will introduce a general technique for accomplishing this, referred to as bootstrap aggregating, or *bagging*.

6.1.1 Variance reduction by averaging

To answer the question posed above our starting point will be a basic property of random variables, namely that averaging reduces variance. To formalize this, let $z_1, \ldots, z_B$ be a collection of identically distributed (but possibly dependent) random variables with mean value $\mathbb{E}[z_b] = \mu$ and variance $\mathrm{Var}[z_b] = \sigma^2$ for $b = 1, \ldots, B$. Furthermore, assume that the average correlation[2] between any pair of variables is ρ. Then,

[1] Both a k-NN model with $k = 1$ or a classification tree with a single data point per leaf node will result in zero training error, typical for severe over-fitting.

[2] That is $\frac{1}{B(B-1)} \sum_{b \neq c} \mathbb{E}[(z_b - \mu)(z_c - \mu)] = \rho\sigma^2$.

computing the mean and the variance of *the average* of these variables we get

$$\mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} z_i\right] = \mu, \tag{6.1a}$$

$$\mathrm{Var}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2. \tag{6.1b}$$

The first equation tells us that the mean is unaltered by averaging a number of identically distributed random variables. Furthermore, the second equation tells us that the variance is reduced by averaging, as long as the correlation $\rho < 1$. Indeed, the first term in the variance expression can be made arbitrarily small by increasing B. We also note that the smaller ρ is, the larger the possible reduction in variance.

To see how this relates to the question posed above, we note that the bias of a prediction model is tightly connected to its mean value. Consequently, by *averaging the predictions from several identically distributed models*, each with a low bias, the bias remains low (cf. (6.1a)) and the variance is reduced (cf. (6.1b)). How then can we construct such a collection—or *ensemble*—of prediction models? To answer this question we will start with an unpractical assumption, which will later be relaxed, namely that we have access to B independent training data sets[3], each of size n. Let these data sets be denoted by $\mathcal{T}^1, \ldots, \mathcal{T}^B$. We can then train a separate low-bias model, such as a deep classification or regression tree, on each separate data set.

For concreteness we consider the regression setting. Each of the B regression models can be used to predict the output for some test input $\mathbf{x}_\star$. Since (by our unpractical assumption) the B training data sets are independent, this will result in B *independent and identically distributed* predictions $\widehat{y}_\star^1, \ldots, \widehat{y}_\star^B$. Taking the average of these predictions we get,

$$\widehat{y}_\star^{\text{avg}} = \frac{1}{B}\sum_{b=1}^{B} \widehat{y}_\star^b. \tag{6.2}$$

In the classification setting we would instead use a majority vote among the B ensemble members.

The average prediction in (6.2) has the same bias as each of the individual $\widehat{y}_\star^b$'s, but its variance is reduced by a factor B. Indeed, in this case we have $\rho = 0$ in the expression (6.1b) since the individual data sets were assumed to be independent. However, as mentioned above this is not a realistic assumption, so in order to use this variance reduction technique in practice we need to do something differently. To this end we will make use of a trick known as *data bootstrapping*.

6.1.2 The bootstrap

The bootstrap is a powerful and widely applicable statistical technique. Its main usage is in fact for quantifying the uncertainties in statistical estimators, but here we will use it for another purpose, namely to mimic the variance reduction approach outlined above. To apply this technique we need to obtain multiple training data sets, but we only have access to one data set $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$. We assume that simply collecting more data is not an option (if it were, we would be better off simply increasing the size of $\mathcal{T}$ instead). Instead, we assume that $\mathcal{T}$ provides a good representation of the real-world data generating process, in the sense that even if we were to collect more training data, these data points would likely be similar to the training data points already contained in $\mathcal{T}$. Therefore, we can simulate "new" training data sets by sampling data points from the original training data. In statistical terms, instead of sampling from the population (collecting more data), we sample from the available training data which is assumed to provide a good representation of the population.

[3] We could of course simply chop the actual training data up into B independent chunks. However, this would mean that the size of each such data set is only n/B, which would counteract the gain offered by variance reduction. In some situations this can still be a good idea though: if n is very large, training a single model on all data can be computationally prohibitive. Instead, we can divide the available data into smaller chucks, train independent models on these chunks in parallel, and average their predictions.

The bootstrapping method is stated in algorithm 5 and illustrated in in Example 6.1 below. Note that the sampling is done with replacement, meaning that the resulting bootstrapped data set may contain multiple copies of some of the original training data points, whereas other data points are not included in the bootstrap sample.

Algorithm 5: The bootstrap.

Data: Training data set $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
Result: Bootstrapped data $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^n$

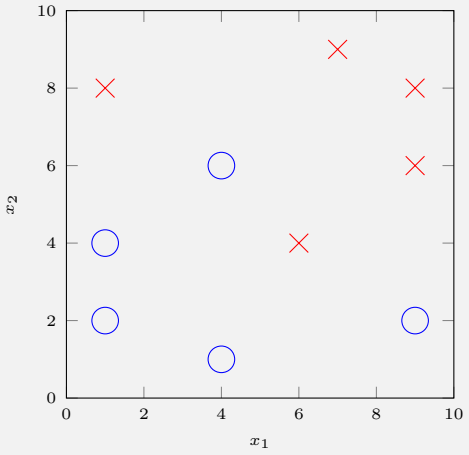
for $i = 1, \ldots, n$ **do**
  Sample ℓ uniformly on the set of integers $\{1, \ldots, n\}$
  Set $\widetilde{\mathbf{x}}_i = \mathbf{x}_\ell$ and $\widetilde{y}_i = y_\ell$
end

Example 6.1: The bootstrap

Consider the same training data set as used in Example 4.2, consisting of $n = 10$ data points with a two-dimensional input $\mathbf{x} = (x_1, x_2)$ and a binary output $y \in \{\text{Blue}, \text{Red}\}$. The data set is shown below.

Original training data, $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{10}$

Index	x_1	x_2	y
1	9.0	2.0	Blue
2	1.0	4.0	Blue
3	4.0	6.0	Blue
4	4.0	1.0	Blue
5	1.0	2.0	Blue
6	1.0	8.0	Red
7	6.0	4.0	Red
8	7.0	9.0	Red
9	9.0	8.0	Red
10	9.0	6.0	Red



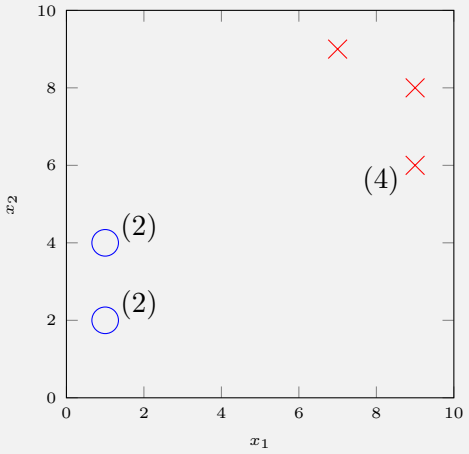
To generate a bootstrapped data set $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^{10}$ we simulate 10 times with replacement from the index set $\{1, \ldots, 10\}$, resulting in the indices

$$\{2, 10, 10, 5, 9, 2, 5, 10, 8, 10\}$$

Thus, $(\widetilde{\mathbf{x}}_1, \widetilde{y}_1) = (\mathbf{x}_2, y_2)$, $(\widetilde{\mathbf{x}}_2, \widetilde{y}_2) = (\mathbf{x}_{10}, y_{10})$, etc. We end up with the following data set, where the numbers in parentheses in the right panel indicate that there are multiple copies of some of the original data points in the bootstrapped data.

Bootstrapped data, $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^{10}$

Index	$\widetilde{x}_1$	$\widetilde{x}_2$	$\widetilde{y}$
2	1.0	4.0	Blue
10	9.0	6.0	Red
10	9.0	6.0	Red
5	1.0	2.0	Blue
9	9.0	8.0	Red
2	1.0	4.0	Blue
5	1.0	2.0	Blue
10	9.0	6.0	Red
8	7.0	9.0	Red
10	9.0	6.0	Red



By running algorithm 5 repeatedly B times we obtain B *identically distributed* bootstrapped data sets $\widetilde{\mathcal{T}}^1, \ldots, \widetilde{\mathcal{T}}^B$. We can then use the bootstrapped data sets to train B low-bias regression (or classification) models and average their predictions, $\widetilde{y}_\star^1, \ldots, \widetilde{y}_\star^B$, analogously to (6.2):

$$\widehat{y}_\star^{\text{bag}} = \frac{1}{B} \sum_{b=1}^{B} \widetilde{y}_\star^b. \tag{6.3}$$

The averaged predictions are in this case not independent since the bootstrapped data sets all come from the same original data $\mathcal{T}$. However the hope is that the predictions are sufficiently uncorrelated (ρ is small enough) to result in a meaningful variance reduction. Experience has shown that this is indeed often the case. We illustrate the bagging method in Example 6.2.

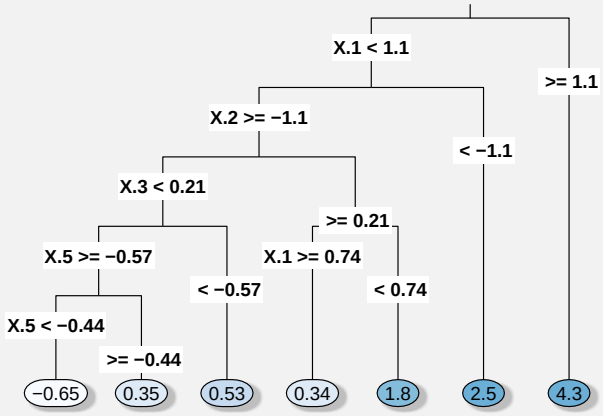
Example 6.2: Illustration of bagging

Consider a toy regression problem, where the 5-dimensional input is multivariate Gaussian $\mathbf{x} \sim \mathcal{N}(0, \Sigma)$ with

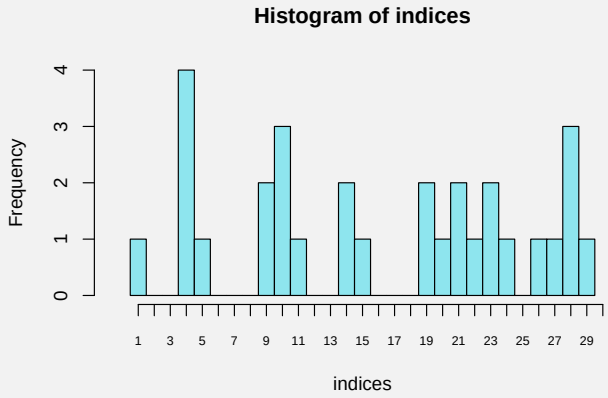
$$\Sigma = \begin{pmatrix} 1 & 0.98 & 0.98 & 0.98 & 0.98 \\ 0.98 & 1 & 0.98 & 0.98 & 0.98 \\ 0.98 & 0.98 & 1 & 0.98 & 0.98 \\ 0.98 & 0.98 & 0.98 & 1 & 0.98 \\ 0.98 & 0.98 & 0.98 & 0.98 & 1 \end{pmatrix}$$

and the output is $y = x_1^2 + \epsilon$ with $\epsilon \sim \mathcal{N}(0, 1)$. We observe a training data set consisting of $n = 30$ inputs and outputs. Due to the strong correlations between the input variables[a], they are all informative about the value of the output, despite the fact that the output only depends directly on the first input variable, x_1.

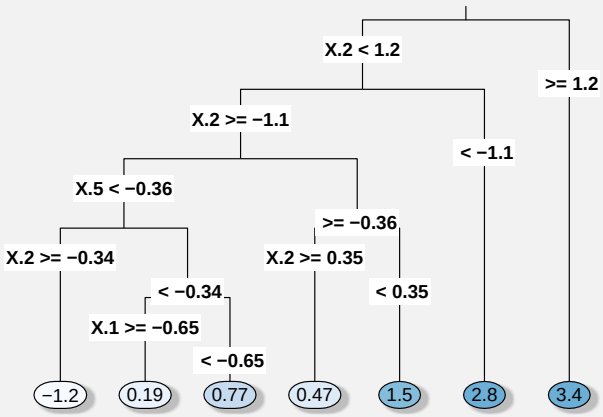
We start by training a single regression model, branching until there are at most 3 observations per leaf node, resulting in the model shown below (the values in the leaf nodes are the constant predictions within each region).



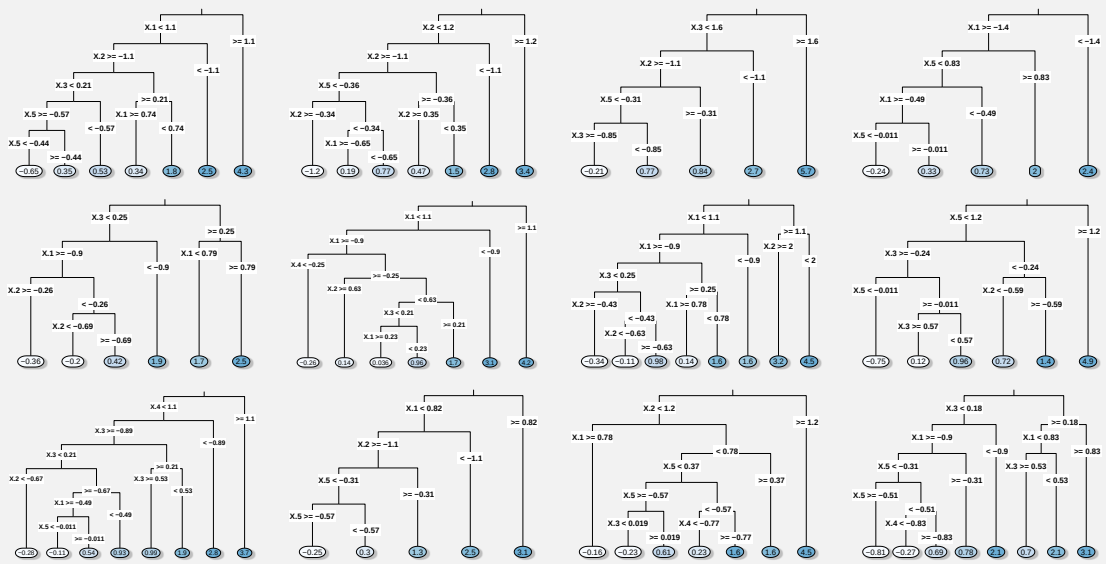
To apply the bagging method, we need to generate bootstrapped data sets. Running algorithm 5 once, we obtain a set of sampled indices, shown by the histogram below. Note that some data points are sampled multiple times, whereas other data points are not included at all in the bootstrapped data.



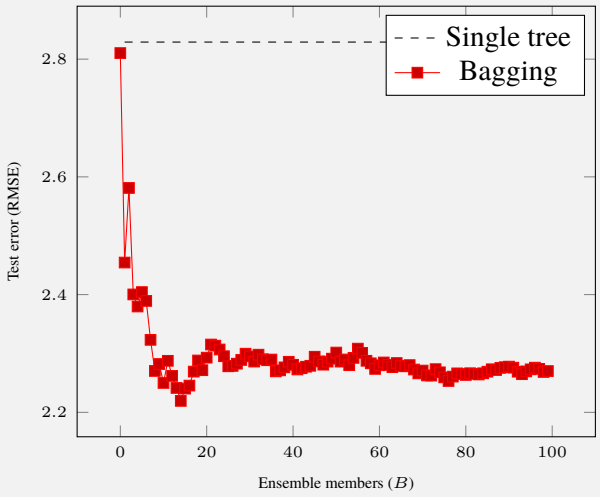
Training another regression tree on the bootstrapped data gives a slightly different model:



This procedure can then be repeated to generate an arbitrary number of bootstrap data sets and regression models, each one possibly different from the ones before due to the randomness in the bootstrap procedure[b]. Below we show the first 12 models that we obtain.



To test the performance of the bagging method for this example, we simulate a large test data set consisting of $n_{\text{test}} = 2000$ samples from the true data distribution (which is known since we are working with a toy model). We can then compute the root mean squared error (RMSE) for varying number of ensemble members B. The results are shown below:



Two things are worth noting in the figure above: First, the bagging method does improve the RMSE over a single tree as a result of variance reduction (here, we get roughly an 18 % reduction in RMSE, but this is of course problem dependent). Second, the performance of the bagging method improves as we increase the number of ensemble members, but saturates at some point beyond which no further improvement (nor degradation) is noticeable.

[a] Recall that the values on the diagonal of the covariance matrix are the marginal variances of the elements x_i, and the off-diagonal values are covariances. Specifically, since all marginal variances in this example are 1, any pair of variables x_i and x_j for $i \neq j$ has a correlation of 0.98.

[b] With n training data points there are in total $n^n/n!$ unique bootstrap datasets.

