Supervised Machine Learning

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

Draft version: June 17, 2020

Contents

1 Introduction

Chapter to be written.

2 Supervised learning: a first approach

In this chapter we will introduce the supervised machine learning problem, as well as two basic machine learning methods for solving it. In short the problem amounts to use data for learning the relationship between an input and an output variable. The methods we will introduce are called k-NN and decision trees. These two methods are relatively simple but useful on their own, and therefore a good place to start. Understanding the inner workings, advantages and shortcomings of these basic methods also lays a good foundation for the more advanced methods that are to come in later chapters.

2.1 The supervised learning problem

The supervised machine learning problem is to use *training data* to *learn* (or, equivalently, *train*) a mathematical model that relates an *output*[1] variable y to an *input*[2] variable $\mathbf{x}$. That mathematical model can thereafter be used for *predicting* the output y for a new, previously unseen, *test data* for which only $\mathbf{x}$ is known. The beauty of machine learning is that it is quite arbitrary what these variables represent, and it varies from application to application. We saw several examples in Chapter 1 of this problem: Inferring the type of a skin lesion (y) from an image of the skin ($\mathbf{x}$), deducing the type of road user (y) from a lidar measurement ($\mathbf{x}$) or predicting the effect on carbon dioxide emissions (y) from a political intervention ($\mathbf{x}$).

Learning from labeled data

In most interesting supervised machine learning applications, the relationship between input $\mathbf{x}$ and output y is difficult to describe explicitly. It may be too cumbersome or complicated to fully unravel from application domain knowledge, or even unknown. The problem can therefore usually not be solved by writing a traditional computer program that takes $\mathbf{x}$ as input and returns y as output from a set of rules. The supervised machine learning approach is instead to learn the relationship between $\mathbf{x}$ and y from data, which contains examples of observed pairs of input and output values. In other words, supervised machine learning amounts to learning from examples.

The data used for learning is called *training data*, and it has to consist of several input-output samples $(\mathbf{x}_i, y_i)$, in total n of them. We will compactly write the training data as $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$. Each sample in the training data provides a snapshot of how y depends on $\mathbf{x}$, and the goal in supervised machine learning is to squeeze as much information as possible out of $\mathcal{T}$. In this book we will only consider problems where the individual data points are assumed to be (probabilistically) *independent*. This excludes, for example, applications in time series analysis where it is of interest to model the correlation between $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$.

The fact that the training data contains not only input values $\mathbf{x}_i$, but also output values y_i, is the reason for the term "supervised" machine learning. We may say that each input $\mathbf{x}_i$ is accompanied with a label y_i, or simply that we have *labeled data*. For some applications, it is only a matter of jointly recording $\mathbf{x}$ and y. In other applications, the output y has to be created by labeling the training data inputs $\mathbf{x}$ by a domain expert. For instance, to construct a training data set for the skin lesion application, a dermatologist needs to look at all training data inputs (images) $\mathbf{x}_i$ and label them by assigning to the variable y_i the type of lesion that is seen in the image. The entire learning process is thus "supervised" by the domain expert.

We use a vector boldface notation $\mathbf{x}$ to denote the input, since we assume it to be a p-dimensional vector, $\mathbf{x} = [x_1\ x_2\ \dots\ x_p]^\mathsf{T}$. Each element of the input vector $\mathbf{x}$ represent some information that is considered to

[1] The output is commonly also called response, regressand, label, explained variable, predicted variable or dependent variable.

[2] The input is commonly also called feature, attribute, predictor, regressor, covariate, explanatory variable, controlled variable and independent variable.

be relevant, for example the outdoor temperature or the unemployment rate. In an application where the input is a black and white image $\mathbf{x}$ can be all pixel values in the image, so $p = h \times w$, where h and w denote the height and width of the input image.[3] The output y, on the other hand, is often of low dimension and throughout most of this book we will assume that it is a scalar value. The *type* of the output value, numerical or categorical, turns out to be important and is used to distinguish between two subtypes of the supervised machine learning problems: *regression* and *classification*. We will discuss this next.

Numerical and categorical variables

The variables contained in our data (input as well as output) can be of two different types, *numerical* or *categorical*. A numerical variable has a natural ordering. We can say that one instance of a numerical variable is larger or smaller than another instance of the same variable. A numerical variable could for instance be represented by a continuous real number, but it could also be discrete, such as an integer. Categorical variables, on the other hand, are always discrete and importantly they lack a natural ordering. In this book we assume that any categorical variable only can take a finite number of different values. A few examples are given in Table 2.1 below.

Table 2.1: Examples of numerical and categorical variables.

Variable type	Example	Handled as
Number (continuous)	32.23 km/h, 12.50 km/h, 42.85 km/h	Numerical
Number (discrete) with natural ordering	0 children, 1 child, 2 children	Numerical
Number (discrete) without natural ordering	1 = Sweden, 2 = Denmark, 3 = Norway	Categorical
Text string	Hello, Goodbye, Welcome	Categorical

The distinction between numerical and categorical is sometimes somewhat arbitrary. We could for instance argue that having no children is something qualitatively different than having children, and use the categorical variable "children: yes/no", instead of the numerical "0, 1 or 2 children". In other words, it is sometimes left to the machine learning engineer to decide whether a certain variable is considered to be numerical or categorical.

The notion of categorical vs. numerical applies to both the output variable y and to the p elements x_j of the input vector $\mathbf{x} = [x_1 \; x_2 \; \ldots \; x_p]^\mathsf{T}$. All p input variables do not have to be of the same type. It is perfectly fine (and common in practice) to have a mix of categorical and numerical inputs.

Classification and regression

We distinguish between different supervised machine learning problems by the type of the output y.

Regression means that the output is numerical, and *classification* means that the output is categorical.

The reason for this distinction is that the regression and classification problems have somewhat different properties, and require different methods to be solved.

Note that the p input variables $\mathbf{x} = [x_1 \; x_2 \; \ldots \; x_p]^\mathsf{T}$ can be either numerical or categorical for both regression and classification problems. It is only the type of the output that determines whether a problem is a regression or a classification problem. A method which solves a classification problems is often called a *classifier*, and when a classifier predicts the wrong output to a data point it is called a *misclassification*.

For classification the output is categorical and can therefore only take values in a finite set. We use M to denote the number of elements in the set of possible output values. It can, for instance, be $\{\texttt{false}, \texttt{true}\}$ ($M = 2$) or $\{\texttt{Sweden}, \texttt{Norway}, \texttt{Finland}, \texttt{Denmark}\}$ ($M = 4$). We will refer to these elements as *classes* or

[3] For image-based problems it is often more convenient to represent the input as a matrix of size $h \times w$, than as a vector of length $p = hw$, but the dimension is nevertheless the same. We will get back to this in Chapter 6 when discussing the convolutional neural network, a model structure tailored to image-type inputs.

labels. The number of classes M is assumed to be known in the classification problem. To prepare for a concise mathematical notation, we use integers $1, 2, \ldots, M$ to denote the output classes if $M > 2$. The ordering of the integers is arbitrary, and does *not* imply any ordering of the classes. When there are only $M = 2$ classes, we have the important special case of *binary* classification. In binary classification we use the labels -1 and 1 (instead of 1 and 2). Occasionally we will also use the equivalent terms *negative* and *positive* class. The only reason for using a different convention for binary classification is that it gives a more compact mathematical notation for some of the methods, and carries no deeper meaning. Let us now have a look at a classification and a regression problem, which we both will return to later in this book.

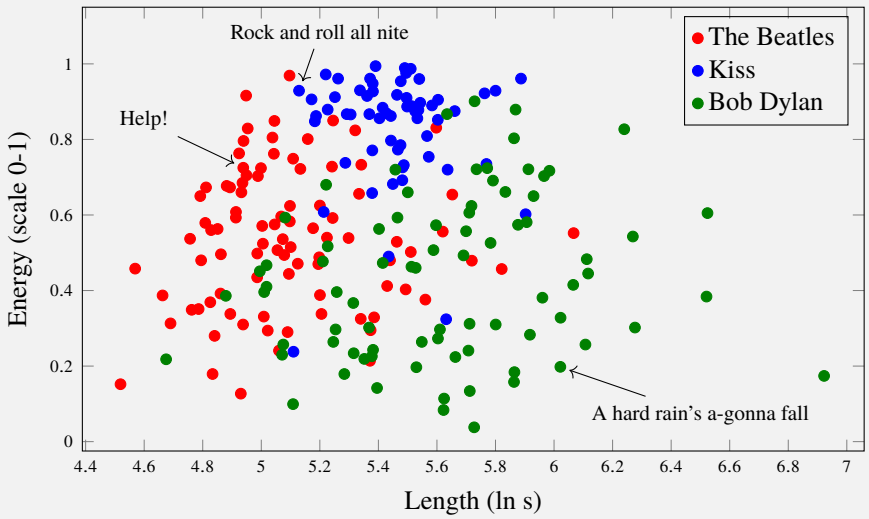
Example 2.1: Classifying songs

Say that we want to build a "song categorizer" app, where the user records a song and the app answers by telling if the song has the artistic style of either the Beatles, Kiss or Bob Dylan. At the heart of this fictitious app there has to be a machinery that takes an audio recording as an input and returns an artist name (`the Beatles`, `Kiss` or `Bob Dylan`).

If we first collect some recordings with songs from the three groups/artists (where we know which artist is behind each song; a labeled dataset), we could use supervised machine learning to *learn* the characteristics of their different styles and therefrom *predict* the artist of the new user-provided song. In the supervised machine learning terminology, the artist name (`the Beatles`, `Kiss` or `Bob Dylan`) is the output y. In this problem y is categorical, and we are hence facing a classification problem.

One of the important design choices for a machine learning engineer is a detailed specification of what the input $\mathbf{x}$ really is. It would in principle be possible to consider the raw audio information as input, but that would give a very high-dimensional $\mathbf{x}$ which (unless an audio-specific machine learning method is used) most likely would require an unrealistically large amount of training data in order to be successful (we will discuss this aspect in detail in Chapter 4). A better option could therefore be to define some summary statistics (features) of audio recordings, and use those as input $\mathbf{x}$ instead. As input we could for example use the length of the audio recording and the amount of energy in it. The length of a recording is easy to measure, and since some songs are rather long it seems natural to take the logarithm of it. The energy is a bit more trickier (the exact definition may even be ambiguous), but we can leave that to the audio experts and re-use a piece of software that they have written for this purpose[a] without bothering too much about its inner workings. As long as this piece of software returns a number for any recording that is sent into it (and always returns the same number for the same recording), we can use it as an input to a machine learning method.

Below we have plotted a dataset with 230 songs from the three artists. Each song is represented by a dot, where the horizontal axis is the logarithm of its length (measured in seconds) and the vertical axis the energy (on a scale 0-1). When we later return to this example, and apply different supervised machine learning methods to it, this data will be the training data.

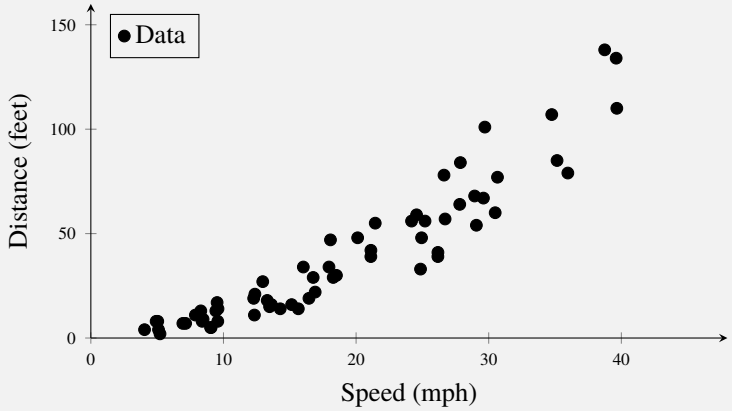


[a] We use `http://api.spotify.com/` here.

Example 2.2: 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 following two 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.



To make a supervised machine learning problem out of it, we interpret `Speed` as the input variable x, and `Distance` as the output variable y. Since y is numerical, this is a regression problem. We then ask ourselves what the stopping distance would be if the initial speed would be, for example, at 33 mph and 45 mph respectively (two speeds at which no data has been recorded). Another way to frame this question is to ask for the *prediction* $\widehat{y}(x_\star)$ for $x_\star = 33$ and $x_\star = 45$.

[a] The data is somewhat dated, so the conclusions are perhaps not applicable to modern cars.

Time to reflect 2.1: *For each application example discussed in Chapter 1, can you determine the input* $\mathbf{x}$*, the output* y*, and whether it is a regression or classification problem?*

Generalizing beyond training data

There are two primary reasons for why it is of interest to design methods for learning mathematical models of input–output relationships from training data.

(i) To *reason about and explore* how input and output variables are connected. An often encountered task in sciences such as medicine and sociology is to determine whether a correlation between a pair of variables exists or not ("does sea food increase the life expectancy?"). Such questions can be addressed by learning a mathematical model and carefully reason about the chance that the learned relationships between input $\mathbf{x}$ and output y are due only to random effects in the data, or if there appear to be some substance to the found relationships.

(ii) To *predict* the output value $y_\star$ for some new, previously unseen input $\mathbf{x}_\star$. By first learning a mathematical model of the input–output relationship from the training data, we can make a prediction by inserting the test input $\mathbf{x}_\star$ into the model and use the output from the model $\widehat{y}(\mathbf{x}_\star)$ as a prediction of the true (but unknown) output $y_\star$ associated with $\mathbf{x}_\star$. The hat $\widehat{\ }$ indicates that the prediction is an estimate (that is, an educated guess) of the true output.

These two objectives are sometimes used to roughly distinguish between classical statistics (i) and machine learning (ii), even though it is not completely true (predictive modeling is a topic of classical statistics and explainable models is a topic within machine learning). The primary focus in this book, however, is on learning models for making predictions, objective (ii) above, which is the foundation of supervised machine learning. Our overall goal is to obtain as accurate predictions $\widehat{y}(\mathbf{x}_\star)$ as possible

(measured in some appropriate way) for a wide range of possible test inputs $\mathbf{x}_\star$. We say that the we are interested in models that *generalize* well beyond the training data.

A model that generalizes well for the music example above would be able to correctly tell the artist of a new song which was not in the training data (assuming that the artist of the new song is one of the three that was present in the training data, of course). The ability to generalize to new data is a key concept of machine learning. It is not difficult to construct models that give very accurate predictions if they are only evaluated on the training data (we will see an example in the next section). If the model is not able to generalize, meaning that the predictions are poor when the model is applied to new test data points, then the model is of little use in practice for making predictions. If that is the case we say that the model is *overfitting* to the training data. We will illustrate the issue of overfitting for a specific machine learning model in the next section and in Chapter 4 we will return to this concept using a more general and mathematical approach.

2.2 A distance-based method: *k*-NN

It is now high time to encounter the first actual machine learning method. We will start with the relatively simple k-NN method, which can be used for both regression and classification. Remember that the setting is that we have access to training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, which consists of n input samples $\mathbf{x}_i$ with corresponding output samples y_i. From this we want to construct a prediction $\widehat{y}(\mathbf{x}_\star)$ for what we believe the output $y_\star$ would be for a new $\mathbf{x}_\star$, which we have not seen previously.

The *k*-nearest neighbors method

Most methods for supervised machine learning builds on the intuition that *if the test data point* $\mathbf{x}_\star$ *is close to training data sample* $\mathbf{x}_i$*, then the prediction* $\widehat{y}(\mathbf{x}_\star)$ *should be close to* y_i. This is a general idea, but one simple way to implement it in practice is the following way: First, compute the distance[4] between the test input and all training inputs, $\|\mathbf{x}_i - \mathbf{x}_\star\|$ for $i = 1, \ldots, n$. Second, we find the data point $\mathbf{x}_j$ with the *shortest distance* to $\mathbf{x}_\star$ and let the prediction be $\widehat{y}(\mathbf{x}_\star) = y_j$.

This simple prediction method is referred to as the 1-nearest neighbor method. It is not very complicated, but for most machine learning applications of interest it is too simplistic. In practice we can rarely say *for certain* what the output value y will be, but we should mathematically describe y as random variable. Consequently, it makes sense to consider the data as *noisy*, that is, being affected by random noise. In this perspective, the shortcoming of 1-nearest neighbor is that the prediction relies only on *one* sample from the training data, which makes it quite "erratic" and sensitive to noisy training data.

To improve the 1-nearest neighbor method we can extend it to make use of the k nearest neighbors instead. 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 aggregate the information from the k outputs y_j for $j \in R_\star$ to make the prediction. For regression problems we take the average of all y_j for $j \in R_\star$, and a majority vote[5] for classification problems. We illustrate the k-nearest neighbors (k-NN) method by Example 2.3 and summarize by Algorithm 1.

Methods that explicitly use the training data when making predictions are referred to as *non-parametric*. This is in contrast with *parametric* methods, where the prediction is given by some function (a model), governed by a fixed number of parameters. For parametric methods the training data is used to *learn* the parameters in an initial training phase, but once the model has been learned, the training data can be discarded since it is not used explicitly when making predictions. We will introduce some parametric models in Chapter 3.

Algorithm 1: k-nearest neighbor, k-NN

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

1 Compute the distances $\|\mathbf{x}_i - \mathbf{x}_\star\|$ for all training data points $i = 1, \ldots, n$
2 Let $R_\star = \{i : \mathbf{x}_i \text{ is one of the } k \text{ data points closest to } \mathbf{x}_\star\}$
3 Compute the prediction $\widehat{y}(\mathbf{x}_\star)$ as

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} \text{Average}\{y_j : j \in R_\star\} & \text{(Regression problems)} \\ \text{MajorityVote}\{y_j : j \in R_\star\} & \text{(Classification problems)} \end{cases}$$

[4] Other distance functions can also be used, and will be discussed in Chapter 8. Categorical input variable can be handled as we will discuss in Section 3.1.

[5] Ties can be handled in different ways, for instance by a coin-flip, or by reporting the actual vote count to the end user who gets to decide what to do with it.

Example 2.3: Predicting colors with k-NN

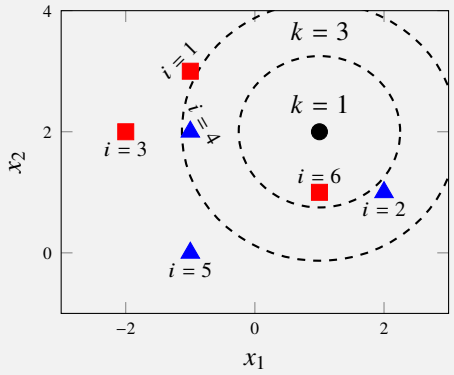
We consider a binary classification problem ($M = 2$). We are given a training data set with $n = 6$ observations of $p = 2$ input variables x_1, x_2 and one categorical output y, the color `Red` or `Blue`,

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 ascending 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 prediction $\widehat{y}(\mathbf{x}_\star) = \texttt{Red}$. For $k = 3$, the 3 nearest neighbors are $i = 6$ (`Red`), $i = 2$ (`Blue`), and $i = 4$ (`Blue`). Taking a majority vote among these three training data points, `Blue` wins with 2 votes against 1, so our prediction becomes $\widehat{y}(\mathbf{x}_\star) = \texttt{Blue}$.

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.

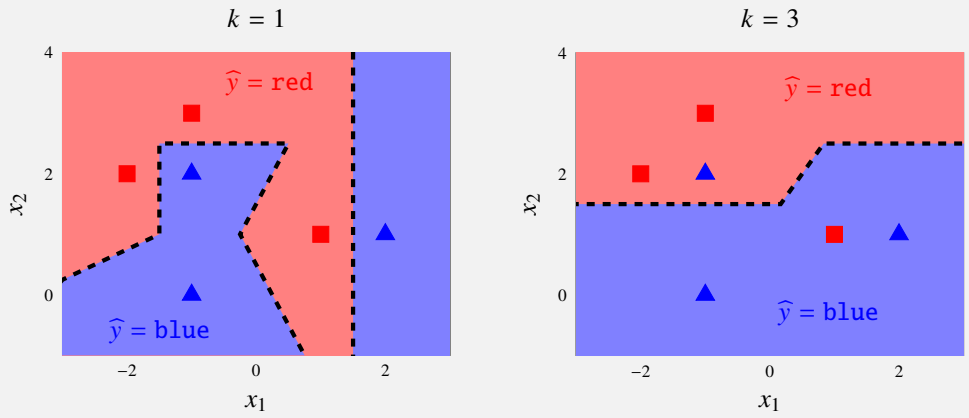
Decision boundaries for a classifier

In Example 2.3 we only computed a prediction for one single test data point $\mathbf{x}_\star$. That prediction might indeed be the ultimate goal of the application, but in order to visualize and better understand a classifier we can also study its *decision boundary*, which illustrates the prediction for all possible test inputs. We introduce the decision boundary using Example 2.4. It is a general concept for classifiers, not only k-NN, but it is only possible to visualize when the dimension of $\mathbf{x}$ is $p = 2$.

Example 2.4: Decision boundaries for the color example

In Example 2.3 we computed the prediction for $\mathbf{x}_\star = [1\ 2]^\mathsf{T}$. 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 $i = 4$ but now $i = 2$ is exchanged for $i = 1$. For $k = 3$ this would give two votes for `Red` and one vote for `Blue` and we would therefore predict $\widehat{y} = \texttt{Red}$. 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 it is undecided if the 3-NN classifier should predict `Red` or `Blue`. (In practice this is most often not a problem, since the test data points rarely end up exactly at the decision boundary. If they do, this can be handled by a coin-flip.) For al classifiers we always end up with such points in the input space where the class prediction abruptly changes from one class to another. These points are said to be on the *decision boundary* of the classifier.

Continuing in a similar way, changing the location of the test input across the entire input space and recording the class prediction, we can compute the full decision boundaries for Example 2.3. We plot the decision boundaries using dashed lines below, both for $k = 1$ and $k = 3$.



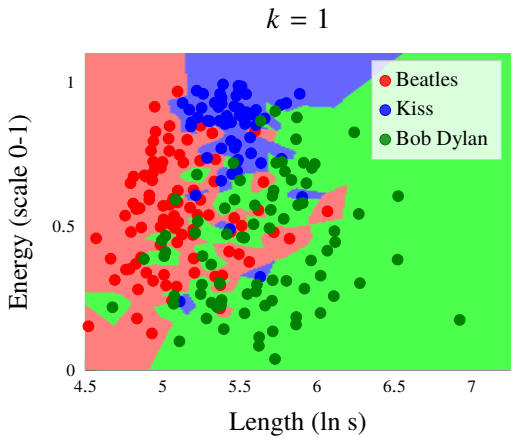
These figures shows the decision boundaries, that is, the points in input space where the class prediction changes. This type of figure gives an concise summary of a classifier, but requires of course that the problem has a 2-dimensional input in order to make such a plot. As we can see, the decision boundaries of k-NN are not linear. In the terminology we will introduce later, k-NN is thereby a nonlinear classifier.

Choosing *k*

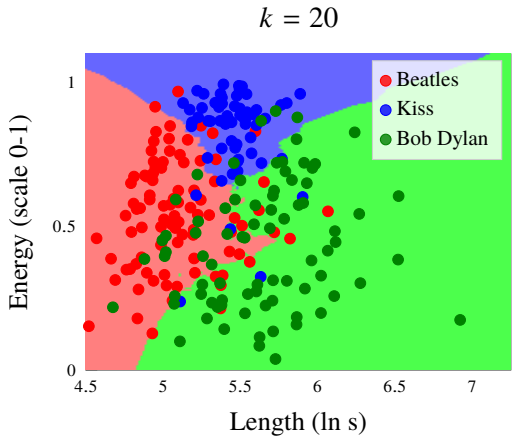
The user has to decide on which k to use in k-NN and this decision has a big impact on the final predictions. To understand the impact of k, we study how the decision boundary changes as k changes in Figure 2.1, where k-NN is applied to the music classification Example 2.1 and car stopping distance Example 2.2, both with $k = 1$ as well as $k = 20$.

With $k = 1$ all training data points will, by construction, be correctly predicted and the model is adapted to the exact $\mathbf{x}$ and y values of the training data. In the classification problem there are for instance small green (Bob Dylan) regions within the red (the Beatles) area that are most likely not relevant when it comes to accurately predict the artist of a new song. In order to make good predictions it would probably be better to instead predict red (the Beatles) for a new song in the entire middle-left region since the vast majority of training data points in that area are red. For the regression problem $k = 1$ gives a quite shaky

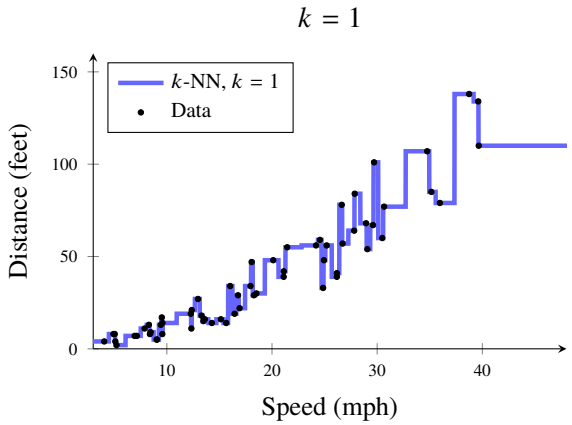
behavior, and also for this problem it is intuitively clear that this does not describe an interesting effect, but rather noise.



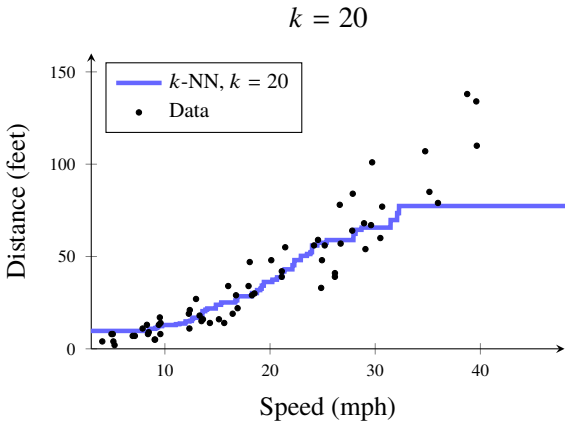
(a) Decision boundaries for the music classification problem using $k = 1$. This is a typical example of overfitting, meaning that the model has adapted too much to the training data so that it does not generalize well to new previously unseen data.



(b) The music classification problem again, now using $k = 20$. A higher value of k gives a more smooth behavior which, hopefully, predicts the artist of new songs more accurately.



(c) The black dots are the car stopping distance data, and the blue line shows the prediction for k-NN with $k = 1$ for any x. As for the classification problem above, k-NN with $k = 1$ overfits to the training data.



(d) The car stopping distance , this time with $k = 20$. Except for the boundary effect at the right, this seems like a much more useful model which captures the interesting effects of the data and ignores the noise.

Figure 2.1: k-NN applied to the music classification Example 2.1 (a and b) and the car stopping distance Example 2.2 (c and d). For both problems k-NN is applied with $k = 1$ and well as $k = 20$.

The drawbacks of using $k = 1$ is not specific to these two examples. In most real world problems there is a certain amount of randomness involved in collecting the training data (or at least some decisions which we know little about, and may consider as being random). In the music example the $n = 230$ songs where somehow selected from all songs ever recorded from these artists, and for the car stopping distance there appears to also be a certain amount of random effects also in y. Thus, by using $k = 1$ and thereby adapting very closely to the training data, the predictions will depend not only on the interesting patterns in the problem, but also on the (more or less) random effects that has shaped the training data. Typically we are not interested in capturing these effects, and we refer to this as *overfitting*.

With the k-NN classifier we can mitigate overfit by increasing the region of the neighborhood used to compute the prediction, that is, increase the parameter k. With, for example, $k = 20$ the predictions are no longer based on only the closest neighbor, but instead a majority vote among the 20 closest neighbors. As a consequence all training data samples are no longer perfectly classified, but some of the songs end up in the wrong region in Figure 2.1b. The predictions are however less adapted to the peculiarities of the

training data and thereby less overfitted, and Figure 2.1b as well as Figure 2.1d are indeed less "noisy" than Figure 2.1a and Figure 2.1c. Selecting k is thus a trade-off between flexibility and rigidity. Indeed, selecting k too big will lead to a meaningless classifier, so there must exist a sweet spot for some moderate k (possibly 20, but could be less or more) where the classifier generalizes the best. Unfortunately there is no general answer for which k this happens, but it is different for different problems. In the music classification it seems reasonable that $k = 20$ will predict new test data points better than $k = 1$, but there might very well be an even better choice of k. For the car stopping problem the behavior is also more reasonable for $k = 20$ than $k = 1$, except for the boundary effect for large x where k-NN is unable to capture the trend in the data as x increases (simply because the 20 nearest neighbors are the same for all test points $x_\star$ around and above 35). A systematic way of choosing a good value of k is to use cross-validation, which we will discuss in Chapter 4.

Time to reflect 2.2: *The prediction $\widehat{y}(\mathbf{x}_\star)$ obtained using the k-NN method is a piece-wise constant function of the input $\mathbf{x}_\star$. For a classification problem this is natural, since the output is categorical (see, e.g., Figure 2.1 where the colored regions correspond to areas of the input space where the prediction is constant according to the color of that region). However, also for regression does k-NN have piece-wise constant predictions. Why?*

Input normalization

A final important practical aspect when using k-NN is the importance of normalization of the input data. Since k-NN is based on the Euclidean distances between points, it is important that it is a relevant measures of how close two data points are. Imagine a training data set with $p = 2$ input variables $\mathbf{x} = [x_1 \; x_2]^\mathsf{T}$ where all values of x_1 are in the range $[100, \; 1100]$ and the values for x_2 are in the much smaller range $[0, \; 1]$. It could for example be that x_1 and x_2 are measured in different units. The Euclidean distance between a test point $\mathbf{x}_\star$ and a training data point $\mathbf{x}_i$ is $\|\mathbf{x}_i - \mathbf{x}_\star\| = \sqrt{(x_{i1} - x_{\star 1})^2 + (x_{i2} - x_{\star 2})^2}$, and this expression will be totally dominated by the first term $(x_{i1} - x_{\star 1})^2$ whereas the second term $(x_{i2} - x_{\star 2})^2$ will almost not matter in practice, due to the different magnitude of x_1 and x_2. That is, the different ranges leads to x_1 being considered much more important than x_2 by k-NN.

To avoid this undesired effect we have to scale the input variables. One option, in the mentioned example, could be to subtract 100 from x_1 and thereafter divide it by 1000 and create $x_{i1}^{\text{new}} = \frac{x_{i1}-100}{1000}$ such that x_1^{new} and x_2 both are in the range $[-1, \; 1]$. More generally, this normalization procedure for the input data can be written as

$$x_{ij}^{\text{new}} = \frac{x_{ij} - \min_\ell(x_{\ell j})}{\max_\ell(x_{\ell j}) - \min_\ell(x_{\ell j})}, \qquad \text{for all } j = 1, \ldots, p, \quad i = 1, \ldots, n. \tag{2.1}$$

Another common way of normalizing (sometimes called standardizing) is by using the mean and standard deviation in the training data:

$$x_{ij}^{\text{new}} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}, \qquad \forall j = 1, \ldots, p, \quad i = 1, \ldots, n, \tag{2.2}$$

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

It is crucial for k-NN to apply some type of input normalization (that was indeed done in Figure 2.1), but it is a good practice to apply also when using other methods, at least for numerical stability. It is however important to compute the scaling factors ($\min_\ell(x_{\ell j})$, $\bar{x}_j$, etc) using training data only and apply that scaling also to future test data points. Failing this, for example by performing normalization before setting test data aside (which we will discuss more in Chapter 4), might lead to wrong conclusions on how well the method will perform in predicting future not yet seen data points.

2.3 A rule-based method: Decision trees

The k-NN method results in a prediction $\widehat{y}(\mathbf{x}_\star)$ that is a piece-wise constant function of the input $\mathbf{x}_\star$. That is, the method partitions the input space into disjoint regions and each region is associated with a certain (constant) prediction. For k-NN, these regions are given implicitly by the k-neighborhood of each possible test input. An alternative approach, that we will study in this section, is to come up with a set of *rules* that define the regions explicitly. For instance, considering the data in Figure 2.1, a simple set of high-level rules for constructing a classifier would be: inputs on the right side are classified as green, in the lower left corner as blue, and in the upper left corner as red. We will now see how such rules can be learned systematically from the training data.

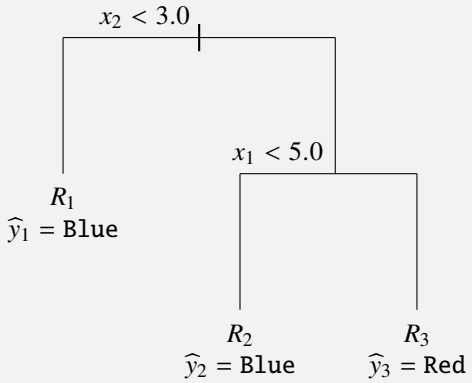
Predicting using a decision tree

The rule-based models that we consider here are referred to as *decision trees*. The reason is that the rules used to define the model can be organized in a graph structure referred to as a binary tree. The decision tree effectively divides the input space into multiple disjoint regions and in each region a constant value is used for the prediction $\widehat{y}(\mathbf{x}_\star)$. We illustrate this with an example.