At first glance, one might think that a bagging model (6.3) becomes more "complex" as the number of ensemble members B increase, and that we therefore run a risk of overfitting for large B. However, from the example above this does not appear to be the case. Indeed, the test RMSE plateaus at some value and does not start start to increase even for large values of B. This is in fact the expected (and intended) behavior. Using more ensemble members does not make the model more *flexible*, but on the contrary,

reduces its overfitting (or equivalently, its variance). This can be seen by considering the limiting behavior as $B \to \infty$, in which case the bagging model becomes

$$\widehat{y}_\star^{\text{bag}} = \frac{1}{B}\sum_{b=1}^{B} \widetilde{y}_\star^b \overset{B\to\infty}{\longrightarrow} \mathbb{E}\left[\widetilde{y}_\star^b \,|\, \mathcal{T}\right], \tag{6.4}$$

where the expectation is w.r.t. the randomness of the bootstrapping algorithm. Since the right hand side of this expression does not depend on B it can be thought of as a regression model with "limited flexibility", and the flexibility of the bagging model will therefore not increase as B becomes large. With this in mind, in practice the choice of B is mainly guided by computational constraints. The larger B the better, but increasing B when there is no further reduction in test error is computationally wasteful.

6.2 Random forests

As pointed out above, the variance reduction obtained by averaging is limited by the correlation between the individual ensemble members (cf. (6.1b)). A natural question to ask is therefore if it is possible to reduce this correlation. One simple trick for accomplishing this is a method known as random forests (Breiman 2001).

While bagging is a general technique that in principle can be used to reduce the variance of any base model, random forests assumes that these base models are given by classification or regression trees. The idea is to inject additional randomness when constructing each tree, in order to reduce the correlation among them. At first this might seem like a silly idea: randomly perturbing the training of a model should intuitively degrade its performance. There is a rationale for this perturbation, however, which we will discuss below, but first we present the details of the algorithm.

Let $\widetilde{\mathcal{T}}^b$ be one of the B bootstrapped data sets. To train a classification or regression tree on this data we proceed as usual (see Section 4.2) , but with one difference. Throughout the training, whenever we are about to split a node we do not consider all possible input variables $x_1, \ldots, x_p$ as splitting variables. Instead, we pick a random subset consisting of $m \leq p$ inputs, and only consider these m variables as possible splitting variables. At the next splitting point we draw a new random subset of m inputs to use as possible splitting variables, and so on. Naturally, this random subset selection is done independently for each of the B ensemble members, so that we (with high probability) end up using different subsets for the different trees. This will cause the B trees to be less correlated and averaging their predictions can therefore result in larger variance reduction compared to bagging. It should be noted, however, that this random perturbation of the training procedure will increase both the bias and the variance of each *individual tree*. That being said, experience has shown that the reduction in correlation is the dominant effect, so that the overall prediction loss is often reduced.

To understand why it can be a good idea to only consider a subset of inputs as splitting variables, recall that tree-building is based on recursive binary splitting which is a greedy algorithm. This means that the algorithm can make choices early on that appear to be good, but which nevertheless turn out to be suboptimal further down the splitting procedure. For instance, consider the case when there is one dominant input variable. If we construct an ensemble of trees using bagging, it is then very likely that all of the ensemble members will pick this dominant variable up as the first splitting variable, making all trees identical (i.e., perfectly correlated) after the first split. If we instead apply random forests, then some of the ensemble members will not even have access to this dominant variable at the first split[4], forcing them to split according to some other variable. While there is of course no reason for why this would improve the performance of the individual tree, it *could* prove to be useful further down the splitting process, and since we average over many ensemble members the overall performance can be improved.

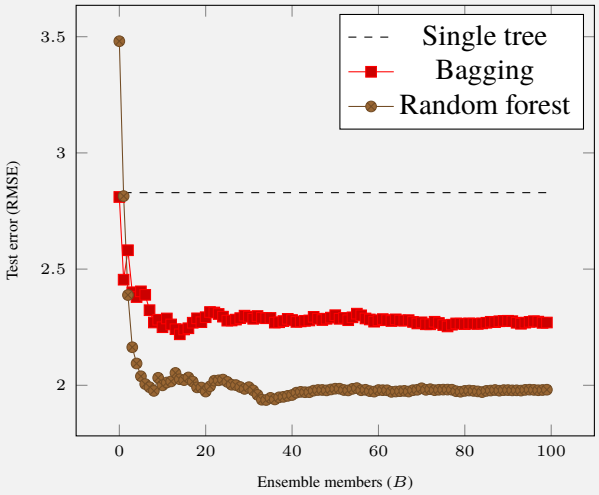
The choice of m is a tuning parameter, where for $m = p$ we recover the basic bagging method described previously. As a rule-of-thumb we can set $m = \sqrt{p}$ for classification problems and $m = p/3$ for regression

[4]That is, it is likely that the dominant variable is not contained in the random subset selected at the first split for some of the ensemble members.

problems (values rounded down to closest integer). A more systematic way of selecting m, however, is by some type of cross-validation. We illustrate the random forest regression model in the example below.

Example 6.3: Random forests

We continue Example 6.2 and apply random forests to the same data. The test root mean squared errors for different number of ensemble members are shown in the figure below.



In this example, tthe decorrelation effect of the random input selection used in random forests result in an additional 13 % drop in RMSE compared to bagging.

6.3 Boosting

6.3.1 The conceptual idea

Boosting is built on the idea that even a simple (or *weak*) high-bias regression or classification model often can capture some of the relationship between the inputs and the output. Thus, by training multiple weak models, each describing part of the input-output relationship, it might be possible to combine the predictions of these models into an overall better prediction. Hence, the intention is to turn an *ensemble* of weak models into one strong model.

Boosting shares some similarities with bagging, as discussed above. Both are ensemble methods, in the sense that they are based on combining the predictions from multiple models (an "ensemble"). Both bagging and boosting can also be viewed as meta-algorithms, in the sense that they can be used to combine essentially any regression or classification algorithm—they are algorithms built on top of other algorithms. However, there are also important differences between boosting and bagging which we will discuss below.

The first key difference is in the construction of the ensemble. In bagging we construct B models in parallel. These models are random (based on randomly bootstrapped datasets) but they are *identically distributed*. Consequently, there is no reason to trust one model more than another, and the final prediction of the bagged model is based on a plain average or majority vote of the individual predictions of the ensemble members.

Boosting, on the other hand, uses a *sequential* construction of the ensemble members. Informally, this is done in such a way that each model tries to correct the mistakes made by the previous one. This is accomplished by modifying the training data set at each iteration in order to put more emphasis on the data points for which the model (so far) has performed poorly. In the subsequent sections we will see exactly how this is done for a specific boosting algorithm known as AdaBoost (Freund and Schapire 1996). First, however, we consider a simple example to illustrate the idea.

Example 6.4: Boosting illustration

We consider a toy binary classification problem with two input variables, x_1 and x_2. The training data consists of $n = 10$ data points, 5 from each of the two classes. We use a decision stump, a classification tree of depth one, as a simple (weak) base classifier. A decision stump means that we select one of the input variables, x_1 or x_2, and split the input space into two half spaces, in order to minimize the training error. This results in a decision boundary that is perpendicular to one of the axes. The left panel below shows the training data, illustrated by red crosses and blue dots for the two classes, respectively. The colored regions shows the decision boundary for a decision stump $\widehat{y}^1(\mathbf{x})$ trained on this data.

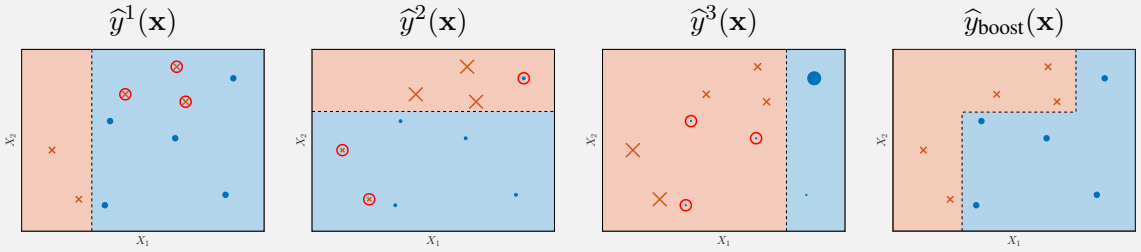


Figure 6.1: Three iterations of AdaBoost for toy problem. Training data points for the two classes is shown by red crosses and blue dots, respectively, scaled according to the weights. The colored regions show the three base classifiers (decision stumps), $\widehat{y}^1$, $\widehat{y}^2$, and $\widehat{y}^3$.

The model $\widehat{y}^1(\mathbf{x})$ misclassifies three data points (red crosses falling in the blue region), which are encircled in the figure. To improve the performance of the classifier we want to find a model that can distinguish these three points from the blue class. To this end, we train another decision stump, $\widehat{y}^2(\mathbf{x})$, on the same data. To put emphasis on the three misclassified points, however, we assign *weights* $\{w_i^2\}_{i=1}^n$ to the data. Points correctly classified by $\widehat{y}^1(\mathbf{x})$ are down-weighted, whereas the three points misclassified by $\widehat{y}^1(\mathbf{x})$ are up-weighted. This is illustrated in the second panel of Figure 6.1, where the marker sizes have been scaled according to the weights. The classifier $\widehat{y}^2(\mathbf{x})$ is then found by minimizing the *weighted* misclassification error, $\frac{1}{n}\sum_{i=1}^n w_i^2 \mathbb{I}\{y_i \neq \widehat{y}^2(\mathbf{x}_i)\}$, resulting in the decision boundary shown in the second panel. This procedure is repeated for a third and final iteration: we update the weights based on the hits and misses of $\widehat{y}^2(\mathbf{x})$ and train a third decision stump $\widehat{y}^3(\mathbf{x})$ shown in the third panel. The final classifier, $\widehat{y}_{\text{boost}}(\mathbf{x})$ is then taken as a combination of the three decision stumps. Its (nonlinear) decision boundaries are shown in the right panel.

6.3.2 Binary classification, margins, and exponential loss

Before diving into the details of the AdaBoost algorithm we will lay the groundwork by introducing some notations and concepts that will be used in its derivation. AdaBoost was originally proposed for binary classification ($K = 2$) and we will restrict our attention to this setting. That is, the output y can take two different values which, in this chapter, we encode as -1 and $+1$. This encoding turns out to be mathematically convenient in the derivation of AdaBoost. However, it is important to realize that the encoding that we choose for the two classes is arbitrary and all the concepts defined below can be generalized to any binary encoding (e.g. $\{0, 1\}$ which we have used before).

Most binary classifiers $\widehat{y}(\mathbf{x})$ can be constructed by thresholding some real-valued function $C(\mathbf{x})$ at 0. That is, using our $+1/-1$ encoding we can write $\widehat{y}(\mathbf{x}) = \text{sign}\{C(\mathbf{x})\}$. In particular, it is the case for the AdaBoost algorithm presented below. Note that the decision boundary is given by values of $\mathbf{x}$ for which $C(\mathbf{x}) = 0$. For simplicity of presentation we will assume that no data points fall exactly on the decision boundary (which always gives rise to an ambiguity), so that we can assume that $\widehat{y}(\mathbf{x})$ as defined above is always either -1 or $+1$.

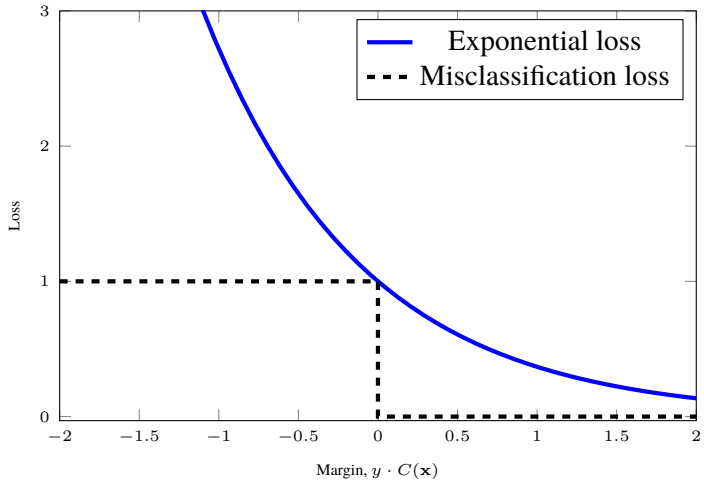


Figure 6.2: Exponential loss.

Based on the function $C(\mathbf{x})$ we define the *margin* of the classifier as

$$y \cdot C(\mathbf{x}). \tag{6.5}$$

It follows that if y and $C(\mathbf{x})$ have the same sign, that is if the classification is correct, then the margin is positive. Analogously, for an incorrect classification y and $C(\mathbf{x})$ will have different signs and the margin is negative. More specifically, since y is either -1 or $+1$, the margin is simply $|C(\mathbf{x})|$ for correct classifications and $-|C(\mathbf{x})|$ for incorrect classifications. The margin can thus be viewed as a measure of certainty in a prediction, where values with small margin in some sense (not necessarily Euclidian!) are close to the decision boundary. The margin plays a similar role for binary classification as the residual $y - f(\mathbf{x})$ does for regression.

Loss functions for classification can be defined in terms of the margin, by assigning small loss to positive margins and large loss to negative margins. One such loss function is the *exponential loss* defined as

$$L(y, C(\mathbf{x})) = \exp(-y \cdot C(\mathbf{x})). \tag{6.6}$$

Figure 6.2 illustrates the exponential loss and compares it against the misclassification loss, which is simply $\mathbb{I}\{y \cdot C(\mathbf{x}) < 0\}$.

Remark 6.1 *The misclassification loss is often used to evaluate the performance of a classifier (in particular if interest only lies in the number of correct and incorrect classification). However, it is typically not suitable to use directly during training of the model. The reason is that it is discontinuous which is problematic in a numerical minimization of the training loss. The exponential loss function, on the other hand, is both convex and (infinitely many times) differentiable. These are nice properties to have when optimizing the training loss. In fact, the exponential loss can in this way be seen as a convenient proxy for misclassification loss during training. Other loss functions of interest are discussed in Section 6.3.5.*

6.3.3 AdaBoost

We are now ready to derive a practical boosting method, the AdaBoost (Adaptive Boosting) algorithm proposed by Freund and Schapire (1996). AdaBoost was the first successful practical implementation of the boosting idea and lead the way for its popularity. Freund and Schapire were awarded the prestigious Gödel Prize in 2003 for their algorithm.

Recall from the discussion above that boosting attempts to construct a sequence of B (weak) classifiers $\widehat{y}^1(\mathbf{x})$, $\widehat{y}^2(\mathbf{x})$, $\ldots$, $\widehat{y}^B(\mathbf{x})$. Any classification model can in principle be used to construct these so

called *base classifiers*—shallow classification trees are common in practice (see Section 6.3.4 for further discussion). The individual predictions of the B ensemble members are then combined into a final prediction. However, all ensemble members are not treated equally. Instead, we assign some positive coefficients $\{\alpha^b\}_{b=1}^B$ and construct the boosted classifier using a weighted majority vote according to

$$\widehat{y}^B_{\text{boost}}(\mathbf{x}) = \operatorname{sign}\left\{\sum_{b=1}^{B} \alpha^b \widehat{y}^b(\mathbf{x})\right\}. \tag{6.7}$$

Note that each ensemble member votes either -1 or $+1$. The output from the boosted classifier is $+1$ if the weighted sum of the individual votes is positive and -1 if it is negative.

How, then, do we learn the individual ensemble members and their coefficients? The answer to this question depends on the specific boosting method that is used. For AdaBoost this is done by greedily minimizing the exponential loss of the boosted classifier at each iteration. Note that we can write the boosted classifier after b iterations as $\widehat{y}^b_{\text{boost}} = \operatorname{sign}\{C^b(\mathbf{x})\}$ where $C^b(\mathbf{x}) = \sum_{j=1}^b \alpha^j \widehat{y}^j(\mathbf{x})$. Furthermore, we can express the function $C^b(\mathbf{x})$ sequentially as

$$C^b(\mathbf{x}) = C^{b-1}(\mathbf{x}) + \alpha^b \widehat{y}^b(\mathbf{x}), \tag{6.8}$$

initialized with $C^0(\mathbf{x}) \equiv 0$. Since we construct the ensemble members sequentially, when at iteration b of the procedure the function $C^{b-1}(\mathbf{x})$ is known and fixed. Thus, what remains to be learned at iteration b is the coefficient α^b and the ensemble member $\widehat{y}^b(\mathbf{x})$. This is done by minimizing the exponential loss of the training data,

$$\begin{aligned}
(\alpha^b, \widehat{y}^b) &= \arg\min_{(\alpha,\widehat{y})} \sum_{i=1}^{n} L(y_i, C^b(\mathbf{x}_i)) && \text{(6.9a)}\\
&= \arg\min_{(\alpha,\widehat{y})} \sum_{i=1}^{n} \exp\left(-y_i\left\{C^{b-1}(\mathbf{x}_i) + \alpha\widehat{y}(\mathbf{x}_i)\right\}\right) && \text{(6.9b)}\\
&= \arg\min_{(\alpha,\widehat{y})} \sum_{i=1}^{n} w_i^b \left(-y_i \alpha \widehat{y}(\mathbf{x}_i)\right) && \text{(6.9c)}
\end{aligned}$$

where for the first equality we have used the definition of the exponential loss function (6.6) and the sequential structure of the boosted classifier (6.8). For the second equality we have defined the quantities

$$w_i^b \overset{\text{def}}{=} \exp\left(-y_i C^{b-1}(\mathbf{x}_i)\right), \qquad i = 1, \ldots, n, \tag{6.10}$$

which can be interpreted as weights for the individual data points in the training data set. Note that these weights are independent of α and $\widehat{y}$ and can thus be viewed as constants when solving the optimization problem (6.9c) at iteration b of the boosting procedure (the fact that we keep previous coefficients and ensemble members fixed is what makes the optimization "greedy").

To solve (6.9) we start by writing the objective function as

$$\sum_{i=1}^{n} w_i^b \exp\left(-y_i \alpha \widehat{y}(\mathbf{x}_i)\right) = e^{-\alpha} \underbrace{\sum_{i=1}^{n} w_i^b \mathbb{I}\{y_i = \widehat{y}(\mathbf{x}_i)\}}_{=W_c} + e^{\alpha} \underbrace{\sum_{i=1}^{n} w_i^b \mathbb{I}\{y_i \neq \widehat{y}(\mathbf{x}_i)\}}_{=W_e}, \tag{6.11}$$

where we have used the indicator function to split the sum into two sums: the first ranging over all training data points correctly classified by $\widehat{y}$ and the second ranging over all point erroneously classified by $\widehat{y}$. Furthermore, for notational simplicity we define W_c and W_e for the sum of weights of correctly classified and erroneously classified data points, respectively. Furthermore, let $W = W_c + W_e$ be the total weight sum, $W = \sum_{i=1}^n w_i^b$.

Minimizing (6.11) is done in two stages, first w.r.t. $\widehat{y}$ and then w.r.t. α. This is possible since the minimizing argument in $\widehat{y}$ turns out to be independent of the actual value of $\alpha > 0$. To see this, note that we can write the objective function (6.11) as

$$e^{-\alpha}W + (e^{\alpha} - e^{-\alpha})W_e. \tag{6.12}$$

Since the total weight sum W is independent of $\widehat{y}$ and since $e^{\alpha} - e^{-\alpha} > 0$ for any $\alpha > 0$, minimizing this expression w.r.t. $\widehat{y}$ is equivalent to minimizing W_e w.r.t. $\widehat{y}$. That is,

$$\widehat{y}^b = \arg\min_{\widehat{y}} \sum_{i=1}^{n} w_i^b \mathbb{I}\{y_i \neq \widehat{y}(\mathbf{x}_i)\}. \tag{6.13}$$

In words, the bth ensemble member should be trained by minimizing the *weighted misclassification loss*, where each data point $(\mathbf{x}_i, y_i)$ is assigned a weight w_i^b. The intuition for these weights is that, at iteration b, we should focus our attention on the data points previously misclassified in order to "correct the mistakes" made by the ensemble of the first $b-1$ classifiers.

How the problem (6.13) is solved in practice depends on the choice of base classifier that we use, i.e. on the specific restrictions that we put on the function $\widehat{y}$ (for example a shallow classification tree). However, since (6.13) is essentially a standard classification objective it can be solved, at least approximately, by standard learning algorithms. Incorporating the weights in the objective function is straightforward for most base classifiers, since it simply boils down to weighting the individual terms of the loss function used when training the base classifier.

When the bth ensemble member, $\widehat{y}^b(\mathbf{x})$, has been learned by solving (6.13) it remains to compute its coefficient α^b. Recall that this is done by minimizing the objective function (6.12). Differentiating this expression w.r.t. α and setting the derivative to zero we get the equation

$$\begin{aligned} &-\alpha e^{-\alpha}W + \alpha\left(e^{\alpha} + e^{-\alpha}\right)W_e = 0 \\ \iff &W = \left(e^{2\alpha} + 1\right)W_e \\ \iff &\alpha = \frac{1}{2}\log\left(\frac{W}{W_e} - 1\right). \end{aligned} \tag{6.14}$$

Thus, by defining

$$E^b_{\text{train}} \overset{\text{def}}{=} \frac{W_e}{W} = \sum_{i=1}^{n} \frac{w_i^b}{\sum_{j=1}^{n} w_j^b} \mathbb{I}\{y_i \neq \widehat{y}^b(\mathbf{x}_i)\} \tag{6.15}$$

to be the weighted misclassification error for the bth classifier, we can express the optimal value for its coefficient as

$$\alpha^b = \frac{1}{2}\log\left(\frac{1 - E^b_{\text{train}}}{E^b_{\text{train}}}\right). \tag{6.16}$$

This completes the derivation of the AdaBoost algorithm, which is summarized in algorithm 6. In the algorithm we exploit the fact that the weights (6.10) can be computed recursively by using the expression (6.8); see line 2. Furthermore, we have added an explicit weight normalization (line 2) which is convenient in practice and which does not affect the derivation of the method above.

Remark 6.2 *One detail worth commenting is that the derivation of the AdaBoost procedure assumes that all coefficients* $\{\alpha^b\}_{b=1}^B$ *are positive. To see that this is indeed the case when the coefficients are computed according to* (6.16)*, note that the function* $\log((1-x)/x)$ *is positive for any* $0 < x < 0.5$*. Thus,* α^b *will be positive as long as the weighted training error for the bth classifier,* E^b_{train}*, is less than 0.5. That is, the classifier just has to be slightly better than a coin flip, which is always the case in practice (note that* E^b_{train} *is the* training *error). Indeed, if* $E^b_{train} > 0.5$*, then we could simply flip the sign of all predictions made by* $\widehat{y}^b(\mathbf{x})$ *to reduce the error!*

Algorithm 6: AdaBoost

1. Assign weights $w_i^1 = 1/n$ to all data points.
2. For $b = 1$ to B
   (a) Train a weak classifier $\widehat{y}^b(\mathbf{x})$ on the weighted training data $\{(\mathbf{x}_i, y_i, w_i^b)\}_{i=1}^n$.
   (b) *Update the weights* $\{w_i^{b+1}\}_{i=1}^n$ *from* $\{w_i^b\}_{i=1}^n$*:*
      i. Compute $E_{\text{train}}^b = \sum_{i=1}^n w_i^b \mathbb{I}\{y_i \neq \widehat{y}^b(\mathbf{x}_i)\}$
      ii. Compute $\alpha^b = 0.5 \log((1 - E_{\text{train}}^b)/E_{\text{train}}^b)$.
      iii. Compute $w_i^{b+1} = w_i^b \exp(-\alpha^b y_i \widehat{y}^b(\mathbf{x}_i))$, $i = 1, \ldots, n$
      iv. *Normalize.* Set $w_i^{b+1} \leftarrow w_i^{b+1} / \sum_{j=1}^n w_j^{b+1}$, for $i = 1, \ldots, n$.
3. Output $\widehat{y}_{\text{boost}}^B(\mathbf{x}) = \text{sign}\left\{\sum_{b=1}^B \alpha^b \widehat{y}^b(\mathbf{x})\right\}$.

In the method discussed above we have assumed that each base classifier outputs a discrete class prediction, $\widehat{y}^b(\mathbf{x}) \in \{-1, 1\}$. However, many classification models used in practice are in fact based on estimating the conditional class probabilities $p(y \mid \mathbf{x})$ as discussed in Section 3.4.1. Hence, it is possible to instead let each base model output a real number and use these numbers as the basis for the "vote". This extension of algorithm 6 is discussed by Friedman, Hastie, and Tibshirani (2000) and is referred to as Real AdaBoost.

6.3.4 Boosting vs. bagging: base models and ensemble size

AdaBoost, and in fact any boosting algorithm, has two important design choices, *(i)* which base classifier to use, an *(ii)* how many iterations B to run the boosting algorithm for. As previously pointed out, we can use essentially any classification method as base classifier. However, the most common choice in practice is to use a shallow classification tree, or even a decision stump (a tree of depth one; see Example 6.4). This choice is guided by the fact that the boosting algorithm can learn a good model despite using very weak base classifiers, and shallow trees can be trained quickly. In fact, using deep (high-variance) classification trees as base classifiers typically deteriorates performance compared to using shallow trees. More specifically, the depth of the tree should be chosen to obtain a desired degree of interactions between input variables. A tree with M terminal nodes is able to model functions depending on maximally $M - 1$ of the input variables; see Hastie, Tibshirani, and Friedman (2009, Chapter 10.11) for a more in-depth discussion on this matter.

The fact that boosting algorithms often use shallow trees as base classifiers is a clear distinction from the (somewhat similar) bagging method. Bagging is a pure variance reduction technique based on averaging and it can not reduce the bias of the individual base models. Hence, for bagging to be successful we need to use base models with low bias (but possibly high variance)—typically very deep decision trees. Boosting on the other hand can reduce both the variance *and* the bias of the base models, making it possible to use very simple base models as described above.

Another important difference between bagging and boosting is that the former is parallel whereas the latter is sequential. Each iteration of a boosting algorithm introduces a new base model aiming at reducing the errors made by the current model. In bagging, on the other hand, all base models are identically distributed and they all try to solve the same problem, with the final model being a simple average over the ensemble. An effect of this is that bagging *does not* overfit as a result of using too many ensemble members (see also the discussion in Section 6.1). Unfortunately, this is not the case for boosting. Indeed, a boosting model becomes more and more flexible as the number of iterations B increase and using too

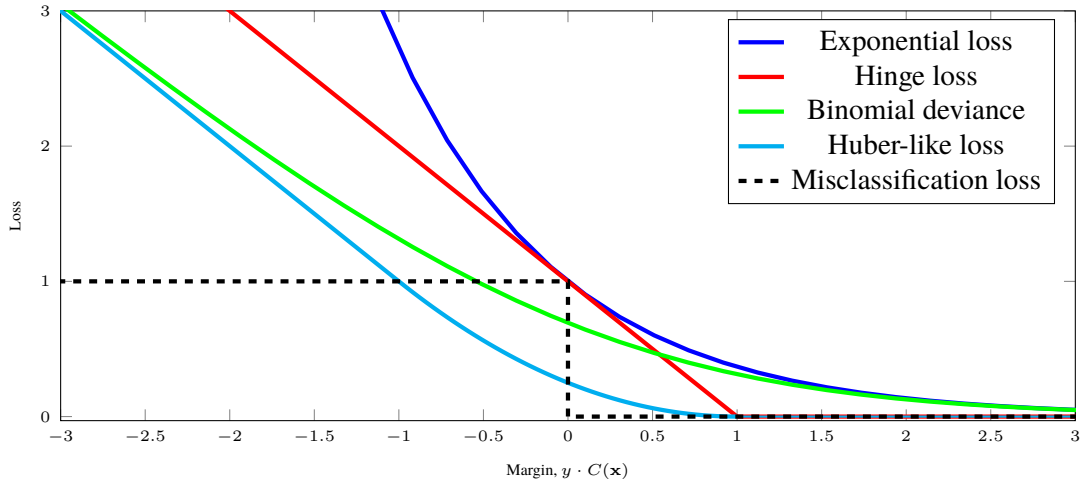


Figure 6.3: Comparison of common loss functions for classification.

many base models can result in overfitting. It has been observed in practice, however, that this overfitting often occurs slowly and the performance tends to be rather insensitive to the choice of B. Nevertheless, it is a good practice to select B in some systematic way, for instance using so called early stopping (this is common also in the context of training neural networks; see Section 7.4.4).

6.3.5 Robust loss functions and gradient boosting

As pointed out above, the margin $y \cdot C(\mathbf{x})$ can be used as a measure of the error made by the classifier $\widehat{y}(\mathbf{x}) = \text{sign}\{C(\mathbf{x})\}$, where negative margins correspond to incorrect classifications and positive margins correspond to correct classifications. It is therefore natural to use a loss function which is a decreasing function of the margin: negative margins should be penalized more than positive margins. The exponential loss function (6.6)—which was used in the derivation of the AdaBoost algorithm—satisfies this requirement, as can be seen in Figure 6.2. However, this loss function also penalizes negative margins very heavily. This can be an issue in practical applications, making the classifier sensitive to noisy data and "outliers", such as mislabeled or atypical data points.

To address these limitations we can consider using some other, more *robust*, loss function in place of the exponential loss. A few examples of commonly used loss functions for classification are shown in Figure 6.3 (see Section 6.A for the mathematical definitions of these functions). An in-depth discussion of the rationale and pros and cons of these different loss functions is beyond the scope of these lecture notes and we refer the interested reader to Hastie, Tibshirani, and Friedman (2009, Chapter 10.6). However, we note that all the alternative loss functions illustrated in the figure have less "aggressive" penalties for large negative margins compared to the exponential loss, i.e., their slopes are not as sharp,[5] making them more robust to noisy data.

Why then have we not used a more robust loss function in the derivation of the AdaBoost algorithm? The reason for this is mainly computational. Using exponential loss is convenient since it leads to a closed form solution to the optimization problem in (6.9). If we instead use another loss function this analytical tractability is unfortunately lost.

However, this difficulty can be dealt with by using techniques from numerical optimization. This approach is complicated to some extent by the fact that the optimization "variable" in (6.9a) is the base classifier $\widehat{y}(\mathbf{x})$ itself. Hence, it is not possible to simply use an off-the-shelf numerical optimization algorithm to solve this problem. That being said, however, it has been realized that it is possible to approximately solve (6.9a) for rather general loss function using a method reminiscent of gradient descent (Appendix B). The resulting method is referred to as *gradient boosting* Friedman 2001; Mason et al. 1999.

[5] Hinge loss, binomial deviance, and the Huber-like loss all increase linearly for large negative margins. Exponential loss, of course, increases exponentially.

We provide pseudo-code for one instance of a gradient boosting method in algorithm 7. As can be seen from the algorithm, the key step involves fitting a base model to the *negative gradient of the loss function*. This can be understood via the intuitive interpretation of boosting, that each base model should try to correct the mistakes made by the ensemble thus far. The negative gradient of the loss function gives an indication of in which "direction" the model should be updated in order to reduce the loss.

Algorithm 7: A gradient boosting algorithm

1. Initialize (as a constant), $C^0(\mathbf{x}) \equiv \arg\min_c \sum_{i=1}^n L(y_i, c)$.
2. For $b = 1$ to B
   (a) Compute the negative gradient of the loss function,
   $$g_i^b = -\left[\frac{\partial L(y_i, c)}{\partial c}\right]_{c=C^{b-1}(\mathbf{x}_i)}, \quad i = 1, \ldots, n.$$
   (b) Train a base *regression* model $\widehat{f}^b(\mathbf{x})$ to fit the gradient values,
   $$\widehat{f}^b = \arg\min_f \sum_{i=1}^n \left(f(\mathbf{x}_i) - g_i^b\right)^2.$$
   (c) Update the boosted model,
   $$C^b(\mathbf{x}) = C^{b-1}(\mathbf{x}) + \gamma \widehat{f}^b(\mathbf{x})$$
3. Output $\widehat{y}^B_{\text{boost}}(\mathbf{x}) = \text{sign}\{C^B(\mathbf{x})\}$.

While presented for classification in algorithm 7, gradient boosting can also be used for regression with minor modifications. In fact, an interesting aspect of the algorithm presented here is that the base models $\widehat{f}^b(\mathbf{x})$ are found by solving a *regression problem* despite the fact that the algorithm produces a classifier. The reason for this is that the negative gradient values $\{g_i^b\}_{i=1}^n$ are quantitative variables, even if the data $\{y_i\}_{i=1}^n$ is qualitative. Here we have considered fitting a base model to these negative gradient values by minimizing a square loss criterion.

The value γ used in the algorithm (line 2(c)) is a tuning parameter which plays a similar role to the step size in ordinary gradient descent. In practice this is usually found by line search (see Appendix B), often combined with a type of regularization via shrinkage (Friedman 2001). When using trees as base models—as is common in practice—optimizing the steps size can be done jointly with finding the terminal node values, resulting in a more efficient implementation (Friedman 2001).

As mentioned above, gradient boosting requires a certain amount of smoothness in the loss function. A minimal requirement is that it is almost everywhere differentiable, so that it is possible to compute the gradient of the loss function. However, some implementations of gradient boosting require stronger conditions, such as second order differentiability. The binomial deviance (see Figure 6.3) is in this respect a "safe choice" which is infinitely differentiable and strongly convex, while still enjoying good statistical properties. As a consequence, binomial deviance is one of the most commonly used loss functions in practice.

6.A Classification loss functions

The classification loss functions illustrated in Figure 6.3 are:

Exponential loss: $L(y, c) = \exp(-yc)$.

Hinge loss: $L(y, c) = \begin{cases} 1 - yc & \text{for } yc < 1, \\ 0 & \text{otherwise.} \end{cases}$

Binomial deviance: $L(y, c) = \log(1 + \exp(-yc))$.

Huber-like loss: $L(y, c) = \begin{cases} -yc & \text{for } yc < -1, \\ \frac{1}{4}(1 - yc)^2 & \text{for } -1 \leq yc \leq 0, \\ 0 & \text{otherwise.} \end{cases}$

Misclassification loss: $L(y, c) = \begin{cases} 1 & \text{for } yc < 0, \\ 0 & \text{otherwise.} \end{cases}$

7 Neural networks and deep learning

Neural networks can be used for both regression and classification, and they can be seen as an extension of linear regression and logistic regression, respectively. Traditionally neural networks with *one* so-called hidden layer have been used and analysed, and several success stories came in the 1980s and early 1990s. In the 2000s it was, however, realized that *deep* neural networks with *several* hidden layers, or simply *deep learning*, are even more powerful. With the combination of new software, hardware, parallel algorithms for training and a lot of training data, deep learning has made a major contribution to machine learning. Deep learning has excelled in many applications, including image classification, speech recognition and language translation. New applications, analysis, and algorithmic developments to deep learning are published literally every day.