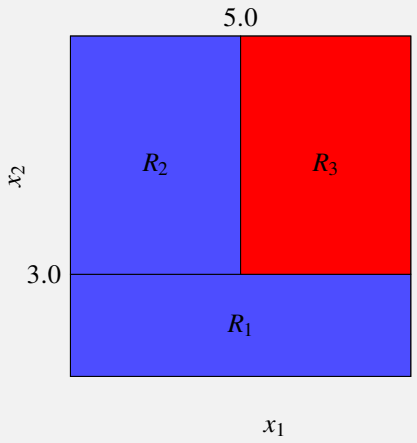
Example 2.5: Predicting colors with a decision tree

We consider a classification problem with two numerical input variables $\mathbf{x} = [x_1 x_2]^\mathsf{T}$ and one categorical output y, the color `Red` or `Blue`. For now we do not consider any training data or how to actually learn the tree, but only how an already existing decision tree can be used to predict $\widehat{y}(\mathbf{x}_\star)$.

The rules defining the model are organized in the graph below (left figure) which is referred to as a binary tree. To use this tree to predict a label for the test input $\mathbf{x}_\star = [x_{\star 1}\ x_{\star 2}]^\mathsf{T}$ we start at the top, referred to as the *root node* of the tree (in the metaphor the tree is growing upside down, with the root at the top and the leaves at the bottom). If the condition stated at the root is true, i.e. if $x_{\star 2} < 3.0$, then we proceed down the left *branch*, otherwise along the right *branch*. If we reach a new *internal node* of the tree, we check the rule associated with that node and pick the left or the right branch accordingly. We continue and work our way down until we reach the end of a branch. Each such final branch corresponds to a constant prediction $\widehat{y}_m$, in this case one of the two classes `Red` or `Blue`.



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 has two internal nodes (including the root) and three leaf nodes.



A region partition, where each region corresponds to a leaf node in the tree. Each border between regions correspond to a split in the tree. Each region is colored with the prediction corresponding to that region, and the boundary between red and blue is therefore the decision boundary.

The decision tree partitions the input space into axis-aligned "boxes", as shown in the right panel above. By increasing the depth of the tree (the number of steps from the root to the leaves), the partitioning can be made finer and finer and thereby describing more complicated functions of the input variable.