We will start in Section 7.1 by generalizing linear regression to a two-layer neural network (i.e., a neural network with one hidden layer), and then generalize it further to a deep neural network. We thereafter leave regression and look at the classification setting in Section 7.2. In Section 7.3 we present a special neural network tailored for images and finally we look in to some of the details on how to train neural networks in Section 7.4.

7.1 Neural networks for regression

A neural network is a nonlinear function that describes the output variable y as a nonlinear function of its input variables

$$y = f(x_1, \ldots, x_p; \boldsymbol{\theta}) + \epsilon, \tag{7.1}$$

where ϵ is an error term and the function f is parametrized by $\boldsymbol{\theta}$. Such a nonlinear function can be parametrized in many ways. In a neural network, the strategy is to use several *layers* of linear regression models and nonlinear *activation functions*. We will explain this carefully in turn below. For the model description it will be convenient to define z as the output without the noise term ϵ,

$$z = f(x_1, \ldots, x_p; \boldsymbol{\theta}). \tag{7.2}$$

7.1.1 Generalized linear regression

We start the description with a graphical illustration of the linear regression model

$$z = \beta_0 1 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p, \tag{7.3}$$

which is shown in Figure 7.1a. Each input variable x_j is represented with a node and each parameter β_j with a link. Furthermore, the output z is described as the sum of all terms $\beta_i x_j$. Note that we use 1 as the input variable corresponding to the offset term β_0.

To describe *nonlinear* relationships between $\mathbf{x} = [1 \; x_1 \; x_2 \; \ldots \; x_p]^\mathsf{T}$ and z we introduce a nonlinear scalar function called the *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$. The linear regression model (7.3) is now modified into a *generalized* linear regresssion model where the linear combination of the inputs is squashed through the (scalar) activation function

$$z = \sigma\big(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\big). \tag{7.4}$$

This extension to the generalized linear regression model is visualized in Figure 7.1b.

(a) **(b)**

Figure 7.1: Graphical illustration of a linear regression model (Figure 7.1a), and a generalized linear regression model (Figure 7.1b). In Figure 7.1a, the output z is described as the sum of all terms β_0 and $\{\beta_j x_j\}_{j=1}^{p}$, as in (7.3). In Figure 7.1b, the circle denotes addition and also transformation through the activation function σ, as in (7.4).

Logistic: $\sigma(x) = \frac{1}{1+e^{-x}}$ ReLU: $\sigma(x) = \max(0, x)$

(a) **(b)**

Figure 7.2: Two common activation functions used in neural networks. The logistic (or sigmoid) function (Figure 7.2a), and the rectified linear unit (Figure 7.2b).

Common choices for activation function are the *logistic function* and the *rectified linear unit* (ReLU). These are illustrated in Figure 7.2a and Figure 7.2b, respectively. The logistic (or sigmoid) function has already been used in the context of logistic regression (Section 3.2). The logistic function is affine close to $x = 0$ and saturates at 0 and 1 as x decreases or increases. The ReLU is even simpler. The function is the identity function for positive inputs and equal to zero for negative inputs.

The logistic function used to be the standard choice of activation function in neural networks for many years, whereas the ReLU has gained in popularity (despite its simplicity!) during recent years and it is now the standard choice in most neural network models.

The generalized linear regression model (7.4) is very simple and is itself not capable of describing very complicated relationships between the input $\mathbf{x}$ and the output z. Therefore, we make two further extensions to increase the generality of the model: We will first make use of *several* generalized linear regression models to build a layer (which will lead us to the *two-layer* neural network) and then stack these layers in a *sequential* construction (which will result in a *deep* neural network, or simply *deep learning*).

7.1.2 Two-layer neural network

In (7.4), the output is constructed by one scalar regression model. To increase its flexibility and turn it into a two-layer neural network, we instead let the output be a sum of M such generalized linear regression models, each of which has its own parameters. The parameter for the ith regression model are $\beta_{0i}, \ldots, \beta_{pi}$ and we denote its output by h_i,

$$h_i = \sigma\left(\beta_{0i} + \beta_{1i}x_1 + \beta_{2i}x_2 + \cdots + \beta_{pi}x_p\right), \qquad i = 1, \ldots, M. \tag{7.5}$$

These intermediate outputs h_i are so-called *hidden units*, since they are not the output of the whole model. The M different hidden units $\{h_i\}_{i=1}^{M}$ instead act as input variables to an additional linear regression model

$$z = \beta_0 + \beta_1 h_1 + \beta_2 h_2 + \cdots + \beta_M h_M. \tag{7.6}$$

To distinguish the parameters in (7.5) and (7.6) we add the superscripts (1) and (2), respectively. The equations describing this two-layer neural network (or equivalently, neural network with one layer of

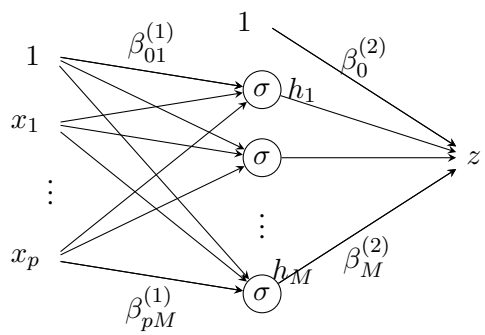


Figure 7.3: A two-layer neural network, or equivalently, a neural network with one intermediate layer of hidden units.

hidden units) are thus

$$\begin{aligned} h_1 &= \sigma\left(\beta_{01}^{(1)} + \beta_{11}^{(1)}x_1 + \beta_{21}^{(1)}x_2 + \cdots + \beta_{p1}^{(1)}x_p\right), \\ h_2 &= \sigma\left(\beta_{02}^{(1)} + \beta_{12}^{(1)}x_1 + \beta_{22}^{(1)}x_2 + \cdots + \beta_{p2}^{(1)}x_p\right), \\ &\vdots \\ h_M &= \sigma\left(\beta_{0M}^{(1)} + \beta_{1M}^{(1)}x_1 + \beta_{2M}^{(1)}x_2 + \cdots + \beta_{pM}^{(1)}x_p\right), \end{aligned} \tag{7.7a}$$

$$z = \beta_0^{(2)} + \beta_1^{(2)}h_1 + \beta_2^{(2)}h_2 + \cdots + \beta_M^{(2)}h_M. \tag{7.7b}$$

Extending the graphical illustration from Figure 7.1, this model can be depicted as a graph with two-layers of links (illustrated using arrows), see Figure 7.3. As before, each link has a parameter associated with it. Note that we include an offset term not only in the input layer, but also in the hidden layer.

7.1.3 Matrix notation

The two-layer neural network model in (7.7) can also be written more compactly using matrix notation, where the parameters in each layer are stacked in a *weight matrix* $\mathbf{W}$ and an *offset vector*[1] $\mathbf{b}$ as

$$\mathbf{b}^{(1)} = \begin{bmatrix} \beta_{01}^{(1)} & \cdots & \beta_{0M}^{(1)} \end{bmatrix}, \quad \mathbf{W}^{(1)} = \begin{bmatrix} \beta_{11}^{(1)} & \cdots & \beta_{1M}^{(1)} \\ \vdots & \cdots & \vdots \\ \beta_{p1}^{(1)} & \cdots & \beta_{pM}^{(1)} \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} \beta_0^{(2)} \end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix} \beta_1^{(2)} \\ \vdots \\ \beta_M^{(2)} \end{bmatrix}. \tag{7.8}$$

The full model can then be written as

$$\mathbf{h} = \sigma\left(\mathbf{W}^{(1)\mathsf{T}}\mathbf{x} + \mathbf{b}^{(1)\mathsf{T}}\right), \tag{7.9a}$$

$$z = \mathbf{W}^{(2)\mathsf{T}}\mathbf{h} + \mathbf{b}^{(2)\mathsf{T}}, \tag{7.9b}$$

where we have also stacked the components in $\mathbf{x}$ and $\mathbf{h}$ as $\mathbf{x} = [x_1, \ldots, x_p]^\mathsf{T}$ and $\mathbf{h} = [h_1, \ldots, h_M]^\mathsf{T}$. The activation function σ acts element-wise. The two weight matrices and the two offset vectors will be

[1] The word "bias" is often used for the offset vector in the neural network literature, but this is really just a model parameter and not a bias in the statistical sense. To avoid confusion we refer to it as an offset instead.

the parameters in the model, which can be written as

$$\boldsymbol{\theta} = \begin{bmatrix} \text{vec}(\mathbf{W}^{(1)})^\mathsf{T} & \text{vec}(\mathbf{W}^{(2)})^\mathsf{T} & \mathbf{b}^{(1)} & \mathbf{b}^{(2)} \end{bmatrix}^\mathsf{T}. \tag{7.10}$$

By this we have described a nonlinear regression model on the form $y = f(\mathbf{x}; \boldsymbol{\theta}) + \epsilon$ where $f(\mathbf{x}; \boldsymbol{\theta}) = z$ according to above. Note that the predicted output z in (7.9b) depends on all the parameters in $\boldsymbol{\theta}$ even though it is not explicitly stated in the notation.

7.1.4 Deep neural network

The two-layer neural network is a useful model on its own, and a lot of research and analysis has been done for it. However, the real descriptive power of a neural network is realized when we stack multiple such layers of generalized linear regression models, and thereby achieve a *deep* neural network. Deep neural networks can model complicated relationships (such as the one between an image and its class), and is one of the state-of-the-art methods in machine learning as of today.

We enumerate the layers with index l. Each *layer* is parametrized with a weight matrix $\mathbf{W}^{(l)}$ and an offset vector $\mathbf{b}^{(l)}$, as for the two-layer case. For example, $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ belong to layer $l = 1$, $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ belong to layer $l = 2$ and so forth. We also have multiple *layers of hidden units* denoted by $\mathbf{h}^{(l-1)}$. Each such layer consists of M_l hidden units $\mathbf{h}^{(l)} = [h_1^{(l)}, \ldots, h_{M_l}^{(l)}]^\mathsf{T}$, where the dimensions M_1, M_2, $\ldots$ can be different for different layers.

Each layer maps a hidden layer $\mathbf{h}^{(l-1)}$ to the next hidden layer $\mathbf{h}^{(l)}$ as

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)\mathsf{T}}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)\mathsf{T}}). \tag{7.11}$$

This means that the layers are stacked such that the output of the first layer $\mathbf{h}^{(1)}$ (the first layer of hidden units) is the input to the second layer, the output of the second layer $\mathbf{h}^{(2)}$ (the second layer of hidden units) is the input to the third layer, etc. By stacking multiple layers we have constructed a *deep* neural network. A deep neural network of L layers can mathematically be described as (cf. (7.9))

$$\begin{aligned} \mathbf{h}^{(1)} &= \sigma(\mathbf{W}^{(1)\mathsf{T}}\mathbf{x} + \mathbf{b}^{(1)\mathsf{T}}), \\ \mathbf{h}^{(2)} &= \sigma(\mathbf{W}^{(2)\mathsf{T}}\mathbf{h}^{(1)} + \mathbf{b}^{(2)\mathsf{T}}), \\ &\vdots \\ \mathbf{h}^{(L-1)} &= \sigma(\mathbf{W}^{(L-1)\mathsf{T}}\mathbf{h}^{(L-2)} + \mathbf{b}^{(L-1)\mathsf{T}}), \\ \mathbf{z} &= \mathbf{W}^{(L)\mathsf{T}}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)\mathsf{T}}. \end{aligned} \tag{7.12}$$

A graphical representation of this model is represented in Figure 7.4.

The weight matrix $\mathbf{W}^{(1)}$ for the first layer $l = 1$ has the dimension $p \times M_1$ and the corresponding offset vector $\mathbf{b}^{(1)}$ the dimension $1 \times M_1$. In deep learning it is common to consider applications where also the output is multi-dimensional $\mathbf{z} = [z_1, \ldots, z_K]^\mathsf{T}$. This means that for the last layer the weight matrix $\mathbf{W}^{(L)}$ has the dimension $M_{L-1} \times K$ and the offset vector $\mathbf{b}^{(L)}$ the dimension $1 \times K$. For all intermediate layers $l = 2, \ldots, L-1$, $\mathbf{W}^{(l)}$ has the dimension $M_{l-1} \times M_l$ and the corresponding offset vector $1 \times M_l$.

The number of inputs p and the number of outputs K are given by the problem, but the number of layers L and the dimensions M_1, M_2, $\ldots$ are user design choices that will determine the flexibility of the model.

7.1.5 Learning the network from data

Analogously to the parametric models presented earlier (e.g. linear regression and logistic regression) we need to learn all the parameters in order to use the model. For deep neural networks the parameters are

$$\boldsymbol{\theta} = \begin{bmatrix} \text{vec}(\mathbf{W}^{(1)})^\mathsf{T} & \text{vec}(\mathbf{W}^{(2)})^\mathsf{T} & \cdots \text{vec}(\mathbf{W}^{(L)})^\mathsf{T} & \mathbf{b}^{(1)\mathsf{T}} & \mathbf{b}^{(2)} & \cdots & \mathbf{b}^{(L)} \end{bmatrix}^\mathsf{T} \tag{7.13}$$

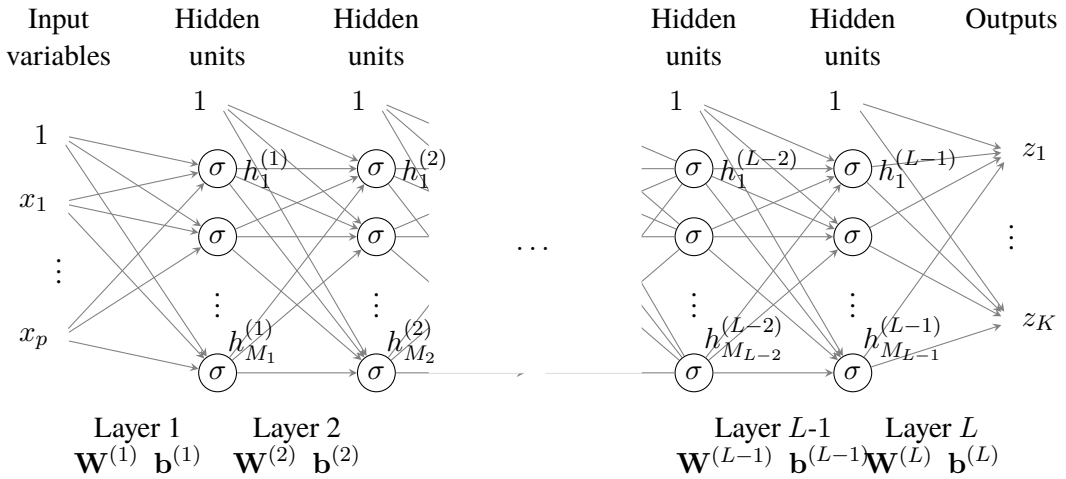


Figure 7.4: A deep neural network with L layers. Each layer is parameterized with $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$.

The wider and deeper the network is, the more parameters there are. Practical deep neural networks can easily have in the order of millions of parameters and these models are therefore also extremely flexible. Hence, some mechanism to avoid overfitting is needed. Regularization such as ridge regression is common (cf. Section 2.6), but there are also other techniques specific to deep learning; see further Section 7.4.4. Furthermore, the more parameters there are, the more computational power is needed to train the model. As before, the training data $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ consists of n samples of the input $\mathbf{x}$ and the output $\mathbf{y}$.

For a regression problem we typically start with maximum likelihood and assume Gaussian noise $\epsilon \sim \mathcal{N}\left(0, \sigma_\epsilon^2\right)$, and thereby obtain the square error loss function as in Section 2.3.1,

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) \qquad \text{where} \qquad L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) = \|\mathbf{y}_i - f(\mathbf{x}_i; \boldsymbol{\theta})\|^2 = \|\mathbf{y}_i - \mathbf{z}_i\|^2. \tag{7.14}$$

This problem can be solved with numerical optimization, and more precisely stochastic gradient descent. This is described in more detail in Section 7.4.

From the model, the parameters $\boldsymbol{\theta}$, and the inputs $\{\mathbf{x}_i\}_{i=1}^n$ we can compute the predicted outputs $\{\mathbf{z}_i\}_{i=1}^n$ using the model $\mathbf{z}_i = f(\mathbf{x}_i; \boldsymbol{\theta})$. For example, for the two-layer neural network presented in Section 7.1.2 we have

$$\mathbf{h}_i^\mathsf{T} = \sigma(\mathbf{x}_i^\mathsf{T}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \tag{7.15a}$$

$$\mathbf{z}_i^\mathsf{T} = \mathbf{h}_i^\mathsf{T}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \tag{7.15b}$$

In (7.15) the equations are transposed in comparison to the model in (7.9). This is a small trick such that we easily can extend (7.15) to include multiple data points i. Similar to (2.4) we stack all data points in matrices, where each data point represents one row

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}_1^\mathsf{T} \\ \vdots \\ \mathbf{y}_n^\mathsf{T} \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix}, \quad \mathbf{Z} = \begin{bmatrix} \mathbf{z}_1^\mathsf{T} \\ \vdots \\ \mathbf{z}_n^\mathsf{T} \end{bmatrix}, \quad \text{and} \quad \mathbf{H} = \begin{bmatrix} \mathbf{h}_1^\mathsf{T} \\ \vdots \\ \mathbf{h}_n^\mathsf{T} \end{bmatrix}. \tag{7.16}$$

We can then write (7.15) as

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \tag{7.17a}$$

$$\mathbf{Z} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \tag{7.17b}$$

where we also have stacked the predicted output and the hidden units in matrices. This is also how the model would be implemented in code. In Tensorflow, which will be used in the laboratory work in the course, it can be written as

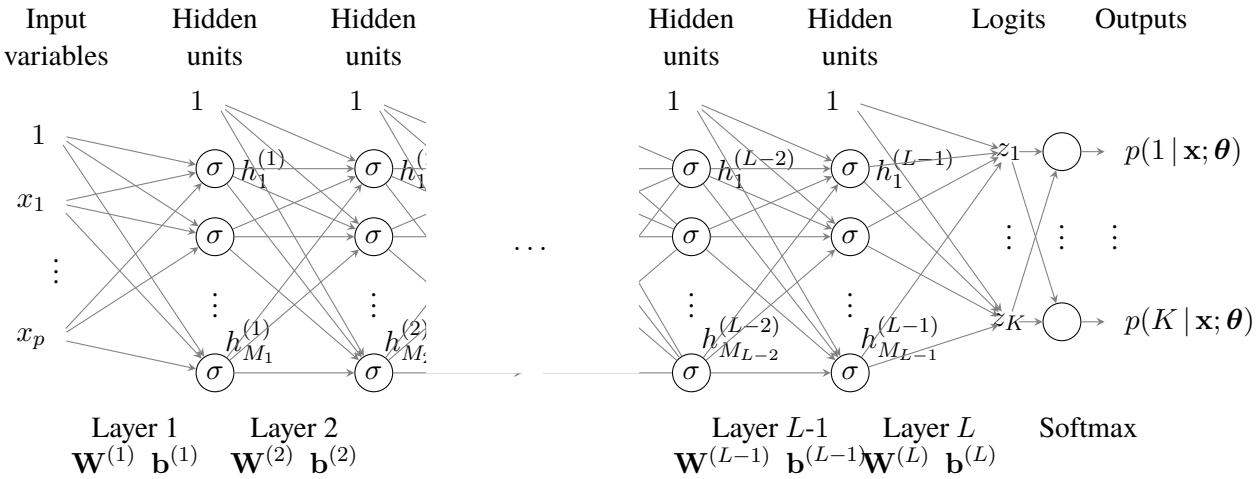


Figure 7.5: A deep neural network with L layers for classification. The only difference to regression (Figure 7.4) is the softmax transformation after layer L.

```
H = tf.sigmoid(tf.matmul(X, W1) + b1)
Z = tf.matmul(H, W2) + b2
```

Note that in (7.17) the offset vectors $\mathbf{b}_1$ and $\mathbf{b}_2$ are added and broadcasted to each row. See more details regarding implementation of a neural network in the instructions for the laboratory work.

7.2 Neural networks for classification

Neural networks can also be used for classification where we have qualitative outputs $y \in \{1, \ldots, K\}$ instead of quantitative. In Section 3.2 we extended linear regression to logistic regression by adding the logistic function to the output. In the same manner we can extend the neural network presented in the last section to a neural network for classification. In doing this extension, we will use the multi-class version of logistic regression presented in Section 3.2.3, and more specifically the softmax parametrization given in (3.13), repeated here for convenience

$$\text{softmax}(\mathbf{z}) = \frac{1}{\sum_{j=1}^{K} e^{z_j}} \begin{bmatrix} e^{z_1} & \cdots & e^{z_K} \end{bmatrix}^\mathsf{T}. \tag{7.18}$$

The softmax function acts as an additional activation function on the final layer of the neural network. In addition to the regression network in (7.12) we add the softmax function at the end of the network as

$$\begin{aligned} &\vdots \\ \mathbf{z} &= \mathbf{W}^{(L)\mathsf{T}} H^{(L-1)} + \mathbf{b}^{(L)\mathsf{T}}, && \text{(7.19a)} \\ \begin{bmatrix} p(1 \mid \mathbf{x}_i) & p(2 \mid \mathbf{x}_i) & \ldots & p(K \mid \mathbf{x}_i) \end{bmatrix}^\mathsf{T} &= \text{softmax}(\mathbf{z}) && \text{(7.19b)} \end{aligned}$$

The softmax function thus maps the output of the last layer $z_1, \ldots, z_K$ to the modeled class probabilities $p(1 \mid \mathbf{x}_i), \ldots, p(K \mid \mathbf{x}_i)$, see also Figure 7.5 for a graphical illustration. The inputs to the softmax function, i.e. the variables $z_1, \ldots, z_K$, are referred to as *logits*.

Note that the softmax function does not come as a layer with additional parameters, it only transforms the previous output $[z_1 \ \ldots \ z_K]^\mathsf{T} \in \mathbb{R}^K$ to the modeled probabilities $[p(1 \mid \mathbf{x}_i) \ \ldots \ p(K \mid \mathbf{x}_i)]^\mathsf{T} \in [0, 1]^K$. Also note that by construction of the softmax function, these values will sum to 1 regardless of the values of $[z_1, \ldots, z_K]$ (otherwise it would not be probabilities).

	$k=1$	$k=2$	$k=3$
y_{ik}	0	**1**	0
$p(k \mid \mathbf{x}_i; \boldsymbol{\theta}_A)$	0.1	**0.8**	0.1

Cross-entropy:
$L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_A) = -1 \cdot \log 0.8 = 0.22$

	$k=1$	$k=2$	$k=3$
y_{ik}	0	**1**	0
$p(k \mid \mathbf{x}_i; \boldsymbol{\theta}_B)$	0.8	**0.1**	0.1

Cross-entropy:
$L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_B) = -1 \cdot \log 0.1 = 2.30$

Figure 7.6: Illustration of the cross-entropy between a data point $\mathbf{y}_i$ and two different prediction outputs.

7.2.1 Learning classification networks from data

As before, the training data consists of n samples of inputs and outputs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$. For the classification problem we use the one-hot encoding for the output $\mathbf{y}_i$. This means that for a problem with K different classes, $\mathbf{y}_i$ consists of K elements $\mathbf{y}_i = \begin{bmatrix} y_{i1} & \dots & y_{iK} \end{bmatrix}^\mathsf{T}$. If a data point i belongs to class k then $y_{ik} = 1$ and $y_{ij} = 0$ for all $j \neq k$. See more about the one-hot encoding in Section 3.2.3.

For a neural network with the softmax activation function on the final layer we typically use the negative log-likelihood, which is also commonly referred to as the *cross-entropy* loss function, to train the model (cf. (3.16))

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) \qquad \text{where} \qquad L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) = -\sum_{k=1}^{K} y_{ik} \log p(k \mid \mathbf{x}_i; \boldsymbol{\theta}). \tag{7.20}$$

The cross-entropy is close to its minimum if the predicted probability $p(k \mid \mathbf{x}_i; \boldsymbol{\theta})$ is close to 1 for the k for which $y_{ik} = 1$. For example, if the ith data point belongs to class $k = 2$ out of in total $K = 3$ classes we have $\mathbf{y}_i = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\mathsf{T}$. Assume that we have a set of parameters of the network denoted $\boldsymbol{\theta}_A$, and with these parameters we predict $p(1 \mid \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.1$, $p(2 \mid \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.8$ and $p(3 \mid \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.1$ meaning that we are quite sure that data point i actually belongs to class $k = 2$. This would generate a low cross-entropy $L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_A) = -(0 \cdot \log 0.1 + 1 \cdot \log 0.8 + 0 \cdot \log 0.1) = 0.22$. If we instead predict $p(1 \mid \mathbf{x}_i; \boldsymbol{\theta}_B) = 0.8$, $p(2 \mid \mathbf{x}_i; \boldsymbol{\theta}_B) = 0.1$ and $p(3 \mid \mathbf{x}_i; \boldsymbol{\theta}_B) = 0.1$, the cross-entropy would be much higher $L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_B) = -(0 \cdot \log 0.8 + 1 \cdot \log 0.1 + 0 \cdot \log 0.1) = 2.30$. For this case, we would indeed prefer the parameters $\boldsymbol{\theta}_A$ over $\boldsymbol{\theta}_B$. This is summarized in Figure 7.6.

Computing the loss function explicitly via the logarithm could lead to numerical problems when $p(k \mid \mathbf{x}_i; \boldsymbol{\theta})$ is close to zero since $\log(x) \to -\infty$ as $x \to 0$. This can be avoided since the logarithm in the cross-entropy loss function (7.20) can "undo" the exponential in the softmax function (7.18),

$$\begin{aligned} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) &= -\sum_{k=1}^{K} y_{ik} \log p(k \mid \mathbf{x}_i; \boldsymbol{\theta}) = -\sum_{k=1}^{K} y_{ik} \log[\operatorname{softmax}(\mathbf{z}_i)]_k \\ &= -\sum_{k=1}^{K} y_{ik} \left(z_{ik} - \log \left\{ \textstyle\sum_{j=1}^{K} e^{z_{ij}} \right\} \right), \qquad (7.21) \\ &= -\sum_{k=1}^{K} y_{ik} \left(z_{ik} - \max_j z_{ij} - \log \left\{ \textstyle\sum_{j=1}^{K} e^{z_{ij} - \max_j z_{ij}} \right\} \right), \qquad (7.22) \end{aligned}$$

where z_{ik} are the logits.

7.3 Convolutional neural networks

Convolutional neural networks (CNN) are a special kind neural networks tailored for problems where the input data has a grid-like topology. In this text we will focus on images, which have a 2D-topology of pixels. Images are also the most common type of input data in applications where CNNs are applied. However, CNNs can be used for any input data on a grid, also in 1D (e.g. audio waveform data) and 3D (volumetric data e.g. CT scans or video data). We will focus on grayscale images, but the approach can easily be extended to color images as well.

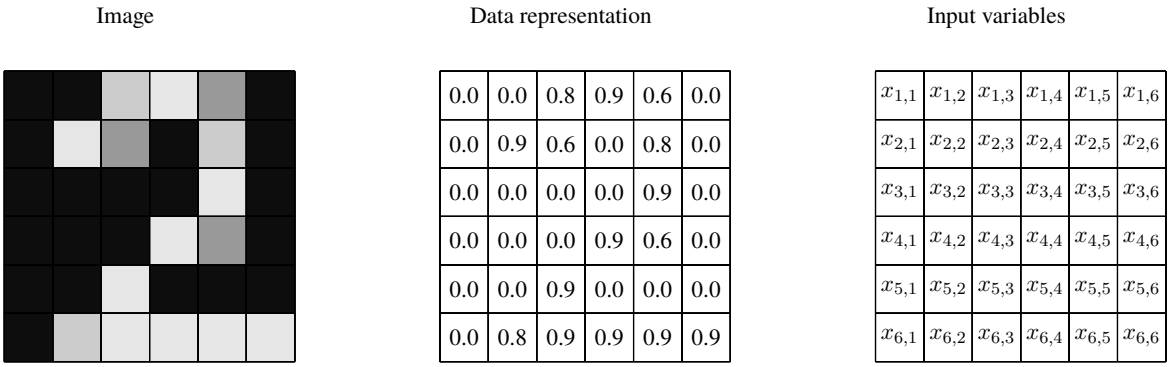


Figure 7.7: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number describing the grayscale color. We denote the whole image as $\mathbf{X}$ (a matrix), and each pixel value is an input variable $x_{j,k}$ (element in the matrix $\mathbf{X}$).

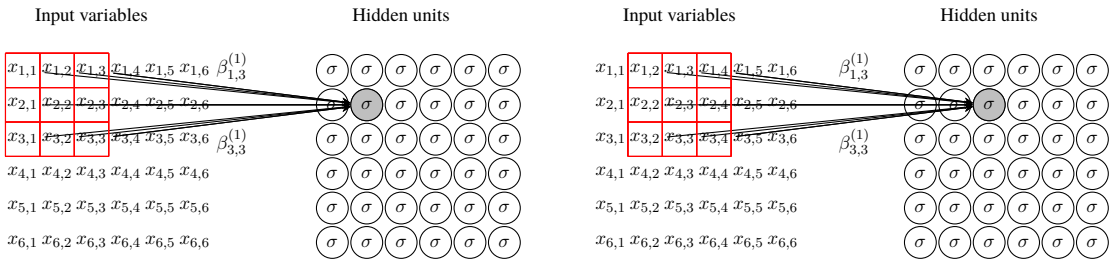


Figure 7.8: An illustration of the interactions in a convolutional layer: Each hidden unit (circle) is only dependent on the pixels in a small region of the image (red boxes), here of size 3×3 pixels. The location of the hidden unit corresponds to the location of the region in the image: if we move to a hidden unit one step to the right, the corresponding region also moves one step to the right, compare Figure 7.8a and Figure 7.8b. Furthermore, the nine parameters $\beta^{(1)}_{1,1}, \beta^{(1)}_{1,2}, \ldots, \beta^{(1)}_{3,3}$ are the same for all hidden units in the layer.

7.3.1 Data representation of an image

Digital grayscale images consist of pixels ordered in a matrix. Each pixel can be represented as a range from 0 (total absence, black) to 1 (total presence, white) and values between 0 and 1 represent different shades of gray. In Figure 7.7 this is illustrated for an image with 6×6 pixels. In an image classification problem, an image is the input $\mathbf{x}$ and the pixels in the image are the input variables $x_{1,1}, x_{1,2}, \ldots, x_{6,6}$. The two indices j and k determine the position of the pixel in the image, as illustrated in Figure 7.7.

If we put all input variables representing the images pixels in a long vector, we can use the network architecture presented in Section 7.1 and 7.2 (and that is what we will do in the laboratory work to start with!). However, by doing that, a lot of the structure present in the image data will be lost. For example, we know that two pixels close to each other have more in common than two pixels further apart. This information would be destroyed by such a vectorization. In contrast, CNNs preserve this information by representing the input variables as well as the hidden layers as matrices. The core component in a CNN is the convolutional layer, which will be explained next.

7.3.2 The convolutional layer

Following the input layer, we use a hidden layer with equally many hidden units as input variables. For the image with 6×6 pixels we consequently have $6 \times 6 = 36$ hidden units. We choose to order the hidden units in a 6×6 matrix, i.e. in the same manner as we did for the input variables, see Figure 7.8a.

The network layers presented in earlier sections (like the one in Figure 7.3) have been *dense layer*. This means that each input variable is connected to each hidden unit in the following layer, and each such connection has a unique parameter β_{jk} associated with it. These layers have empirically been found to

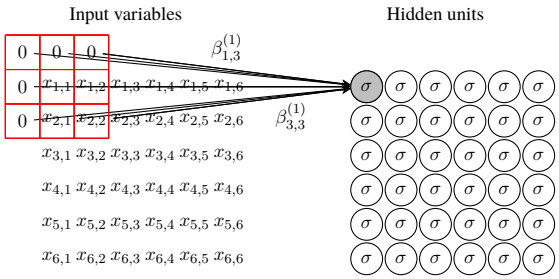


Figure 7.9: An illustration of zero-padding. If the region is partly is outside the image. With zero-padding, the size of the image can be preserved in the following layer.

provide too much flexibility for images and we might not be able to capture the patterns of real importance, and hence not generalize and perform well on unseen data. Instead, a convolutional layer appears to exploit the structure present in images to find a more efficiently parameterized model. In contrast to a dense layer, a convolutional layer leverages two important concepts – *sparse interactions* and *parameter sharing* – to achieve such a parametrization.

Sparse interactions

With sparse interactions we mean that most of the parameters in a dense layer are forced to be equal to zero. More specifically, a hidden unit in a convolutional layer only depends on the pixels in a small region of the image and not on all pixels. In Figure 7.8 this region is of size 3×3. The position of the region is related to the position of the hidden unit in its matrix topology. If we move to the hidden unit one step to the right, the corresponding region in the image also moves one step to the right, as displayed by comparing Figure 7.8a and Figure 7.8b. For the hidden units on the border, the corresponding region is partly located outside the image. For these border cases, we typically use zero-padding where the missing pixels are replaced with zeros. Zero-padding is illustrated in Figure 7.9.

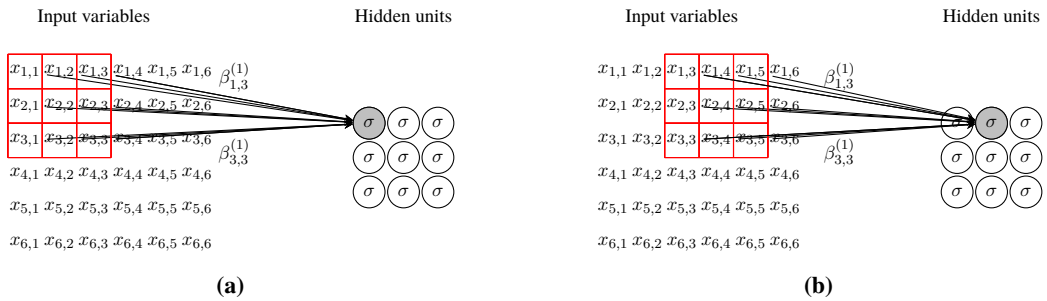
Parameter sharing

In a dense layer each link between an input variable and a hidden unit has its own unique parameter. With parameter sharing we instead let the same parameter be present in multiple places in the network. In a convolutional layer the set of parameters for the different hidden units are all the same. For example, in Figure 7.8a we use the same set of parameters to map the 3×3 region of pixels to the hidden unit as we do in Figure 7.8b. Instead of learning separate sets of parameters for every position we only learn one such set of a few parameters, and use it for all links between the input layer and the hidden units. We call this set of parameters a *kernel*. The mapping between the input variables and the hidden units can be interpreted as a convolution between the input variables and the kernel, hence the name convolutional neural network.