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

```
if x_2 < 3.0 then
    return Blue
else
    if x_1 < 5.0 then
        return Blue
    else
        return Red
    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 $x_{\star 1} = 2.5 < 5.0$. The prediction for this test point would be $\widehat{y}(\mathbf{x}_\star) = \texttt{Blue}$.

To set the terminology, the endpoint of each branch R_1, R_2 and R_3 in Example 2.5 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*. The tree is referred to as *binary* since each internal node splits into exactly two branches.

With more than two input variables it is difficult to illustrate the partitioning of the inputs space into regions (right figure in the example), but the tree representation can still be use in precisely the same way. Each internal node corresponds to a rule where one of the p input variables x_j, $j = 1, \ldots, p$, is compared with a threshold s. If $x_j < s$ we continue along the left branch and if $x_j \geq s$ we continue along the right branch.

The constant predictions that we associate with the leaf nodes can be either categorical, as in the example above, or numerical and decision trees can thus be used to address both classification and regression problems. In fact, the models that we describe here also go by the name Classification And Regression Trees (CART). Example 2.5 Illustrates how a decision tree is used for making a prediction. We will now turn to the question of how the tree can be learned from training data.

Learning a regression tree

We will now discuss how to learn (or, equivalently, train) a decision tree for a regression problem. Training a decision tree for classification is conceptually similar and explained in the next section.

As discussed above, the prediction $\widehat{y}(\mathbf{x}_\star)$ corresponding to a classification or regression tree is a piece-wise constant function of the input $\mathbf{x}_\star$. We can write this mathematically as,

$$\widehat{y}(\mathbf{x}_\star) = \sum_{\ell=1}^{L} \widehat{y}_\ell \mathbb{I}\{\mathbf{x}_\star \in R_\ell\}, \tag{2.3}$$

where L is the total number of regions (leaf nodes) in the tree, R_ℓ is the ℓth region, and $\widehat{y}_\ell$ is the constant prediction for the ℓth region. Note that in the regression setting $\widehat{y}_\ell$ is a numerical variable, and we will consider it to be a real number for simplicity. In the equation above we have used the indicator function, $\mathbb{I}\{\mathbf{x} \in R_\ell\} = 1$ if $\mathbf{x} \in R_\ell$ and $\mathbb{I}\{\mathbf{x} \in R_\ell\} = 0$ if $\mathbf{x} \notin R_\ell$ otherwise.

Learning the tree from data corresponds to finding suitable values for the parameters defining the function (2.3), namely the regions R_ℓ and the constant predictions $\widehat{y}_\ell$, $\ell = 1, \ldots, L$, as well as the total size of the tree L. If we start by assuming that the shape of the tree, the partition $(L, \{R_\ell\}_{\ell=1}^L)$ is known, then we can compute the constants $\{\widehat{y}_\ell\}_{\ell=1}^L$ in a natural way, simply as the average of the training data points falling in each region:

$$\boxed{\widehat{y}_\ell = \text{Average}\{y_i : \mathbf{x}_i \in R_\ell\}}$$

It remains however to find the shape of the tree, the regions R_ℓ, which requires a bit more work. The basic idea is of course to select the regions so that the tree fits the training data. This means that the output predictions from the tree should match the output values in the training data. Unfortunately, even when

restricting ourselves to seemingly simple regions such as the "boxes" obtained from a decision tree, finding the tree (a collection of splitting rules) that optimally partitions the input space to fit the training data as well as possible turns out to be computationally infeasible. The problem is that there is a combinatorial explosion in the number of ways in which we can partition the input space. Searching through all possible binary trees is not possible in practice unless the tree size is so small that it is practically useless.

To handle this situation we use a *greedy* algorithm known as *recursive binary splitting* for learning the tree. The word "recursive" means that we will determine the splitting rules one after the other, starting with the first split at the root and then build the tree from top to bottom. The word greedy means that tree is constructed one split at a time, without having the complete tree "in mind". That is, when determining the splitting rule at the root node, the objective is to obtain a model that explains the training data as well as possible after *a single split*, without taking into consideration that additional splits may be added before arriving at the final model. When we have decided on the first split of the input space (corresponding to the root node of the tree), this split is kept fixed and we continue in a similar way for the two resulting half-spaces (corresponding to the two branches of the tree), etc.

To see in detail how one step of this algorithm works, consider the setting when we are about to do our very first split at the root of the tree. Hence, we want to select one of the p input variables $x_1, \ldots, x_p$ and a corresponding cutpoint s which divides the input space into two half-spaces,

$$R_1(j,s) = \{\mathbf{x} \,|\, x_j < s\} \qquad \text{and} \qquad R_2(j,s) = \{\mathbf{x} \,|\, x_j \geq s\}. \tag{2.4}$$

Note that the regions depend on the index j of the splitting variable as well as the value of the cutpoint s, which is why we write them as functions of j and s. This is the case also for the predictions associated with the two regions,

$$\widehat{y}_1(j,s) = \text{Average}\{y_i : \mathbf{x}_i \in R_1(j,s)\} \qquad \text{and} \qquad \widehat{y}_2(j,s) = \text{Average}\{y_i : \mathbf{x}_i \in R_2(j,s)\}$$

since the sums in these expression range over different data points depending on the regions.

For each training data point $(\mathbf{x}_i, y_i)$ we can compute a *prediction error* by first determining which region the data point falls in, and then computing the difference between y_i and the constant prediction associated with that region. Doing this for all training data points the sum of squared errors can be written as

$$\sum_{i:\mathbf{x}_i \in R_1(j,s)} (y_i - \widehat{y}_1(j,s))^2 + \sum_{i:\mathbf{x}_i \in R_2(j,s)} (y_i - \widehat{y}_2(j,s))^2 . \tag{2.5}$$

The square is added to ensure that the expression above is non-negative. The squared error is a common *loss function* used for measuring the closeness of a model's prediction and the training data, but other loss functions can also be used. We will discuss the choice of loss function in more detail in later chapters.

To find the optimal split we select the values for j and s that minimize the squared error (2.5). This minimization problem can be solved easily by looping through all possible values for $j = 1, \ldots, p$. For each j we can scan through the finite number of possible splits, and pick the pair (j, s) for which the expression above is minimized. As pointed out above, when we have found the optimal split at the root node, this splitting rule is fixed. We then continue in the same way for the left and right branch independently. Each branch (corresponding to a half-space) is split again by minimizing the squared prediction error over all training data points correspond to that branch.

In principle, we can continue in this way until there is only a single training data point in each of the regions, i.e. $L = n$. Such a *full grown tree* will result in predictions that exactly match the training data points, and the resulting model is quite similar to k-NN with $k = 1$. As pointed out above, this will typically result in a too erratic model that has overfitted to (possibly noisy) training data. To mitigate this issue, it is common to stop the growth of the tree at an earlier stage, for instance by adding a constraint on the minimum number of training data points associated with each leaf node. Forcing the model to have more training data points in each leaf will result in an averaging effect, similarly to increasing the value of k in the k-NN method. Note that using such a stopping criteria means that the value of L is not set manually, but determined adaptively based on the result of the learning procedure. The full algorithm is

summarized in Algorithm 2 and Algorithm 3. Note that Algorithm 2 includes a recursive call, whre we in each recursion grow one branch of the tree one step further. Also the Algorithm 2 can be formulated as a recursion, but here we have chosen an easier presentation.

Algorithm 2: Learning a decision tree using recursive binary splitting

Data: Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
Result: Decision tree with regions $R_1, \ldots, R_L \in R$

Initialize the tree with R being the whole input space
return `Split(`R`,`$\mathcal{T}$`)`

Function `Split(`R`,`$\mathcal{T}$`):`
    if *splitting criteria fulfilled* **then**
        return R
    else
        Go through all possible splits $x_j < s$ for all input variables $j = 1, \ldots, p$.
        Pick the pair (j, s) that minimizes (2.5)/(2.6) for regression/classification problems.
        Split region R into R_1 and R_2 according to (2.4).
        Split data $\mathcal{T}$ into $\mathcal{T}_1$ and $\mathcal{T}_2$ accordingly.
        return `Split(`R_1`,`$\mathcal{T}_1$`)`, `Split(`R_2`,`$\mathcal{T}_2$`)`
    end
end

Algorithm 3: Making predictions from a decision tree

Data: Decision tree with regions $R_1, \ldots, R_L$, training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ and test data point $\mathbf{x}_\star$
Result: Predicted test output $\widehat{y}(\mathbf{x}_\star)$

Find the region R_ℓ which $\mathbf{x}_\star$ belongs to.
Compute the prediction $\widehat{y}(\mathbf{x}_\star)$ as

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} \text{Average}\{y_i : \mathbf{x}_i \in R_\ell\} & \text{(Regression problems)} \\ \text{MajorityVote}\{y_i : \mathbf{x}_i \in R_\ell\} & \text{(Classification problems)} \end{cases}$$

Classification trees

Trees can also be used for classification. We use the same procedure of recursive binary splitting but with two main differences.

Firstly, we then use a majority vote instead of an average to compute the prediction associated with each region,

$$\widehat{y}_\ell = \text{MajorityVote}\{y_i : \mathbf{x}_i \in R_\ell\}.$$

Secondly, when learning the tree we need a different splitting criteria than the squared prediction error to take into account the fact that the output is categorical. To define these criteria we first introduce

$$\widehat{\pi}_{\ell m}(j, s) = \frac{1}{n_\ell} \sum_{i:\mathbf{x}_i \in R_\ell(j,s)} \mathbb{I}\{y_i = m\}$$

to be the proportion of training observations in the ℓth region that belong to the mth class.

Similar to (2.5), we now want to minimize

$$\min_{j,s} n_1 Q_1 + n_2 Q_2 \tag{2.6}$$

where n_ℓ is the number of data points in region R_ℓ and where Q_ℓ is the splitting criteria.

For Q_ℓ, we have a few options. One simple alternative is the *misclassification rate*

$$Q_\ell = 1 - \max_m \widehat{\pi}_{\ell m}, \tag{2.7}$$

which is simply the proportion of data points in region R_ℓ which do not belong to the most common class. Another common splitting criteria is the *Gini index*

$$Q_\ell = \sum_{m=1}^{M} \widehat{\pi}_{\ell m}(1 - \widehat{\pi}_{\ell m}) \tag{2.8}$$

and the *entropy* criteria.

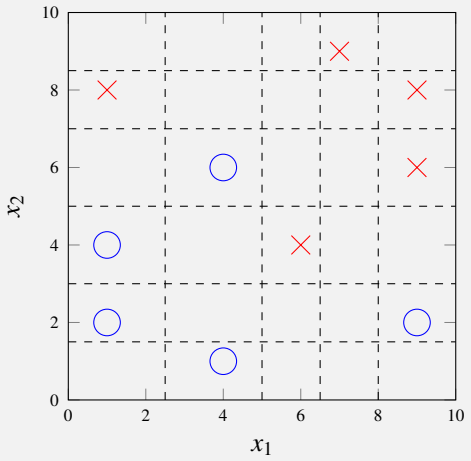
$$Q_\ell = -\sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln \widehat{\pi}_{\ell m}. \tag{2.9}$$

In Example 2.5 we illustrate how to construct a classification tree using recursive binary splitting and with the entropy as the slitting criteria.

Example 2.6: Learning a classification tree (continuation of Example 2.5)

We consider the same setup as in Example 2.5 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 (2.9) 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 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

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

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

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

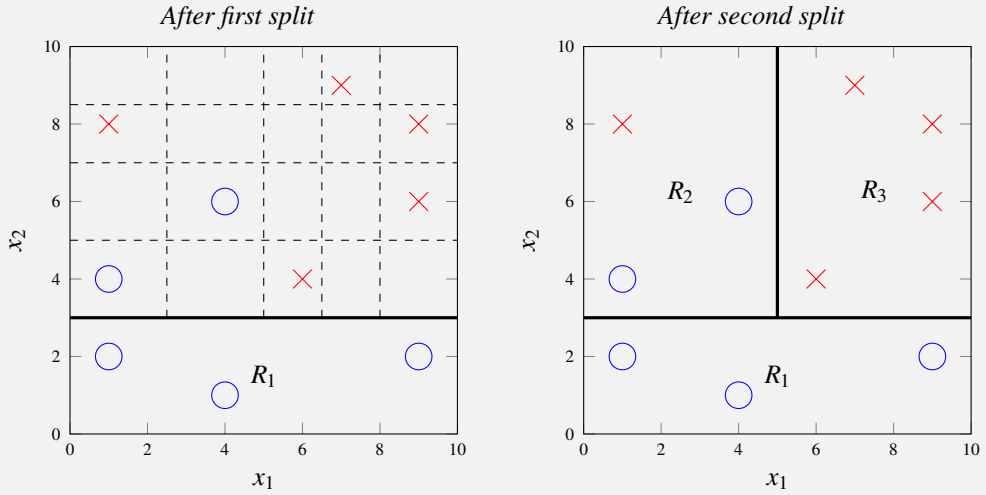
and inserted in 2.6 the total weighted entropy for this split becomes

$$n_1 Q_1 + n_2 Q_2 = 3 \cdot 0.64 + 7 \cdot 0.68 = 6.69.$$

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

Split (R_1)	n_1	$\widehat{\pi}_{1\mathrm{B}}$	$\widehat{\pi}_{1\mathrm{R}}$	Q_1	n_2	$\widehat{\pi}_{2\mathrm{B}}$	$\widehat{\pi}_{2\mathrm{R}}$	Q_2	$n_1Q_1 + n_2Q_2$
$x_1 < 2.5$	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	$\widehat{\pi}_{2\mathrm{B}}$	$\widehat{\pi}_{2\mathrm{R}}$	Q_2	n_3	$\widehat{\pi}_{3\mathrm{B}}$	$\widehat{\pi}_{3\mathrm{R}}$	Q_3	$n_2Q_2 + n_3Q_3$
$x_1 < 2.5$	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 2.5. 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 predict blue if $\mathbf{x}_\star \in R_1$ or $\mathbf{x}_\star \in R_2$ since the blue training data points are in majority in each of these two regions. Similarly, we predict red if $\mathbf{x}_\star \in R_3$. This tree is also visualized in Example 2.5.

When choosing between the different splitting criteria mentioned above, the misclassification rate sounds like a reasonable choice since that is typically the criteria we want the final model to do well on[6]. However, one drawback is that it does not favor pure nodes. With pure nodes we mean nodes where most

[6]This is not always true, for example for imbalanced and asymmetric classification problems, see further in Section 4.5

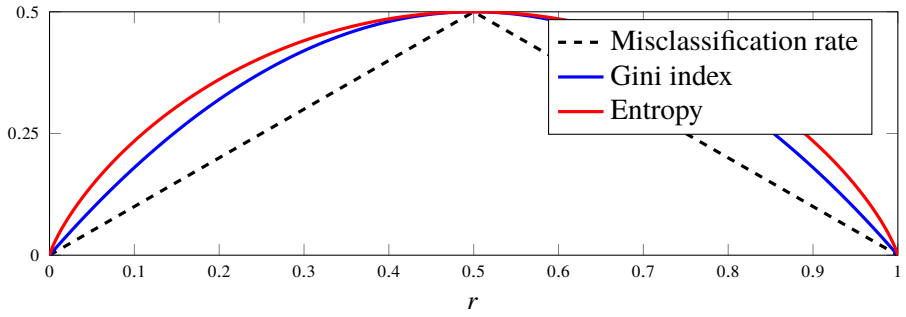


Figure 2.2: Three splitting criteria for classification trees as a function of the proportion of the first class $r = \pi_{\ell 1}$ in a certain region R_ℓ. The entropy criteria has been scaled such that it passes through (0.5,0.5).

of the 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. Both the entropy criteria and Gini index favors node purity more than misclassification rate does.

This advantage can also be illustrated in Example 2.6. Consider the first split in this example. If we would use the misclassification rate as the splitting criteria, both the split $x_2 < 5.0$ as well as the split $x_2 < 3.0$ 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 $x_2 < 5.0$.

To generalize this discussion, consider a problem with two classes where we denote the proportion of the first class as $\pi_{\ell 1} = r$ and hence the proportion of the second class as $\pi_{\ell 2} = 1 - r$, the three criteria can then in terms of r be expressed as

$$\begin{aligned} \text{Misclassification rate:} \quad & Q_\ell = 1 - \max(r, 1-r), \\ \text{Gini index:} \quad & Q_\ell = 2r(1-r), \\ \text{Entropy:} \quad & Q_\ell = -r \ln r - (1-r)\ln(1-r). \end{aligned}$$

These functions are shown in Figure 2.2. All three citeria are similar in the sense that they provide zero loss if all data points belongs to either of the two classes, and maximum loss the data points are equally divided between the two classes. However, the Gini index and entropy have a higher loss for all other proportions. In other words, the gain of having a pure node (r close to 0 or 1) is higher for the Gini index and the entropy than for the miscalsification rate. As a consequence, when doing a split using Gini index or the entropy, they both tend to favour (more than misclassification does) make one of the two nodes pure (or close to being pure) since that provides a smaller total loss.

How deep should a decision tree be?

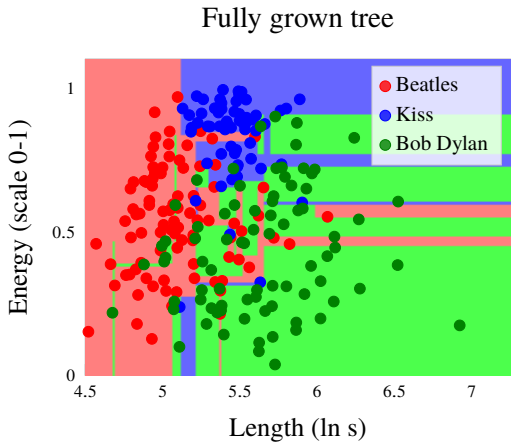
The depth of a decision tree (the maximum number of splits in a certain branch) has a big impact on the final predictions. This impact is very similar the discussion of k in k-NN, and we again use the music classification Example 2.1 and car stopping distance Example 2.2 to study how the decision boundaries change depending on the depth of the trees. In Figure 2.3 the decision boundaries are illustrated for different trees. In Figure 2.3a and Figure 2.3c we have used fully grown trees In Figure 2.3b and Figure 2.3d the depth has been restricted to 4 and 3 splits in each branch, respectively.

Similar to choosing $k = 1$ in k-NN, for a fully grown tree all training data points will by construction be correctly predicted since each region only contains data points with the same output. As a result, for the music classification problem we get thin and small regions adapted to single training data points and for the car stopping distance problem we get an overly adaptive line passing exactly through the observations. Even though these trees give excellent performance on the training data, they are not likely to be the best

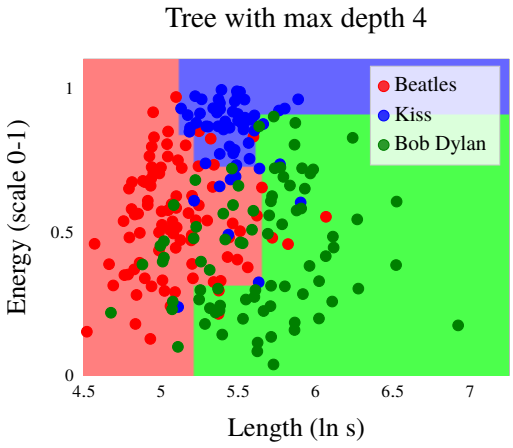
models for new yet unseen data. As we discussed previously in the context of k-NN, we refer to this as overfitting.

In decision trees we can mitigate overfitting by using trees that are not as deep. Consequently, we get fewer and larger regions and the decision boundaries are less adapted to the peculiarities of the training data. This is illustrated in Figure 2.3b and Figure 2.3d for the two problems. As for k in k-NN, the optimal size of the tree depends on many properties of the problem and it is a trade-off between flexibility and rigidity. Similar trade-offs have to be made in almost all models presented in this book and systematic strategies will be discussed in Chapter 4.

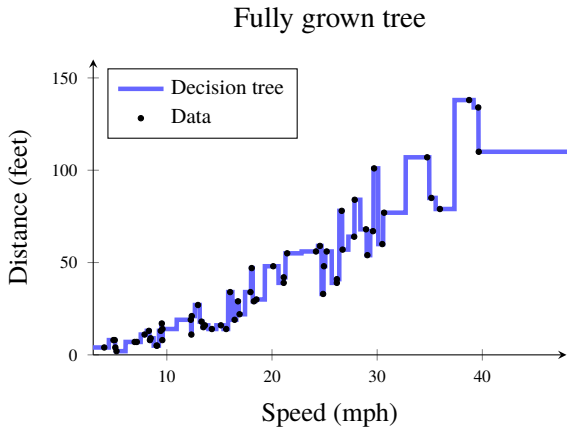
How can the user control the depth of the tree? Here we have different strategies. The most obvious strategy is to adjust the stopping criteria, i.e. the criteria that should be fulfilled for not proceeding with further splits in a certain node. As mentioned earlier, this criteria could be that we do not attempt further splits if there are less than a certain number of training data points the corresponding region, or as in the example above, stop splitting when we reach a certain depth. Another strategy of controlling the depth is to use *pruning*. In pruning we grow a deep and then in a second post-processing step prune it back to a smaller subtree. However, we will not discuss pruning further here.



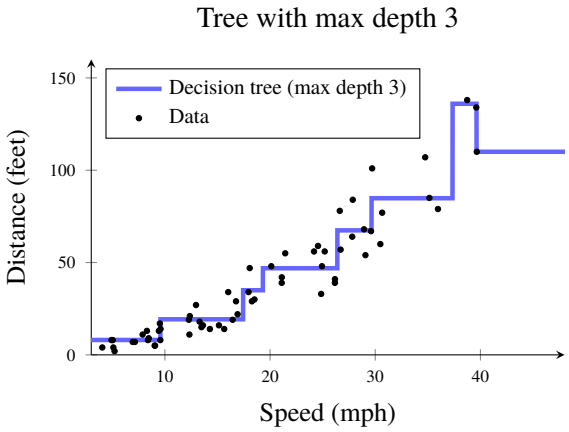
(a) Decision boundaries for the music classification problem for a fully grown classification tree with Gini index. This model overfits the data.



(b) The same problem and data as in 2.3a for which a tree restricted to depth 4 has been learned, again using Gini index. This models will hopefully make better predictions for new data.



(c) The prediction for a fully grown regression tree. As for the classification problem above, this model overfits to the training data.



(d) The same problem and data as in 2.3c for which a tree restricted to depth 3 has been learned.

Figure 2.3: Decision trees applied to the music classification Example 2.1 (a and b) and the car stopping distance Example 2.2 (c and d).

3 Basic parametric models for regression and classification

In the previous chapter we saw the supervised machine learning problem, as well as two methods for solving it. In this chapter we introduce a systematic approach referred to as *parametric* modeling, by first looking at *linear regression* and *logistic regression* which are two such parametric models. The key point of a parametric model is that it contains some parameters $\boldsymbol{\theta}$, which are learned from training data. However, once the parameters are learned, the training data may be discarded, since the prediction only depends on $\boldsymbol{\theta}$.

3.1 Linear regression

Regression is one of the two fundamental tasks of supervised learning (the other one is classification). We will now introduce the *linear regression* model, which might (at least historically) be the most popular method for solving regression problems. Despite its relative simplicity, it is a surprisingly useful and is an important stepping stone for more advanced methods, such as deep learning, see Chapter 6.

As discussed in the previous chapter, regression amounts to learning the relationships between some input variables $\mathbf{x} = [x_1 \; x_2 \; \dots \; x_p]^\mathsf{T}$ and a numerical output variable y. The inputs can be either categorical or numerical, but we will start by assuming that they also are numerical and discuss categorical inputs later. In a more statistical framework, regression is about learning a model f

$$y = f(\mathbf{x}) + \varepsilon, \tag{3.1}$$

mapping the input to the output, where ε is an error term that describes everything about the input–output relationship that cannot be captured by the model. With a statistical perspective, we consider ε as random variable, referred to as a *noise*, that is independent of $\mathbf{x}$ and has mean value of zero. As a running example of regression, we will use the car stopping distance regression problem introduced in the previous chapter as Example 2.2.

The linear regression model

The linear regression model assumes that the output variable y (a scalar) can be described as an affine[1] combination of the input variables $x_1, x_2, \dots, x_p$ (each a scalar) plus a noise term ε,

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \varepsilon. \tag{3.2}$$

We refer to the coefficients $\theta_0, \theta_1, \dots \theta_p$ as the *parameters* of the model, and we sometimes refer to θ_0 specifically as the intercept (or offset) term. The noise term ε accounts for random errors in the data not captured by the model. The noise is assumed to have mean zero and to be independent of $\mathbf{x}$. The zero-mean assumption is nonrestrictive, since any (constant) non-zero mean can be incorporated in the offset term θ_0.

To have a more compact notation, we introduce the parameter vector $\boldsymbol{\theta} = [\theta_0 \; \theta_1 \; \dots \; \theta_p]^\mathsf{T}$ and extend the vector $\mathbf{x}$ with a constant one in its first position, such that we can write the linear regression model (3.2)

[1] An affine function is a linear function plus a constant offset.

compactly

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \varepsilon = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_p \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_p \end{bmatrix} + \varepsilon = \boldsymbol{\theta}^{\mathsf{T}} \mathbf{x} + \varepsilon \tag{3.3}$$

This notation means that the symbol $\mathbf{x}$ unfortunately is used both for the $p+1$ and the p-dimensional version of the input vector, with or without the constant one in the leading position. This is only a matter of book-keeping for handling the intercept term θ_0 and will be clear from context, and carries no deeper meaning,

The linear regression model is a *parametric* function of the form (3.3). The parameters $\boldsymbol{\theta}$ can take arbitrary values, and the actual values that we assign to them will control the input–output relationship described by the model. *Learning* of the model therefore amounts to finding suitable values for $\boldsymbol{\theta}$ based on observed training data. Before discussing how to do this, however, let us have a closer look at how the model can be used to make predictions once it has been learned.

Making predictions with the linear regression model

The goal in supervised machine learning is making predictions $\widehat{y}(\mathbf{x}_\star)$ for new, previously unseen, test inputs $\mathbf{x}_\star = \begin{bmatrix} 1 & x_{\star 1} & x_{\star 2} & \ldots & x_{\star p} \end{bmatrix}^{\mathsf{T}}$. Let us assume that we have already learned some parameter values $\widehat{\boldsymbol{\theta}}$ for the linear regression model (how this is done will be described next). We use the symbol $\widehat{\ }$ to indicate that $\widehat{\boldsymbol{\theta}}$ contains learned values of the otherwise unknown parameter vector $\boldsymbol{\theta}$. Since we assume that the noise term ε is random with zero mean and independent of all observed variables, it makes statistical sense to let $\varepsilon = 0$ n the prediction. That is, a prediction from the linear regression model takes the form

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\theta}_0 + \widehat{\theta}_1 x_{\star 1} + \widehat{\theta}_2 x_{\star 2} + \cdots + \widehat{\theta}_p x_{\star p} = \widehat{\boldsymbol{\theta}}^{\mathsf{T}} \mathbf{x}_\star. \tag{3.4}$$

This is in fact applicable for all regression models of the type (3.1), where the model used for prediction in general is $\widehat{y}(\mathbf{x}_\star) = f(\mathbf{x}_\star)$.. The noise term ε is often referred to as an *irreducible error* or an *aleatoric*[2] uncertainty in the prediction. We illustrate the predictions made by a linear regression model in Figure 3.1.

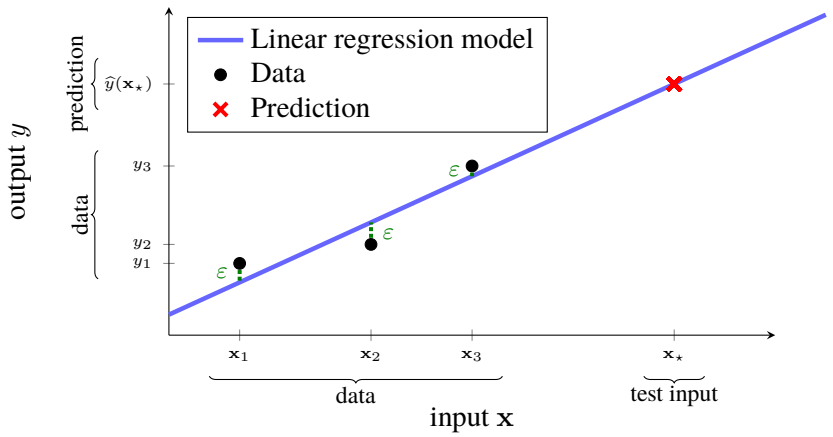


Figure 3.1: Linear regression with $p = 1$: The black dots represent $n = 3$ data samples, from which a linear regression model (blue line) is learned. The model does not fit the data perfectly, but there is a remaining error/noise ε (green). The model can be used to *predict* (red cross) the output $\widehat{y}(\mathbf{x}_\star)$ for a test input $\mathbf{x}_\star$.

[2]From the Latin word *aleator*, meaning dice-player.

Learning linear regression from training data

Let us now discuss how to learn a linear regression model, that is learn $\boldsymbol{\theta}$, from training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$. We collect the training data, which consists of n samples $\mathbf{x}_i$ of the input and corresponding n samples y_i of the output in the $n \times (p+1)$ matrix $\mathbf{X}$ and n-dimensional vector $\mathbf{y}$,

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1^\mathsf{T}- \\ -\mathbf{x}_2^\mathsf{T}- \\ \vdots \\ -\mathbf{x}_n^\mathsf{T}- \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \text{where each } \mathbf{x}_i = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix}. \tag{3.5}$$

Example 3.1: Car stopping distances

We continue Example 2.2, and attempt to learn a liner regression model for it. We therefore 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

$$\mathbf{X} = \begin{bmatrix} 1 & 4.0 \\ 1 & 4.9 \\ 1 & 5.0 \\ 1 & 5.1 \\ 1 & 5.2 \\ \vdots & \vdots \\ 1 & 39.6 \\ 1 & 39.7 \end{bmatrix}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}, \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} 4.0 \\ 8.0 \\ 8.0 \\ 4.0 \\ 2.0 \\ \vdots \\ 134.0 \\ 110.0 \end{bmatrix}. \tag{3.6}$$

Altogether we can use this vector ant matrix notation to describe the linear regression model for each and every training data point $i = 1, \ldots, n$ $\mathbf{X}$ in one equation as a matrix multiplication

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon}, \tag{3.7}$$

where $\boldsymbol{\epsilon}$ is a vector of errors/noise. Moreover we can also define a vector of predicted outputs for the training data $\widehat{\mathbf{y}} = \begin{bmatrix} \widehat{y}(\mathbf{x}_1) & \widehat{y}(\mathbf{x}_2) & \ldots & \widehat{y}(\mathbf{x}_n) \end{bmatrix}^\mathsf{T}$ which also allows a compact matrix formulation

$$\widehat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}. \tag{3.8}$$

Note that whereas $\mathbf{y}$ is a vector of recorded training data values, $\widehat{\mathbf{y}}$ is a vector whose entries are functions of $\boldsymbol{\theta}$. Learning the unknown parameters $\boldsymbol{\theta}$ amounts to finding values such that $\widehat{\mathbf{y}}$ is similar to $\mathbf{y}$, that is, the model should fit the training data well. There are multiple ways to define what 'similar' or 'well' actually means, but it somehow amounts to find $\boldsymbol{\theta}$ such that $\widehat{\mathbf{y}} - \mathbf{y} = \boldsymbol{\epsilon}$ is small. We will approach this by formulating a loss function, which will imply a meaning of 'fitting the data well'. We will thereafter interpret the loss function from a statistical perspective, by understanding this as selecting the value of $\boldsymbol{\theta}$ which makes the observed training data $\mathbf{y}$ as likely as possible to have been observed—the so-called *maximum likelihood* solution. Later in Chapter 9 we will also introduce a conceptually different way to learn $\boldsymbol{\theta}$.

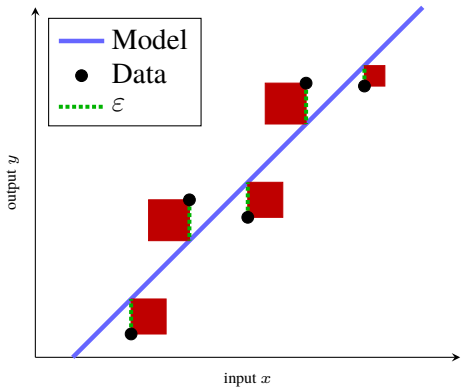


Figure 3.2: A graphical explanation of the squared error loss function: the goal is to choose the model (blue line) such that the sum of the squares (red) of each error ε (green dashed lines) is minimized. That is, the blue line is to be chosen so that the amount of red color is minimized. The black dots, the training data, are fixed. This motivates the name *least squares*.

Loss functions and cost functions

One principled way to define the learning problem is to introduce a *loss function* $L(\widehat{y}, y)$ which measures how close the model's prediction $\widehat{y}$ is to the observed data y. If the model fits the data well, i.e. if $\widehat{y} \approx y$, then the loss function should take a small value, and vice versa. Based on the chosen loss function we also define the *cost function* as the average loss over the training data. Learning a model then amounts to finding the parameter values that minimize the cost

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \overbrace{L(\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta}), y_i)}^{\text{loss function}}}_{\text{cost function } J(\boldsymbol{\theta})}. \tag{3.9}$$

Note that each term in the expression above corresponds to evaluating the loss function for the prediction $\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta})$, given by (3.4), for the training point with index i and the true output value y_i at that point. To emphasize that the prediction depends on the parameters $\boldsymbol{\theta}$, we have included $\boldsymbol{\theta}$ as an argument to $\widehat{y}$ for clarity. The operator $\arg\min_{\boldsymbol{\theta}}$ means "the value of $\boldsymbol{\theta}$ for which the cost function attains it minimum". The relationship between loss and cost functions (3.9) is general for all cost functions in this book.

Least squares and the normal equations

For regression, a commonly used loss function is the *squared error* loss

$$L(\widehat{y}(\mathbf{x};\boldsymbol{\theta}), y) = \big(\widehat{y}(\mathbf{x};\boldsymbol{\theta}) - y\big)^2. \tag{3.10}$$

This loss function attains 0 if $\widehat{y}(\mathbf{x};\boldsymbol{\theta}) = y$, and grows fast (quadratic) as the difference between y and the prediction $\widehat{y}(\mathbf{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}$ increases. The corresponding cost function for the linear regression model (3.7) can be written with matrix multiplications as

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta}) - y_i)^2 = \frac{1}{n}\|\widehat{\mathbf{y}} - \mathbf{y}\|_2^2 = \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \frac{1}{n}\|\boldsymbol{\epsilon}\|_2^2, \tag{3.11}$$

where $\|\cdot\|_2$ denotes the usual Euclidean vector norm, and $\|\cdot\|_2^2$ its square. Due to the square, this particular cost function is also commonly referred to as the *least squares* cost. It is illustrated in Figure 3.2. We will discuss other loss functions in Chapter 5.

When using the squared error loss for learning a linear regression model from $\mathcal{T}$, we thus need to solve the problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{\theta}^\mathsf{T} \mathbf{x}_i - y_i)^2 = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2. \tag{3.12}$$

From a linear algebra point of view, this can be seen as the problem of finding the closest vector to $\mathbf{y}$ (in an Euclidean sense) 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 $\widehat{\boldsymbol{\theta}}$ can be shown (see Section 3.A) to fulfill

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}. \tag{3.13}$$

Equation (3.13) is often referred to as the *normal equations*, and gives the solution to the least squares problem (3.12). If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, which is often the case, then $\widehat{\boldsymbol{\theta}}$ has the closed form expression

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{3.14}$$

The fact that this closed-form solution exists is important, and is probably the reason for why the linear regression with squared error loss is so extremely commonly used in practice. Other loss functions lead to optimization problems that often lack closed-form solutions.

We now have everything in place for using linear regression, and we summarize by Algorithm 4 and Algorithm 5, as well as Example 3.2.

Time to reflect 3.1: *What does it mean in practice if* $\mathbf{X}^\mathsf{T}\mathbf{X}$ *is not invertible?*

Time to reflect 3.2: *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* $\mathbf{y} = \mathbf{X}\boldsymbol{\theta}$ *exactly, i.e., the model fits the training data perfectly. If that is the case,* (3.14) *reduces to* $\boldsymbol{\theta} = \mathbf{X}^{-1}\mathbf{y}$, *and the model fits the data perfectly. Why would that not be a desired property in practice?*

Algorithm 4: Learning a linear regression model with squared error loss

Data: Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
Result: Learned parameter vector $\widehat{\boldsymbol{\theta}}$

1 Construct the matrix $\mathbf{X}$ and vector $\mathbf{y}$ according to (3.5).
2 Compute $\widehat{\boldsymbol{\theta}}$ by solving (3.13).

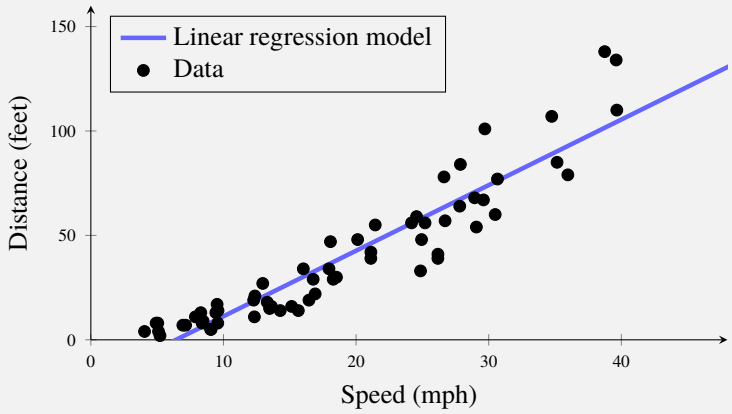
Algorithm 5: Making predictions from a linear regression model

Data: Learned parameters $\widehat{\boldsymbol{\theta}}$ and test input $\mathbf{x}_\star$
Result: Prediction $\widehat{y}(\mathbf{x}_\star)$

1 Compute $\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T}\mathbf{x}_\star$.

Example 3.2: Car stopping distances

By inserting the matrices (3.6) from Example 3.1 into the normal equations (3.7), we obtain $\widehat{\theta}_0 = -20.1$ and $\widehat{\theta}_1 = 3.14$. If we plot the resulting model, it looks like this:



This can be compared to how k-NN and decision trees solves the same problem, Figure 2.1 and 2.3. Clearly the linear regression model behaves differently to k-NN and decision trees; linear regression does not share the "local" nature of k-NN and decision trees (only training data points close to $\mathbf{x}_\star$ affects $\widehat{y}(\mathbf{x}_\star)$), which is related to the fact that linear regression is a parametric model.

The maximum likelihood perspective

To get another perspective of the squared error loss we will now reinterpret the least squares method above as a *maximum likelihood* solution. The word 'likelihood' refers to the statistical concept of the likelihood function, and maximizing the likelihood function amounts to finding the value of $\boldsymbol{\theta}$ that makes observing $\mathbf{y}$ as likely as possible. That is, instead of (somewhat arbitrarily) selecting a loss function, we start with the problem

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}). \tag{3.15}$$

Here $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$ is the probability density of all observed outputs $\mathbf{y}$ in the training data, given all inputs $\mathbf{X}$ and parameters $\boldsymbol{\theta}$. This determines mathematically what 'likely' means, but we need to specify it in more detail. We do that by considering the noise term ε as a stochastic variable with a certain distribution. A common assumption is the noise terms are independent, each with a Gaussian distribution with zero mean and variance σ_ε^2,

$$\varepsilon \sim \mathcal{N}\left(0, \sigma_\varepsilon^2\right). \tag{3.16}$$

This implies that the n observed training data points are independent, and $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$ factorizes as

$$p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}). \tag{3.17}$$

Together with the linear regression model (3.3) we have

$$p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}) = \mathcal{N}\left(y_i; \boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i, \sigma_\varepsilon^2\right) = \frac{1}{\sqrt{2\pi\sigma_\varepsilon^2}} \exp\left(-\frac{1}{2\sigma_\varepsilon^2}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2\right). \tag{3.18}$$

Recall that we want to maximize the likelihood w.r.t. $\boldsymbol{\theta}$. For numerical reasons, it is usually better to work with the logarithm of $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$,

$$\ln p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}). \tag{3.19}$$

Since the logarithm is a monotonically increasing function, maximizing the log-likelihood (3.19) is equivalent to maximizing the likelihood itself. Putting (3.18) and (3.19) together, we get

$$\ln p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = -\frac{n}{2} \ln(2\pi\sigma_\varepsilon^2) - \frac{1}{2\sigma_\varepsilon^2} \sum_{i=1}^{n} \left(\boldsymbol{\theta}^\mathsf{T} \mathbf{x}_i - y_i \right)^2. \tag{3.20}$$

Removing terms and factors independent of $\boldsymbol{\theta}$ does not change the maximizing argument, and we see that we can rewrite (3.15) as

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} -\sum_{i=1}^{n} \left(\boldsymbol{\theta}^\mathsf{T} \mathbf{x}_i - y_i \right)^2 = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \frac{1}{n} \left(\boldsymbol{\theta}^\mathsf{T} \mathbf{x}_i - y_i \right)^2. \tag{3.21}$$

This is indeed linear regression with the least squares cost (the cost function implied by the squared error loss function (3.10)). The key to arrive at this expression was the assumption of ε having a Gaussian distribution. Other assumptions on ε leads to other loss functions, as we will discuss more in Chapter 5.

Categorical input variables

The regression problem is characterized by a numerical output y, and inputs $\mathbf{x}$ of arbitrary type. We have, however, only discussed the case of numerical inputs so far, but categorical inputs are perfectly possible as well.

Assume that we have a categorical input variable that only takes two different values. We refer to those two values as `A` and `B`. We can then create a *dummy variable* x as

$$x = \begin{cases} 0 & \text{if } \texttt{A}, \\ 1 & \text{if } \texttt{B}, \end{cases} \tag{3.22}$$

and use this variable in any supervised machine learning method as if it was numerical. For linear regression, this effectively gives us a model which looks like

$$y = \theta_0 + \theta_1 x + \varepsilon = \begin{cases} \theta_0 + \varepsilon & \text{if } \texttt{A}, \\ \theta_0 + \theta_1 + \varepsilon & \text{if } \texttt{B}. \end{cases} \tag{3.23}$$

The choice is somewhat arbitrary, since `A` and `B` of course can be switched. If the categorical variable takes more than two values, let us say `A`, `B`, `C` and `D`, we can make a so-called one-hot encoding by constructing a four-dimensional vector

$$\mathbf{x} = \begin{bmatrix} x_A & x_B & x_C & x_D \end{bmatrix}^\mathsf{T} \tag{3.24}$$

where $x_A = 1$ if `A`, $x_B = 1$ if `B`, and so on. That is, only one element of $\mathbf{x}$ will be 1, the rest are 0. Again, this construction can be used for any supervised machine learning method, not only linear regression.

3.2 Classification and logistic regression

After presenting a parametric method for solving the regression problem, we now turn our attention to classification. As we will see, with a modification of the linear regression model we can apply it to the classification problem as well, however at the cost of not being able to use the convenient normal equations for learning the parameters. To perform Instead we have to resort to numerical optimization, which we will discuss later in Section 5.3.

A statistical view of the classification problem

Supervised machine learning amounts to predicting the output from the input. With a statistical perspective, classification amounts to predicting the conditional class probabilities

$$p(y = m \mid \mathbf{x}), \tag{3.25}$$

where y is the output $(1, 2, \ldots,$ or $M)$ and $\mathbf{x}$ is the input.[3] In words, $p(y = m \mid \mathbf{x})$ describes *the probability for class m given that we know the input* $\mathbf{x}$. 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 (much like the random error ε in regression). Let us illustrate with an example:

Example 3.3: Describing voting behavior using probabilities

We want to construct a model that can predict voting preferences (= y, the categorical output) for different population groups (= $\mathbf{x}$, the input). However, we then have to face the fact that not everyone in a certain population group will vote for the same political party. We can therefore 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 (here we have $M = 3$), we could describe it as

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

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

(a) all 45 year old women do not 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.

Thus, it can be useful to have a classifier which predicts not only a class $\widehat{y}$ (one party), but a distribution over classes $p(y \mid \mathbf{x})$.

We now aim to construct a classifier which can not only predict classes, but also learn the class probabilities $p(y \mid \mathbf{x})$. More specifically, for binary classification problems ($M = 2$, and y is either 1 or -1), we learn a model $g(\mathbf{x})$ for which

$$\boxed{p(y = 1 \mid \mathbf{x}) \text{ is modeled by } g(\mathbf{x}).} \tag{3.26a}$$

By the laws of probabilities, it holds that $p(y = 1 \mid \mathbf{x}) + p(y = -1 \mid \mathbf{x}) = 1$, which gives that

$$\boxed{p(y = -1 \mid \mathbf{x}) \text{ is modeled by } 1 - g(\mathbf{x}).} \tag{3.26b}$$

Since $g(\mathbf{x})$ is a model for a probability, it is natural to require that $0 \leq g(\mathbf{x}) \leq 1$ for any $\mathbf{x}$. We will see how this constraint can be enforced below.

[3] We use the notation $p(y \mid \mathbf{x})$ to denote probability masses (y discrete) as well as probability densities (y continuous).

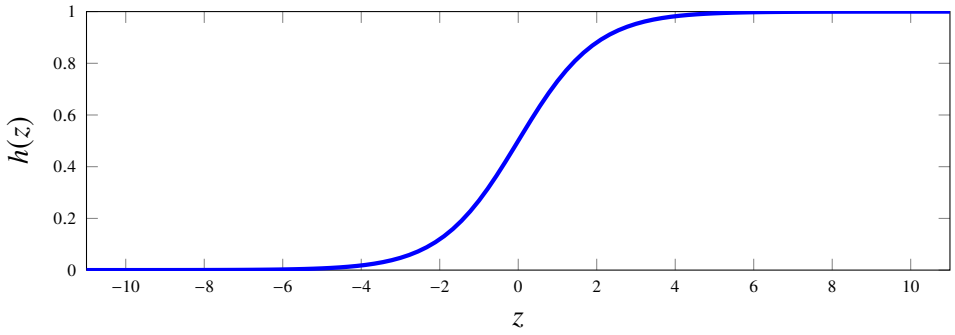


Figure 3.3: The logistic function $h(z) = \frac{e^z}{1+e^z}$.

For the multiclass problem, we instead let the classifier return a vector-valued function $\mathbf{g}(\mathbf{x})$, where

$$\begin{bmatrix} p(y=1 \mid \mathbf{x}) \\ p(y=2 \mid \mathbf{x}) \\ \vdots \\ p(y=M \mid \mathbf{x}) \end{bmatrix} \text{ is modeled by } \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ \vdots \\ g_M(\mathbf{x}) \end{bmatrix} = \mathbf{g}(\mathbf{x}). \tag{3.27}$$

In words, each element $g_m(\mathbf{x})$ of $\mathbf{g}(\mathbf{x})$ corresponds to the conditional class probability $p(y = m \mid \mathbf{x})$. Since $\mathbf{g}(\mathbf{x})$ models a probability vector, we require that each element $g_m(\mathbf{x}) \geq 0$ and that $\|\mathbf{g}(\mathbf{x})\|_1 = \sum_{m=1}^{M} |g_m(\mathbf{x})| = 1$ for any $\mathbf{x}$.

The logistic regression model

We will now introduce the logistic regression model, which is one possible way of modeling conditional class probabilities. Logistic regression can be viewed as a modification of the linear regression model so that it fits the classification (rather than regression) problem.

Let us start with binary classification, that is, learning a function $g(\mathbf{x})$ that approximates the conditional probability of the positive class. The linear regression model, without the noise term, is given by

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p = \boldsymbol{\theta}^\mathsf{T} \mathbf{x}. \tag{3.28}$$

This is a mapping which takes $\mathbf{x}$ and returns z, which in this context is called the *logit*. Note that z takes values on the entire real line, whereas we need a function which instead returns a value on the interval $[0, 1]$. The key idea of logistic regression is thus to 'squeeze' z from (3.28) to the interval $[0, 1]$ by using the logistic function $h(z) = \frac{e^z}{1+e^z}$, see Figure 3.3. This gives the function

$$g(\mathbf{x}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}, \tag{3.29a}$$

which is restricted to $[0, 1]$ and hence can be interpreted as a probability. (3.29a) is the logistic regression model for $p(y = 1 \mid \mathbf{x})$. Note that this implicitly also gives a model for $p(y = -1 \mid \mathbf{x})$,

$$1 - g(\mathbf{x}) = 1 - \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}. \tag{3.29b}$$

These equations defines the logistic regression model, which in a nutshell is linear regression appended with the logistic function. The reason why there is no noise term ε in (3.28) when comparing to the linear regression model (3.3) is that the randomness in classification is statistically modeled by the class probability construction $p(y = m \mid \mathbf{x})$ instead of an additive noise ε.

As for linear regression, we have a model (3.29) which contains unknown parameters $\boldsymbol{\theta}$. Logistic regression is thereby also a parametric model, and we learn the parameters from training data.

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

Learning the logistic regression model by maximum likelihood

By using the logistic function, we have transformed linear regression (a model for regression problems) into logistic regression (a model for classification problems). As it will turn out, the price to pay is that we will not be able to use the convenient normal equations for learning $\boldsymbol{\theta}$ in logistic regression (as we could for linear regression if we used the squared error loss).

In order to derive a principled way of learning $\boldsymbol{\theta}$ in (3.29) from training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, we start with the maximum likelihood approach. From a maximum likelihood perspective, learning a classifier amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}), \tag{3.30}$$

where we, similarly to linear regression (3.19), assume that the training data points are independent and we consider the logarithm of the likelihood function for numerical reasons. We have also added $\boldsymbol{\theta}$ explicitly to the notation to emphasize the dependence on the model parameters. Remember that our model of $p(y = 1 \mid \mathbf{x}; \boldsymbol{\theta})$ is $g(\mathbf{x}; \boldsymbol{\theta})$, which gives

$$\ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}) = \begin{cases} \ln g(\mathbf{x}_i; \boldsymbol{\theta}) & \text{if } y_i = 1, \\ \ln (1 - g(\mathbf{x}_i; \boldsymbol{\theta})) & \text{if } y_i = -1. \end{cases} \tag{3.31}$$

It is common to turn the maximization problem (3.30) into an equivalent minimization problem by using the negative log-likelihood as cost function, $J(\boldsymbol{\theta}) = -\frac{1}{n} \sum \ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta})$, that is

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\begin{cases} -\ln g(\mathbf{x}_i; \boldsymbol{\theta}) & \text{if } y_i = 1, \\ -\ln (1 - g(\mathbf{x}_i; \boldsymbol{\theta})) & \text{if } y_i = -1. \end{cases}}_{\text{Binary cross-entropy loss } L(g(\mathbf{x}_i;\boldsymbol{\theta}), y_i)} \tag{3.32}$$

The loss function in the expression above is called the *cross-entropy loss*. It is not specific to logistic regression, but can be used for any binary classifier that predicts class probabilities $g(\mathbf{x}; \boldsymbol{\theta})$.

Considering specifically the logistic regression model we can write out the cost function (3.32) in more detail. In doing so, the particular choice of labeling $\{-1, 1\}$ turns out to be convenient since we for $y_i = 1$ we can write

$$g(\mathbf{x}_i; \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}, \tag{3.33a}$$

and for $y_i = -1$

$$1 - g(\mathbf{x}_i; \boldsymbol{\theta}) = \frac{e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}. \tag{3.33b}$$

Since we get the same expression in both cases, we write (3.32) compactly as

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} -\ln \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{1}{n} \sum_{i=1}^{n} -\ln \frac{1}{1 + e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\ln \left(1 + e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}\right)}_{\text{Logistic loss } L(\mathbf{x}_i, y_i, \boldsymbol{\theta})}. \tag{3.34}$$

The loss function $L(\mathbf{x}, y_i, \boldsymbol{\theta})$ above, which is the cross-entropy loss specialized to logistic regression, is called the *logistic loss* (or sometimes binomial deviance). Learning a logistic regression model thus amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \ln \left(1 + e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}\right). \tag{3.35}$$

Contrary to linear regression with squared error loss, (3.35) has no closed-form solution, so we have to use numerical optimization instead. We will come back to that topic in Section 5.3.

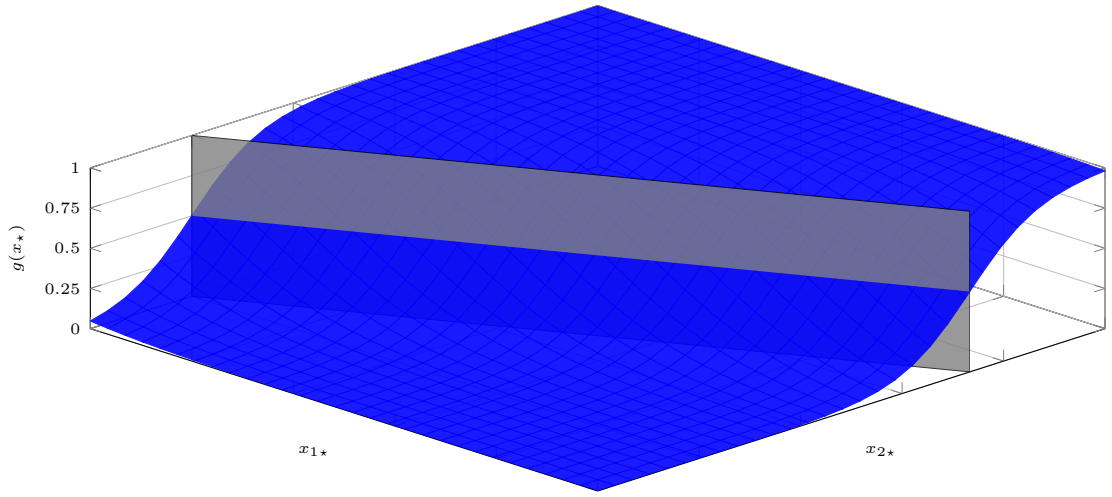


Figure 3.4: In binary classification ($y = -1$ or 1) logistic regression predicts $g(\mathbf{x}_\star)$ ($\mathbf{x}$ is two-dimensional here), which is an attempt to determine $p(y = 1 \mid \mathbf{x}_\star)$. This implicitly also gives a prediction for $p(y = -1 \mid \mathbf{x}_\star)$ as $1 - g(\mathbf{x}_\star)$. To turn these probabilities into actual class predictions ($\widehat{y}(\mathbf{x}_\star)$ is either -1 or 1), the class which is modeled to have the highest probability can be taken as the prediction, as in Equation (3.36). The point(s) where the prediction changes from from one class to another is the decision boundary (gray plane).

Predictions and decision boundaries

So far, we have discussed logistic regression as a method for predicting class probabilities for a test input $\mathbf{x}_\star$ by first learning $\boldsymbol{\theta}$ from training data and thereafter computing $g(\mathbf{x}_\star)$ (our model for $p(y = 1 \mid \mathbf{x}_\star)$ in binary classification) or $\mathbf{g}(\mathbf{x}_\star)$ (our model for $p(y = m \mid \mathbf{x}_\star)$ in multiclass classification). However, sometimes we want to make a "hard" prediction for the test input $\mathbf{x}_\star$, that is, predicting $\widehat{y}(\mathbf{x}_\star) = -1$ or $\widehat{y}(\mathbf{x}_\star) = 1$ in binary classification, just like with k-NN or decision trees. We then have to append logistic regression with a final step in which the predicted probabilities are turned into a class prediction. The most common approach is to *let the most probable class be* $\widehat{y}(\mathbf{x}_\star)$. For binary classification, we can express this[4] as

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > r \\ -1 & \text{if } g(\mathbf{x}) \leq r \end{cases}, \tag{3.36}$$

with decision threshold $r = 0.5$, which is illustrated in Figure 3.4. With binary logistic regression, this is equivalent to

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_\star). \tag{3.37}$$

We now have everything in place for summarizing binary logistic regression in Algorithm 6.

Algorithm 6: Logistic regression for binary classification

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

Learn

1 Compute $\widehat{\boldsymbol{\theta}}$ by solving (3.35) numerically.

Predict

2 Compute $g(\mathbf{x}_\star)$ (3.29a).
3 If $g(\mathbf{x}_\star) > 0.5$, return $\widehat{y}(\mathbf{x}_\star) = 1$, otherwise return $\widehat{y}(\mathbf{x}_\star) = -1$.

In some applications, however, it can be beneficial to explore different thresholds than $r = 0.5$. It can be shown that *if* $g(\mathbf{x}) = p(y = 1 \mid \mathbf{x})$, that is, the model provides a correct description of the real-world

[4] It is arbitrary what happens if $g(\mathbf{x}) = 0.5$.

class probabilities, then the choice $r = 0.5$ will give the smallest possible number of misclassification on average. In other words, $r = 0.5$ minimizes the so-called *misclassification rate*. The misclassification rate is, however, not always the most important aspect of classifier. Many classification problems are asymmetric (the classes are of different importance to correctly predict) or imbalanced (the classes occur with very different frequency). In a medical diagnosis application, for example, it can be of higher importance not to falsely predict the negative class (that is, by mistake predict a sick patient being healthy) than falsely predict the positive class (by mistake predict a healthy patient as sick). For such a problem, minimizing the misclassification rate might lead not lead to the desired performance. Furthermore, the medical diagnosis problem could be imbalanced if the disorder is very rare, meaning that the vast majority of the data samples (patients) belong to the negative class. By only considering the misclassification rate in such a situation we implicitly value accurate predictions of the negative class higher than accurate predictions of the positive class, simply because the negative class is more common in the data. We will discuss how we can evaluate such situations more systematically in Section 4.5. In the end, however, the decision threshold r is a choice that the user has to make.

For the multiclass problem, taking the prediction as the most probable class amounts to solving

$$\widehat{y}(\mathbf{x}_\star) = \arg\max_m g_m(\mathbf{x}_\star). \tag{3.38}$$

As for the binary case, it is possible to modify this when working with an asymmetric or imbalanced problem.

The decision boundary for binary classification can be computed by solving the equation

$$g(\mathbf{x}) = 1 - g(\mathbf{x}). \tag{3.39}$$

The solutions to this equation are points in the input space for which the two classes are predicted to be equally probable. These points therefore lie on the decision boundary. For binary logistic regression, this means

$$\frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} \Leftrightarrow e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}} = 1 \Leftrightarrow \boldsymbol{\theta}^\mathsf{T}\mathbf{x} = 0. \tag{3.40}$$

The equation $\boldsymbol{\theta}^\mathsf{T}\mathbf{x} = 0$ parameterizes a (linear) hyperplane. Hence, the decision boundaries in logistic regression always have the shape of a (linear) hyperplane. The same argument can be generalized to multiclass logistic regression, but the decision boundaries will then be given by a combination of $M - 1$ hyperplanes.

In general we distinguish between different types of classifiers by the shape of their decision boundaries.

A classifier whose decision boundaries are linear hyperplanes is a *linear classifier.*

All other classifiers are *nonlinear classifiers*. Logistic regression is an example of a linear classifier, whereas k-NN and decision trees are nonlinear classifiers.

Note that the term "linear" is used in a different sense for linear regression; linear regression is a model which is linear in its parameters, whereas a linear classifier is a model whose decision boundaries are linear.

Logistic regression for more than two classes

Logistic regression can be used also for the multiclass problem when there are more than two classes, $M > 2$. There are several ways of generalizing logistic regression to the multiclass problem. We will follow one path using the so-called softmax function which will be useful also later when introducing deep learning models in Chapter 6.

For the binary problem, we used the logistic function to design a model for $g(\mathbf{x})$ (a scalar-valued function representing $p(y = 1 \mid \mathbf{x})$). For the multiclass problem we instead have to design a vector-valued function $\mathbf{g}(\mathbf{x})$, whose elements should be non-negative and sum to one. For this purpose, we first use M instances of (3.28), each denoted z_m and each with a different set of parameters $\boldsymbol{\theta}_m$. We stack all z_m into a vector of logits $\mathbf{z} = [z_1 \; z_2 \; \ldots \; z_M]^\mathsf{T}$ and use the *softmax function* as a vector-valued generalization of the logistic function,

$$\operatorname{softmax}(\mathbf{z}) \triangleq \frac{1}{\sum_{m=1}^{M} e^{z_m}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_M} \end{bmatrix}. \tag{3.41}$$

Note that the argument $\mathbf{z}$ to the softmax function is an M-dimensional vector, and that it also returns a vector of the same dimension. By construction, the output vector from the softmax function always sums to 1, and each element is always ≥ 0. Similarly to how we combined linear regression and the logistic function for the binary classification problem (3.29), we have now combined linear regression and the softmax function to model the class probabilities,

$$\mathbf{g}(\mathbf{x}) = \operatorname{softmax}(\mathbf{z}), \qquad \text{where } \mathbf{z} = \begin{bmatrix} \boldsymbol{\theta}_1^\mathsf{T}\mathbf{x}_i \\ \boldsymbol{\theta}_2^\mathsf{T}\mathbf{x}_i \\ \vdots \\ \boldsymbol{\theta}_M^\mathsf{T}\mathbf{x}_i \end{bmatrix}. \tag{3.42}$$

Equivalently, we can write out the individual class probabilities, i.e. the elements of the vector $\mathbf{g}(\mathbf{x})$, as

$$g_m(\mathbf{x}) = \frac{e^{\boldsymbol{\theta}_m^\mathsf{T}\mathbf{x}_i}}{\sum_{j=1}^{M} e^{\boldsymbol{\theta}_j^\mathsf{T}\mathbf{x}_i}}, \qquad m = 1, \ldots, M. \tag{3.43}$$

This is the multiclass logistic regression model. Note that this construction uses M parameter vectors $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M$ (one for each class), meaning that the number of parameters to learn grows with M. As for binary logistic regression, we can learn those parameters using the maximum likelihood method. We use $\boldsymbol{\theta}$ to denote *all* model parameters, $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M\}$. Since $g_m(\mathbf{x}; \boldsymbol{\theta})$ is our model for $p(y_i = m \mid \mathbf{x}_i)$, the cost function for the cross-entropy (or negative log-likelihood) loss for the multiclass problem is

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{-\ln g_{y_i}(\mathbf{x}_i; \boldsymbol{\theta})}_{\substack{\text{Multi-class cross-entropy} \\ \text{loss } L(\mathbf{g}(\mathbf{x}_i;\boldsymbol{\theta}), y_i)}}. \tag{3.44}$$

Note that we use the training data labels y_i as index variables to select the correct conditional probability for the loss function. That is, the ith term of the sum is the negative logarithm of the y_ith element of the vector $\mathbf{g}(\mathbf{x}_i; \boldsymbol{\theta})$. We illustrate the meaning of this further in example 3.4.

Inserting the model (3.43) into the loss function (3.44) gives the cost function to optimize when learning multiclass logistic regression (the softmax version),

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(-\boldsymbol{\theta}_{y_i}^\mathsf{T}\mathbf{x}_i + \ln \sum_{j=1}^{M} e^{\boldsymbol{\theta}_j^\mathsf{T}\mathbf{x}_i} \right). \tag{3.45}$$

Example 3.4: The cross-entropy loss for multiclass problems

Consider the following (very small) data set with $n = 6$ data samples, $p = 2$ input dimensions and $M = 3$ classes, which we want to use for learning a multiclass classifier:

$$\mathbf{X} = \begin{bmatrix} 0.20 & 0.86 \\ 0.41 & 0.18 \\ 0.96 & -1.84 \\ -0.25 & 1.57 \\ -0.82 & -1.53 \\ -0.31 & 0.58 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 1 \\ 3 \end{bmatrix}.$$

Multiclass logistic regression with softmax, or any other multiclass classifier which predicts conditional class probabilities, return a 3-dimensional probability vector $\mathbf{g}(\mathbf{x};\boldsymbol{\theta})$ for any $\mathbf{x}$ and $\boldsymbol{\theta}$. If we stack the logarithms of the transpose of all vectors $\mathbf{g}(\mathbf{x}_i;\boldsymbol{\theta})$ for $i = 1,\ldots,6$, we obtain the matrix

$$\mathbf{G} = \begin{bmatrix} \ln g_1(\mathbf{x}_1;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_1;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_1;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_2;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_2;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_2;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_3;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_3;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_3;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_4;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_4;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_4;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_5;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_5;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_5;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_6;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_6;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_6;\boldsymbol{\theta}) \end{bmatrix}.$$

Computing the multi-class cross-entropy cost (3.44) now simply amounts to compute the average of all circled elements, and multiply that with -1. The element that we have circled in row i is given by the training label y_i. Training the model amounts to finding $\boldsymbol{\theta}$ such that this average is maximized.

Time to reflect 3.3: *Can you derive* (3.32) *as a special case of* (3.44)*?*
Hint: think of the binary case as a special case of the multiclass case with $\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g(\mathbf{x}) \\ 1 - g(\mathbf{x}) \end{bmatrix}$.

Time to reflect 3.4: *The softmax-based logistics regression is actually over-parameterized, in the sense that we can construct an equivalent model with fewer parameters. That is often not a problem in practice, but compare the multiclass model* (3.42) *for the case* $M = 2$ *with binary logistic regression* (3.29) *and see if you can spot the over-parametrization!*

3.3 Polynomial regression and regularization

In comparison to k-NN and decision trees in Chapter 2, linear and logistic regression might appear to be rigid and non-flexible models with their straight lines (such as Figure 3.1 and 3.4). However, both models are able to adapt to the training data well if the input dimension p is large, or the number of data samples n is small.

A common way of increasing the input dimension in linear and logistic regression, which we will discuss more thoroughly in Chapter 8, is to make a nonlinear transformation of the input. A simple nonlinear transformation is to replace a one-dimensional input x with itself raised to different powers, which makes the linear regression model a polynomial

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots + \varepsilon. \tag{3.46}$$

This is called *polynomial regression*, and can be applied also to logistic regression. Note that if we let $x_1 = x$, $x_2 = x^2$ and $x_3 = x^3$, this is still a linear model (3.2) with input $\mathbf{x} = [1\ x\ x^2\ x^3]$, but we have 'lifted' the input from being one-dimensional ($p = 1$) to three-dimensional ($p = 3$). Using nonlinear input transformations can be very useful in practice, but it effectively increases p and we may easily end up *overfitting* the model to the noise—rather than the interesting patterns—in the training data, as in the example below.

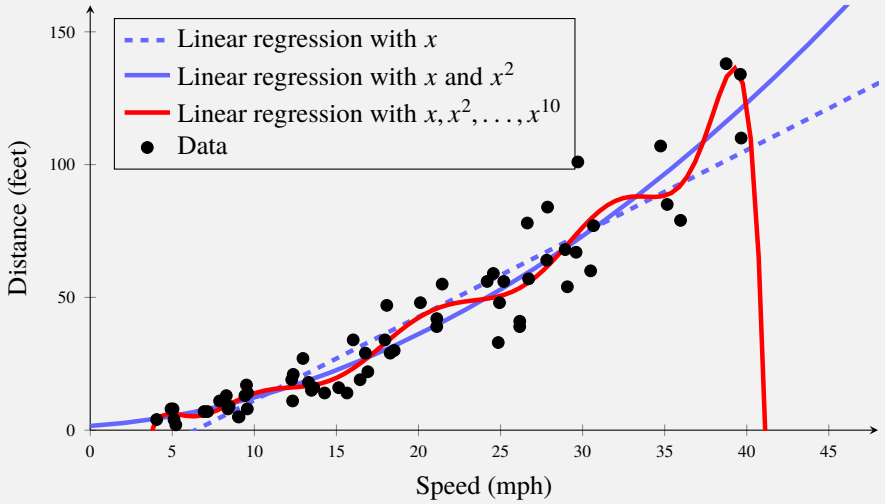
Example 3.5: Car stopping distances with polynomial regression

We return to Example 2.2, but this time we also add the squared speed as an input and thereby use a 2nd order polynomial in linear regression. This gives the new matrices (instead of Example 3.1)

$$\mathbf{X} = \begin{bmatrix} 1 & 4.0 & 16.0 \\ 1 & 4.9 & 24.0 \\ 1 & 5.0 & 25.0 \\ \vdots & \vdots & \vdots \\ 1 & 39.6 & 1568.2 \\ 1 & 39.7 & 1576.1 \end{bmatrix}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4.0 \\ 8.0 \\ 8.0 \\ \vdots \\ 134.0 \\ 110.0 \end{bmatrix}, \tag{3.47}$$

and when we insert them into the normal equations (3.13), the new parameter estimates are $\widehat{\theta}_0 = 1.58$, $\widehat{\theta}_1 = 0.42$ and $\widehat{\theta}_2 = 0.07$. (Note that also $\widehat{\theta}_0$ and $\widehat{\theta}_1$ change, compared to Example 3.2.)

In a completely analogous way we also learn a 10th order polynomial, and we illustrate them all below.



The second order polynomial (blue solid line) appears sensible, and using a 2nd order polynomial seems to give some advantage compared to plain linear regression (blue dashed line, from Example 3.2). However, using a 10th order polynomial (red solid line) seems to make the model less useful than even plain linear regression, and instead suffering from overfit. In conclusion it seems that there is some merit to the idea of polynomial regression, but it has to be applied carefully.

For linear (or logistic) regression with nonlinear input transformations, one could indeed mitigate overfit issues by very carefully selecting the nonlinear transformations. Such a careful selection can be quite hard to perform in practice, and overfit can occur also in other situations when p is large compared to n.

A useful approach in practice is therefore to use *regularization*. The idea of regularization can be described as 'keeping the parameters $\widehat{\boldsymbol{\theta}}$ small unless the data really convinces us otherwise', or alternatively 'if a model with small parameter $\widehat{\boldsymbol{\theta}}$ values 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 idea mathematically, which leads to different regularization methods. We will give a more complete treatment of this in Section 5.2, and only discussed the so-called L^2 regularization for now. When paired with regularization, the idea of using nonlinear input transformations can be very powerful and enables a whole family of supervised machine learning methods that we will properly introduce and discuss in

Chapter 8.

To keep $\widehat{\boldsymbol{\theta}}$ small (in order to prevent overfit) an extra penalty term $\lambda\|\boldsymbol{\theta}\|_2^2$ is added to the cost function when using L^2 regularization. The purpose of the penalty term is to prevent overfit, whereas the original cost function only rewards the fit to training data (which does not prevent overfit). The regularization parameter $\lambda \geq 0$ is therefore important to select wisely, in order to obtain the right amount of regularization. With $\lambda = 0$ the regularization has no effect, whereas $\lambda \to \infty$ will force all parameters $\widehat{\boldsymbol{\theta}}$ to 0. A common solution is to use cross-validation (Chapter 4) to select λ. Applied to linear regression with square error lost (3.12), it becomes[5]

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{3.48}$$

It turns out that (3.48) has a cosed-form solution, as a modified version of the normal equations, namely

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{3.49}$$

where $\mathbf{I}_{p+1}$ is the identity matrix of size $(p+1)\times(p+1)$.

Regularization is not restricted to linear regression. The very same L^2 regularization idea can be applied to any method that involves optimizing a cost function, such as logistic regression

$$\widehat{\boldsymbol{\theta}} = \frac{1}{n}\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i\right)\right) + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{3.50}$$

It is common in practice to train logistic regression using (3.50) instead of (3.29). One reason is indeed to decrease possible overfit issues. Another reason is that for the non-regularized cost function (3.29), the optimal $\widehat{\boldsymbol{\theta}}$ is not finite if the training data is linearly separable (meaning there exists a linear decision boundary which separates the classes perfectly). In practice it means that the logistic regression learning diverges with some datasets, unless (3.50) (with $\lambda > 0$) is used instead of (3.29).

3.4 Nonlinear regression and generalized linear models

In this chapter we have so far introduced two basic parametric models for regression and classification, linear regression and logistic regression, respectively. The concept of parametric modeling is however much more general. Before leaving this chapter we will briefly discuss how the models introduced above can be generalized to describe more intricate input–output relationships. We will also dig deeper into a special case of this later in Chapter 6, where we discuss neural networks.

Generalized linear models

To be written.

Nonlinear parametric functions

Let us remind ourselves about the general regression model (3.1),

$$y = f_{\boldsymbol{\theta}}(\mathbf{x}) + \varepsilon. \tag{3.51}$$

We have introduced an explicit dependence on the parameters $\boldsymbol{\theta}$ in the notation to emphasize that $f_{\boldsymbol{\theta}}$ is a model of the input–output relationship which depends on some parameters $\boldsymbol{\theta}$. To turn this model into a linear regression, that could be trained using least squares with a closed form solution, we made two assumptions in Section 3.1. First, the function $f_{\boldsymbol{\theta}}$ was assumed to be linear in the model parameters, $f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$. Second, the noise term ε was assumed to be Gaussian, $\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$. The latter assumption

[5] In practice, it can be wise to exclude θ_0, the intercept, from the regularization.

is sometimes implicit, but as we saw above it makes the maximum likelihood formulation equivalent to least squares.

Both of these assumptions can be replaced with other assumptions. Replacing the Gaussian assumption will be discussed in Section 5.1, and we will here discuss what happens if we allow the function $f_{\boldsymbol{\theta}}$ to be some arbitrary nonlinear function. We still want to learn the model from data, so $f_{\boldsymbol{\theta}}$ has to depend on some parameters $\boldsymbol{\theta}$ which we can fit to the training data, no matter if it is a linear function or not. Let us give an example of a nonlinear $f_{\boldsymbol{\theta}}$ from biochemistry.