The sparse interactions and parameter sharing in a convolutional layer makes the CNN fairly invariant to translations of objects in the image. If the parameters in the kernel are sensitive to a certain detail (such as a corner, an edge, etc.) a hidden unit will react to this detail (or not) *regardless of where in the image that detail is present*! Furthermore, a convolutional layer uses a lot fewer parameters than the corresponding dense layer. In Figure 7.8 only $3 \cdot 3 + 1 = 10$ parameters are required (including the offset parameter). If we had used a dense layer $(36 + 1) \cdot 36 = 1332$ parameters would have been needed! Another way of interpreting this is: with the same amount of parameters, a convolutional layer can encode more properties of an image than a dense layer.

7.3.3 Condensing information with strides

In the convolutional layer presented above we have equally many hidden units as we have pixels in the image. As we add more layers to the CNN we usually want to condense the information by reducing the number of hidden units at each layer. One way of doing this is by not applying the kernel to every pixel



(a) **(b)**

Figure 7.10: A convolutional layer with stride [2,2] and kernel of size 3×3.

but to say every two pixels. If we apply the kernel to every two pixels both row-wise and column-wise, the hidden units will only have half as many rows and half as many columns. For a 6×6 image we get 3×3 hidden units. This concept is illustrated in Figure 7.10.

The *stride* controls how many pixels the kernel shifts over the image at each step. In Figure 7.8 the stride is [1,1] since the kernel moves by one pixel both row- and column-wise. In Figure 7.10 the stride is [2,2] since it moves by two pixels row- and column-wise. Note that the convolutional layer in Figure 7.10 still requires 10 parameters, as the convolutional layer in Figure 7.8 does. Another way of condensing the information after a convolutional layer is by subsampling the data, so-called *pooling*. The interested can read further about pooling in Goodfellow, Bengio, and Courville 2016.

7.3.4 Multiple channels

The networks presented in Figure 7.8 and 7.10 only have 10 parameters each. Even though this parameterization comes with a lot of advantages, one kernel is probably not sufficient to encode all interesting properties of the images in our data set. To extended the network, we add multiple kernels, each with their own set of kernel parameters. Each kernel produces its own set of hidden units—a so-called *channel*—using the same convolution operation as explained in Section 7.3.2. Hence, each layer of hidden units in a CNN are organized into a tensor with the dimensions (rows $\times$ columns $\times$ channels). In Figure 7.11, the first layer of hidden units has four channels and that hidden layer consequently has dimension $6 \times 6 \times 4$.

When we continue to stack convolutional layers, each kernel depends not only on one channel, but on all the channels in the previous layer. This is displayed in the second convolutional layer in Figure 7.11. As a consequence, each kernel is a tensor of dimension (kernel rows $\times$ kernel columns $\times$ input channels). For example, each kernel in the second convolutional layer in Figure 7.11 is of size $3 \times 3 \times 4$. If we collect all kernels parameters in one weight tensor W, that tensor will be of dimension (kernel rows $\times$ kernel columns $\times$ input channels $\times$ output channels). In the second convolutional layer in Figure 7.11, the corresponding weight matrix $\mathbf{W}^{(2)}$ is a tensor of dimension $3 \times 3 \times 4 \times 6$. With multiple kernels in each convolutional layer, each of them can be sensitive to different features in the image, such as certain edges, lines or circles enabling a rich representation of the images in our training data.

7.3.5 Full CNN architecture

A full CNN architecture consists of multiple convolutional layers. Typically, we decrease the number of rows and columns in the hidden layers as we proceed though the network, but instead increase the number of channels to enable the network to encode more high level features. After a few convolutional layers we usually end a CNN with one or more dense layers. If we consider an image classification task, we place a softmax layer at the very end to get outputs in the range [0,1]. The loss function when training a CNN will be the same as in the regression and classification networks explained earlier, depending on which type of problem we have at hand. In Figure 7.11 a small example of a full CNN architecture is displayed.

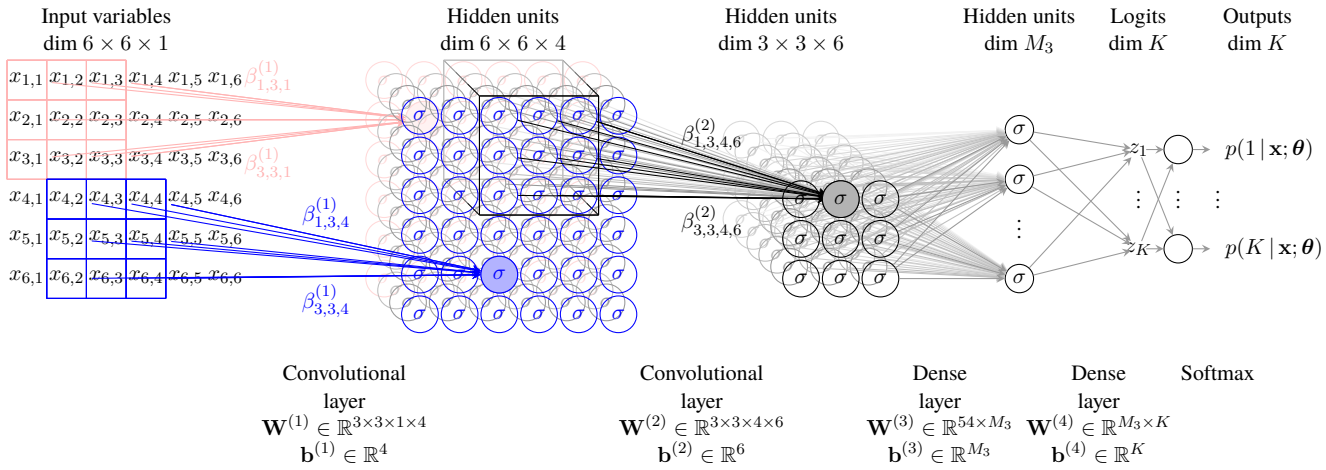


Figure 7.11: A full CNN architecture for classification of grayscale 6×6 images. In the first convolutional layer four kernels, each of size 3×3, produce a hidden layer with four channels. The first channel (in the bottom) is visualized in red and the forth (on the top in blue). We use the stride [1,1] which maintains the number of rows and columns. In the second convolutional layer, six kernels of size $3 \times 3 \times 4$ and the stride [2,2] are used. They produce a hidden layer with 3 rows, 3 columns and 6 channels. After the two convolutional layers follows a dense layer where all $3 \cdot 3 \cdot 6 = 54$ hidden units in the second hidden layer are densely connected to the third layer of hidden units where all links have their unique parameters. We add an additional dense layer mapping down to the K logits. The network ends with a softmax function to provide predicted class probabilities as output.

7.4 Training a neural network

To use a neural network for prediction we need to find an estimate for the parameters $\widehat{\boldsymbol{\theta}}$. To do that we solve an optimization problem on the form

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \qquad \text{where } J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}). \tag{7.23}$$

We denote $J(\boldsymbol{\theta})$ as the *cost function* and $L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$ as the loss function. The functional form of the loss function depends on if we have regression or a classification problem at hand, see e.g. (7.14) and (7.20).

These optimization problems can not be solved in closed form, so numerical optimization has to be used. In Appendix B, an introduction to numerical optimization is provided. In all numerical optimization algorithms the parameters are updated in an iterative manner. In deep learning we typically use various versions of gradient descent:

1. Pick an initialization $\boldsymbol{\theta}_0$.
2. Update the parameters as $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ for $t = 1, 2, \ldots$. (7.24)
3. Terminate when some criterion is fulfilled, and take the last $\boldsymbol{\theta}_t$ as $\widehat{\boldsymbol{\theta}}$.

In many applications of deep learning we cannot afford to compute the exact gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ at each iteration. Instead we use approximations, which are explained in Section 7.4.2. In Section 7.4.3 strategies how to tune the learning rate γ are presented and in Section 7.4.4 a popular regularization method called dropout is described. First, however, a few words on how to initialize the training.

7.4.1 Initialization

The previous optimization problems (LASSO (2.31), logistic regression (3.8)) that we have encountered have all been convex. This means that we can guarantee global convergence regardless of what initialization $\boldsymbol{\theta}_0$ we use. In contrast, the cost functions for training neural networks is usually non-convex. This means that the training is sensitive to the value of the initial parameters. Typically, we initialize all the parameters

to small random numbers such that we ensure that the different hidden units encode different aspects of the data. If the ReLU activation functions are used, offset elements b_0 are typically initialized to a small positive value such that it operates in the non-negative range of the ReLU.

7.4.2 Stochastic gradient descent

Many problems that are addressed with deep learning contain more than a million training data points n, and the design of the neural network is typically made such that $\boldsymbol{\theta}$ has more than a million elements. This provides a computational challenge.

A crucial component is the computation of the gradient required in the optimization routine (7.24)

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}). \tag{7.25}$$

If the number of data points n is big, this operation is costly. However, we can often assume that the data set is redundant meaning that many of the data points are similar. Then the gradient based on the first half of the dataset $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{i=1}^{\frac{n}{2}} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$ is almost identical to the gradient based on the second half of the dataset $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{i=\frac{n}{2}+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$. Consequently, it is a waste of time of compute the gradient based on the whole data set. Instead, we could compute the gradient based on the first half of the data set, update the parameters, and then get the gradient for the new parameters based on the second half of the data,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \frac{1}{n/2} \sum_{i=1}^{\frac{n}{2}} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_t), \tag{7.26a}$$

$$\boldsymbol{\theta}_{t+2} = \boldsymbol{\theta}_{t+1} - \gamma \frac{1}{n/2} \sum_{i=\frac{n}{2}+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_{t+1}). \tag{7.26b}$$

These two steps would only require roughly half the computational time in comparison to if we had used the whole data set for each gradient computation.

The extreme version of this strategy would be to use only a single data point each time when computing the gradient. However, most commonly when training a deep neural network we do something in between, using more than one data point but not all data points when computing the gradient We use a smaller set of training data called a *mini-batch*. Typically, a mini-batch contains $n_b = 10$, $n_b = 100$ or $n_b = 1000$ data points.

One important aspect when using mini-batches is that the different mini-batches are balanced and representative for the whole data set. For example, if we have a big training data set with a few different classes and the data set is sorted after the classes (i.e. samples belonging to class $k = 1$ are first, and so on), a mini-batch with first n_b samples would only include one class and hence not give a good approximation of the gradient for the full data set.

For this reason, we prefer to draw n_b training data points *at random* from the training data to form a mini-batch. One implementation of this is to first randomly shuffle the training data, before dividing it into mini-batches in an order manner. One complete pass through the training data is called an *epoch*. When we have completed one epoch, we do another random reshuffling of the training data and do another pass though the data set. We call this procedure *stochastic gradient descent* or *mini-batch gradient descent*. A pseudo algorithm is presented in Algorithm 8.

Since the neural network model is a composition of multiple layers, the gradient of the loss function with respect to all the parameters $\nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$ can be analytically and efficiently computed by applying the chain rule of differentiation. This is called back-propagation and is not described further here. The interested reader can find more in, for example, Goodfellow, Bengio, and Courville 2016, Section 6.5.

Algorithm 8: Mini-batch gradient descent

1. Initialize all the parameters $\boldsymbol{\theta}_0$ in the network and set $t \leftarrow 1$.
2. For $i = 1$ to E
   a) Randomly shuffle the training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$.
   b) For $j = 1$ to $\frac{n}{n_b}$
      i. Approximate the gradient of the loss function using the mini-batch $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=(j-1)n_b+1}^{jn_b}$,
      $\hat{\mathbf{g}}_t = \frac{1}{n_b} \sum_{i=(j-1)n_b+1}^{jn_b} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$.
      ii. Do a gradient step $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \hat{\mathbf{g}}_t$.
      iii. Update the iteration index $t \leftarrow t + 1$.

7.4.3 Learning rate

An important tuning parameter for (stochastic) gradient descent is the *learning rate* γ. The learning rate γ decides the length of the gradient step that we take at each iteration. If we use a too low learning rate, the estimate $\boldsymbol{\theta}_t$ from one iteration to the next will not change much and the learning will progress slower than necessarily. This is illustrated in Figure 7.12a for a small optimization problem with only one parameter $\boldsymbol{\theta}$.

In contrast, with a too big learning rate, the estimate will pass the optimum and never converge since the step is too long, see Figure 7.12b. For a learning rate which neither is too slow nor too fast, convergence is achieved in a reasonable amount of iterations. A good strategy to find a good learning rate is:

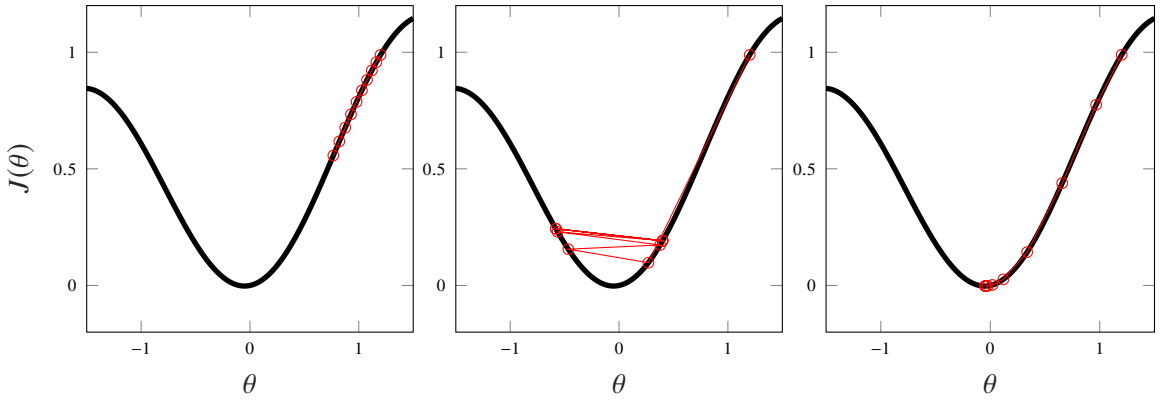
- if the error keeps getting worse or oscillates widely, reduce the learning rate
- if the error is fairly consistently but slowly increasing, increase the learning rate.

Convergence with gradient descent can be achieved with a constant learning rate since the gradient itself approaches zero when we reach the optimum, hence also the gradient step $\gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$. However, this is not true for stochastic gradient descent since the gradient $\hat{\mathbf{g}}_t$ is only an approximation of the true gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$, and $\hat{\mathbf{g}}_t$ will not necessarily approach 0 as $J(\boldsymbol{\theta})$ approaches its minimum. As a consequence, we will make a too big updates as we start approaching the optimum and the stochastic gradient algorithm will not converge. In practice, we instead adjust the learning rate. We start with a fairly high learning rate and then decay the learning rate to a certain level. This can, for example, be achieved by the rule

$$\gamma_t = \gamma_{\text{min}} + (\gamma_{\text{max}} - \gamma_{\text{min}}) e^{-\frac{t}{\tau}}. \tag{7.27}$$

Here the learning rate starts at γ_{max} and goes to γ_{min} as $t \to \infty$. How to pick the parameters γ_{min}, γ_{max} and τ is a more of an art than science. As a rule of thumb γ_{min} can be chosen approximately as 1% of γ_{max}. The parameter τ depends on the size of the data set and the complexity of the problem, but should be chosen such that multiple epochs have passed before we reach γ_{min}. The strategy to pick γ_{max} can be the same as for normal gradient descent explained above.

Under certain regularity conditions and if the so called Robbins-Monro condition holds: $\sum_{t=1}^{\infty} \gamma_t = \infty$ and $\sum_{t=1}^{\infty} \gamma_t^2 < \infty$, then stochastic gradient descent converges almost surely to a local minimum. However, to be able to satisfy the Robbins-Monro condition we need $\gamma_t \to 0$ as $t \to \infty$. In practice this it typically not the case and we instead let the learning rate approach a non-zero level $\gamma_{\text{min}} > 0$ by using a scheme like the one in (7.27). This has been found to work better in practice in many situations, despite sacrificing the theoretical convergence of the algorithm.



(a) Low learning rate $\gamma = 0.05$ **(b)** High learning rate $\gamma = 1.2$ **(c)** Good learning rate $\gamma = 0.3$

Figure 7.12: Optimization using gradient descent of a cost function $J(\theta)$ where θ is a scalar parameter. In the different subfigures we use a too low learning rate (a), a too high learning rate (b), and a good learning rate (c).

7.4.4 Dropout

Like all models presented in this course, neural network models can suffer from overfitting if we have a too flexible model in relation to the complexity of the data. Bagging (James et al. 2013, Chapter 8.2) is one way to reduce the variance and by that also the overfitting of the model. In bagging we train an entire *ensemble* of models. We train all models (ensemble members) on a different data set each, which has been bootstrapped (sampled with replacement) from the original training data set. To make a prediction, we first make one prediction with each model (ensemble member), and then average over all models to obtain the final prediction.

Bagging is also applicable to neural networks. However, it comes with some practical problems; a large neural network model usually takes quite some time to train and it also has quite some parameters to store. To train not just one but an entire ensemble of many large neural networks would thus be very costly, both in terms of runtime and memory. *Dropout* (Srivastava et al. 2014) is a bagging-like technique that allows us to combine many neural networks without the need to train them separately. The trick is to let the different models share parameters with each other, which reduces the computational cost and memory requirement.