Example 3.6: Michaelis–Menten kinetics

An example of a relatively simple nonlinear parametric function is the Michaelis–Menten equation for modeling enzyme kinetics. The model is given by

$$y = \underbrace{\frac{\theta_1 x}{\theta_2 + x}}_{=f_{\boldsymbol{\theta}}(x)} + \varepsilon$$

where y corresponds to a reaction rate and x a substrate concentration. The model is parameterized by the maximum reaction rate $\theta_1 > 0$ and the so called Michaelis constant of the enzyme $\theta_2 > 0$. Just like the parameters in linear regression, these parameters can be learned from input-output data $\{x_i, y_i\}_{i=1}^n$ by formulating a cost function and solving the resulting optimization problem.

Within biochemistry this model is often expressed as a deterministic relationship without the noise term ε, but we include it as an error term for consistency with our framework.

In the example above the parameters θ_1 and θ_2 have physical interpretations and are restricted to be positive. However, in machine learning we most often lack such physical interpretations of the parameters. The way we use a model is more of a "black box" which is adapted to fit the training data as well as possible. The archetype of such nonlinear black-box models are neural networks, which we will discuss in more detail in Chapter 6.

Nonlinear classification models can be constructed in a very similar way, as a generalization of the logistic regression model (3.42). In multiclass logistic regression we compute a vector of logits $\mathbf{z} = [z_1 \; z_2 \; \dots \; z_M]^\mathsf{T}$ where each element of the vector is given by a linear model $z_m = \boldsymbol{\theta}_m^\mathsf{T}\mathbf{x}$. The class probabilities are then obtained by propagating the logit vector through the softmax function as $g_{\boldsymbol{\theta}}(\mathbf{x}) = \text{softmax}(\mathbf{z})$. To turn this into a nonlinear classification model we can replace the expression for the logit vector with

$$\mathbf{z} = \boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x})$$

where $\boldsymbol{f}_{\boldsymbol{\theta}}$ is some arbitrary function that maps $\mathbf{x}$ to an M-dimensional real-valued vector. Propagating this logit vector through the softmax function, analogously to the logistic regression case, results in a nonliear model for the conditional class probabilities, $g_{\boldsymbol{\theta}}(\mathbf{x}) = \text{softmax}(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}))$. We will return to nonlinear classification models of this form in Chapter 6, where we use neural networks to construct the function $\boldsymbol{f}_{\boldsymbol{\theta}}$.

3.A Derivation of the normal equations

The normal equations (3.13)

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

can be derived from (3.12) (the scaling $\frac{1}{n}$ does not affect the minimizing argument)

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \; \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2,$$

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

No matter how (3.13) is derived, if $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, it (uniquely) gives

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

If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, then (3.13) has infinitely many solutions $\widehat{\boldsymbol{\theta}}$, which all are equally good solutions to the problem (3.12).

A calculus approach

Let

$$V(\boldsymbol{\theta}) = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^\mathsf{T}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y} - 2\mathbf{y}^\mathsf{T}\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta}, \tag{3.52}$$

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

$$\frac{\partial}{\partial \boldsymbol{\theta}} V(\boldsymbol{\theta}) = -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta}. \tag{3.53}$$

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

$$\frac{\partial}{\partial \boldsymbol{\theta}} V(\widehat{\boldsymbol{\theta}}) = 0 \Leftrightarrow -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta} = 0 \Leftrightarrow \mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{3.54}$$

i.e., the normal equations.

A linear algebra approach

Denote the $p+1$ columns of $\mathbf{X}$ as $c_j, j = 1, \ldots, p+1$. We first show that $\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$ is minimized if $\boldsymbol{\theta}$ is chosen such that $\mathbf{X}\boldsymbol{\theta}$ 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 $\mathbf{X}\boldsymbol{\theta}$, it follows that

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\theta} - (\mathbf{y}_\perp + \mathbf{y}_\parallel)\|_2^2 = \|(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\parallel) - \mathbf{y}_\perp\|_2^2 \geq \|\mathbf{y}_\perp\|_2^2, \tag{3.55}$$

and the triangle inequality also gives us

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\perp - \mathbf{y}_\parallel\|_2^2 \leq \|\mathbf{y}_\perp\|_2^2 + \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\parallel\|_2^2. \tag{3.56}$$

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

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\theta}})^\mathsf{T} c_j = 0, j = 1, \ldots, p+1 \tag{3.57}$$

(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

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\theta}})^\mathsf{T}\mathbf{X} = 0, \tag{3.58}$$

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

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

i.e., the normal equations.

4 Understanding, evaluating and improving the performance

We have so far encountered four different methods for supervised machine learning, and more are to come in later chapters. We always learn the models by adapting them to training data, and 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 sound like a trivial question, but on a second thought it is perhaps not (so) obvious, and we will give it some attention in this chapter before we dive into even more complicated models in later chapters. By doing so, we will unveil some interesting concepts, and discover some practical tools for evaluating, choosing between and improving supervised machine learning methods.

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

We start by introducing some concepts and notation. First, we define an error function $E(\widehat{y}, y)$ which encodes the purpose of classification or regression. The error function compares a prediction $\widehat{y}(\mathbf{x})$ to a measured data point y, and returns a small value (possibly zero) if $\widehat{y}(\mathbf{x})$ is a good prediction of y, and a larger value otherwise. One could consider many different error functions, but our default choices are misclassification and squared error, respectively:

$$\text{Misclassification:} \qquad E(\widehat{y}, y) \triangleq \begin{cases} 0 & \text{if } \widehat{y} = y \\ 1 & \text{if } \widehat{y} \neq y \end{cases} \qquad \text{(classification)} \tag{4.1a}$$

$$\text{Squared error:} \qquad E(\widehat{y}, y) \triangleq (\widehat{y} - y)^2 \qquad \text{(regression)} \tag{4.1b}$$

When we compute the average misclassification (4.1a), we usually refer to it as the *misclassification rate*. The misclassification rate is often a natural quantity to consider in classification, but for imbalanced or asymmetric problems other aspects might be more important, as we discuss in Section 4.5.

The error function $E(\widehat{y}, y)$ has similarities to a loss function $L(\widehat{y}, y)$. However, they are used differently: A loss function is used to *train* a model, whereas we use the error function to *analyze performance* of an already trained model. There are also reasons for choosing $E(\widehat{y}, y)$ and $L(\widehat{y}, y)$ differently, which we will come back to soon.

In the end, supervised machine learning cares about designing a method which performs well 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 incoming 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 the regression method predicts. To be able to mathematically describe the endless stream of new data, we introduce a *distribution over data* $p(\mathbf{x}, y)$. Most of the time, we only consider the output y as a random variable whereas the inputs $\mathbf{x}$ are considered fixed. In this chapter, however, 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 write down. We will nevertheless use $p(\mathbf{x}, y)$ to *reason* about supervised machine learning methods, and the bare notion of $p(\mathbf{x}, y)$ (even though it is unknown in practice) will be helpful for that.

No matter which specific classification or regression method we consider, once it has been trained on training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, it will return predictions $\widehat{y}(\mathbf{x}_\star)$ for any new input $\mathbf{x}_\star$ we give to it. We will in this chapter write $\widehat{y}(\mathbf{x}; \mathcal{T})$ to emphasize that the training data $\mathcal{T}$ was used to train the model. Indeed, different training datasets will train the model differently and, consequently, give different predictions.

In the other chapters we mostly discuss how a model predicts one, or a few, test inputs $\mathbf{x}_\star$. Let us take that to the next level by integrating (averaging) the error function (4.1) over *all* possible test data points with respect to the distribution $p(\mathbf{x}, y)$. We refer to this as the *expected new data error*

$$\boxed{E_{\text{new}} \triangleq \mathbb{E}_\star \left[E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) \right],} \tag{4.2}$$

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,

$$\mathbb{E}_\star \left[E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) \right] = \int E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) p(\mathbf{x}_\star, y_\star) \, d\mathbf{x}_\star dy_\star. \tag{4.3}$$

Remember that the model (regardless if it is linear regression, a classification tree, an ensemble of trees, a neural network or something else) is trained on a given training dataset $\mathcal{T}$ and represented by $\widehat{y}(\cdot; \mathcal{T})$. What is happening in equation (4.2) 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.

We also introduce the *training error*

$$\boxed{E_{\text{train}} \triangleq \frac{1}{n} \sum_{i=1}^{n} E(\widehat{y}(\mathbf{x}_i; \mathcal{T}), y_i),} \tag{4.4}$$

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, but gives no information on how well the method will perform for new unseen data points.

Time to reflect 4.1: *What is E_{train} for k-NN with $k = 1$?*

Whereas the training error E_{train} describes how well the method is able to "reproduce" the data from which it was learned, 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.

This actually sheds some additional light upon the comment we made previously, that the loss function $L(\widehat{y}, y)$ end the error function $E(\widehat{y}, y)$ do not have to be the same. As we will discuss thoroughly in this chapter, a model which fits the training data well and consequently has a small E_{train} might still have a high E_{new} when faced with new unseen data. In order to minimize E_{new}, the best strategy is therefore not necessarily to minimize E_{train}.[1] Besides the fact that the misclassification (4.1a) is unsuited as optimization objective (it is discontinuous and has derivative zero almost everywhere) it can also, depending on the method, be argued that E_{new} can be made smaller by a more clever choice of loss function. Such examples include gradient boosting and support vector machines. (Of course, this only applies to methods that are trained using a loss function. k-NN, for example, is not.)

In practical cases we can, unfortunately, 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 analyze how E_{new} behaves, to better understand how we can decrease it.

[1] The term "risk function" is used in some literature both for loss and error functions, assuming they are chosen equally. In that terminology, E_{new} is referred to as "expected risk", E_{train} as "empirical risk", and the idea of minimizing the cost function as "empirical risk minimization".

Remark 4.1 *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 logistic regression" in general, but instead we have to make more specific statements, like "E_{new} for the handwritten digit recognition problem, with a logistic regression classifier trained with the MNIST data[2]".*

4.2 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 will lead us to a very useful concept known as cross-validation.

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

We have both introduced the expected new data error, E_{new}, and the training error E_{train}. In contrast to E_{new}, we can always compute E_{train}.

We assume for now 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.

When an integral is hard to compute, it can be numerically approximated with a sum. Now, the question is if the integral in E_{new} can be well approximated by the sum in E_{train}, like

$$E_{\text{new}} = \int E(\widehat{y}(\mathbf{x}; \mathcal{T}), y))p(\mathbf{x}, y)d\mathbf{x}dy \overset{??}{\approx} \frac{1}{n}\sum_{i=1}^{n} E(\widehat{y}(\mathbf{x}_i; \mathcal{T}), y_i) = E_{\text{train}}. \tag{4.5}$$

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 4.2:*** *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}), even though the training data is drawn from the distribution $p(\mathbf{x}, y)$?*

Equation (4.5) does *not* hold, and the reason is that the training data are not just any data points, but the predictions $\widehat{y}$ depends on them since they are used for training the model. We can therefore not expect (4.5) to hold. (Technically, the conditions for approximating an integral with a sum are not fulfilled since $\widehat{y}$ depends on $\mathcal{T}$.)

As we will discuss more thoroughly later, 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}.*

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

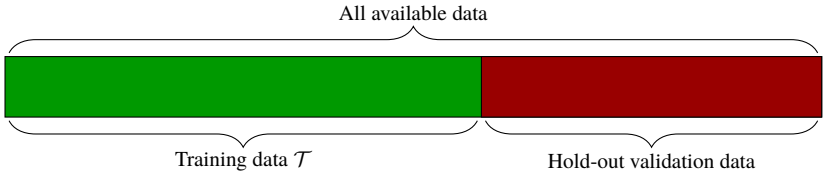


Figure 4.1: The hold-out validation dataset approach: If we split the available data in two sets and train the model on the training set, we can compute $E_{\text{hold-out}}$ using the hold-out validation set. The more data that are in the hold-out validation dataset, the less variance (better estimate) in $E_{\text{hold-out}}$, but the less data left for training the model. The split here is only pictorial, in practice one should always split the data randomly.

$E_{\text{hold-out}} \approx E_{\text{new}}$: We can estimate E_{new} from hold-out validation 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 (4.4)) and second, to evaluate the error function (the sum in (4.4)). A remedy is to set aside some *hold-out validation data* $\{\mathbf{x}_j, y_j\}_{j=1}^{n_v}$, which are not in $\mathcal{T}$ used for training, and then use the hold-out validation data only for estimating the model performance as the *hold-out validation error*

$$\boxed{E_{\text{hold-out}} \triangleq \frac{1}{j}\sum_{j=1}^{n_v} E(\widehat{y}(\mathbf{x}_j; \mathcal{T}), y_j)}. \tag{4.6}$$

In this way, not all data will be used for training, but some data points (the hold-out validation data) will be saved and used only for computing $E_{\text{hold-out}}$. This procedure is a simple version of *cross-validation*, and is illustrated by Figure 4.1.

Be aware! *If you are splitting your data, always do it randomly! Someone might—intentionally or unintentionally—have sorted the dataset 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 hold-out validation data . . .*

With the conditions $n_v \geq 1$ and that all data is drawn from $p(\mathbf{x}, y)$, it can be shown that $E_{\text{hold-out}}$ is an unbiased estimate of E_{new} (meaning that if the entire procedure is repeated multiple times, the average value of $E_{\text{hold-out}}$ would be E_{new}). That is reassuring, but it does not tell us how close $E_{\text{hold-out}}$ will be to E_{new} in a single experiment. However, the variance of $E_{\text{hold-out}}$ decreases when the size of hold-out validation data n_v increases; a small variance of $E_{\text{hold-out}}$ means that we can expect it to be close to E_{new}. Thus, if we take the hold-out validation dataset big enough, $E_{\text{hold-out}}$ will be close to E_{new}. However, setting aside a big validation dataset means that the training dataset is smaller. Typically the more training data, the smaller E_{new} (which we will discuss later in Section 4.3), and achieving a small E_{new} is our ultimate goal.

Sometimes, the number of available data points counts in hundreds of thousands, millions, or even more. When we have a lot of data, we can afford to set aside a few percent data into a reasonably large hold-out validation dataset, without sacrificing the size of the training dataset too much. *In such data-rich situations, the hold-out validation data approach is sufficient.*

If the amount of available data is more limited, this becomes more of a problem. We are in practice faced with the following dilemma: *the better we want to know* E_{new} (more hold-out validation data gives less variance in $E_{\text{hold-out}}$), *the worse we have to make it* (less training data increases E_{new}). That is not very satisfying, and we have to look for an alternative to the hold-out validation data approach.

k-fold cross-validation: $E_{k\text{-fold}} \approx E_{\text{new}}$ without setting aside validation data

To avoid setting aside validation data, but still obtain an estimate of E_{new}, one could suggest a two-step procedure of

(i) splitting the available data in one training and one hold-out validation set, train the model on the training data and compute $E_{\text{hold-out}}$ using hold-out validation data (as in Figure 4.1), and then

(ii) training the model again, this time using the entire dataset.

By such a procedure, we both get an estimate of E_{new} and a model trained on the entire dataset. That is not bad, but not perfect either. Why? To achieve small variance in the estimate, we have to put lots of data in the hold-out validation dataset. That means the model trained in (i) will possibly be very different from step (ii), and the estimate of E_{new} concerns the model from step (i), not the possibly very different model from step (ii). Hence, this will not give us a good estimate of E_{new}, but it will still lead us the the useful k-fold cross-validation idea.

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. By *k-fold cross-validation*, we can achieve (almost) this. The idea of k-fold cross-validation is simply to repeat the hold-out validation dataset approach multiple times with a *different* hold-out dataset each time, in the following way:

(i) split the dataset in k batches of similar size (see Figure 4.2), and let $\ell = 1$

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

(iii) train the model on the training data, and compute $E^{(\ell)}_{\text{hold-out}}$ as the average error on the hold-out validation data, (4.6)

(iv) if $\ell < k$, set $\ell \leftarrow \ell + 1$ and return to (ii). If $\ell = k$, compute the *k-fold cross-validation error*

$$\boxed{E_{k\text{-fold}} \triangleq \frac{1}{k}\sum_{\ell=1}^{k} E^{(\ell)}_{\text{hold-out}}} \tag{4.7}$$

(v) train the model again, this time using the entire dataset

This illustrated in Figure 4.2.

With k-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 $E_{k\text{-fold}}$. Whereas $E_{\text{hold-out}}$ (Section 4.2) was an unbiased estimate of E_{new} (to the cost of setting aside hold-out validation data), $E_{k\text{-fold}}$ is only approximately unbiased. However, with k large enough, it turns out to often be a sufficiently good approximation. Let us try to understand why k-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 $1/k$ fraction of the data in step (iii). The key in k-fold cross-validation is that if k is large enough, the intermediate models are quite similar to the final model (since they are trained on almost the same dataset, only a fraction $1/k$ of the data is missing). Furthermore, each intermediate $E^{(\ell)}_{\text{hold-out}}$ is an unbiased but high-variance estimate of E_{new} for the corresponding intermediate model ℓ. Since all intermediate and the final model are similar, $E_{k\text{-fold}}$ (4.7) is approximately the average of k high-variance estimates of E_{new} for the final model. When averaging estimates, the variance decreases and $E_{k\text{-fold}}$ will have a lower variance.

Be aware! *For the same reason as with the hold-out validation 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 dataset, and thereafter split it into batches.*

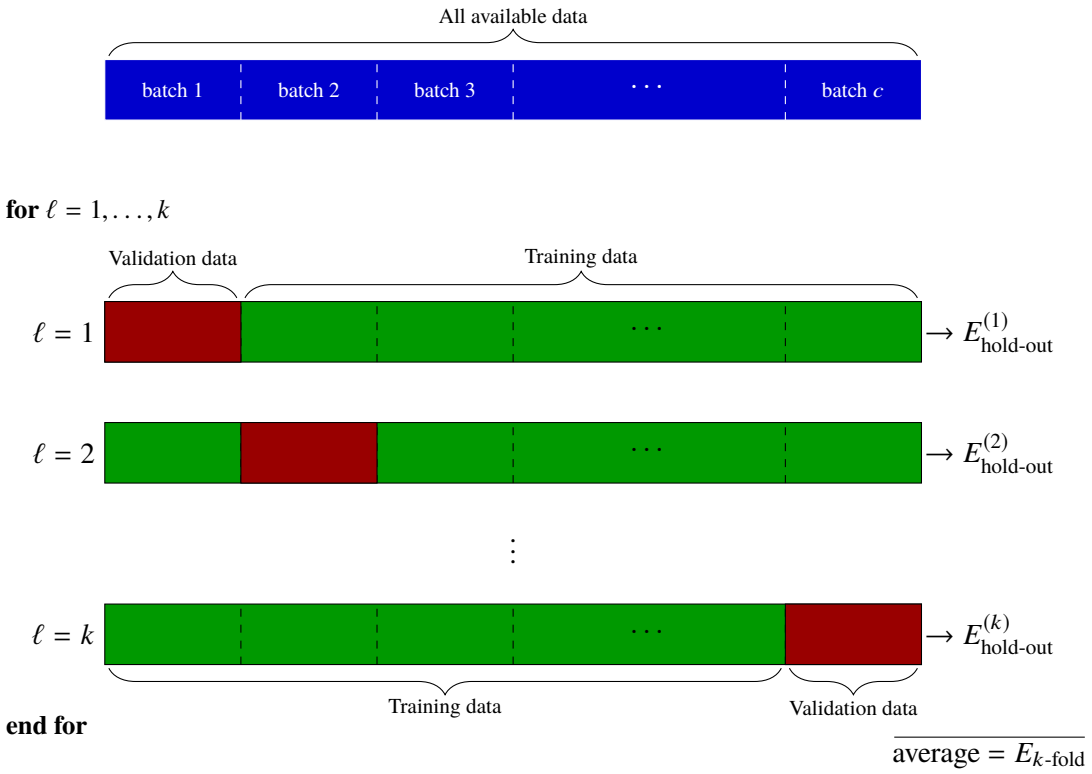


Figure 4.2: Illustration of k-fold cross-validation. The data is split in k batches of similar sizes. When looping over $\ell = 1, 2, \ldots, k$, batch ℓ is held out as validation data, and the model is trained on the remaining $k - 1$ data batches. Each time, the trained model is used to compute the average error $E^{(\ell)}_{k\text{-fold}}$ for the validation data. The final model is trained using all available data, and the estimate of E_{new} for that model is $E_{k\text{-fold}}$, the average of all $E^{(\ell)}_{k\text{-fold}}$.

We usually talk about training (or learning) as a procedure that is executed once. However, in k-fold cross-validation the training is repeated k (or even $k + 1$) times. A special case is $k = n$ which is also called *leave-one-out cross-validation*. However, a common value for k in practice 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 k times is usually not really a problem. When 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 $k + 1$. This aspect and how much data is available determines whether the hold-out or k-fold cross-validation should be used.

Using a test dataset

A very important use of $E_{k\text{-fold}}$ (or $E_{\text{hold-out}}$) in practice is to choose between methods and select different types of hyperparameters such that $E_{k\text{-fold}}$ (or $E_{\text{hold-out}}$) becomes as small as possible. Typical hyperparameters to choose in this way are k in k-NN, tree depths or regularization parameters. However, much like we cannot use the training data error E_{train} to estimate the new data error E_{new}, selecting models and hyperparameters based on $E_{k\text{-fold}}$ (or $E_{\text{hold-out}}$) will invalidate its use as an estimator of E_{new}. If it is important to have a good estimate of the final E_{new}, it is wise to first set aside another hold-out dataset, which we refer to as a *test set*. This test set should be used only once (after selecting models and hyperparameters) to estimate E_{new} for the final model.

In problems where the training data is expensive, it is common to increase the training dataset using more or less artificial techniques. Such techniques can be to duplicate the data and add noise to the duplicated versions, to use simulated data, or to use data from a different but related problem. With such techniques (which indeed can be very successful), the training data $\mathcal{T}$ is no longer drawn from $p(\mathbf{x}, y)$.

In the worst case (if, for example, the simulations are very poor), $\mathcal{T}$ might not provide any information about $p(\mathbf{x}, y)$, and we can not really expect the model to learn anything useful. It can therefore be very useful to have a good estimate of E_{new} if such techniques were used during training, but a reliable estimate of E_{new} can only be achieved from data that we *know* are drawn from $p(\mathbf{x}, y)$ (that is, collected under production-like circumstances). If the training data is extended artificially, it is therefore extra important to set aside a test data set *before* that extension is done.

Remark 4.2 *The error on the test data could be called "test error". To avoid confusion, we do not use that term since it in other places is used ambiguously for both the error on a test dataset as well as E_{new}.*

4.3 The training error–generalization gap decomposition of E_{new}

Designing a method with small E_{new} is the goal in supervised machine learning, and some form of cross-validation is important for *estimating* E_{new}. However, more can be said to also *understand* 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},

$$\bar{E}_{\text{new}} \triangleq \mathbb{E}_{\mathcal{T}}\left[E_{\text{new}}\right], \tag{4.8a}$$

$$\bar{E}_{\text{train}} \triangleq \mathbb{E}_{\mathcal{T}}\left[E_{\text{train}}\right]. \tag{4.8b}$$

Here, $\mathbb{E}_{\mathcal{T}}$ denotes the expected value when the training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ (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 datasets*, 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 $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$, than we can say about E_{new} and E_{train} when the model is trained on one specific training dataset $\mathcal{T}$. Even though we most often care about E_{new} in the end (the training data is usually fixed), insights from studying $\bar{E}_{\text{new}}$ are still useful. In fact, k-fold cross-validation estimates in practice $\bar{E}_{\text{new}}$ rather than E_{new}.

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

$$\bar{E}_{\text{train}} < \bar{E}_{\text{new}}, \tag{4.9}$$

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. We consequently call the difference between $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$ the *generalization gap*[3], as

$$\text{generalization gap} \triangleq \bar{E}_{\text{new}} - \bar{E}_{\text{train}}. \tag{4.10}$$

The generalization gap is the difference between performance on training data and the performance 'in production' on new, previously unseen data. Since we can always compute E_{train} and cross-validation gives an estimate of E_{new}, we can also estimate the generalization gap in practice.

With the decomposition of $\bar{E}_{\text{new}}$ into

$$\bar{E}_{\text{new}} = \bar{E}_{\text{train}} + \text{generalization gap}, \tag{4.11}$$

we also have an opening for digging deeper and trying to understand what affects $\bar{E}_{\text{new}}$ in practice. We will refer to (4.11) as the *training error–generalization gap decomposition*.

[3] We use a loose terminology and refer also to $E_{\text{new}} - E_{\text{train}}$ as the generalization gap.

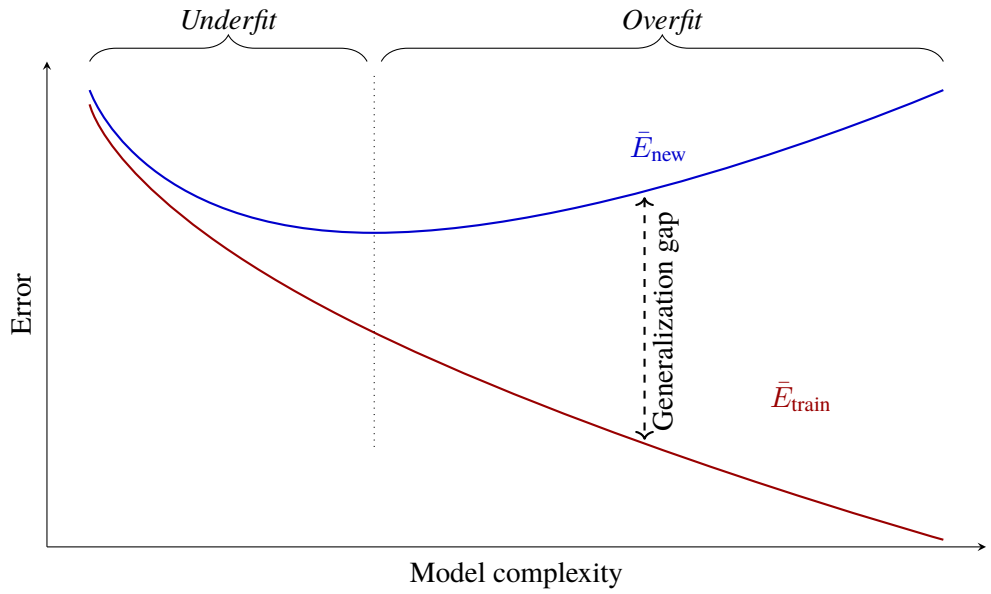


Figure 4.3: Behavior of $\bar{E}_{\text{train}}$ and $\bar{E}_{\text{new}}$ for many supervised machine learning methods, as a function of model complexity. We have not made a formal definition of complexity, but a rough proxy is the number of parameters that are learned from the data. The difference between the two curves is the generalization gap. The training error $\bar{E}_{\text{train}}$ decreases as the model complexity increases, whereas the new data error $\bar{E}_{\text{new}}$ 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 commonly 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) could be called a balanced fit. When we, for example, use cross-validation to select hyperparameters (that is, tuning the model complexity), we are searching for a balanced fit.

Generalization gap and model complexity

The generalization gap depends on the method and the problem. Concerning the method, one can typically say that *the more a method adapts to training data, the larger the generalization gap*. A theoretical framework for how much method a method adapts to training data is given by the so-called VC dimension. From the VC dimension framework, probabilistic bounds on the generalization gap can be derived, but those bounds are unfortunately rather conservative, and we will not pursue that approach any further. Instead, we only use the vague terms *model complexity* or *model flexibility* (we use them as synonyms), by which we mean the ability of a method to adopt to patterns in the training data. A model with high complexity/flexibility (such as a fully connected deep neural network, deep trees and k-NN with small k) can describe almost arbitrarily complicated relationships, whereas a model with low complexity/flexibility (such as logistic regression) is less flexible in what functions it can describe. For parametric methods, the model complexity is somewhat related to the number of parameters that are trained, but is also affected by regularization techniques.

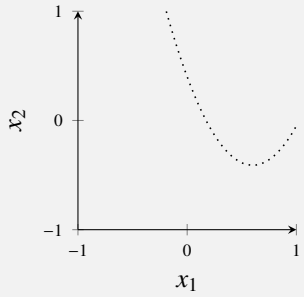
Typically, *higher model complexity implies larger generalization gap*. Furthermore, $\bar{E}_{\text{train}}$ decreases as the model complexity increases, whereas $\bar{E}_{\text{new}}$ typically attains a minimum for some intermediate model complexity value: too small *and* too high model complexity both raises $\bar{E}_{\text{new}}$. This is illustrated in Figure 4.3. A too high model complexity, meaning that $\bar{E}_{\text{new}}$ is higher than it had been with a less complex model, is called *overfit*. The other situation, when the model complexity is too low, is sometimes called *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 sweet spot. We also illustrate this by Example 4.1.

Remark 4.3 *We discuss the usual behavior of $\bar{E}_{new}$, $\bar{E}_{train}$ and the generalization gap. We use the term 'usual' 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, and pathological*

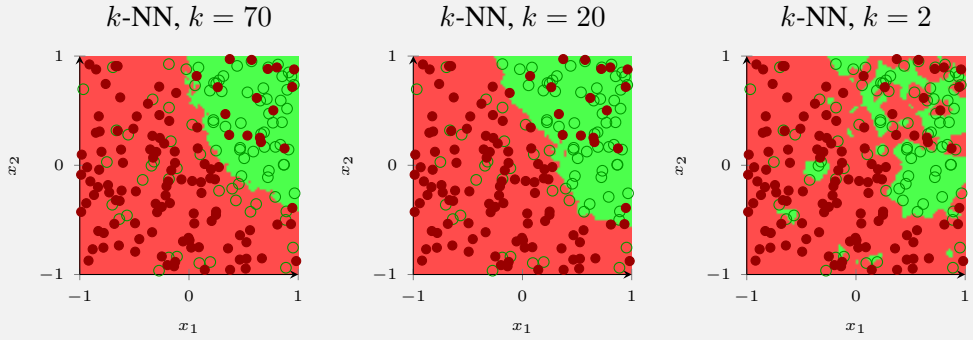
counter-examples may exist. One should also keep in mind that claims about $\bar{E}_{train}$ and $\bar{E}_{new}$ are about the average *behavior, which hopefully is clear in Example 4.1.*

Example 4.1: The training error–generalization gap decomposition for k-NN

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 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 0.8, and points below the curve are red with probability 0.8. (The optimal classifier, in terms of minimal E_{new}, would have the dotted line as its decision boundary and achieve $E_{\text{new}} = 0.2$.)



We have $n = 200$ in the training data, and learn three classifiers: k-NN with $k = 70$, $k = 20$ and $k = 2$, respectively. In model complexity sense, $k = 70$ gives the least flexible model, and $k = 2$ the most flexible model. We plot their decision boundaries, together with the training data:

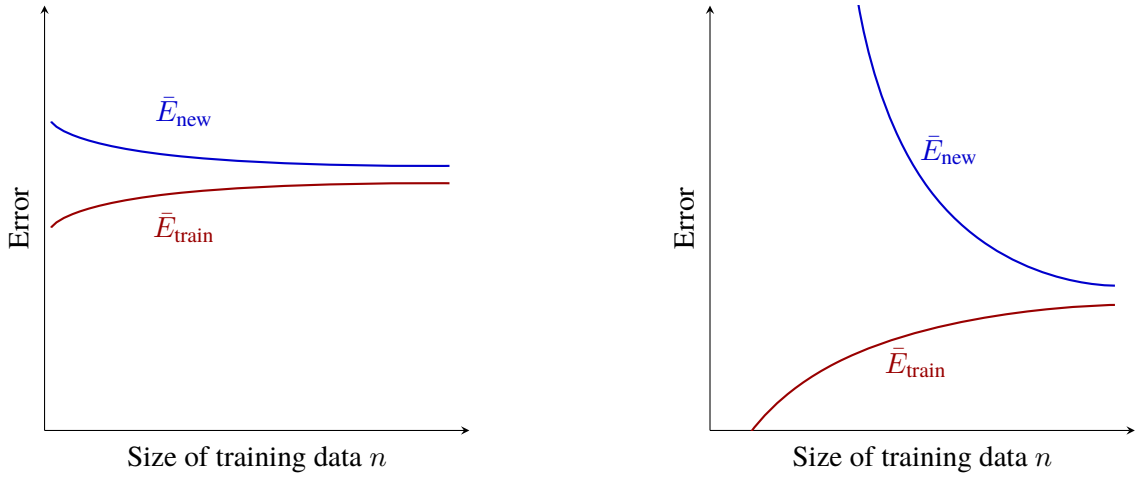


Intuitively we see that $k = 2$ (right) adapts too well to the data. With $k = 70$, on the other hand, the model is rigid enough not to adapt to the noise, but appears to possibly be too inflexible to adapt well to the true dotted line above.

We can compute E_{train} by counting the fraction of misclassified training data points in the figures above. From left to right, we get $E_{\text{train}} = 0.27, 0.24, 0.22$. Since this is 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 $E_{\text{new}} = 0.26, 0.23, 0.33$. This pattern resembles Figure 4.3, except for the fact that E_{new} is acutally smaller than E_{train} for some values of j. This is, however, not unexpected. 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. To study $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$, we therefore repeat this experiment 100 times, and compute the average over those 100 experiments:

	k-NN with $k = 70$	k-NN with $k = 20$	k-NN with $k = 2$
$\bar{E}_{\text{train}}$	0.24	0.22	0.17
$\bar{E}_{\text{new}}$	0.25	0.23	0.30

This table follows Figure 4.3 well: The generalization gap (difference between $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$) is positive and increases with model complexity (decreasing k in k-NN), whereas $\bar{E}_{\text{train}}$ decreases with model complexity. Among these values for k, $\bar{E}_{\text{new}}$ has its minimum for $k = 20$. This suggests that k-NN with $k = 2$ suffers from overfitting for this problem, whereas $k = 70$ is a case of underfitting.



(a) Simple model

(b) Complex model

Figure 4.4: Typical relationship between $\bar{E}_{\text{new}}$, $\bar{E}_{\text{train}}$ and the number of data points n in the training dataset for a simple model (low model flexibility, left) and a complex model (high model flexibility, right). The generalization gap (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 gap is typically larger for a more complex model, in particular when the training dataset is small.

Generalization gap and size n of training data

The previous section and Figure 4.3 are concerned about the relationship between the generalization gap and the model complexity. Another very important aspect is the size of the training dataset, n. We do not make a formal derivation, but we can in general expect that *the more training data, the smaller the generalization gap*. 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 4.4.

Reducing E_{new} in practice

Our overall goal is to achieve a small error "in production", that is, small E_{new}. To achieve that, according to the decomposition $E_{\text{new}} = E_{\text{train}} + \text{generalization gap}$, we need to have E_{train} as well as the generalization gap small. Let us first draw two practically useful conclusions from this.

- The new data error E_{new} will (most often) not be smaller than the training error E_{train}. Thus, if E_{train} is much bigger than the E_{new} you need for your application to be successful, you do not even need to waste time on implementing cross-validation for estimating E_{new}. Instead, you should re-think the problem and which method you are using.

- The generalization gap and E_{new} decreases as good as always as n increases. Thus, increasing the size of the training data may help a lot, and will at least not hurt.

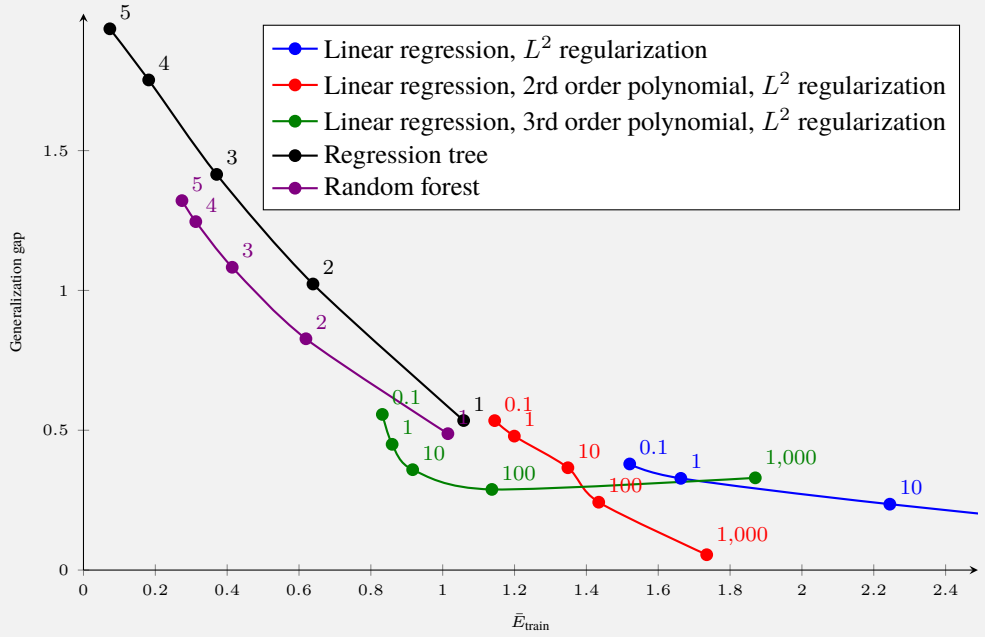
It is also important to realize that the one-dimensional model complexity scale in Figure 4.3 does not make justice for the space of all supervised machine learning methods. For a given problem, one method can have a smaller generalization gap than another method *without* having a larger training error. Some methods are simply better for certain problems, as we illustrate in Example 4.2 below. It is therefore important to use your knowledge about the problem and how different methods are designed to make a good choice. It is good to choose models that are known to work well for a specific type of data (such as using convolutional neural networks for images, Chapter 6) as well as using experience from similar problems.

Example 4.2: Training error and generalization gap for a regression problem

To be able to explore how the approximation and generalization can behave, we to consider a simulated problem so that we can compute E_{new}. We let $n = 10$ data points be generated as $x \sim \mathcal{U}[-5, 10]$, $y \sim \min(0.1x^2, 3) + \varepsilon$, and $\varepsilon \sim \mathcal{N}(0, 1)$, and consider the following regression methods:

- Linear regression with L^2 regularization
- Linear regression with a quadratic polynomail and L^2 regularization
- Linear regression with a third order polynomial and L^2 regularization
- Regression tree
- A random forest (Chapter 7) with 10 regression trees

For each of these methods, we try a few different values of the hyperparameters, compute $\bar{E}_{\text{train}}$ and the generalization gap. The hyperparameter for linear regression is the regularization parameter (larger number means heavier regularization), and the maximum tree depth for the trees and random forests.



The hyperparameter that minimizes E_{new} is, for each method, the value which is closest, in the 1-norm sense, to the origin. Having decided a certain model, and only having the hyperparameter left to choose, corresponds well to the situation in Figure 4.3. But when comparing the *different* models, however, a more complicated situation is revealed. Compare, for example, the second (green) to the third order polynomial (red): for some values of the regularization parameter, the training error decreases *without* increasing the generalization gap. Similarly is the generalization gap smaller, while the training error remains the same, for the random forest (black) than for the tree (purple) for a maximum tree depth of 2. These type of relationships are quite intricate, problem-dependent and hard to anticipate. The main message is that the one-dimensional model complexity scale in Figure 4.3 gives a simplified picture, and that a good choice of model can possibly decrease the generalization gap *and* the training error simultaneously.

(Remember, for real problems we cannot make such plots. This is only possible for such simulated problems. In practice we have to use cross-validation and previous experience for selecting between different models and choosing hyperparameters.)

Once the choice of method is done, there are usually still a few design choices to make. The design choices can, for instance, include some order selection, regularization hyperparameters, and hyperparameters for numerical optimization (such as learning rate). In the end, many such choices boils down to moving along the model complexity axis of Figure 4.3. Making the model more flexible decreases E_{train} but increases, often, the generalization gap. Making the model less flexible, on the other hand, typically decreases the

generalization gap but increases E_{train}.

The optimal tradeoff, in terms of small E_{new}, is for many models achieved when neither the generalization gap nor the training error E_{train} is zero. Thus, by monitoring E_{train} and estimating E_{new} with, say, $E_{\text{hold-out}}$, we get the following advice:

- If $E_{\text{hold-out}} \approx E_{\text{train}}$ (small generalization gap), it might be beneficial to increase the model flexibility by loosening the regularization, increasing the model order (more parameters to learn), etc.
- If E_{train} is close to zero, it might be beneficial to decrease the model flexibility by tightening the regularization, decreasing the order (fewer parameters to learn), etc.

4.4 The bias-variance decomposition of E_{new}

We will now introduce another decomposition of $\bar{E}_{\text{new}}$ into a (squared) *bias* and a *variance* term, as well as an unavoidable component of irreducible noise. This decomposition is somewhat more abstract than the training-generalization gap, but provides some additional insights into E_{new} and how different models behave.

Let us first make a short reminder of the general concepts of bias and variance. Consider an experiment with an unknown constant z_0, which we would like to estimate. To our help for estimating z_0 we have a random variable z. Think, for example, of z_0 as being the (true) position of an object, and z of being noisy GPS measurements of that position. Since z is a random variable, it has some mean $\mathbb{E}[z]$ which we denote by $\bar{z}$. We now define

$$\text{Bias: } \bar{z} - z_0 \tag{4.12a}$$

$$\text{Variance: } \mathbb{E}\big[(z-\bar{z})^2\big] = \mathbb{E}\big[z^2\big] - \bar{z}^2. \tag{4.12b}$$

The *variance* describes how much the experiment varies each time we perform it (the noise in the GPS measurements), whereas the *bias* describes the systematic error in z that remains no matter how many times we repeat the experiment (a possible shift or offset in the GPS measurements). If we consider the expected squared error between z and z_0 as a metric of how good the estimator z is, we can re-write it in terms of the variance and the squared bias,

$$\begin{aligned}\mathbb{E}\big[(z-z_0)^2\big] &= \mathbb{E}\Big[\big((z-\bar{z})+(\bar{z}-z_0)\big)^2\Big] = \\ &= \underbrace{\mathbb{E}\big[(z-\bar{z})^2\big]}_{\text{Variance}} + 2\underbrace{(\mathbb{E}[z]-\bar{z})}_{0}(\bar{z}-z_0) + \underbrace{(\bar{z}-z_0)^2}_{\text{bias}^2}.\end{aligned} \tag{4.13}$$

In words, the average squared error between z and z_0 is the sum of the squared bias and the variance. The main point here is that to obtain a small expected squared error, we have to consider the bias *and* the variance. Only a small bias *or* little variance in the estimator is not enough, but both aspects are important.

We will now apply the bias and variance concepts to our supervised machine learning setting, and in particular to the regression problem. The intuition, however, carries over also to the classification problem[4]. In this setting, z_0 corresponds to the true relationship between inputs and output, and the random variable z corresponds to the model learned from training data. (Since the training data collection includes randomness, the model learned from it will also be random.) Let us spell out the details:

We first make the assumption that the true relationship between input $\mathbf{x}$ and output y can be described as some (possibly very complicated) function $f_0(\mathbf{x})$ plus independent noise ε,

$$y = f_0(\mathbf{x}) + \varepsilon, \quad \text{with } \mathbb{E}[\varepsilon] = 0 \text{ and } \operatorname{var}(\varepsilon) = \sigma^2. \tag{4.14}$$

[4]For intuition, we may think of classification problems as regression in terms of the decision boundaries.

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

$$\bar{f}(\mathbf{x}) \triangleq \mathbb{E}_{\mathcal{T}}\left[\widehat{y}(\mathbf{x}; \mathcal{T})\right]. \tag{4.15}$$

As before, $\mathbb{E}_{\mathcal{T}}$ denotes the expected value over training data drawn from $p(\mathbf{x}, y)$. Thus, $\bar{f}(\mathbf{x})$ is the (hypothetical) average model we would achieve, if we could re-train the model an infinite number of times on different training datasets and compute the average.

Our definition of $\bar{E}_{\text{new}}$ is

$$\bar{E}_{\text{new}} = \mathbb{E}_{\mathcal{T}}\left[\mathbb{E}_{\star}\left[\left(\widehat{y}(\mathbf{x}_{\star}; \mathcal{T}) - y_{\star}\right)^2\right]\right], \tag{4.16}$$

and assuming these integrals (expressed as expected values) fulfill some technical assumptions, we can change the order of integration and write (4.16) as

$$\bar{E}_{\text{new}} = \mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}\left[\left(\widehat{y}(\mathbf{x}_{\star}; \mathcal{T}) - f_0(\mathbf{x}_{\star}) - \varepsilon\right)^2\right]\right] \tag{4.17}$$

With a slight extension of (4.13) to also include the zero-mean noise term ε (which is independent of $\widehat{y}(\mathbf{x}_{\star}; \mathcal{T})$), we can rewrite the expression inside the expected value $\mathbb{E}_{\star}$ in (4.17) as

$$\mathbb{E}_{\mathcal{T}}\Big[\big(\underbrace{\widehat{y}(\mathbf{x}_{\star}; \mathcal{T})}_{z} - \underbrace{f_0(\mathbf{x}_{\star})}_{z_0} - \varepsilon\big)^2\Big] = \big(\bar{f}(\mathbf{x}_{\star}) - f_0(\mathbf{x}_{\star})\big)^2 + \mathbb{E}_{\mathcal{T}}\left[\big(\widehat{y}(\mathbf{x}_{\star}; \mathcal{T}) - \bar{f}(\mathbf{x}_{\star})\big)^2\right] + \varepsilon^2. \tag{4.18}$$

This is (4.13) applied to supervised machine learning. In $\bar{E}_{\text{new}}$, which we are interested in decomposing, we also have the expectation over new data points $\mathbb{E}_{\star}$. By incorporating also that expected value in the expression, we can decompose $\bar{E}_{\text{new}}$ as

$$\bar{E}_{\text{new}} = \underbrace{\mathbb{E}_{\star}\left[\big(\bar{f}(\mathbf{x}_{\star}) - f_0(\mathbf{x}_{\star})\big)^2\right]}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}\left[\big(\widehat{y}(\mathbf{x}_{\star}; \mathcal{T}) - \bar{f}(\mathbf{x}_{\star})\big)^2\right]\right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\substack{\text{Irreducible} \\ \text{error}}}. \tag{4.19}$$

The squared bias term $\mathbb{E}_{\star}\left[\big(\bar{f}(\mathbf{x}_{\star}) - f_0(\mathbf{x}_{\star})\big)^2\right]$ now describes how much the average trained model $\bar{f}(\mathbf{x}_{\star})$ differs from the true $f_0(\mathbf{x}_{\star})$, averaged over all possible data points $\mathbf{x}_{\star}$. In a similar fashion the variance term $\mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}[\big(\widehat{y}(\mathbf{x}_{\star}; \mathcal{T}) - \bar{f}(\mathbf{x}_{\star})\big)^2]\right]$ describes how much $\widehat{y}(\mathbf{x}; \mathcal{T})$ varies each time the model is trained on a different training dataset. If the variance term is small, the model is not very sensitive to exactly which data points happened to be in the training data, and vice versa. The irreducible error σ^2 is simply an effect of the assumption (4.14) —it is not possible to predict ε since it is truly random. There is thus not so much more to say about the irreducible error, but we will focus on the bias and variance terms.

Bias, variance and its relation to model complexity

We have not properly defined model complexity, but we can actually use the bias and variance concept to give it a more concrete meaning: A high model complexity means low bias and high variance, and a low model complexity means high bias and low variance, as illustrated by Figure 4.5.

This resonates well with the intuition. The more flexible a model is, the more it will adapt to the training data $\mathcal{T}$—not only to the interesting patterns, but also to the actual data points and noise that happened to be in $\mathcal{T}$. That is exactly what is described by the variance term. On the other hand, a model with low flexibility can be too rigid to capture the true relationship $\bar{f}(\mathbf{x})$ between inputs and outputs well. This effect is described by the squared bias term.

Figure 4.5 may very well be compared to Figure 4.3, which builds on the training error–generalization gap decomposition of $\bar{E}_{\text{new}}$ instead. From Figure 4.5 we can talk about the challenge of finding the right model complexity level also as the *bias-variance tradeoff*. We give an example of this in Example 4.3.

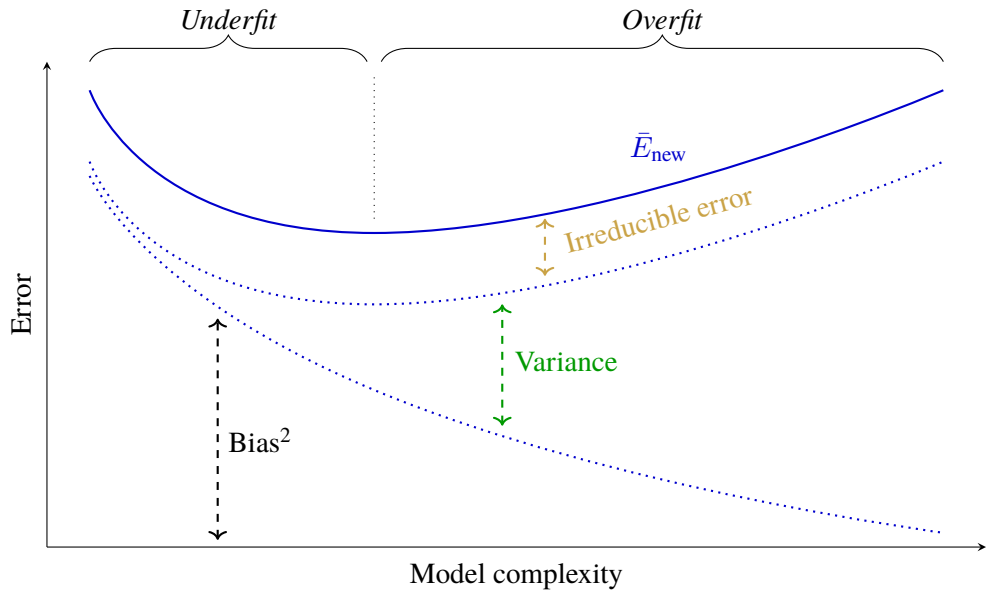


Figure 4.5: The bias-variance decomposition of $\bar{E}_{\text{new}}$ (cf. Figure 4.3). Low model complexity means high bias. The more complicated the model is, the more it adapts to training data, and the higher variance. The irreducible error is always constant. The problem of achieving a small E_{new} by selecting a good model complexity level is often called the bias-variance tradeoff.

Example 4.3: The bias–variance tradeoff for L^2 regularized linear regression

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

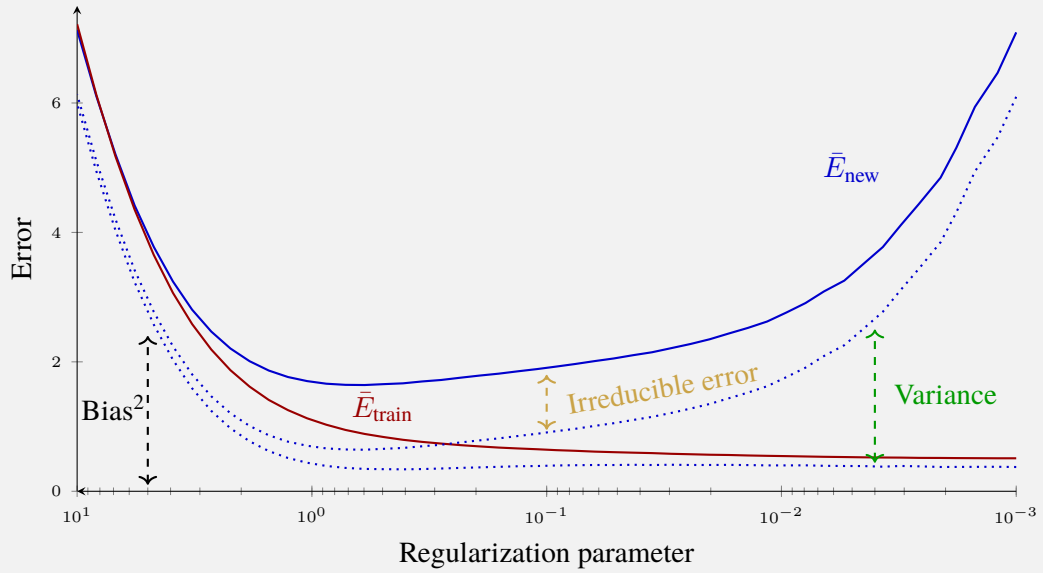
$$y = 5 - 2\mathbf{x} + \mathbf{x}^3 + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1)\,. \tag{4.20}$$

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

$$y = \beta_0 + \beta_1\mathbf{x} + \beta_2\mathbf{x}^2 + \beta_3\mathbf{x}^3 + \beta_4\mathbf{x}^4 + \varepsilon. \tag{4.21}$$

Since (4.20) is a special case of (4.21) and the squared error loss corresponds to Gaussian noise, we do actually have zero bias for this model if we train it using squared error loss. However, learning 5 parameters from only 10 data points leads to very high variance, so we decide to train the model with squared error loss and L^2 regularization, which will decrease the variance (but increase the bias). The more regularization (bigger λ), the more bias and less variance.

Since this is a simulated example, we can repeat the experiment multiple times and estimate the bias and variance terms (since we can simulate as much training and test data as needed). We plot them in the very same style as Figures 4.3 and 4.5 (note the reversed x-axis: a smaller regularization parameter corresponds to a higher model complexity). For this problem the optimal value of λ would have been about 0.7 since $\bar{E}_{\text{new}}$ attains its minimum there. Finding this optimal λ is a typical example of the bias–variance tradeoff.



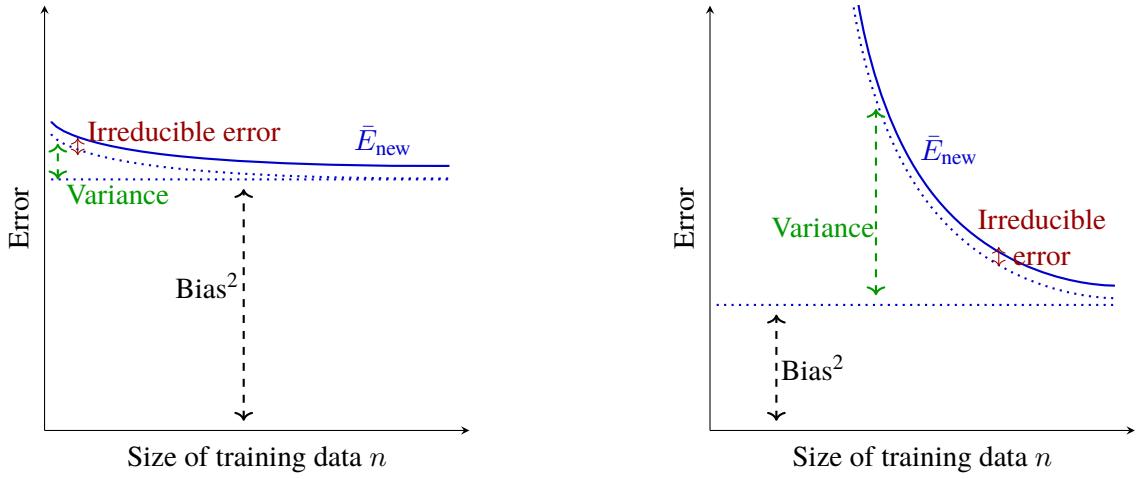
Bias, variance and its relation to the size n of training data

The squared bias term is mostly a property of the model rather than of the training dataset, and we may think[5] 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 4.6, which can be compared to Figure 4.4.

Connections between bias, variance and the generalization gap

The bias and variance are theoretically well defined properties, but often intangible in practice since they are defined in terms of $p(\mathbf{x}, y)$. In practice, we only have an estimate of the generalization gap (for example

[5] This is not exactly true. The average model $\bar{f}$ might indeed be different if all training datasets (which we average over) contain $n = 2$ or $n = 100\,000$ data points, but we neglect that effect here.



(a) Simple model

(b) Complex model

Figure 4.6: The typical relationship between bias, variance and the size n of the training dataset (cf. Figure 4.4). The bias is (approximately) constant, whereas the variance decreases as the size of the training dataset increases.

as $E_{\text{hold-out}} - E_{\text{train}}$). It is therefore interesting to explore what E_{train} and the generalization gap says about the bias and variance.

Considering the regression problem, and assuming that the error function (squared error) also is the loss function and that the global minimum is found during training, we can write

$$\sigma^2 + \text{bias}^2 = \mathbb{E}_\star\left[(\bar{f}(x_\star) - y_\star)^2\right] \approx \frac{1}{n}\sum_{i=1}^{n}(\bar{f}(x_i) - y_i)^2 \geq \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}(x_i;\mathcal{T}) - y_i)^2 = E_{\text{train}}. \tag{4.22}$$

In the approximate equality, we used the "replace the integral with a sum"-trick[6]. In the next step, equality is obtained if $\widehat{y} = \bar{f}$ (which we assume is possible), and inequality otherwise. Remembering that $\bar{E}_{\text{new}} = \sigma^2 + \text{bias}^2 + \text{variance}$, and allowing ourselves to write $\bar{E}_{\text{new}} - E_{\text{train}} = \text{generalization gap}$, we have

$$\text{generalization gap} \gtrapprox \text{variance}, \tag{4.23a}$$

$$E_{\text{train}} \lessapprox \text{bias}^2 + \sigma^2. \tag{4.23b}$$

We have made several assumptions in this derivation that are not always met in practice, but it at least gives us some rough idea.

As we discussed previously, the choice of method is crucial for what E_{new} is obtained. Again Figure 4.5 and the notion of a bias-variance tradeoff is a simplified picture; decreased bias does not always lead to increased variance, and vice versa. However, in contrast to the decomposition of E_{new} into training error and generalization gap, the bias and variance decomposition can shed some more light over why E_{new} decreases for different methods: sometimes, the superiority of one method over another can sometimes be attributed to either a lower bias or a lower variance.

A simple (and useless) way to increase the variance without decreasing the bias in linear regression, is to first learn the parameters using the normal equations and thereafter add zero-mean random noise to them. The extra noise does not affect the bias, since the noise has zero mean and hence leaves the average model $\bar{f}$ unchanged, but the variance increases. (This effects also the training error and the generalization gap, but in a less clear way.) This way of training linear regression would be pointless in practice since it increases E_{new}, but it illustrates the fact that increased variance does *not* automatically leads to decreased bias.

A much more useful way of dealing with bias and variance is the meta-method called bagging, Chapter 7. It makes use of several copies (an ensemble) of a base model, each trained on a slightly different version

[6]Since neither $\bar{f}(x_\star)$ nor $y_\star$ depends on the training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, we can use $\{\mathbf{x}_i, y_i\}_{i=1}^n$ for approximating the integral, cf. (4.2)

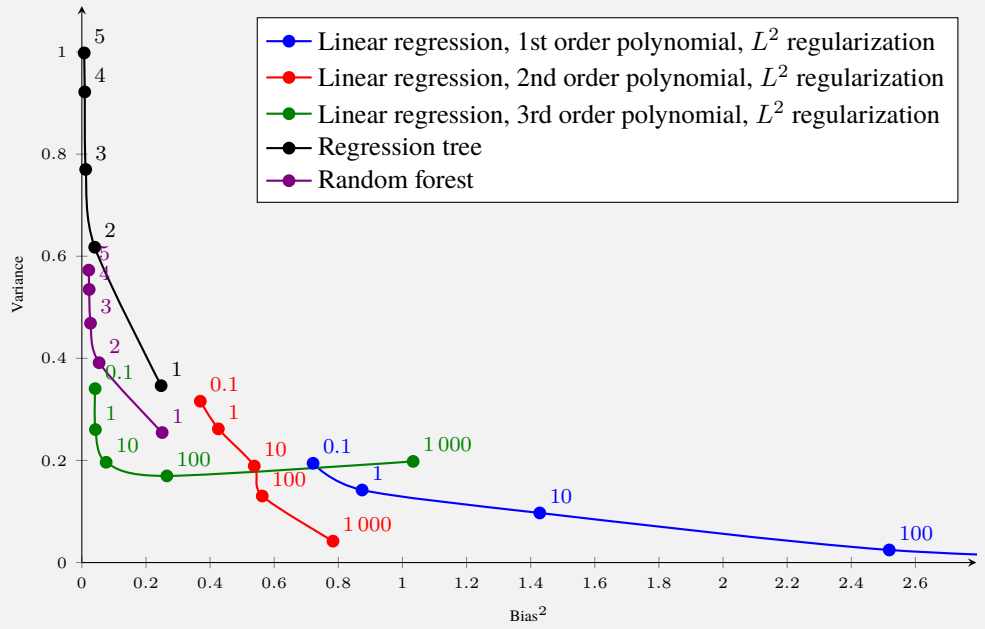
of the training dataset. Since bagging averages over many base models, it reduces the variance, but the bias remains (essentially) unchanged. Hence, by using bagging instead of the base model, the variance is decreased but (almost) without increasing the bias, which consequently decreases E_{new}.

To conclude, the world is more complex than just the one-dimensional model complexity scale used in Figure 4.3 and 4.5, which we illustrate by Example 4.4.

> ***Time to reflect 4.3:*** *Can you modify linear regression such that the bias increases, without decreasing the variance? Hint: You may change not only the model, but also the training procedure.*

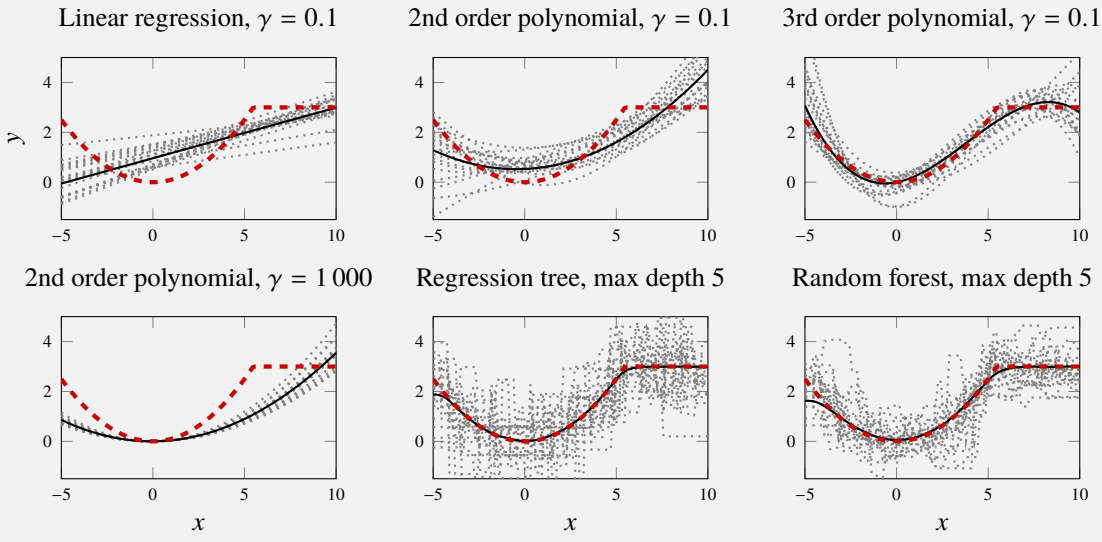
Example 4.4: Bias and variance for a regression problem

We consider the exact same setting as in Example 4.2, but decompose $\bar{E}_{\text{new}}$ into bias and variance instead. This gives us the figure below.



There are clear resemblances to Example 4.2, as expected from (4.23). The effect of bagging (random forest) is, however, more clear, namely that it reduces the variance compared to the regression tree with no noteworthy increase in bias.

For another illustration of what bias and variance means, we illustrate some of these cases in more detail. First we plot some of the linear regression models. The dashed red line is the true $f_0(\mathbf{x})$, the dotted gray lines are different models $\widehat{y}(\mathbf{x}_\star)$ learned from different training datasets $\mathcal{T}$, and the solid black line their mean $\bar{f}(\mathbf{x})$. In these figures, bias is the difference between the dashed red and solid black lines, whereas variance is the spread of the dotted gray lines around the solid black. The variance appears to be roughly the same for all three models, perhaps somewhat smaller for the first order polynomial, whereas the bias is clearly smaller for the higher order polynomials. This can be compared to the figure above.



Comparing the second order polynomial with little ($\gamma = 0.1$) and heavy ($\gamma = 1\,000$) regularization, it is clear that regularization reduces variance, but also increases bias. Furthermore has random forest a smaller variance than the regression tree, but without any change in the black line $\bar{f}(\mathbf{x})$, and hence no change in bias.

4.5 Evaluation for imbalanced and asymmetric classification problems

Sometimes classification problems are *imbalanced* and/or *asymmetric*. The typical example is perhaps when binary classification is used to detect the presence of something, such as a disease, an object on the radar, etc. The convention is that $y = 1$ (positive) denotes presence, and $y = -1$ (negative) denotes absence. We say that a problem is

(i) *imbalanced* if the vast majority of the data points belongs to one class, typically negative $y = -1$. This imbalance implies that a (useless) classifier which always predicts $\widehat{y}(\mathbf{x}) = -1$ will score very well in terms of misclassification rate (4.1a).

(ii) *asymmetric* if a missed detection (predicting $\widehat{y}(\mathbf{x}) = -1$, when in fact $y = 1$) is considered more severe than a false detection (predicting $\widehat{y} = 1$, when in fact $y = -1$), or vice versa. That asymmetry is not reflected in the misclassifcation rate (4.1a).

If a classification problem is imbalanced and/or asymmetric, the misclassification rate, and hence E_{new} as we defined it in Section 4.1, is not always very useful. The concepts of generalization gap and bias-variance tradeoff are still applicable, but for those problems other metrics than misclassification rate might be more relevant. We will now briefly introduce some useful tools for asymmetric and/or imbalanced classification problems, which can be used either as an alternative or as a complement to the misclassification error function (4.1a) in E_{new}. For simplicity we consider the binary problem and use a hold-out validation dataset approach, but the ideas can be extended to the multiclass problem as well as to k-fold cross-validation.

The confusion matrix and the F_1 score

If we learn a binary classifier and evaluate it on a hold-out validation dataset, a simple yet useful way to inspect the performance more than just computing $E_{\text{hold-out}}$ is a *confusion matrix*. By separating the validation data in four groups depending on y (the actual output) and $\widehat{y}(\mathbf{x})$ (the output predicted by the classifier), we can make the confusion matrix,

	$y = -1$	$y = 1$	*total*
$\widehat{y}(\mathbf{x}) = -1$	True neg (TN)	False neg (FN)	N*
$\widehat{y}(\mathbf{x}) = 1$	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 in Example 4.5. The confusion matrix provides a quick and informative overview of the characteristics of a classifier. For asymmetric problems, it is 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 in practice one usually has to decide which ones are considered most important. That tradeoff can often be done by tuning the decision threshold for a binary classifier as we discussed in **??**.

There is also a wide body of terminology related to the confusion matrix, which is summarized in Table 4.1. Some particularly common terms are the *recall* (TP/P) and the *precision* (TP/P*). Recall describes how big proportion among the true positive points that are predicted as positive. A high recall (close to 1) is good, and a low recall (close to 0) indicates a problem with false negatives. Precision describes what the ratio of true positive points are among the ones predicted as positive. A high precision (close to 1) is good, and a low recall (close to 0) indicates a problem with false positives.

For imbalanced (but not asymmetric) problems where the negative class $y = -1$ is the most common class, it makes sense to summarize the precision and recall by their harmonic mean into the F_1 score. Instead of using the misclassification rate for imbalanced problems, the F_1 score can be used instead. Note, however, the scale for the F_1 score: the perfect classifier attains F_1 score equal to 1 whereas 0 is worst (for misclassification rate, 1 is worst). It is also possible to use k-fold cross-validation to estimate the F_1 score for new unseen datapoints if the amount of validation data is limited.

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
(FN+FP)/*n*	*Misclassification rate*
(TN+TP)/*n*	Accuracy
2TP/(P*+P)	*F_1 score*

Table 4.1: Some common terms related to the quantities (TN, FN, FP, TP) in the confusion matrix. The terms written in italics are discussed in the text.

Example 4.5: The 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 {`normal`, `hyperthyroid`, `hypothyroid`}, which we convert into the binary problem with the output classes {`normal`, `not normal`}. The dataset is split into a training and hold-out validation part, with 3772 and 3428 samples respectively. The problem is imbalanced since only 7% of the data points are `not normal`, and possibly asymmetric if false negatives (not indicating the disease) are considered more problematic than false positives (falsely indicating the disease). We train a logistic regression classifier and evaluate it on the validation dataset (using the default decision threshold $r = 0.5$, see (3.36)). We obtain the confusion matrix

	y = `normal`	y = `not normal`
$\widehat{y}(\mathbf{x})$ = `normal`	3177	237
$\widehat{y}(\mathbf{x})$ = `not normal`	1	13

Most validation 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. The misclassification rate is 0.069 and the F_1 score is 0.106. (The useless predictor of always predicting `normal` has a very similar misclassification rate of 0.073, but worse F_1 score 0.)

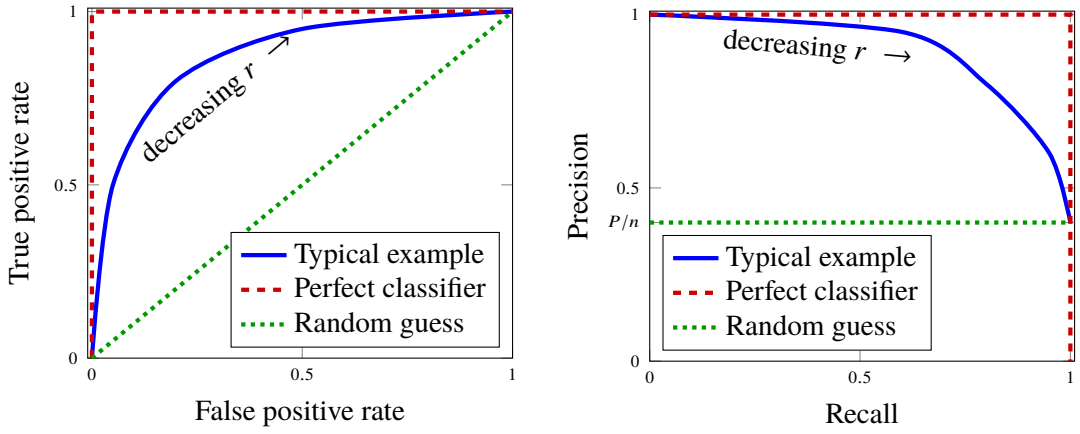
To change the picture, we change the threshold to $r = 0.15$, and obtain new predictions with the following confusion matrix instead:

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

This change gives more true positives (85 instead of 13 patients are correctly predicted as `not normal`), but this happens at the expense of more false positives (111, instead of 1, patients are now falsely predicted as `not normal`). As expected, the misclassification rate is now higher (worse) at 0.081, but the F_1 score is higher (better) at 0.381. Remember, however, that the F_1 score does not take the asymmetry into account, but only the imbalance. We have to decide ourselves whether this classifier is a good tradeoff between the false negative and false positive rates, by considering which type of error has the most severe consequences.

ROC and precision-recall curve

As suggested by the example above, the tuning of the threshold r in (3.36) can be crucial for the performance in binary classification. If we want to compare different classifiers for a certain problem without specifying



(a) The ROC curve

(b) The precision-recall curve

Figure 4.7: The ROC (left) and the precision-recall (right) curve. Both plots summarizes the performance of a classifier for *all* decision thresholds r (cf. (**??**), but the ROC curve is most relevant for balanced problems, whereas the precision-recall curve is more informative for imbalanced problems.

a certain decision threshold r, 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, a high value is good) is drawn against the false positive rate (FP/N, a low value is good) for all values of $r \in [0, 1]$. The curve typically looks as shown in Figure 4.7a. 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, ROC-AUC*. From Figure 4.7a, we conclude that a perfect classifier has ROC-AUC 1, whereas a classifier which only assigns random guesses has ROC-AUC 0.5. The ROC-AUC is thus summarizing the performance of a classifier for *all* possible values of the decision threshold r.

In the previous section, we discussed how the misclassification rate might be misleading for imbalanced problems, but we should consider the precision, recall and F_1 score instead. Similarly might the ROC curve be misleading for imbalanced problems, but the precision-recall curve can (for imbalanced problems where $y = -1$ is the most common class) be more useful. As the name suggests, the precision-recall curve plots the precision (TP/P*, a high value is good) against the recall (TP/P, a high value is good) for all values of $r \in [0, 1]$, much like the ROC curve. The precision-recall curve for the perfect classifier touches the upper right corner, and a classifier which only assigns random guesses gives a horizontal line with height P/n, as shown in Figure 4.7b.

Also for the precision-recall curve, we can define *area under the curve, precision-recall AUC*. The best possible precision-recall AUC is 1, and the classifier which only makes random guesses has precision-recall AUC equal to P/n.

5 Learning parametric models

This chapter resolves around three different topics, namely loss functions, regularization and optimization. We have touched upon all of them already, mostly in connection to the parametric models in Chapter 3, linear and logistic regression. These topics are however central for many more supervised machine learning methods and deserve a dedicated discussion, which we will give now. The understanding of loss functions is particularly helpful when we introduce boosting (Chapter 7) and support vector machines (Chapter 8) later on. Regularization is crucial in preventing overfit (Chapter 4) and should be in the back of the mind of any machine learning engineer. Finally, it is always helpful to be familiar with the main ideas of optimization, in particular since they are a key enabler for deep learning (Chapter 6).

5.1 Loss functions

Most parametric regression and classification models, including linear and logistic regression (Chapter 3), are trained by minimizing a cost function $J(\boldsymbol{\theta})$. The cost function is the average of the loss function[1] L evaluated on the training data,

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \overbrace{L(\widehat{y}(\mathbf{x};\boldsymbol{\theta}), y_i)}^{\text{loss function}}}_{\text{cost function } J(\boldsymbol{\theta})}. \tag{5.1}$$

The choice of loss function is, in the end, a user choice. Different loss functions give rise to different solutions $\widehat{\boldsymbol{\theta}}$, which in turn results in models with different characteristics. There is never a "right" or "wrong" loss function, but one loss functions may of course perform better than another on a given problem. Certain combinations of models and loss functions have proven particularly fruitful and have historically been branded as different methods carrying different names. For example, the term "linear regression" most often refers to the combination of a linear-in-the-parameter model and the squared error loss, whereas the term "support vector classification" (Chapter 8) refers to a linear-in-the-parameter model trained using the hinge loss. In this section, however, we provide a general discussion about different loss functions and their properties, without connections to a specific method.

One important aspect of a loss function is its *robustness*. Robustness is tightly connected to outliers, meaning spurious data points that do not describe the relationship we are interested in modeling. If outliers in the training data only have a minor impact to the learned model, we say that the loss function is robust. Conversely, a loss function is not robust if the outliers have a major impact to the learned model. Robustness is therefore a very important property in applications where the training data is contaminated with outliers. Some of the common "default" loss-functions, such as the squared error loss, are unfortunately not particularly robust, and it is therefore important to make an informed user choice. Robustness is however not a binary property, instead loss functions can be robust to a greater or smaller degree.

Another important property of loss functions for binary classification is their so-called *asymptotic minimizers*. The asymptotic minimizer refers to the function that minimizes the loss function when $n \to \infty$, and reveals whether the model will possibly learn the full conditional class probability $p(y = 1 \mid \mathbf{x})$ or only the decision boundary. We will discuss this in more detail later.

[1] As you might already have noticed, the arguments to the loss function (here $\widehat{y}$ and $y_\star$) varies with context. This is unfortunate but unavoidable, since different loss functions are formulated in terms of different quantities, for example the prediction $\widehat{y}$, the predicted conditional class probability $g(\mathbf{x})$, the classifier margin, etc.

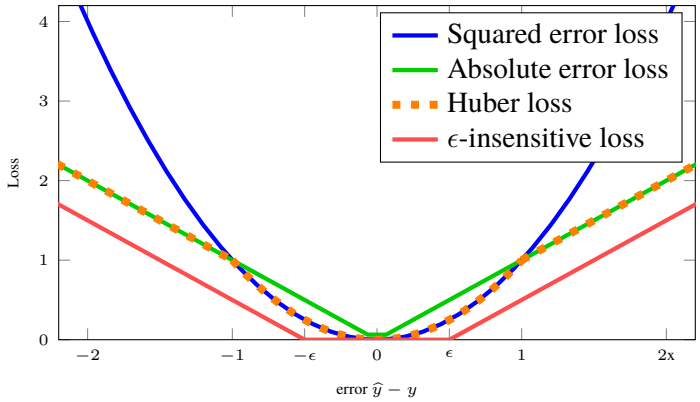


Figure 5.1: The loss functions for regression presented in the text, each as a function of the error $\widehat{y} - y$.

Loss functions for regression

In Chapter 3 we introduced the *squared error loss*

$$L(\widehat{y}, y) = \big(\widehat{y} - y\big)^2, \tag{5.2}$$

which is the default choice for linear regression since it simplifies the training to only solving the normal equations. The squared error loss is often used also for other regression models, such as neural networks. Another common, but more robust, loss function is the *absolute error loss*,

$$L(\widehat{y}, y) = |\widehat{y} - y|. \tag{5.3}$$

In Chapter 3 we introduced the maximum likelihood motivation of the squared error loss by assuming that the output y is measured with additive uncorrelated noise ε from a Gaussian distribution. We can similarly motivate the absolute error loss by instead assuming ε to have a Laplace distribution, $\varepsilon \sim \mathcal{L}(0, b_\varepsilon)$. This statistical perspective is one way to understand the fact that the absolute error loss is more robust (less sensitive to outliers) than the squared error loss, since the Laplace distribution has a thicker tail compared to the Gaussian distribution, see Figure 5.1. The Laplace distribution therefore encodes sporadic large noise values (i.e., outliers) as more probable, compared to the Gaussian distribution.

If we have statistical knowledge about how the noise ε in a regression problem behaves (for example a skewed normal distribution), it is perfectly possible to use that information and make a maximum likelihood derivation like (3.17)-(3.21) to derive a loss function for regression tailored to that information. There are, however, a few additional off-the-shelf loss functions commonly used for regression which are less natural to derive from a maximum likelihood perspective.

It is sometimes argued that the squared error loss is a good choice because of its quadratic shape which penalizes small errors ($\varepsilon < 1$) less than linearly. After all, the Gaussian distribution appears (at least approximately) quite often in nature. However, the quadratic shape for large errors ($\varepsilon > 1$) is the reason for its non-robustness, and the *Huber loss* has therefore been suggested as a hybrid between the absolute loss and squared error loss:

$$L(\widehat{y}, y) = \begin{cases} (\widehat{y} - y)^2 & \text{if } |\widehat{y} - y| < 1, \\ |\widehat{y} - y| & \text{otherwise.} \end{cases} \tag{5.4}$$

Another extension to the absolute error loss is the *ϵ-insensitive loss*,

$$L(\widehat{y}, y) = \begin{cases} 0 & \text{if } |\widehat{y} - y| < \epsilon, \\ |\widehat{y} - y| - \epsilon & \text{otherwise,} \end{cases} \tag{5.5}$$

where ϵ is a user-chosen design parameter. This loss places a "tolerance" of width 2ϵ around the true y and behaves like the absolute error loss outside this region. Furthermore the robustness properties of the

ϵ-insensitive loss are very similar to those of the absolute error loss. The ϵ-insensitive loss turns out to be a useful loss function for support vector regression in Chapter 8. We illustrate all these loss functions for regression in Figure 5.1.

Loss functions for binary classification

An intuitive loss function for binary classification is provided by the *misclassification loss*,

$$L(\widehat{y}, y) = \begin{cases} 1 & \text{if } y_\star = y, \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

However, even though a small misclassification loss often is the ultimate goal in practice, this loss function is rarely used when training models. There are (at least) two reasons for this. The most apparent reason in that the resulting cost function is a piecewise constant function, which is a poor optimization objective since the gradient is zero everywhere, except where it is undefined. Furthermore, a more subtle reason is that for many models the final prediction $\widehat{y}$ does not reveal all aspects of the classifier. Thinking intuitively in terms of decision boundaries, we may prefer to not have the decision boundary close to the training data points, but instead push the boundary further away to have some "safety leeway" if possible. The misclassification loss (5.6) cannot achieve this, sine it only encourages training data points to be on the right side of the decision boundary.

For binary classifiers that predicts conditional class probabilities $p(y = 1 \mid \mathbf{x})$ in terms of a function $g(\mathbf{x})$, the *cross entropy loss* is, as introduced in Chapter 3,

$$L(g(\mathbf{x}), y) = \begin{cases} \ln g(\mathbf{x}) & \text{if } y = 1, \\ \ln(1 - g(\mathbf{x})) & \text{if } y = -1. \end{cases} \tag{5.7}$$

This loss was derived from a maximum likelihood perspective, but unlike regression (where we had to specify a distribution for ε) there are no user choices left in the cross entropy loss, other than what model to use for $g(\mathbf{x})$.

The misclassification and cross entropy loss are however not the only loss functions for binary classification. To introduce an entire family of loss functions for logistic regression, let us first introduce the concept of *margins* in binary classification. Many binary classifiers $\widehat{y}(\mathbf{x})$ can be constructed by thresholding some real-valued function $c(\mathbf{x})$ at 0. That is, we can write

$$\widehat{y}(\mathbf{x}) = \operatorname{sign}\{c(\mathbf{x})\}. \tag{5.8}$$

Logistic regression, for example, can be brought into this form by simply using $c(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$. The decision boundary for any classifier on the form (5.8) is given by the values of $\mathbf{x}$ for which $c(\mathbf{x}) = 0$. To simplify our discussion we will assume that none of the data points fall exactly on the decision boundary (which always gives rise to an ambiguity). This will imply that we can assume that $\widehat{y}(\mathbf{x})$ as defined above is always either -1 or $+1$.

Based on the function $c(\mathbf{x})$, we say that

$$\boxed{\text{the margin of a classifier is } \; y \cdot c(\mathbf{x}).} \tag{5.9}$$

It follows that if y and $c(\mathbf{x})$ have the same sign, meaning 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. The margin can be viewed as a measure of certainty in a prediction, where data points with small margins in some sense (not necessarily Euclidian) are close to the decision boundary. The margin plays a similar role for binary classification as the error $\widehat{y} - y$ does for regression.

We can now define loss functions for binary classification in terms of the margin, by assigning a small loss to positive margins (correct classifications) and a large loss to negative margins (misclassifications). We can, for instance, re-formulate the logistic loss (3.34) as

$$L(y \cdot c(\mathbf{x})) = \ln\left(1 + \exp\left(-y \cdot c(\mathbf{x})\right)\right), \tag{5.10}$$

with $c(\mathbf{x}) = \theta^\mathsf{T}\mathbf{x}$. That is, learning $\boldsymbol{\theta}$ by minimizing the loss function (5.10) and making predictions according to (5.8) is equivalent to logistic regression as described in Chapter 3, except for the fact that we have lost the notion of a conditional class probability estimate $g(\mathbf{x})$ and only have a "hard" prediction $\widehat{y}(\mathbf{x}_\star)$. We will, however, recover the class probability estimate later when we discuss the asymptotic minimizer of (5.10).

We can also formulate the misclassification loss in terms of the margin,

$$L(y \cdot c(\mathbf{x})) = \begin{cases} 1 & \text{if } y \cdot c(\mathbf{x}) < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5.11}$$

The benefit of formulating a loss function in terms of the margin is that it is easy to come up with new loss functions. Let us now introduce the *exponential loss* defined as

$$L(y \cdot c(\mathbf{x})) = \exp(-y \cdot c(\mathbf{x})), \tag{5.12}$$

which turns out to be a useful loss function when we later will derive the AdaBoost in Chapter 7 due to its properties. The downside of the exponential loss is that it is not particularly robust against outliers, compared to the logistic loss. We also have the *hinge loss*, which we will use later for support vector classification in Chapter 8,

$$L(y \cdot c(\mathbf{x})) = \begin{cases} 1 - y \cdot c(\mathbf{x}) & \text{for } y \cdot c(\mathbf{x}) \leq 1, \\ 0 & \text{otherwise.} \end{cases} \tag{5.13}$$

As we will see in Chapter 8, the hinge loss has an attractive so-called support-vector property. However, as we soon will see its asymptotic minimizer is somewhat unfortunate. As a remedy to that, one may instead consider the *squared hinge loss*

$$L(y \cdot c(\mathbf{x})) = \begin{cases} (1 - y \cdot c(\mathbf{x}))^2 & \text{for } y \cdot c(\mathbf{x}) \leq 1, \\ 0 & \text{otherwise,} \end{cases} \tag{5.14}$$

which on the other hand is much less robust than the hinge loss. A more elaborate alternative might therefore be the *Huberized squared hinge* loss

$$L(y \cdot c(\mathbf{x})) = \begin{cases} -4y \cdot c(\mathbf{x}) & \text{for } y \cdot c(\mathbf{x}) \leq -1, \\ (1 - y \cdot c(\mathbf{x}))^2 & \text{for } -1 \leq y \cdot c(\mathbf{x}) \leq 1, \quad \text{(squared hinge loss)} \\ 0 & \text{otherwise,} \end{cases} \tag{5.15}$$

whose name refers to its similarities to the Huber loss for regression, namely that the quadratic function is replaced with a linear function for margins < -1. The three loss functions presented above are all particularly interesting for support vector classification, due to the fact that they are all exactly 0 for margins > 1.

We summarize this cascade of loss functions for binary classification in Figure 5.2, which illustrates all these losses as a function of the margin.

We have already made some claims about robustness. Let us motivate them using Figure 5.2. One characterization of an outlier is as a data point on the "wrong" side of and far away from the decision boundary. In a margin perspective, that is equivalent to a large negative margin. The robustness of a loss function is therefore tightly connected to the shape of the loss function for large negative margins. The steeper slope and heavier penalization of large negative margins, the more sensitive it is to outliers. We can therefore tell from Figure 5.2 that the exponential loss is most sensitive to outliers, due to its exponential left asymptote, whereas the squared hinge loss is somewhat more robust with a quadratic asymptote instead. However, even more robust are the Huberized squared hinge loss, the hinge loss and the logistic loss which all have an asymptotic behavior which is linear. Most robust is the misclassification loss, but as already discussed that loss has other disadvantages.

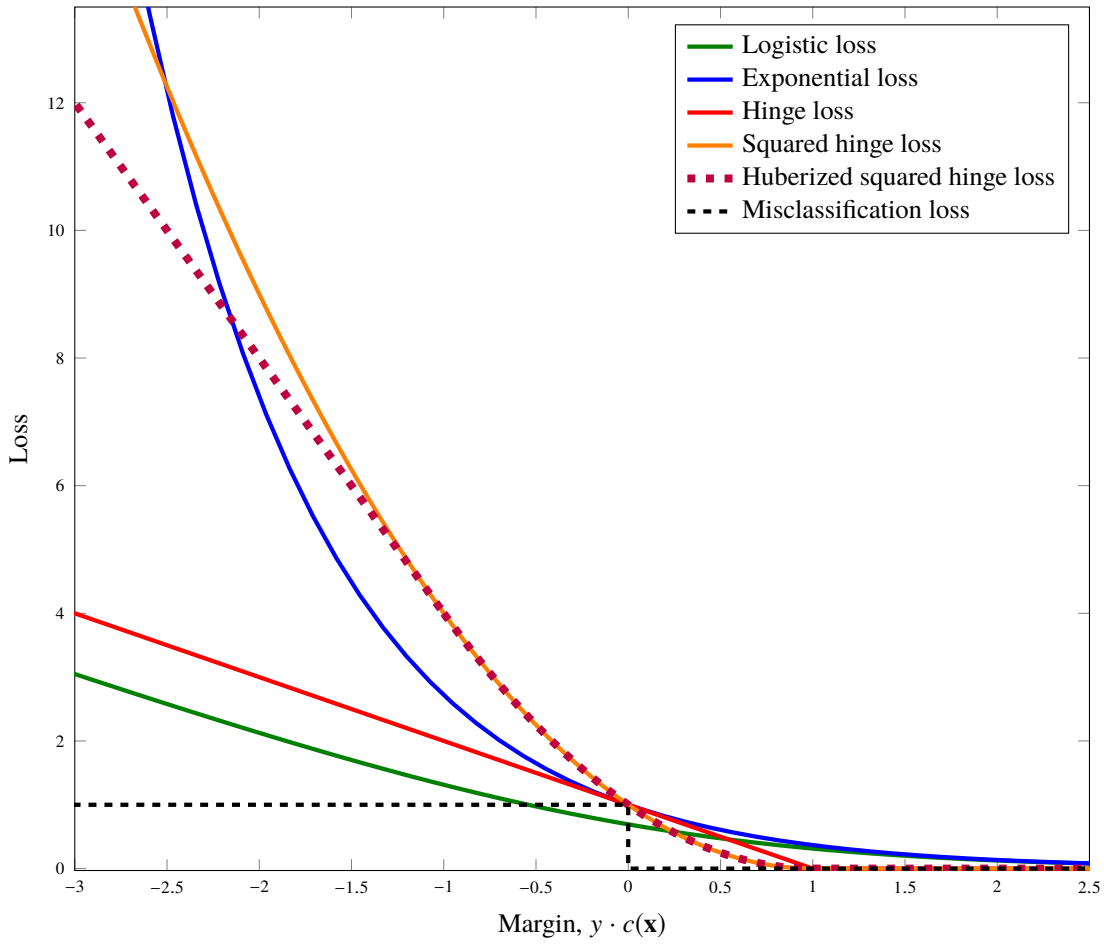


Figure 5.2: Comparison of some common loss functions for classification, plotted as a function of the margin.

The asymptotic minimizer in binary classification

As mentioned earlier the *asymptotic minimizer* is a useful theoretical concept un understanding loss functions for classification. The asymptotic minimizer is the function ($g(\mathbf{x})$ or $c(\mathbf{x})$, depending on the parametrization of the loss function) which minimizes the cost function when the number of training data $n \to \infty$, hence the name asymptotic. It contains valuable information whether the model will, eventually as $n \to \infty$, learn the conditional class probability $p(y = 1 \mid \mathbf{x})$ correctly or not.

In the following discussion we will make heavy use of the notion of a ground truth conditional class probability $p(y = 1 \mid \mathbf{x})$, which we assume the training data is drawn from. Learning $p(y = 1 \mid \mathbf{x})$, or some aspect of it, is the ultimate goal with binary classification in supervised machine learning. We assume that the model is flexible enough such that it can describe $p(y = 1 \mid \mathbf{x})$ well. This assumption is not always met in practice, and we will discuss the implications of this assumption at the end.

There is a notion of a *strictly proper* loss function. A strictly proper loss function is a loss function for which the asymptotic minimizer (i) is unique and (ii) is an invertible transformation of $p(y = 1 \mid \mathbf{x})$. That is, a strictly proper loss function forces the model to learn the full[2] distribution $p(y = 1 \mid \mathbf{x})$ if given enough training data. All models can be used for predicting classes $\widehat{y}(\mathbf{x}_\star)$, but only a model trained with a strictly proper loss function can be used for predicting conditional class probabilities (like $g(\mathbf{x})$ in logistic regression). Deriving the asymptotic minimizer of a loss function is most often a straightforward calculation, but for brevity we do not include the derivations here.

Starting with the binary maximum likelihood loss (5.7), its asymptotic minimizer can be shown to be $g(\mathbf{x}) = p(y = 1 \mid \mathbf{x})$. In other words, when $n \to \infty$ the loss function (5.7) trains the model such that $g(\mathbf{x})$ becomes the conditional class probability. Since we derived (5.7) with this intention in Chapter 3, this was indeed expected.

The asymptotic minimizer of the logistic loss (5.10) is $c(\mathbf{x}) = \ln \frac{p(y=1 \mid \mathbf{x})}{1-p(y=1 \mid \mathbf{x})}$. This is an invertible expression and hence the logistic loss is a strictly proper loss function. By inverting $c(\mathbf{x})$ we obtain $p(y = 1 \mid \mathbf{x}) = \frac{\exp c(\mathbf{x})}{1+\exp c(\mathbf{x})}$ which shows how conditional class probability predictions can be obtained from $c(\mathbf{x})$. (With the "margin formulation" of logistic regression, we seemingly lost the class probability predictions $g(\mathbf{x})$. We have now recovered it.)

For the misclassification loss (5.11) the asymptotic minimizer is not unique but only fulfills

$$\operatorname{sign}\{c(\mathbf{x})\} = \begin{cases} 1 & \text{if } p(y = 1 \mid \mathbf{x}) > 0.5, \\ -1 & \text{if } p(y = 1 \mid \mathbf{x}) < 0.5. \end{cases}$$

From this expression we see that a model trained with the misclassification loss has the correct decision boundary (when $n \to \infty$). However, the misclassification loss does not train the model such that $p(y = 1 \mid \mathbf{x})$ can be inferred from $c(\mathbf{x})$, since the asymptotic minimizer is not an invertible transformation of it. The misclassification loss is therefore not a strictly proper loss function, and $c(\mathbf{x})$ can not be used for predicting conditional class probabilities. This is yet another downside of the misclassification loss.

Turning to the exponential loss (5.12), its asymptotic minimizer is $c(\mathbf{x}) = \frac{1}{2} \ln \frac{p(y=1 \mid \mathbf{x})}{1-p(y=1 \mid \mathbf{x})}$, almost like the logistic loss. The exponential loss is therefore also strictly proper, and $c(\mathbf{x})$ can be transformed and used for predicting conditional class probabilities.

The asymptotic minimizer of the hinge loss is

$$c(\mathbf{x}) = \begin{cases} 1 & \text{if } p(y = 1 \mid \mathbf{x}) > 0.5, \\ -1 & \text{if } p(y = 1 \mid \mathbf{x}) < 0.5. \end{cases}$$

This is a non-invertible transformation of $p(y = 1 \mid \mathbf{x})$, which means that $p(y = 1 \mid \mathbf{x})$ is not possible to infer from the asymptotic minimizer $c(\mathbf{x})$. This implies that a classifier learned using hinge loss (such as support vector classification, Section 8.4) can *not* predict conditional class probabilities.

The squared hinge loss, on the other hand, is a strictly proper loss function, since its asymptotic minimizer is $c(\mathbf{x}) = 2p(y \mid \mathbf{x}) - 1$. This also holds for the Huberized square hinge loss. Recalling our

[2] Loss functions that are proper, but not strictly proper, may for example learn only the correct decision boundary, that is $p(y = 1 \mid \mathbf{x}) = 0.5$, which is not a full characterization of $p(y = 1 \mid \mathbf{x})$.

robustness discussion, we see that by squaring the hinge loss we make it strictly proper but a the same time we impact its robustness. However, the "huberization" (replacing the quadratic curve with a linear one for margins < -11) improves the robustness while keeping the property of being strictly proper.

We have now seen that some (but not all) of the loss functions are strictly proper, meaning they could potentially predict conditional class probabilities correctly. However, this is only under the assumption that the model is sufficiently flexible such that $g(\mathbf{x})$ or $c(\mathbf{x})$ actually can take the shape of the asymptotic minimizer. This is possibly problematic; remember that $c(\mathbf{x})$ is a linear function in logistic regression, whereas $p(y = 1 \mid \mathbf{x})$ can be almost arbitrarily complicated in real world applications. It is therefore not sufficient to use a strictly proper loss function in order to predict conditional class probabilities, but our model also has to be flexible enough. This discussion is also only valid in the limit as $n \to \infty$. However, in practice n is always bounded, and we may ask how large n has to be for a flexible enough model to at least approximately learn the asymptotic minimizer? Unfortunately we cannot give any general numbers, but following the same principles as the overfit discussion in Chapter 4, the more flexible the model the larger n is required. If n is not large enough, the predicted conditional class probabilities tend to be "overfitted" to the training data. We therefore unfortunately have to conclude that the concept of the asymptotic minimizer only give limited guidance on how to interpret the learned $c(\mathbf{x})$ in practice.

Multiclass classification

So far we have only discussed the binary classification problem with $M = 2$. The maximum likelihood loss is indeed possible to generalize to the multiclass problem, that is $M > 2$, as we did in Chapter 3 for logistic regression. The generalization of the other loss functions is, however, a more complicated problem and would require a generalization of the margin to the multiclass problem to start with. That is possible but we do not discuss it in this book. A more pragmatic approach is instead to reformulate the problem as several binary problems. This re-formulation can be done using either a one-versus-rest or one-versus-one scheme.

The one-versus-rest (or one-versus-all or binary relevance) idea is to train M binary classifiers. Each classifier in this scheme is trained for predicting one class against all the other classes. To make a prediction for a test data point, all M classifiers are used, and the class which, for example, is predicted with the largest margin is taken as the predicted class. This approach is a pragmatic solution, which may turn out to work well for some problems.

The one-versus-one idea is instead to train one classifiers for each pair of classes. If there are M classes in total, there are $\frac{1}{2}M(M-1)$ such pairs. To make a prediction each classifier predicts either of its two classes, and the class which overall obtains most "votes" is chosen as the final prediction. The predicted margins can be used to break a tie if that happens. Compared to one-versus-rest, the one-versus-one approach has the disadvantage of involving $\frac{1}{2}M(M-1)$ classifiers, instead of only M. On the other hand each of these classifiers is trained on much smaller datasets (only the data points that belong to either of the two classes) compared to one-versus-rest which uses the entire original training dataset for all M classifiers.

5.2 Regularization

We will now take a closer look at regularization, which was briefly introduced in Section 3.3 as a useful tool for avoiding overfit if the model was too flexible, such as a polynomial of high degree. We have also discussed thoroughly in Chapter 4 the need for tuning the model flexibility, which is the purpose of regularization. Finding the right level of flexibility, and thereby avoiding overfit, is very important in practice.

Regularization is applicable to any parametric model, and the idea it to 'keep the parameters $\widehat{\boldsymbol{\theta}}$ small unless the data really convinces us otherwise', or alternatively 'if a model with small values of the parameters $\widehat{\boldsymbol{\theta}}$ fits the data almost as well as a model with larger parameter values, the one with small parameter values should be preferred'. We will now have a closer look at the so-called L^2 and L^1 regularization.

L^2 regularization

The L^2 regularization (also known as *Tikhonov regularization*, *ridge regression* and *weight decay*) amounts to adding an extra penalty term $\|\boldsymbol{\theta}\|_2^2$ to the cost function. Linear regression with squared error loss and L^2 regularization, as an example, amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{5.16}$$

By choosing the regularization parameter $\lambda \geq 0$, a trade-off between the original cost function (fitting the training data as well as possible) and the regularization term (keeping the parameters $\widehat{\boldsymbol{\theta}}$ close to zero) is made. In the setting $\lambda = 0$ we recover the original least squares problem (3.12), whereas $\lambda \to \infty$ will force all parameters $\widehat{\boldsymbol{\theta}}$ to 0. A good choice of λ is usually somewhere in between and depends on the actual problem, and can be selected using cross-validation.

It is actually possible to derive a version of the normal equations for (5.16), namely

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{5.17}$$

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

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{5.18}$$

This also reveals another reason for using regularization in linear regression, namely if $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible. When $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, the ordinary normal equations (3.13) have no unique solution $\widehat{\boldsymbol{\theta}}$, whereas the L^2-regularized version always has the unique solution (5.18) if $\lambda > 0$.

L^1 regularization

With L^1 regularization (also called *LASSO*, an abbreviation for Least Absolute Shrinkage and Selection Operator), the penalty term $\|\boldsymbol{\theta}\|_1$ is added to the cost function. Here $\|\boldsymbol{\theta}\|_1$ is the 1-norm or 'taxicab norm' $\|\boldsymbol{\theta}\|_1 = |\theta_0| + |\theta_1| + \cdots + |\theta_p|$. The L^1 regularized cost function for linear regression (with squared error loss) (3.12) then becomes

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1. \tag{5.19}$$

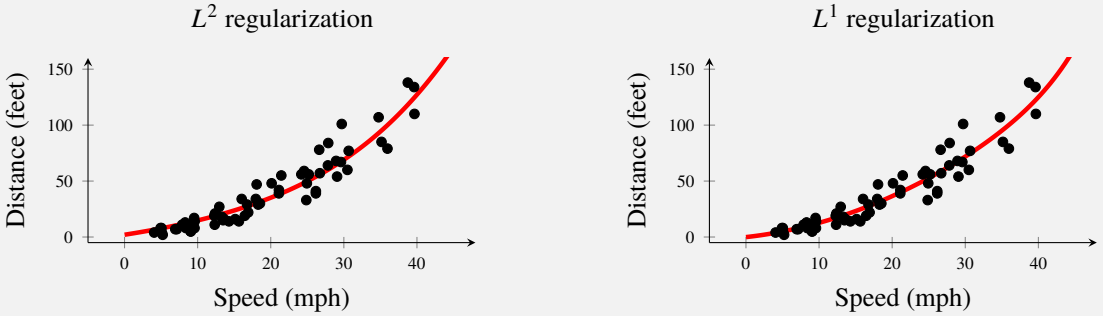
Contrary to linear regression with L^2 regularization (3.48), there is no closed-form solution available for (5.19). However, as we will see in Section 5.3, it is possible to design an efficient numerical optimization algorithm for solving (5.19).

As for L^2 regularization, the regularization parameter λ has to be chosen by the user, and has a similar meaning: $\lambda = 0$ gives the ordinary least squares solution and $\lambda \to \infty$ gives $\widehat{\boldsymbol{\theta}} = 0$. Between these extremes,

however, L^1 and L^2 tend to give different solutions. Whereas L^2 regularization pushes all parameters towards small values (but not necessarily exactly zero), L^1 tends to favor so-called *sparse* solutions where only a few of the parameters are non-zero, and the rest are exactly zero. Thus, L^1 regularization can effectively 'switch off' some inputs (by setting the corresponding parameter θ_k to zero) and it can therefore be used as an input (or feature) selection method.

Example 5.1: Regularization for car stopping distance

Consider again Example 2.2 with the car stopping distance regression problem. We use the 10th order polynomial that was considered meaningless in Example 3.5 and apply L^2 and L^1 regularization to it, respectively. With manually chosen λ, we obtain the following models



Both models suffer less from overfit than the non-regularized 10th order polynomial in Example 3.5. The two models here are, however, not identical. Whereas all parameters are relatively small but non-zero in the L^2-regularized model (left panel), only 4 (out of 11) parameters are non-zero in the L^1-regularized model (right panel). It is typical for L^1 regularization to give sparse models, where some parameters are set to exactly zero.

General cost function regularization

The L^1 and L^2 regularization are two commonly used regularization methods, and they are both formulated as additions to the cost function. Together they suggest a more general regularization scheme

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})}_{\text{(i)}} + \underbrace{\lambda}_{\text{(iii)}} \underbrace{R(\boldsymbol{\theta})}_{\text{(ii)}}. \tag{5.20}$$

This expression contains three important elements:

(i) the cost function, which encourages a good fit to training data,

(ii) the regularization term, which encourages small parameter values, and

(iii) the regularization parameter λ, which determines the trade-off between (i) and (ii).

The regularization term can be designed in many ways. As a combination of the L^1 and L^2 terms, one option is $R(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 + \|\boldsymbol{\theta}\|_2^2$, which often is referred to as elastic net regularization. Regardless of the exact expression of the regularization term, its purpose is to encourage small parameter values and thereby decrease the flexibility of the model. As we discussed in depth in Chapter 4, a too flexible (or complex) model may overfit to training data and not be able to generalize well to previously unseen test data. By using regularization we get a systematic and practically useful way of controlling the model flexibility and thereby counteracting overfit.

Implicit regularization

Any supervised machine learning method that is trained by minimizing a cost function can be regularized as (5.20). There are, however, alternative ways to achieve similar effects. One such example of *implicit*

regularization is early stopping. Early stopping is applicable to any method that is trained using numerical optimization, and amounts to aborting the optimization before it has reached the minimum of the cost function. Even though it might appear counter-intuitive to pre-maturely abort an optimization procedure, early stopping has shown to be a of practical importance to avoid overfit for some models, most notably deep learning Chapter 6.

5.3 Parameter optimization

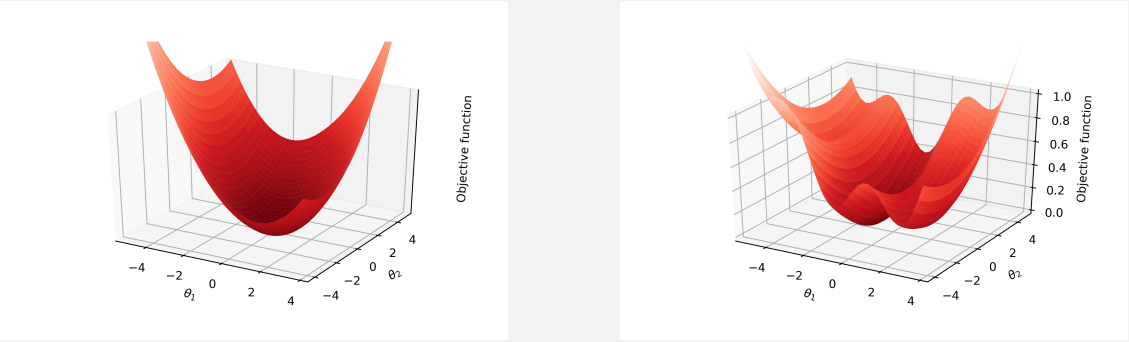
Many supervised machine learning methods, including linear and logistic regression, involves one (or more) optimization problems, such as (3.12), (3.35) or (5.19). It is therefore good to be familiar with some of the main strategies for how to solve optimization problems fast. Starting in the optimization problems from linear and logistic regression, we will introduce the ideas behind some common methods.

Optimization is about finding the minimum (or maximum) of an *objective function*. Since the maximization problem can be formulated as minimization of the negative objective function, it is sufficient to describe only minimization. When we use optimization for training models in machine learning, that objective function is the cost function J. Optimization is, however, often used also in combination with cross-validation (Chapter 4) for tuning hyperparameters, such as selecting the regularization parameter λ by minimizing the prediction error on a held-out validation dataset.

An important class of objective functions are *convex* functions. Optimization is often easier to carry out for convex objective functions, and it is a general advise to spend some extra effort to consider whether a non-convex optimization problem can be re-formulated into a convex problem (which sometimes, but not always, is possible). The most important property of a convex function, for this discussion, is that a convex function has a unique minimum[3], and no other local minima. Examples of convex functions are the cost functions for logistic regression, linear regression and L^1-regularized linear regression. Examples of non-convex functions will come later in the book, including the cost function for a deep neural network. Two examples of convex and non-convex functions are given in Example 5.2 below.

Example 5.2: Example of objective functions

These figures are examples of what an objective function can look like.



Both examples are functions of a two-dimensional parameter vector $\boldsymbol{\theta} = [\theta_1\ \theta_2]^\mathsf{T}$. The left is convex and has a finite unique global minimum, whereas the right is non-convex and has three local minima (of which only one is the global minimum). We will in the following examples illustrate these objective functions using contour plots instead, as shown below.

[3] The minimum does, however, not have to be finite. Consider for example the exponential function, which is convex but attains its minimum in $-\infty$. Convexity is, however, a relatively strong property, and a function may have only one minimum even if it is not convex.

Time to reflect 5.1: *After reading the rest of this book, return here and try to fill out this table, summarizing how optimization is used by the different methods.*

Method	*What is optimization used for?*			*What type of optimization?*			
	Training	*Hyper-parameters*	*Nothing*	*Closed-form**	*Grid search*	*Gradient-based*	*Stochastic gradient*
Linear regression							
Linear regression with L2-regularization							
Linear regression with L1-regularization							
Logistic regression							
k-NN							
Trees							
Random forests							
AdaBoost							
Gradient boosting							
Deep learning							
Gaussian processes							

**including coordinate descent*

Optimization using closed-form expressions

For linear regression with squared error loss, training the model amounts to solving the optimization problem (3.12)

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2.$$

As we have discussed, and also prove in Appendix 3.A, the solution (3.14) to this problem can (under the assumption that $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible) be derived analytically. If we only once spend some time to efficiently implement (3.14), for example using the Cholesky or QR factorization, we can use that every time we want to train a linear regression model with squared error loss. Each time we use it we know that we have found the optimal solution in a computationally efficient way.

If we instead want to learn the L^1-regularized version, we instead have to solve (5.19)

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1.$$

This problem can, unfortunately, not be solved analytically. Instead we have to use computer power to solve it, by constructing an iterative procedure for seeking the solution. With a certain choice of such an optimization algorithm, we can make use of some analytical expressions along the way, which turns out to yield an efficient way to solve (5.19). Remember that $\boldsymbol{\theta}$ is a vector containing $p+1$ parameters we want to learn from the training data. As it turns out, if we seek the minimum for only one of these parameters, say θ_j, while keeping the other parameters fix, we can find the optimum as

$$\arg\min_{\theta_j} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1 = \text{sign}(t)(|t| - \lambda), \text{ where } t = \sum_{i=1}^{n} x_{ij}(y_i - \sum_{k\neq j} x_{ik}\theta_k). \tag{5.21}$$

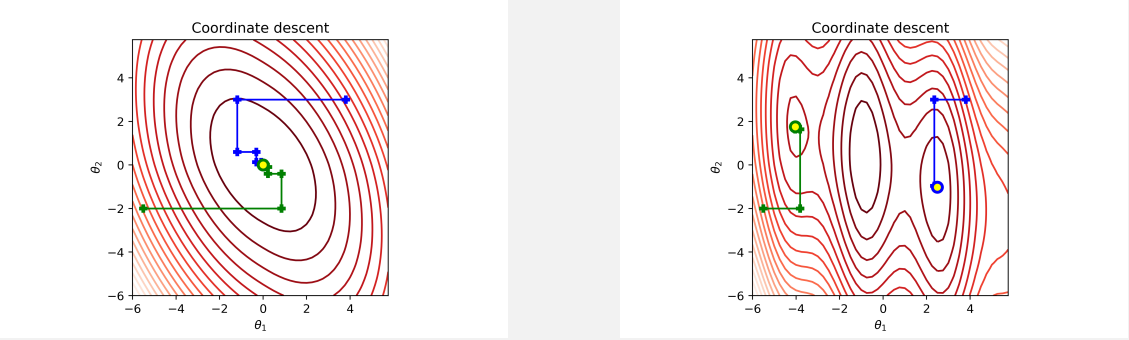
It turns out that making repeated "sweeps" through the vector $\boldsymbol{\theta}$ and updating one-parameter-at-a-time according to (5.21) is a good way to solve (5.19). This type of algorithm, where we update one-parameter-at-a-time, is referred to as *coordinate descent*, and we illustrate it in Example 5.3

It can be shown that the cost function in (5.19) is convex. Only convexity is not sufficient to guarantee that coordinate descent will find its (global) minimum, but for the L^1-regularized cost function (5.19) it can be shown that coordinate descent actually finds the (global) minimum. In practice we know that we have found the global minimum when no parameters have changed during a full "sweep" of the parameter vector.

It turns out that coordinate descent is a very efficient method for L^1-regularized linear regression (5.19). The keys are (i) that (5.21) exists and is cheap to compute, and (ii) many updates will simply set $\theta_j = 0$ due to the sparsity of the optimal $\widehat{\boldsymbol{\theta}}$. This makes the algorithm fast. For most machine learning optimization problems it can, however, *not* be said that coordinate descent is the preferred method. We will now have a look at some more general families of optimization methods that are widely used in machine learning.

Example 5.3: Coordinate descent

We apply coordinate descent to the objective functions from Example 5.2. For coordinate descent to be an efficient alternative in practice, closed-form solutions for updating one-parameter-at-a-time, similar to (5.21), have to be available.



The figures show how the parameters are updated in the coordinate descent algorithm, for two different initial parameter vectors (blue and green trajectory, respectively). It is clear from the figures that only one parameter is updated each time, which gives the trajectory a characteristic shape. The obtained minimum is marked with a yellow dot. Note how the different initializations lead to different (local) minima in the non-convex (right) case.

Grid search

Most often, we do not have closed-form solutions to the optimization problems we need to solve. Sometimes, we can only evaluate the objective function in certain points, and we have no knowledge abouts its derivatives. This typically happens when selecting hyperparameters, such as the regularization parameter λ in (5.19). The objective is then to minimize the prediction error for a held-out validation data set (or some other version of cross-validation). That is a very complicated objective function to write down, and it is even harder to find its derivative (in fact, this objective function includes an optimization problem itself—learning $\widehat{\boldsymbol{\theta}}$ for a given value of λ), but we can nevertheless evaluate it for any given λ (that is, run the entire learning procedure and see how good the predictions become for the validation data set).

The perhaps simplest way to solve an optimization problem, which also is the idea of grid search, is to "try a few different parameter values, and pick the one which works best". That is the idea of *grid search*. The term "grid" here refers to some (more or less arbitrary chosen) set of different parameter values to try out, and we illustrated it in Example 5.4.

Although simple to implement, grid search can be computationally quite inefficient, in particular if the parameter vector has a high dimension. As an example, having a grid with a resolution of 10 grid points per dimension (which is a very coarse-grained grid) for a 5-dimensional parameter vector requires $10^5 = 100\,000$ evaluations of the objective function. If possible, one should avoid using grid search for this reason. However, with low-dimensional hyperparameters (in L^1 and L^2 regularization, λ is 1-dimensional, for example), grid search can be feasible. We summarize grid search in Algorithm 7, where we use it for determining a regularization parameter λ.

Algorithm 7: Grid search for regularization parameter λ

Input: Training data $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$, validation data $\{\mathbf{x}_j, y_j\}_{j=1}^{n_v}$

Result: $\widehat{\lambda}$

for $\lambda = 10^{-3}, 10^{-2}, \ldots, 10^{3}$ **do**

  Learn $\widehat{\boldsymbol{\theta}}$ with regularization parameter λ from training data

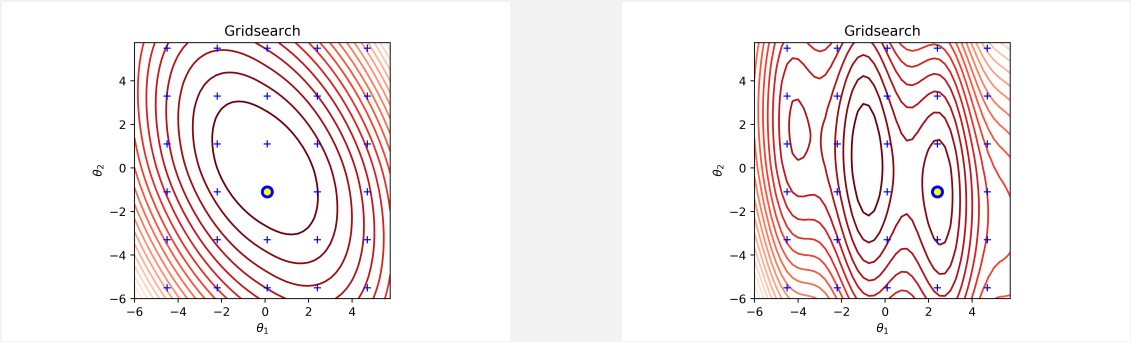
  Compute error on validation data $E_{\text{val}}(\lambda) \leftarrow \sum_{j=1}^{n_v} (\widehat{y}(\mathbf{x}_j; \widehat{\boldsymbol{\theta}}) - y_j)^2$

end

return $\widehat{\lambda}$ as $\arg\min_{\lambda} E_{\text{val}}(\lambda)$

Example 5.4: Grid search

We apply grid search to the objective functions from Example 5.2, with an arbitrary chosen grid indicated below by blue marks. The found minimum, which is the grid point with the smallest value of the objective functions, is marked with a yellow dot.



Due to the unfortunate selection of the grid, the global minimum is not found in the non-convex problem (right). That problem could be handled by increasing the resolution of the grid, which however requires more computations (more evaluations of the objective function).

Some hyperparameters (for example k in k-NN, Chapter 2) are integers, and sometimes it is feasible to simply try all reasonable integer values in grid search. However, most of the time the major challenge in grid search is to select a good grid. The grid used in Algorithm 7 is logarithmic between 0.001 and 1 000, but that is of course only an example. Of course, one could do some manual work by first selecting a coarse grid to get an initial guess, and then refine the grid only around the promising candidates, etc. In practice, if the problem has more than one dimension, it can also be useful to select the grid points randomly instead of using an equally spaced linear or logarithmic grid.

The manual procedure of choosing a grid might, however, become quite tedious, and one could wish for an automated method. That is, in fact, possible by treating the grid point selection problem as a machine learning problem itself. If we consider the points in which the objective function already has been evaluated as a training data set, we can use a regression method to learn a model for the objective function. That model can, in turn, be used to answer questions on where to evaluate the objective function next, and thereby automatically selecting the next grid point. A concrete method built from this idea is the Gaussian process optimization method, which uses Gaussian processes (Chapter 9) for learning a model of the objective function.