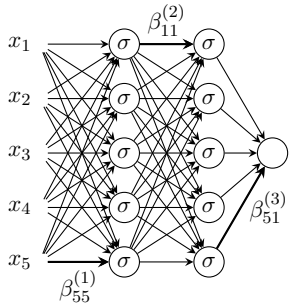
Ensemble of sub-networks

Consider a neural network like the one in Figure 7.13a. In dropout we construct the equivalent to an ensemble member by randomly removing some of the hidden units. We say that we drop the units, hence the name dropout. By this we achieve a sub-network of our original network. Two such sub-networks are displayed in Figure 7.13b. We randomly sample with a pre-defined probability which units to drop, and the collection of dropped units in one sub-network is independent from the collection of dropped units in another sub-network. When a unit is removed, also all its incoming and outgoing connections are removed. Not only hidden units can be dropped, but also input variables.

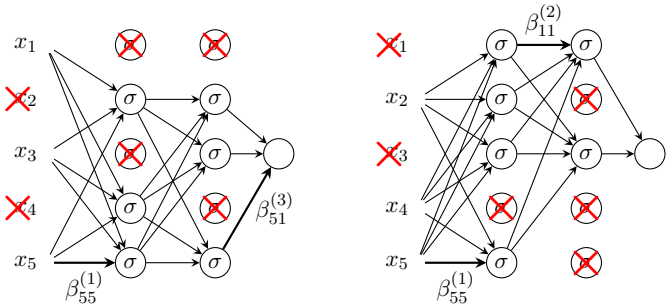
Since all sub-networks are of the very same original network, the different sub-networks share some parameters with each other. For example, in Figure 7.13b the parameter $\beta_{55}^{(1)}$ is present in both sub-networks. The fact that they share parameters with each other allow us to train the ensemble of sub-networks in an efficient manner.

Training with dropout

To train with dropout we use the mini-batch gradient descent algorithm described in Algorithm 8. In each gradient step a mini-batch of data is used to compute an approximation of the gradient, as before. However, instead of computing the gradient for the full network, we generate a random sub-network by



(a) A standard neural network



(b) Two sub-networks

Figure 7.13: A neural network with two hidden layers (a), and two sub-networks with dropped units (b). The collection of units that have been dropped are independent between the two sub-networks.

randomly dropping units as described above. We compute the gradient for that sub-network as if the dropped units were not present and then do a gradient step. This gradient step only updates the parameters present in the sub-network. The parameters that are not present are left untouched. In the next gradient step we grab another mini-batch of data, remove another randomly selected collection of units and update the parameters present in that sub-network. We proceed in this manner until some terminal condition is fulfilled.

Dropout vs bagging

This procedure to generate an ensemble of models differs from bagging in a few ways:

- In bagging all models are independent in the sense that they have their own parameters. In dropout the different models (the sub-networks) share parameters.
- In bagging each model is trained until convergence. In dropout each sub-network is only trained for a singe gradient step. However, since they share parameters all models will be updated also when the other networks are trained.
- Similar to bagging, in dropout we train each model on a data set that has been randomly selected from our training data. However, in bagging we usually do it on a bootstrapped version of the whole data set whereas in dropout each model is trained on a randomly selected mini-batch of data.

Even though dropout differs from bagging in some aspects it has empirically been shown to enjoy similar properties as bagging in terms of avoiding overfittning and reducing the variance of the model.

Prediction at test time

After we have trained the sub-networks, we want to make a prediction based on an unseen input data point $\mathbf{x}_\star$. In bagging we evaluate all the different models in the ensemble and combine their results. This would be infeasible in dropout due to the very large (combinatorial) number of possible sub-networks.

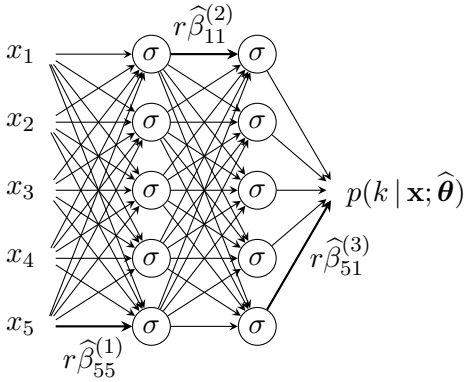


Figure 7.14: The network used for prediction after being trained with dropout. All units and links are present (no dropout) but the weights going out from a certain unit is multiplied with the probability of that unit being included during training. This is to compensate for the fact that some of them where dropped during training. Here all units have been kept with the probability r during training (and dropped with the probability $1 - r$).

However, there is a simple trick to approximately achieve the same result. Instead of evaluating all possible sub-networks we simply evaluate the full network containing all the parameters. To compensate for the fact that the model was trained with dropout, we multiply each estimated parameter going out from a unit with the probability of that unit being included during training. This ensures that the expected value of the input to a unit is the same during training and testing, as during training only a fraction of the incoming links were active. For instance, assume that we during training kept a unit with probability p in all layers, then during testing we multiply all estimated parameters with p before we do a prediction based on network. This is illustrated in Figure 7.14. This procedure of approximating the average over all ensemble members has been shown to work surprisingly well in practice even though there is not yet any solid theoretical argument for the accuracy of this approximation.

Dropout as a regularization method

As a way to reduce the variance and avoid overfitting, dropout can be seen as a regularization method. There are plenty of other regularization methods for neural networks including parameter penalties (like we did in ridge regression and LASSO in Section 2.6.1 and 2.6.2), early stopping (you stop the training before the parameters have converged, and thereby avoid overfitting) and various sparse representations (for example CNNs can be seen as a regularization method where most parameters are forced to be zero) to mention a few. Since its invention, dropout has become one of the most popular regularization techniques due to its simplicity, computationally cheap training and testing procedure and its good performance. In fact, a good practice of designing a neural network is often to extended the network until you see that it starts overfitting, extended it a bit more and add a regularization like dropout to avoid that overfitting.

7.5 Perspective and further reading

Although the first conceptual ideas of neural networks date back to the 1940s (McCulloch and Pitts 1943), they had their first main success stories in the late 1980s and early 1990s with the use of the so-called back-propagation algorithm. At that stage, neural networks could, for example, be used to classify handwritten digits from low-resolution images (LeCun, Boser, et al. 1990). However, in the late 1990s neural networks were largely forsaken because it was widely thought that they could not be used to solve any challenging problems in computer vision and speech recognition. In these areas, neural networks could not compete with hand-crafted solutions based on domain specific prior knowledge.

This picture has changed dramatically since the late 2000s, with multiple layers under the name deep learning. Progress in software, hardware and algorithm parallelization made it possible to address more complicated problems, which were unthinkable only a couple of decades ago. For example, in image

recognition, these deep models are now the dominant methods of use and they reach almost human performance on some specific tasks (LeCun, Bengio, and Hinton 2015). Recent advances based on deep neural networks have generated algorithms that can learn how to play computer games based on pixel information only (Mnih et al. 2015), and automatically understand the situation in images for automatic caption generation (Xu et al. 2015).

A fairly recent and accessible introduction and overview of deep learning is provided by LeCun, Bengio, and Hinton (2015), and a recent textbook by Goodfellow, Bengio, and Courville (2016).

A Probability theory

A.1 Random variables

A random variable z is a variable that can take any value on a certain set, and its value depends on the outcome of a random event. For example, if z describes the outcome of rolling a die, the possible outcomes are $\{1, 2, 3, 4, 5, 6\}$ and the probability of each possible outcome of a die roll is typically modeled to be $1/6$. To denote this, we use the *probability mass function* (pmf) p and write in this case $p(z) = 1/6$ for $z = 1, \ldots, 6$.

In these lecture notes we will primarily consider random variables where z is continuous, for example taking values in $\mathbb{R}$ (z is a scalar) or in $\mathbb{R}^d$ (z is a d-dimensional vector). Since there are infinitely many possible outcomes, we cannot speak of the probability of an outcome—it is almost always zero—but we use the *probability density function* (pdf) p (as for the pmf). The probability density function p describes the probability of z to be within a certain set C

$$\text{Probability of } z \text{ to be in the set } \mathsf{C} = \int_{z \in \mathsf{C}} p(z) dz. \tag{A.1}$$

A random variable z with a uniform distribution on the interval $[0, 3]$ has the pdf $p(z) = 1/3$ for $z \in [0, 3]$, and otherwise $p(z = 0)$. Note that pmfs are upper bounded by 1, whereas a pdf can possibly take values larger than 1. However, it holds for pdfs that they always integrates[1] to 1: $\int p(z) = 1$.

A common probability distribution is the *Gaussian* (or *Normal*) distribution, whose density is defined as

$$p(z) = \mathcal{N}\left(z \,|\, \mu, \sigma^2\right) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right), \tag{A.2}$$

where we have made use of $\exp$ to denote the exponential function; $\exp(x) = e^x$. We also use the notation $z \sim \mathcal{N}\left(\mu, \sigma^2\right)$ to say that z has a Gaussian distribution with parameters μ and σ^2 (i.e., its probability density function is given by (A.2)). The symbol $\sim$ reads 'distributed according to'.

The *expected value* or *mean* of the random variable z is given by

$$\mathbb{E}[z] = \int z p(z) dz. \tag{A.3}$$

We can also compute the expected value of some arbitrary function $g(z)$ applied to z as

$$\mathbb{E}[g(z)] = \int g(z) p(z) dz. \tag{A.4}$$

For a scalar random variable with mean $\mu = \mathbb{E}[z]$ the *variance* is defined as

$$\mathrm{Var}[z] = \mathbb{E}[(z-\mu)^2] = \mathbb{E}[z^2] - \mu^2. \tag{A.5}$$

The variance measures the 'spread' of the distribution, i.e. how far a set of random number drawn from the distribution are spread out from their mean. The variance is always non-negative. For the Gaussian distribution (A.2) the mean and variance are given by the parameters μ and σ^2 respectively.

Now, consider two random variables z_1 and z_2 (both of which could be vectors). An important property of pairs of random variables is that of *independence*. The variables z_1 and z_2 are said to be independent

[1] For notational convenience, when the integration is over the whole domain of z we simply write $\int$.

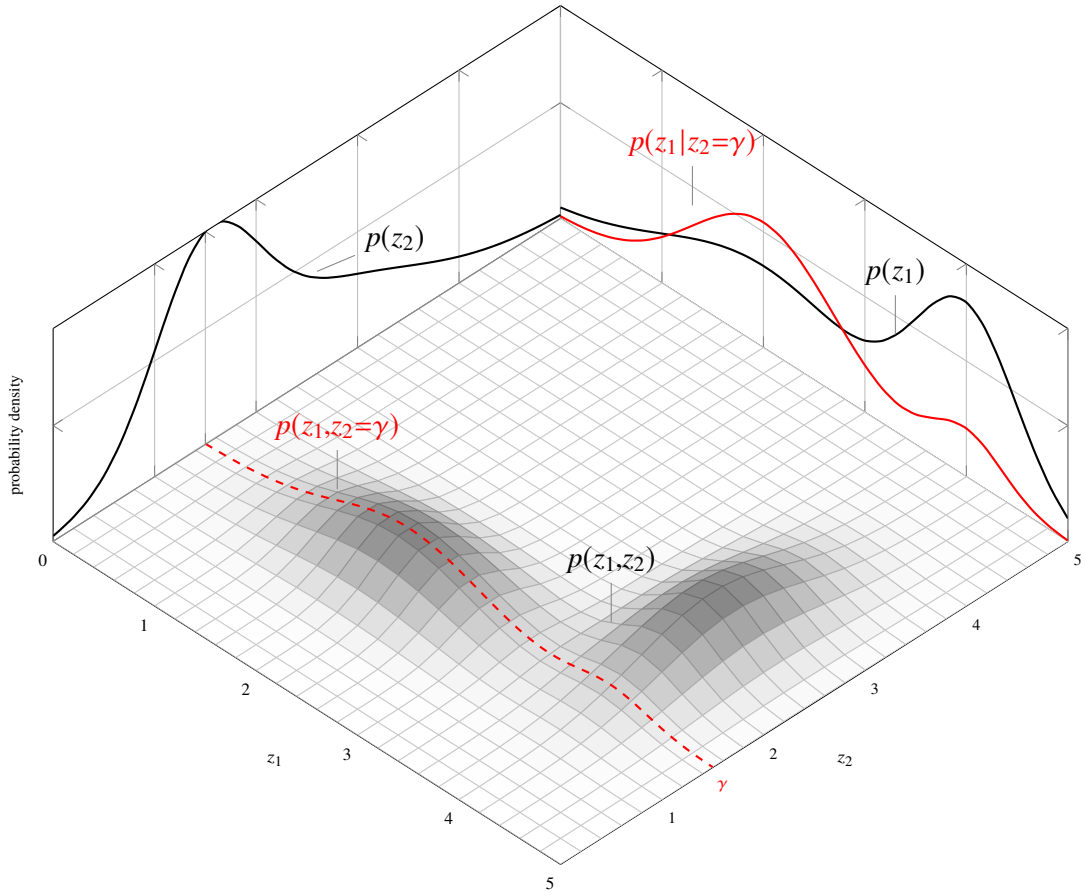


Figure A.1: Illustration of a two-dimensional *joint* probability distribution $p(z_1, z_2)$ (the surface) and its two *marginal* distributions $p(z_1)$ and $p(z_2)$ (the black lines). We also illustrate the *conditional* distribution $p(z_1|z_2 = \gamma)$ (the red line), which is the distribution of the random variable z_1 conditioned on the observation $z_2 = \gamma$ ($\gamma = 1.5$ in the plot).

if the joint pdf factorizes according to $p(z_1, z_2) = p(z_1)p(z_2)$. Furthermore, for independent random variables the expected value of any separable function factorizes as $\mathbb{E}[g_1(z_1)g_2(z_2)] = \mathbb{E}[g_1(z_1)]\mathbb{E}[g_2(z_2)]$.

From the joint probability density function we can deduce both its two marginal densities $p(z_1)$ and $p(z_2)$ using *marginalization*, as well as the so called conditional probability density function $p(z_2 \mid z_1)$ using *conditioning*. These two concepts will be explained below.

A.1.1 Marginalization

Consider a multivariate random variable z which is composed of two components z_1 and z_2, which could be either scalars or vectors, as $z = [z_1^\mathsf{T}, z_2^\mathsf{T}]^\mathsf{T}$. If we know the (joint) probability density function $p(z) = p(z_1, z_2)$, but are interested only in the *marginal* distribution for z_1, we can obtain the density $p(z_1)$ by *marginalization*

$$p(z_1) = \int p(z_1, z_2) dz_2. \tag{A.6}$$

The other marginal $p(z_2)$ is obtained analogously by integrating over z_1 instead. In Figure A.1 a joint two-dimensional density $p(z_1, z_2)$ is illustrated along with their marginal densities $p(z_1)$ and $p(z_2)$.

A.1.2 Conditioning

Consider again the multivariate random variable z which can be partitioned in two parts $z = [z_1^\mathsf{T},\ z_2^\mathsf{T}]^\mathsf{T}$. We can now define the *conditional* distribution of z_1, conditioned on having observed a value $z_2 = z_2$, as

$$p(z_1 \mid z_2) = \frac{p(z_1, z_2)}{p(z_2)}. \tag{A.7}$$

If we instead have observed a value of $z_1 = z_1$ and want to use that to find the conditional distribution of z_2 given $z_1 = z_1$, it can be done analogously. In Figure A.1 a joint two-dimensional probability density function $p(z_1, z_2)$ is illustrated along with a conditional probability density function $p(z_1 \mid z_2)$.

From (A.7) it follows that the joint probability density function $p(z_1, z_2)$ can be factorized into the product of a marginal times a conditional,

$$p(z_1, z_2) = p(z_2 \mid z_1)p(z_1) = p(z_1 \mid z_2)p(z_2). \tag{A.8}$$

If we use this factorization for the denominator of the right-hand-side in (A.7) we end up with the relationship

$$p(z_1 \mid z_2) = \frac{p(z_2 \mid z_1)p(z_1)}{p(z_2)}. \tag{A.9}$$

This equation is often referred to as *Bayes' rule*.

A.2 Approximating an integral with a sum

An integral over a given smooth function $h(z)$ and a probability density $p(z)$ can be approximated with a sum over M samples in the following fashion

$$\int h(z)p(z)dz \approx \frac{1}{M}\sum_{j=1}^{M} h(z_i) \tag{A.10}$$

if each z_i is drawn independently from $z_i \sim p(z)$. This is called Monte Carlo integration. The approximate equality becomes exact with probability one in the limit as the number of samples $M \to \infty$.

B Unconstrained numerical optimization

Given a function $L(\theta)$, the *optimization* problem is about finding the value of the variable x for which the function $L(\theta)$ is either minimized or maximized. To be precise it will here be formulated as finding the value $\widehat{\theta}$ that minimizes[1] the function $L(\theta)$ according to

$$\min_{\theta} L(\theta), \tag{B.1}$$

where the vector θ is allowed to be anywhere in $\mathbb{R}^n$, motivating the name *unconstrained* optimization. The function $L(\theta)$ will be referred to as the *cost function*[2], with the motivation that the minimization problem in (B.1) is striving to minimize some cost. We will make the assumption that the cost function $L(\theta)$ is continuously differentiable on $\mathbb{R}^n$. If there are requirements on θ (e.g. that its components θ have to satisfy a certain equation $g(\theta) = 0$) the problem is instead referred to as a constrained optimization problem.

The unconstrained optimization problem (B.1) is ever-present across the sciences and engineering, since it allows us to find the best—in some sense—solution to a particular problem. One example of this arises when we are searching for the parameters in a linear regression problem by finding the parameters that make the available measurements as likely as possible by maximizing the likelihood function. For a linear model with Gaussian noise, this resulted in a least squares problem, for which there is an explicit expression (the normal equations, (2.17)) describing the solution. However, for most optimization problems that we face there are no explicit solutions available, forcing us to use approximate *numerical* methods in solving these problems. We have seen several concrete examples of this kind, for example the optimization problems arising in deep learning and logistic regression. This appendix provides a brief introduction to the practical area of unconstrained numerical optimization.

The key in assembling a working optimization algorithm is to build a simple and useful *model* of the complicated cost function $L(\theta)$ around the current value for θ. The model is often local in the sense that it is only valid in a neighbourhood of this value. The idea is then to exploit this model to select a new value for θ that corresponds to a smaller value for the cost function $L(\theta)$. The procedure is then repeated, which explains why most numerical optimization algorithms are of *iterative* nature. There are of course many different ways in which this can be done, but they all share a few key parts which we outline below. Note that we only aim to provide the overall strategies underlying practical unconstrained optimization algorithms, for precise details we refer to the many textbooks available on the subject, some of which are referenced towards the end.

B.1 A general iterative solution

What do we mean by a solution to the unconstrained minimization problem in (B.1)? The best possible solution is the *global minimizer*, which is a point $\widehat{\theta}$ such that $L(\widehat{\theta}) \leq L(\theta)$ for all $\theta \in \mathbb{R}^n$. The global minimizer is often hard to find and instead we typically have to settle for a local minimizer instead. A point $\widehat{\theta}$ is said to be a *local minimizer* if there is a neighbourhood $\mathcal{M}$ of $\widehat{\theta}$ such that $L(\widehat{\theta}) \leq L(\theta)$ for all $\theta \in \mathcal{M}$.

In our search for a local minimizer we have to start somewhere, let us denote this starting point by θ_0. Now, if θ_0 is not a local minimizer of $L(\theta)$ then there must be an increment d_0 that we can add to θ_0 such that $L(\theta_0 + d_0) < L(\theta_0)$. By the same argument, if $\theta_1 = \theta_0 + d_0$ is not a local minimizer then there must

[1] Note that it is sufficient to cover minimization problem, since any maximization problem can be considered as a minimization problem simply by changing the sign of the cost function.

[2] Throughout the course we have talked quite a lot about different loss functions. These loss functions are examples of cost functions.

be another increment d_1 that we can add to θ_1 such that $L(\theta_1 + d_1) < L(\theta_1)$. This procedure is repeated until it is no longer possible to find an increment that decrease the value of the objective function. We have then found a local minimizer. Most of the algorithms capable of solving (B.1) are iterative procedures of this kind. Before moving on, let us mention that the increment d is often resolved into two parts according to

$$d = \gamma p. \tag{B.2}$$

Here, the scalar and positive parameter γ is commonly referred to as the *step length* and the vector $p \in \mathbb{R}^n$ is referred to as the *search direction*. The intuition is that the algorithm is searching for the solution by moving in the search direction and how far it moves in this this direction is controlled by the step length.

The above development does of course lead to several questions, where the most pertinent are the following:

1. How can we compute a useful search direction p?
2. How big steps should we make, i.e. what is a good value of the step length γ?
3. How do we determine when we have reached a local minimizer, and stop searching for new directions?

Throughout the rest of this section we will briefly discuss these questions and finally we will assemble the general form of an algorithm that is often used for unconstrained minimization.

A straightforward way of finding a general characterization of all search directions p resulting in a decrease in the value of the cost function, i.e. directions p such that

$$L(\theta + p) < L(\theta) \tag{B.3}$$

is to build a local model of the cost function around the point θ. One model of this kind is provided by Taylor's theorem, which builds a local polynomial approximation of a function around some point of interest. A linear approximation of the cost function $L(\theta)$ around the point θ is given by

$$L(\theta + p) \approx L(\theta) + p^\mathsf{T} \nabla L(\theta). \tag{B.4}$$

By inserting the linear approximation (B.4) of the objective function into (B.3) we can provide a more precise formulation of how to find a search direction p such that $L(\theta + p) < L(\theta)$ by asking for which p it holds that $L(\theta) + p^\mathsf{T} \nabla L(\theta) < L(\theta)$, which can be further simplified into

$$p^\mathsf{T} \nabla L(\theta) < 0. \tag{B.5}$$

Inspired by the inequality above we chose a generic description of the search direction according to

$$p = -V \nabla L(\theta), \qquad V \succ 0, \tag{B.6}$$

where we have introduced some extra flexibility via the positive definite scaling matrix V. The inspiration came from the fact that by inserting (B.6) into (B.5) we obtain

$$p^\mathsf{T} L(\theta) = -\nabla^\mathsf{T} L(\theta) V^\mathsf{T} \nabla L(\theta) = -\|\nabla L(\theta)\|^2_{V^\mathsf{T}} < 0, \tag{B.7}$$

where the last inequality follows from the positivity of the squared weighted two-norm, which is defined as $\|a\|^2_W = a^\mathsf{T} W a$. This shows that $p = -V \nabla L(\theta)$ will indeed result in a search direction that decreases the value of the objective function. We refer to such a search direction as a *descent direction*.

The strategy summarized in Algorithm 9 is referred to as *line search*. Note that we have now introduced subscript t to clearly show the iterative nature. The algorithm searches along the line defined by starting at the current iterate θ_t and then moving along the search direction p_t. The decision of how far to move along this line is made by simply minimizing the cost function along the line

$$\min_\gamma L(\theta_t + \gamma p_t). \tag{B.8}$$

Algorithm 9: General form of unconstrained minimization

1. Set $t = 0$.
2. **while** *stopping criteria is not satisfied* **do**
   a) Compute a search direction $p_t = -V_t \nabla L(\theta_t)$ for some $V_t \succ 0$.
   b) Find a step length $\gamma_t > 0$ such that $L(\theta_t + \gamma_t p_t) < L(\theta_t)$.
   c) Set $\theta_{t+1} = \theta_t + \gamma_t p_t$.
   d) Set $t \leftarrow t + 1$.
3. **end while**

Note that this is a one-dimensional optimization problem, and hence simpler to deal with compared to the original problem. The *step length* γ_t that is selected in (B.8) controls how far to move along the current search direction p_t. It is sufficient to solve this problem approximately in order to find an acceptable step length, since as long as $L(\theta_t + \gamma_t p_t) < L(\theta_t)$ it is not crucial to find the global minimizer for (B.8).

There are several different indicators that can be used in designing a suitable stopping criteria for row 2 in Algorithm 9. The task of the stopping criteria is to control when to stop the iterations. Since we know that the gradient is zero at a stationary point it is useful to investigate when the gradient is close to zero. Another indicator is to keep an eye on the size of the increments between adjacent iterates, i.e. when θ_{t+1} is close to θ_t.

In the so-called *trust region* strategy the order of step 2a and step 2b in Algorithm 9 is simply reversed, i.e. we first decide how far to step and then we chose in which direction to move.

B.2 Commonly used search directions

Three of the most popular search directions corresponds to specific choices when it comes to the positive definite matrix V_t in step 2a of Algorithm 9. The simplest choice is to make use of the identity matrix, resulting in the so-called *steepest descent* direction described in Section B.2.1. The *Newton* direction (Section B.2.2) is obtained by using the inverse of the Hessian matrix and finally we have the *quasi-Newton* direction (Section B.2.3) employing an approximation of the inverse Hessian.

B.2.1 Steepest descent direction

Let us start by noting that according to the definition of the scalar product[3], the descent condition (B.5) imposes the following requirement of the search direction

$$p^{\mathsf{T}} \nabla L(\theta_t) = \|p\|_2 \|\nabla L(\theta_t)\|_2 \cos(\varphi) < 0, \tag{B.9}$$

where φ denotes the angle between the two vectors p and $\nabla L(\theta_t)$. Since we are only interested in finding the direction we can without loss of generality fix the length of p, implying the scalar product $p^{\mathsf{T}} \nabla L(\theta_t)$ is made as small as possible by selecting $\varphi = \pi$, corresponding to

$$p = -\nabla L(\theta_t). \tag{B.10}$$

Recall that the gradient vector at a point is the direction of maximum rate of change of the function at that point. This explains why the search direction suggested in (B.10) is referred to as the *steepest descent direction*.

[3]The scalar (or dot) product of two vectors a and b is defined as $a^{\mathsf{T}} b = \|a\| \|b\| \cos(\varphi)$, where $\|a\|$ denotes the length (magnitude) of the vector a and φ denotes the angle between a and b.

Sometimes, the use of the steepest descent direction can be very slow. The reason for this is that there is more information available about the cost function that the algorithm can make use of, which brings us to the Newton and the quasi-Newton directions described below. They make use of additional information about the local geometry of the cost function by employing a more descriptive local model.

B.2.2 Newton direction

Let us now instead make use of a better model of the objective function, by also keeping the quadratic term of the Taylor expansion. The result is the following quadratic approximation $m(\theta_t, p_t)$ of the cost function around the current iterate θ_t

$$L(\theta_t + p_t) \approx \underbrace{L(\theta_t) + p_t^\mathsf{T} g_t + \frac{1}{2} p_t^\mathsf{T} H_t p_t}_{=m(\theta_t, p_t)} \tag{B.11}$$

where $g_t = \nabla L(\theta)|_{\theta=\theta_t}$ denotes the cost function gradient and $H_t = \nabla^2 L(\theta)|_{\theta=\theta_t}$ denotes the Hessian, both evaluated at the current iterate θ_t. The idea behind the Newton direction is to select the search direction that minimizes the quadratic model in (B.11), which is obtained by setting its derivative

$$\frac{\partial m(\theta_t, p_t)}{\partial p_t} = g_t + H_t p_t \tag{B.12}$$

to zero, resulting in

$$p_t = -H_t^{-1} g_t. \tag{B.13}$$

It is often too difficult or too expensive to compute the Hessian, which has motivated the development of search directions employing an approximation of the Hessian. The generic name for these are quasi-Newton directions.

B.2.3 Quasi-Newton

The quasi-Newton direction makes use of a local quadratic model $m(\theta_t, p_t)$ of the cost function according to (B.11), similarly to what was done in finding the Newton direction. However, rather than assuming that the Hessian is available, the Hessian will now instead be learned from the information that is available in the cost function values and its gradients.

Let us first denote the line segment connecting two adjacent iterates θ_t and θ_{t+1} by

$$r_t(\tau) = \theta_t + \tau(\theta_{t+1} - \theta_t), \qquad \tau \in [0, 1]. \tag{B.14}$$

From the fundamental theorem of calculus we know that

$$\int_0^1 \frac{\partial}{\partial \tau} \nabla L(r_t(\tau)) d\tau = \nabla L(r_t(1)) - \nabla L(r_t(0)) = \nabla L(\theta_{t+1}) - \nabla L(\theta_t) = g_{t+1} - g_t, \tag{B.15}$$

and from the chain rule we have that

$$\frac{\partial}{\partial \tau} \nabla L(r_t(\tau)) = \nabla^2 L(r_t(\tau)) \frac{\partial r_t(\tau)}{\partial \tau} = \nabla^2 L(r_t(\tau))(\theta_{t+1} - \theta_t). \tag{B.16}$$

Hence, in combining (B.15) and (B.16) we obtain

$$y_t = \int_0^1 \frac{\partial}{\partial \tau} \nabla L(r_t(\tau)) d\tau = \int_0^1 \nabla^2 L(r_t(\tau)) s_t d\tau. \tag{B.17}$$

where we have defined $y_t = g_{t+1} - g_t$ and $s_t = \theta_{t+1} - \theta_t$. An interpretation of the above equation is that the difference between two consecutive gradients y_t is given by integrating the Hessian times s_t for

points θ along the line segment $r_t(\tau)$ defined in (B.14). The approximation underlying quasi-Newton methods is now to assume that this integral can be described by a constant matrix B_{t+1}, resulting in the following approximation

$$y_t = B_{t+1} s_t \tag{B.18}$$

of the integral (B.17), which is sometimes referred to as the *secant condition* or the quasi-Newton equation. The secant condition above is still not enough to determine the matrix B_{t+1}, since even though we know that B_{t+1} is symmetric there are still too many degrees of freedom available. This is solved using regularization and B_{t+1} is selected as the solution to

$$\begin{aligned} B_{t+1} = \min_{B} \quad & \|B - B_t\|_W^2, \\ \text{s.t.} \quad & B = B^\mathsf{T}, \quad B s_t = y_t, \end{aligned} \tag{B.19}$$

for some weighting matrix W. Depending on which weighting matrix that is used we obtain different algorithms. The most common quasi-Newton algorithms are referred to as BFGS (named after Broyden, Fletcher, Goldfarb and Shanno), DFP (named after Davidon, Fletcher and Powell) and Broyden's method. The resulting Hessian approximation B_{t+1} is then used in place of the true Hessian.

B.3 Further reading

This appendix is heavily inspired by the solid general introduction to the topic of numerical solutions to optimization problems given by Nocedal and Wright (2006) and by Wills (2017). In solving optimization problems the initial important classification of the problem is whether it is convex or non-convex. Here we have mainly been concerned with the numerical solution of non-convex problems. When it comes to convex problems Boyd and Vandenberghe (2004) provide a good engineering introduction. A thorough and timely introduction to the use of numerical optimization in the machine learning context is provided by Bottou, Curtis, and Nocedal (2017). The focus is naturally on large scale problems and as we have explained in the deep learning chapter this naturally leads to stochastic optimization problems.

Bibliography

Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin (2012). *Learning From Data. A short course*. AMLbook.com.

Barber, David (2012). *Bayesian reasoning and machine learning*. Cambridge University Press.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bottou, L., F. E. Curtis, and J. Nocedal (2017). *Optimization methods for large-scale machine learning*. Tech. rep. arXiv:1606.04838v2.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge, UK: Cambridge University Press.

Breiman, Leo (Oct. 2001). "Random Forests". In: *Machine Learning* 45.1, pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

Deisenroth, M. P., A. Faisal, and C. O. Ong (2019). *Mathematics for machine learning*. Cambridge University Press.

Dheeru, Dua and Efi Karra Taniskidou (2017). *UCI Machine Learning Repository*. URL: http://archive.ics.uci.edu/ml.

Efron, Bradley and Trevor Hastie (2016). *Computer age statistical inference*. Cambridge University Press.

Ezekiel, Mordecai and Karl A. Fox (1959). *Methods of Correlation and Regression Analysis*. John Wiley & Sons, Inc.

Freund, Yoav and Robert E. Schapire (1996). "Experiments with a new boosting algorithm". In: *Proceedings of the 13th International Conference on Machine Learning (ICML)*.

Friedman, Jerome (2001). "Greedy function approximation: A gradient boosting machine". In: *Annals of Statistics* 29.5, pp. 1189–1232.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani (2000). "Additive logistic regression: a statistical view of boosting (with discussion)". In: *The Annals of Statistics* 28.2, pp. 337–407.

Gelman, Andrew et al. (2013). *Bayesian data analysis*. 3rd ed. CRC Press.

Ghahramani, Zoubin (May 2015). "Probabilistic machine learning and artificial intelligence". In: *Nature* 521.7553, pp. 452–459.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The elements of statistical learning. Data mining, inference, and prediction*. 2nd ed. Springer.

Hastie, Trevor, Robert Tibshirani, and Martin J. Wainwright (2015). *Statistical learning with sparsity: the Lasso and generalizations*. CRC Press.

Hoerl, Arthur E. and Robert W. Kennard (1970). "Ridge regression: biased estimation for nonorthogonal problems". In: *Technometrics* 12.1, pp. 55–67.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani (2013). *An introduction to statistical learning. With applications in R*. Springer.

Jordan, M. I. and T. M. Mitchell (2015). "Machine learning: trends, perspectives, and prospects". In: *Science* 349.6245, pp. 255–260.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521, pp. 436–444.

LeCun, Yann, Bernhard Boser, et al. (1990). "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 396–404.

MacKay, D. J. C. (2003). *Information theory, inference and learning algorithms*. Cambridge University Press.

Mason, Llew, Jonathan Baxter, Peter Bartlett, and Marcus Frean (1999). "Boosting Algorithms as Gradient Descent". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS)*.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Murphy, Kevin P. (2012). *Machine learning – a probabilistic perspective*. MIT Press.

Nocedal, J. and S. J. Wright (2006). *Numerical Optimization*. 2nd ed. Springer Series in Operations Research. New York, USA: Springer.

Srivastava, Nitish et al. (2014). "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Tibshirani, Robert (1996). "Regression Shrinkage and Selection via the LASSO". In: *Journal of the Royal Statistical Society (Series B)* 58.1, pp. 267–288.

Wills, A. G. (2017). "Real-time optimisation for embedded systems". Lecture notes.

Xu, Kelvin et al. (2015). "Show, attend and tell: Neural image caption generation with visual attention". In: *Proceedings of the International Conference on Learning representations (ICML)*.