Gradient methods

In many situation, we do have access to more than just the value of the objective function in certain points. For example when training parameters it is most often possible to compute the derivative (or rather gradient) of the objective function, and sometimes even the second derivative (or rather Hessian). In those situations, it is often a good idea to use a *gradient descent* method, or even *Newton's method*. In certain situations Newthon's method offers a great speedup, but may also break if the circumstances are less fortunate. In practice we therefore have to modify Netwon's method, for example by using *trust regions*. We will now introduce the fundamentals of these methods now.

Gradient descent

Gradient descent is typically used for learning parameters, where the parameter vector $\boldsymbol{\theta}$ often has a high dimension and where the objective function $J(\boldsymbol{\theta})$ is simple enough such that its gradient is possible to compute. Let us therefore consider the parameter learning problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{5.22}$$

(even though gradient descent possibly can be used for hyperparameters as well). We will assume[4] that the gradient of the cost function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ exists for all $\boldsymbol{\theta}$. As an example, the gradient of the cost function for logistic regression (3.34) is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{1+e^{y_i\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}_i}}\right) y_i\mathbf{x}_i. \tag{5.23}$$

Note that $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is a vector of the same dimension as $\boldsymbol{\theta}$, which describes the direction in which $J(\boldsymbol{\theta})$ increases. Consequently, and more useful for us, $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ describes the direction in which $J(\boldsymbol{\theta})$ decreases. That is,

$$J\big(\boldsymbol{\theta} - \gamma\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\big) \leq J\big(\boldsymbol{\theta}\big) \tag{5.24}$$

for some (possibly very small) $\gamma > 0$. If $J(\boldsymbol{\theta})$ is convex, the inequality in (5.24) is strict except at the minimum (where $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is zero). This suggests that if we have $\boldsymbol{\theta}^{(t)}$ and want to select $\boldsymbol{\theta}^{(t+1)}$ such that $J(\boldsymbol{\theta}^{(t+1)}) \leq J(\boldsymbol{\theta}^{(t)})$, we should

$$\boxed{\text{update } \boldsymbol{\theta}^{(t+1)} \text{ as } \boldsymbol{\theta}^{(t)} - \gamma\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)}).} \tag{5.25}$$

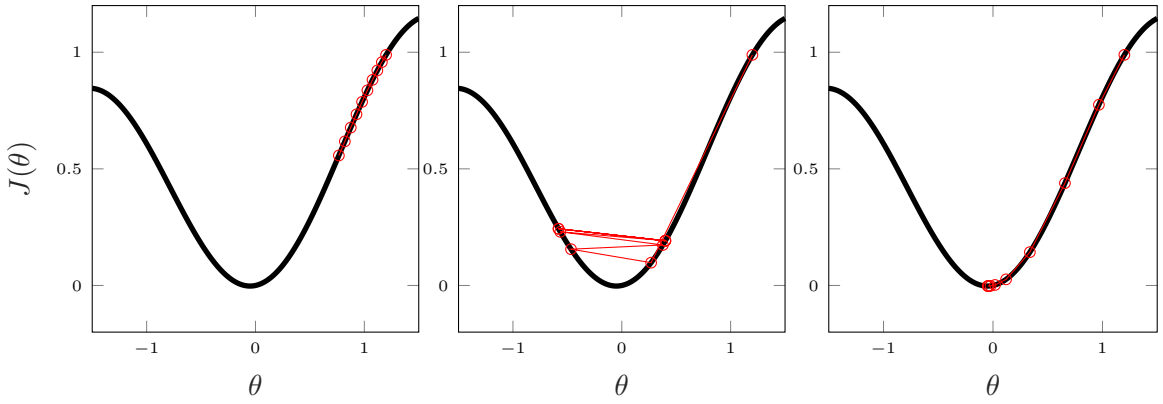
Repeating (5.25) gives the gradient descent Algorithm 8.

Algorithm 8: Gradient descent

Input: Objective function $J(\boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, learning rate γ
Result: $\widehat{\boldsymbol{\theta}}$

Set $t \leftarrow 0$
while $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\|$ *not small enough* **do**
  Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})$
  Update $t \leftarrow t + 1$
end
return $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

[4] This assumption is primarily made for the theoretical discussion. In practice, there are successful examples of gradient descent being applied to objective functions not differentiable everywhere, such as neural networks with ReLu activation functions (Chapter 6).



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

Figure 5.3: 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). Remember that a good value of γ is very much related to the shape of the cost function; $\gamma = 0.3$ might be too small (or high) for a different $J(\theta)$.

In practice, we do not know γ, which determines how big the $\boldsymbol{\theta}$-step is at each iteration. It is possible to formulate the selection of γ as an optimization problem itself to be solved at each iteration, a so-called line-search problem. Here we will consider the simpler solution where we leave the choice of γ to the user. When left as a user choice, γ is often referred to as the *learning rate* or step-size. Note that the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ will typically decrease, and eventually attain 0 at a stationary point (possibly, but not necessarily, a minimum), so Algorithm 8 may converge if γ is kept constant. This is in contrast to what you later will learn about the *stochastic* gradient algorithm.

The choice of learning rate γ is important. Some typical situations with too small, too high and a good choice of learning rate are shown in Figure 5.3. With the intuition from these figures, we advise to monitor $J(\boldsymbol{\theta}^{(t)})$ during your optimization, and

- decrease the learning rate γ if the cost function values $J(\theta^{(t)})$ are getting worse or oscillates widely (cf. Figure 5.3b),
- increase the learning rate γ if the cost function values $J(\theta^{(t)})$ are fairly constant and only slowly decreasing (cf. Figure 5.3a).

No general convergence guarantees can be given for gradient descent, basically because a bad learning rate γ may break the method. However, with the "right" choice of γ, the value of $J(\boldsymbol{\theta})$ will decrease for each iteration (cf. (5.24)) until a point with zero gradient is found, a so-called stationary point. That is, however, not necessarily a minimum, but can also be a maximum or a saddle-point of the objective function. In practice, one typically monitor the value of $J(\boldsymbol{\theta})$ and terminate the algorithm when it seems not to decrease anymore, and hope it has arrived at a minimum.

In non-convex problem with multiple local minima, we can not expect gradient descent to always find the global one either. The initialization is usually critical for determining which minimum (or stationary point) is found, as illustrated by Example 5.5. It can therefore be a good practice (if time and computational resources permits) to run the optimization multiple times with different initializations. For computationally heavy non-convex problems such as training a deep neural network (Chapter 6) when we cannot afford to re-run the training, we usually employ method-specific heuristics and tricks to find a good initialization point.

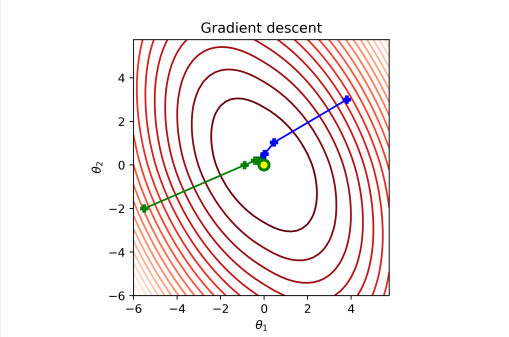
For convex problems there is only one stationary point, which also is the global minimum. Hence, the initialization for convex problem can be done arbitrarily. However, by *warm-starting* the optimization with a good initial guess, we may still save valuable computational time. Sometimes, such as when doing k-fold cross validation (Chapter 4), we have to train k models on similar (but not identical) datasets. In

situations like that we can typically make use of that by initializing Algorithm 8 with the parameters learned for the previous model.

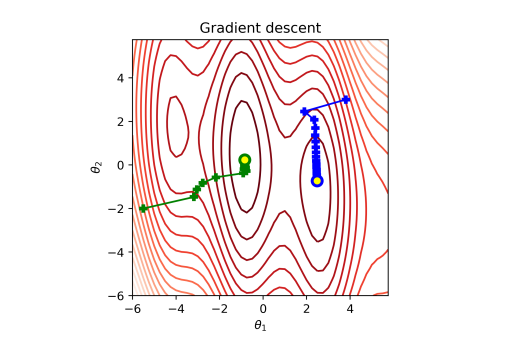
For learning logistic regression (3.35), gradient descent can be used. If we remember that its cost function is convex, we know that once gradient descent has converged to a minimum, it has reached the global minimum and we are done. However, for logistic regression, there are more advanced alternatives that usually performs even better.

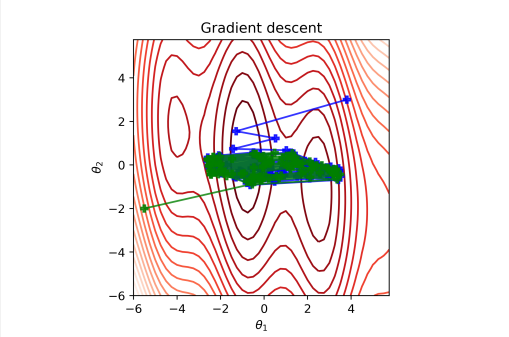
Example 5.5: Gradient descent

We first consider the convex objective function from Example 5.2, and apply gradient descent to it with a seemingly reasonable learning rate. Note that each step is perpendicular to the level curves at the point where it starts, which is a property of the gradient. As expected, we find the (global) minimum with both of the two different initializations.



For the non-convex objective function from Example 5.2 we apply gradient descent with two different learning rates. In the left plot, the learning rate seems well chosen and the optimization converges nicely, albeit to different minima depending on the initialization. Note that it *could* have converged also to one of the saddle points between the different minima. In the right plot the learning rate is too big, and the procedure does not seem to converge.





Second order methods

We can think of gradient descent as approximating $J(\boldsymbol{\theta})$ with a first order Taylor expansion around $\boldsymbol{\theta}^{(t)}$, that is, a (hyper-)plane. The next parameter $\boldsymbol{\theta}^{(t+1)}$ is selected by taking a step in the steepest direction of the (hyper-)plane. Let us now see what happen if we instead use a second order Taylor expansion,

$$J(\boldsymbol{\theta} + \mathbf{v}) \approx \underbrace{J(\boldsymbol{\theta}) + \mathbf{v}^{\mathsf{T}}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})] + \frac{1}{2}\mathbf{v}^{\mathsf{T}}[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]\mathbf{v}}_{\triangleq s(\boldsymbol{\theta}, \mathbf{v})}, \tag{5.26}$$

where $\mathbf{v}$ is a vector of same dimension as $\boldsymbol{\theta}$. This expression does not only contain the gradient of the cost function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, but also the Hessian matrix of the cost function $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Remember that we are searching for the minimum of $J(\boldsymbol{\theta})$. If the Hessian $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is positive definite, that minimum is obtained where the derivative of $s(\boldsymbol{\theta}, \mathbf{v})$ is zero,

$$\frac{\partial}{\partial \mathbf{v}} s(\boldsymbol{\theta}, \mathbf{v}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]\mathbf{v} = 0 \Leftrightarrow \mathbf{v} = -[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]. \tag{5.27}$$

This suggests to update

$$\boldsymbol{\theta}^{(t+1)} \text{ as } \boldsymbol{\theta}^{(t)} - [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})], \tag{5.28}$$

which is *Netwton's method* for minimization. Unfortunately, no general convergence guarantees can be given for Newton's method either. For certain cases, Newton's method can be much faster than gradient descent. In fact, *if* the cost function $J(\boldsymbol{\theta})$ is a quadratic function in $\boldsymbol{\theta}$ then (5.26) is exact and Newton's method (5.28) will find the optimum in only one iteration! Quadratic objective functions are, however, rare in machine learning[5]. It is not even guaranteed that the Hessian $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ always is positive definite in practice, which may result in rather strange parameter updates in (5.28). To still make use of the potentially valuable second order information, but at the same time also have a robust and practically useful algorithm, we have to introduce some modification of Newton's method. There are multiple options, and we will look at so-called *trust regions*.

We derived Newton's method using the second order Taylor expansion (5.26) as a model for how $J(\boldsymbol{\theta})$ behave around $\boldsymbol{\theta}^{(t)}$. We should perhaps not trust the Taylor expansion to be a good model for *all* values of $\boldsymbol{\theta}$, but only in the vicinity of $\boldsymbol{\theta}^{(t)}$. One natural restriction is to trust the second order Taylor expansion (5.26) only within a ball of radius D around $\boldsymbol{\theta}^{(t)}$, which we refer to as our trust region. This suggests that we could make a Newton update (5.28) of the parameters, unless the step is longer than D, in which case we downscale the step to never leave our trust region. In the next iteration, the trust region is moved to be centered around the updated $\boldsymbol{\theta}^{(t+1)}$, and another step is taken from there. We can express this as

$$\boxed{\text{update } \boldsymbol{\theta}^{(t+1)} \text{ as } \boldsymbol{\theta}^{(t)} - \eta[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})],} \tag{5.29}$$

where $\eta \leq 1$ is chosen as large as possible such that $\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\| \leq D$. The radius of the trust region D can be updated and adapted as the optimization proceeds, but for simplicity we consider here D to be a user choice (much like the step size for gradient descent). We summarize this by Algorithm 9. The trust-region Newton method, with a certain set of rules on how to update D is one of the methods commonly used for training logistic regression in practice.

It can be computationally expensive or even impossible to compute the inverse of the Hessian matrix $[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}$. To this end, there is an entire class of methods called *quasi-Newton methods* that all use different ways to *approximate* the inverse of the Hessian matrix $[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]^{-1}$ in (5.28). This class includes, among other, the Broyden method and the BFGS method (an abbreviation of Broyden, Fletcher, Goldfarb and Shanno). A further approximation of the latter, called limited-memory BFGS or L-BFGS, has proven to be another good choice for the logistic regression problem.

[5] For regression, we often use the squared error loss $L(\widehat{y}, y) = (\widehat{y} - y)^2$, which indeed is a quadratic function in $\widehat{y}$. That does *not* imply that $J(\boldsymbol{\theta})$ (the objective function) is a quadratic function in $\boldsymbol{\theta}$.

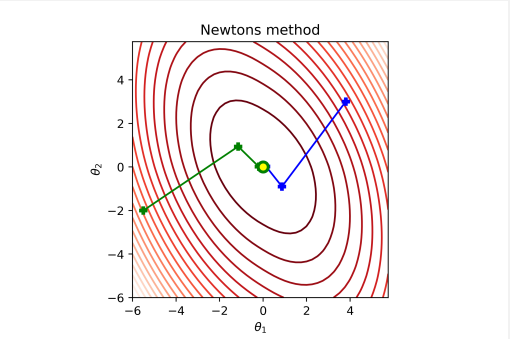
Algorithm 9: Truncated Newton's method

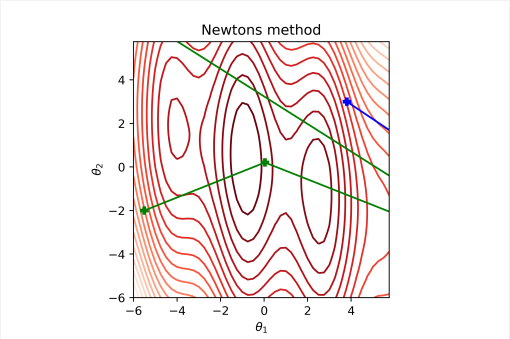
Input: Objective function $J(\boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, trust region radius D
Result: $\widehat{\boldsymbol{\theta}}$

Set $t \leftarrow 0$
while $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\|$ *not small enough* **do**
  Compute $\mathbf{v} \leftarrow [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1} [\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]$
  Compute $\eta \leftarrow \frac{D}{\max(\|\mathbf{v}\|, D)}$
  Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \mathbf{v}$
  Update $t \leftarrow t + 1$
end
return $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

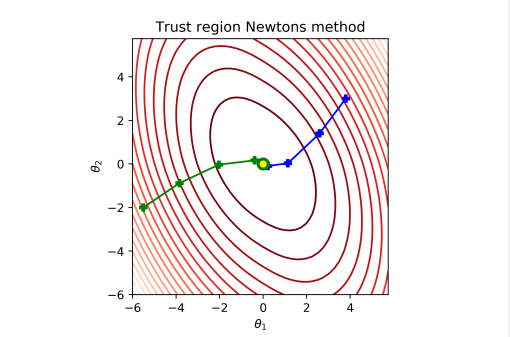
Example 5.6: Newton's method

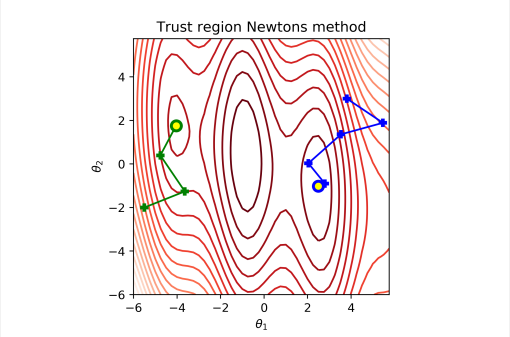
We first apply Newton's method to the cost functions from Example 5.2. Since the convex cost (left) function also happens to be close to a quadratic function, the Newton's method works well and finds, for both initializations, the minimum in only two iterations. For the non-convex problem (right), Newton's method diverges for both initializations, since the second order Taylor expansion (5.26) is a poor approximation of this function and leads the method wrong.





We apply also the trust-region Newton's method to both problems. Note that the first step direction is identical to the non-truncated version above, but the steps are now limited to stay within the trust region (here a circle of radius 2). This prevents the severe divergence problems for the non-convex case, and all cases converges nicely. Indeed, the convex case (left) requires more iterations than for the non-truncated version above, but that is a price we have to pay in order to have a method which also works for the non-convex case shown to the right.





5.4 Optimization with large datasets

In machine learning the training data may have n = millions (or more) of data samples, the so-called *big data* problem. Computing, for example, the gradient of the cost function

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}) \tag{5.30}$$

thus involves summing a million of elements. Besides taking lot of time to sum, it can also be an issue to keep all data samples in the computer memory at the same time. However, with so many data samples it is usually the case that many of them are similar, and in practice we do not always need to consider them all at once, but it might be sufficient to have a look at only a subset of them. This is a general idea called *subsampling*, and we will have a closer look at how subsampling can be combined with gradient descent into a very useful optimization method called *stochastic gradient*. It is, however, possible to combine the subsampling idea also with other methods.

With n very big, we can expect the gradient computed only for the first half of the dataset $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{i=1}^{n/2} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$ to be almost identical to the gradient based on the second half of the dataset $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{i=n/2+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$. Consequently, it might be a waste of time to compute the gradient based on the whole training data set for each iteration of gradient descent. Instead, we could compute the gradient based on the first half of the training data set, update the parameters according to the gradient descent method Algorithm 8, and then compute the gradient for the new parameters based on the second half of the training data,

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \frac{1}{n/2} \sum_{i=1}^{\frac{n}{2}} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}^{(t)}), \tag{5.31a}$$

$$\boldsymbol{\theta}^{(t+2)} = \boldsymbol{\theta}^{(t+1)} - \gamma \frac{1}{n/2} \sum_{i=\frac{n}{2}+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}^{(t+1)}). \tag{5.31b}$$

In other words, we use only a *subsample* of the training when we compute the gradient. These two steps would only require roughly half the computational time compared to using the entire training data set for each gradient computation, and this computational saving illustrates well the benefit of the subsampling idea.

The extreme version of this strategy would be to use only one single data sample each time when computing the gradient. However, most commonly we do something in between, using more than one but not all data samples when computing the gradient. We call this small subsample of data a *mini-batch*, which typically can contain $n_b = 10$, $n_b = 100$ or $n_b = 1000$ data samples. One complete pass through the training data is called an *epoch*, and consists consequently of n/n_b iterations.

When using mini-batches it is important to ensure 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 output classes and the dataset is sorted with respect to the output, the mini-batch with the 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, the mini-batches should be formed randomly. One implementation of this is to first randomly shuffle the training data, and thereafter dividing it into mini-batches in an ordered manner. When we have completed one epoch, we do another random reshuffling of the training data and do another pass through the data set. We summarize gradient descent with, *stochastic gradient*, by Algorithm 10.

Stochastic gradient is widely used in machine learning, and there are many extensions tailored for different methods. For training deep neural networks (Chapter 6), some commonly used methods include automatic adaption of the learning rate and an idea called momentum to counteract the subsampling noise. The AdaGrad (short for adaptive gradient), RMSProp (short for root mean square propagation) and Adam (short for adaptive moments) methods are such examples. For logistic regression in the big data setting, the stochastic average gradient (SAG) method, which averages over all previous gradient estimates, has proven useful, to only mention a few.

Algorithm 10: Stochastic gradient

Input: Objective function $J(\boldsymbol{\theta}) = \sum_{i=1}^{n} L(\mathbf{x}_i, y_i; \boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, learning rate $\gamma^{(t)}$
Result: $\widehat{\boldsymbol{\theta}}$

Set $t \leftarrow 0$
while *Convergence criteria not met* **do**
  for $i = 1, 2, \ldots, E$ **do**
    Randomly shuffle the training data $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$
    for $j = 1, 2, \ldots, \frac{n}{n_b}$ **do**
      Approximate the gradient using the mini-batch $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=(j-1)n_b+1}^{jn_b}$,
      $\widehat{\mathbf{d}}^{(t)} = \frac{1}{n_b} \sum_{i=(j-1)n_b+1}^{jn_b} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}^{(t)})$.
      Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma^{(t)} \mathbf{d}^{(t)}$
      Update $t \leftarrow t + 1$
    end
  end
end
return $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

Learning rate and convergence for stochastic gradient

Standard gradient descent converges if the learning rate is constant, since the gradient itself is zero at the minimum (or any other stationary point). For stochastic gradient, on the other hand, we do *not* obtain convergence if we use a constant learning rate. The reason is that we only use an *estimate* of the true gradient, and this estimate will not necessarily be zero at the minimum of the objective function, but there might still be a considerable amount of "noise" in the gradient estimate due to the subsampling. As a consequence, the stochastic gradient algorithm with a fixed learning rate will not converge towards a point, but walk around somewhat randomly. However, if we do not use a constant learning rate, but gradually decrease it towards zero, the parameter updates will be smaller and smaller, and eventually converge. We hence start at $t = 0$ with a fairly high learning rate $\gamma^{(t)}$ (meaning that we take big steps) and then decay $\gamma^{(t)}$ as t increases. Under certain regularity conditions of the cost function and with a learning rate fulfilling the Robbins-Monro conditions $\sum_{t=0}^{\infty} \gamma^{(t)} = \infty$ and $\sum_{t=0}^{\infty} (\gamma^{(t)})^2 < \infty$, the stochastic gradient algorithm can be shown to converge almost surely to a local minimum. The Robbins-Monro conditions are, for example, fulfilled if using $\gamma^{(t)} = \frac{1}{t}$.

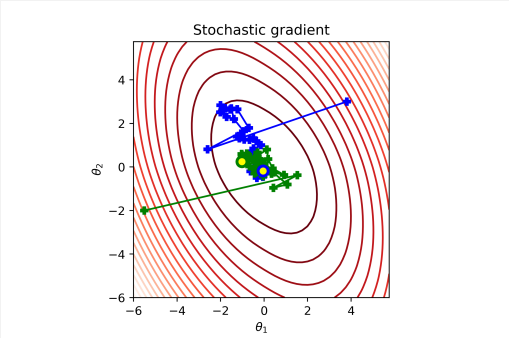
For many machine learning problems, however, it has been found that better performance in practice often is obtained if not letting $\gamma^{(t)} \to 0$, but to cap it at some small value $\gamma_{\text{min}} > 0$. This will cause stochastic gradient not to exactly converge, but the algorithm will in fact walk around indefinitely (theoretically, in practice until the algorithm is aborted by the user). For practical purposes this seemingly undesired property does usually not cause any major issue if γ_{min} is only small enough, and one possible rule for setting the learning rate in practice is

$$\gamma^{(t)} = \gamma_{\text{min}} + (\gamma_{\text{max}} - \gamma_{\text{min}}) e^{-\frac{t}{\tau}}. \tag{5.32}$$

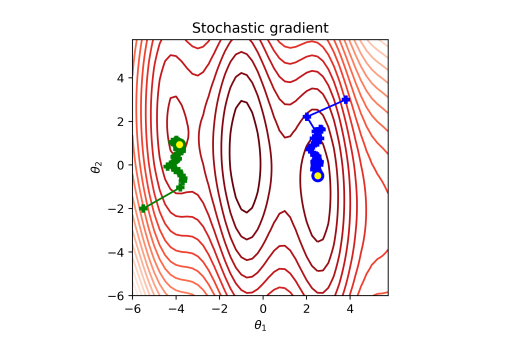
Now the learning rate $\gamma^{(t)}$ starts at γ_{max} and goes to γ_{min} as $t \to \infty$. How to pick the parameters γ_{min}, γ_{max} and τ is more of an art than science. As a rule of thumb γ_{min} can be chosen approximately as 1% of γ_{max}. The parameter τ depends on the size of the dataset and 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 done by monitoring the cost function as for standard gradient descent in Figure 5.3.

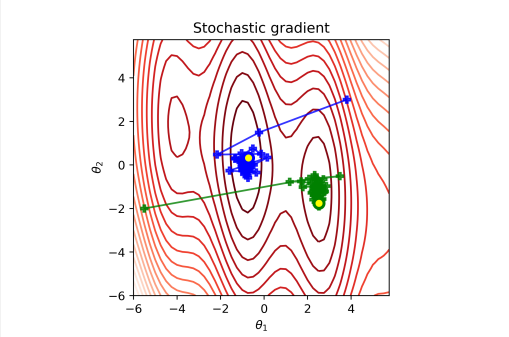
Example 5.7: Optimization

We apply the stochastic gradient method to the objective functions from Example 5.2. For the convex function below, the choice of learning rate is not very crucial. Note, however, that the algorithm does not converge as nicely as, for example, gradient descent, due to the "noise" in the gradient estimate caused by the subsampling. This is the price we have to pay for the substantial computational savings offered by the subsampling.



For the objective function with multiple local minima, we apply stochastic gradient with two decaying learning rates, but with different initial $\gamma^{(0)}$. With a smaller learning rate, left, stochastic gradient converges to the closest minima, whereas it converges to other minima with a bigger initial learning rate (right).





5.5 Further reading

Draft (*June* 17, 2020) *of Supervised Machine Learning. Feedback and exercise problems*: `http://smlbook.org`

6 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 a lot of training data, new software, hardware and parallel algorithms for training, deep learning has made a major contribution to machine learning and several other fields. 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 6.1 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 6.1. In Section 6.2 we present a special neural network tailored for images and finally we look into some details on how to train neural networks in Section 6.3.

6.1 Neural networks

A neural network is a nonlinear function that describes a prediction of the output $\widehat{y}$ as a nonlinear function of its input variables

$$\widehat{y} = f_{\boldsymbol{\theta}}(x_1, \ldots, x_p), \tag{6.1}$$

where 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.

Generalized linear regression

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

$$\widehat{y} = W_1 x_1 + W_2 x_2 + \cdots + W_p x_p + b, \tag{6.2}$$

which is shown in Figure 6.1a. Each input variable x_j is represented with a node and each parameter W_j with a link. Furthermore, the output z is described as the sum of all terms $W_j x_j$. Note that we use 1 as the input variable corresponding to the offset term b.

To describe *nonlinear* relationships between $\mathbf{x} = [1 \; x_1 \; x_2 \; \ldots \; x_p]^\mathsf{T}$ and $\widehat{y}$ we introduce a nonlinear scalar function called the *activation function* $h : \mathbb{R} \to \mathbb{R}$. The linear regression model (6.2) is now modified into a *generalized* linear regresssion model where the linear combination of the inputs is squashed through the (scalar) activation function

$$\widehat{y} = h\big(W_1 x_1 + W_2 x_2 + \cdots + W_p x_p + b\big). \tag{6.3}$$

This extension to the generalized linear regression model is visualized in Figure 6.1b.

Common choices for activation function are the *logistic function* and the *rectified linear unit* (ReLU). These are illustrated in Figure 6.2a and Figure 6.2b, respectively. The logistic (or sigmoid) function has already been used in the context of logistic regression (Section 3.2). The logistic function is linear close to

(a) **(b)**

Figure 6.1: Graphical illustration of a linear regression model (Figure 6.1a), and a generalized linear regression model (Figure 6.1b). In Figure 6.1a, the output z is described as the sum of all terms b and $\{W_j x_j\}_{j=1}^{p}$, as in (6.2). In Figure 6.1b, the circle denotes addition and also transformation through the activation function h, as in (6.3).

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

(a) **(b)**

Figure 6.2: Two common activation functions used in neural networks. The logistic (or sigmoid) function (Figure 6.2a), and the rectified linear unit (Figure 6.2b).

$z = 0$ and saturates at 0 and 1 as z 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 is now the standard choice (despite its simplicity!) in most neural network models.

The generalized linear regression model (6.3) is very simple and is itself not capable of describing very complicated relationships between the input $\mathbf{x}$ and the output $\widehat{y}$. 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*).

Two-layer neural network

In (6.3), 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 U such generalized linear regression models, each of which has its own parameters. The parameter for the ith regression model are $b_i, \ldots, W_{ip}$ and we denote its output by q_i,

$$q_i = h\left(W_{i1}x_1 + W_{i2}x_2 + \cdots + W_{ip}x_p + b_i\right), \qquad i = 1, \ldots, U. \tag{6.4}$$

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

$$\widehat{y} = W_1 q_1 + W_2 q_2 + \cdots + W_U q_U + b. \tag{6.5}$$

To distinguish the parameters in (6.4) and (6.5) we add the superscripts (1) and (2), respectively. The equations describing this two-layer neural network (or equivalently, a neural network with one layer of

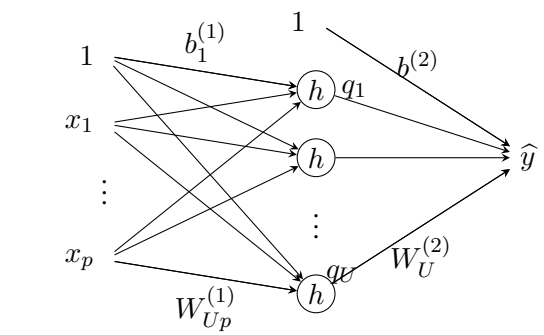


Figure 6.3: A two-layer neural network, or equivalently, a neural network with one intermediate layer of hidden units.

hidden units) are thus

$$\begin{aligned} q_1 &= h\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + \cdots + W_{1p}^{(1)}x_p + b_1^{(1)}\right), \\ q_2 &= h\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + \cdots + W_{2p}^{(1)}x_p + b_2^{(1)}\right), \\ &\vdots \\ q_U &= h\left(W_{U1}^{(1)}x_1 + W_{U2}^{(1)}x_2 + \cdots + W_{Up}^{(1)}x_p + b_U^{(1)}\right), \end{aligned} \tag{6.6a}$$

$$\widehat{y} = W_1^{(2)}q_1 + W_2^{(2)}q_2 + \cdots + W_U^{(2)}q_U + b^{(2)}. \tag{6.6b}$$

Extending the graphical illustration from Figure 6.1, this model can be depicted as a graph with two layers of links (illustrated using arrows), see Figure 6.3. 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.

Matrix notation

The two-layer neural network model in (6.6) 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

$$\mathbf{W}^{(1)} = \begin{bmatrix} W_{11}^{(1)} & \cdots & W_{1p}^{(1)} \\ \vdots & & \vdots \\ W_{U1}^{(1)} & \cdots & W_{Up}^{(1)} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_U^{(1)} \end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix} W_1^{(2)} & \cdots & W_U^{(2)} \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} b^{(2)} \end{bmatrix}. \tag{6.7}$$

The full model can then be written as

$$\mathbf{q} = h\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right), \tag{6.8a}$$

$$\widehat{y} = \mathbf{W}^{(2)}\mathbf{q} + \mathbf{b}^{(2)}, \tag{6.8b}$$

where we have also stacked the components in $\mathbf{x}$ and $\mathbf{q}$ as $\mathbf{x} = [x_1, \ldots, x_p]^\mathsf{T}$ and $\mathbf{q} = [q_1, \ldots, q_U]^\mathsf{T}$. The activation function h acts element-wise. The two weight matrices and the two offset vectors will be the parameters in the model, which can be written as

$$\boldsymbol{\theta} = \begin{bmatrix} \mathrm{vec}(\mathbf{W}^{(1)})^\mathsf{T} & \mathrm{vec}(\mathbf{W}^{(2)})^\mathsf{T} & \mathbf{b}^{(1)\mathsf{T}} & \mathbf{b}^{(2)\mathsf{T}} \end{bmatrix}^\mathsf{T}. \tag{6.9}$$

[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.

By this we have described a nonlinear regression model on the form $\widehat{y} = f(\mathbf{x}; \boldsymbol{\theta})$ according to above. Note that the predicted output $\widehat{y}$ in (6.8b) depends on all the parameters in $\boldsymbol{\theta}$ even though it is not explicitly stated in the notation.

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 $\mathbf{q}^{(l-1)}$. Each such layer consists of U_l hidden units $\mathbf{q}^{(l)} = [q_1^{(l)}, \ldots, q_{U_l}^{(l)}]^\mathsf{T}$, where the dimensions U_1, U_2, ... can be different across the various layers.

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

$$\mathbf{q}^{(l)} = h(\mathbf{W}^{(l)}\mathbf{q}^{(l-1)} + \mathbf{b}^{(l)}). \tag{6.10}$$

This means that the layers are stacked such that the output of the first layer $\mathbf{q}^{(1)}$ (the first layer of hidden units) is the input to the second layer, the output of the second layer $\mathbf{q}^{(2)}$ (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. (6.8))

$$\begin{aligned}
\mathbf{q}^{(1)} &= h(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}),\\
\mathbf{q}^{(2)} &= h(\mathbf{W}^{(2)}\mathbf{q}^{(1)} + \mathbf{b}^{(2)}),\\
&\vdots\\
\mathbf{q}^{(L-1)} &= h(\mathbf{W}^{(L-1)}\mathbf{q}^{(L-2)} + \mathbf{b}^{(L-1)}),\\
\widehat{\mathbf{y}} &= \mathbf{W}^{(L)}\mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}.
\end{aligned} \tag{6.11}$$

A graphical representation of this model is provided in Figure 6.4.

The weight matrix $\mathbf{W}^{(1)}$ for the first layer $l = 1$ has the dimension $U_1 \times p$ and the corresponding offset vector $\mathbf{b}^{(1)}$ has the dimension U_1. In deep learning it is common to consider applications where also the output is multi-dimensional $\widehat{\mathbf{y}} = [\widehat{y}_1, \ldots, \widehat{y}_M]^\mathsf{T}$. This means that for the last layer the weight matrix $\mathbf{W}^{(L)}$ has the dimension $M \times U_{L-1}$ and the offset vector $\mathbf{b}^{(L)}$ has the dimension M. For all intermediate layers $l = 2, \ldots, L-1$, $\mathbf{W}^{(l)}$ has the dimension $U_l \times U_{l-1}$ and the corresponding offset vector U_l.

The number of inputs p and the number of outputs M are given by the problem, but the number of layers L and the dimensions U_1, U_2, ... are user design choices that will determine the flexibility of the model.

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

$$\boldsymbol{\theta} = \begin{bmatrix} \mathrm{vec}(\mathbf{W}^{(1)})^\mathsf{T} & \mathrm{vec}(\mathbf{W}^{(2)})^\mathsf{T} & \cdots \mathrm{vec}(\mathbf{W}^{(L)})^\mathsf{T} & \mathbf{b}^{(1)\mathsf{T}} & \mathbf{b}^{(2)\mathsf{T}} & \cdots & \mathbf{b}^{(L)\mathsf{T}} \end{bmatrix}^\mathsf{T} \tag{6.12}$$

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. **??**), but there are also other techniques specific to deep learning; see further Section 6.3. Furthermore,

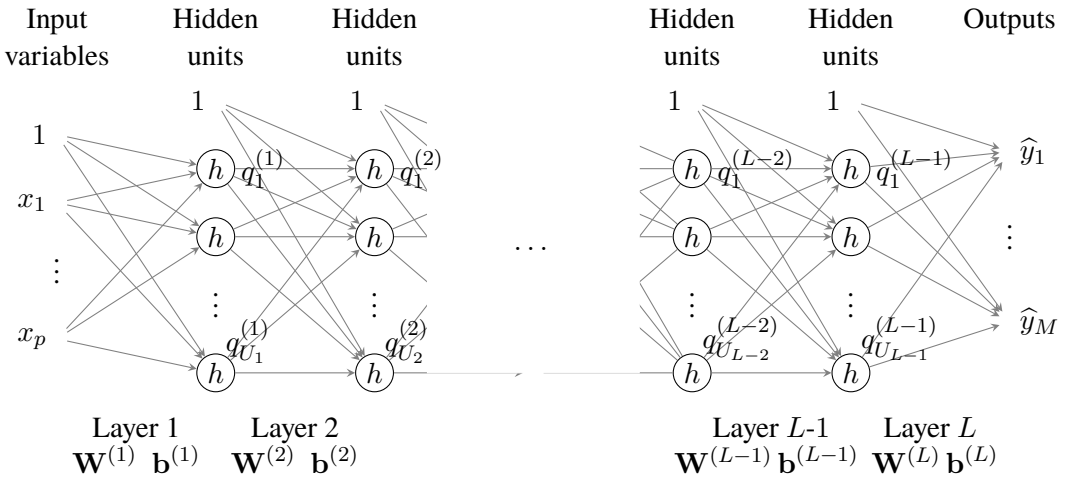


Figure 6.4: A deep neural network with L layers. Each layer l is parameterized by $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$.

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 $\varepsilon \sim \mathcal{N}(0, h_\varepsilon^2)$, and thereby obtain the square error loss function as in Section 1,

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}), \qquad \text{where} \qquad L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) = \|\mathbf{y}_i - f(\mathbf{x}_i; \boldsymbol{\theta})\|^2 = \|\mathbf{y}_i - \widehat{\mathbf{y}}_i\|^2. \tag{6.13}$$

This problem can be solved with numerical optimization, and more precisely stochastic gradient. This is described in more detail in Section 6.3.

From the parameters $\boldsymbol{\theta}$ and inputs $\{\mathbf{x}_i\}_{i=1}^n$ we can compute the predicted outputs $\{\widehat{\mathbf{y}}_i\}_{i=1}^n$ using the model $\widehat{\mathbf{y}}_i = f(\mathbf{x}_i; \boldsymbol{\theta})$. For example, for the two-layer neural network presented in Section 6.1 we have

$$\mathbf{q}_i^\mathsf{T} = h(\mathbf{x}_i^\mathsf{T}\mathbf{W}^{(1)\mathsf{T}} + \mathbf{b}^{(1)\mathsf{T}}), \tag{6.14a}$$

$$\widehat{\mathbf{y}}_i^\mathsf{T} = \mathbf{q}_i^\mathsf{T}\mathbf{W}^{(2)\mathsf{T}} + \mathbf{b}^{(2)\mathsf{T}}, \qquad i = 1, \ldots, n \tag{6.14b}$$

In (6.14) the equations are transposed in comparison to the model in (6.8). This is a small trick such that we easily can extend (6.14) to include multiple data points i. Similar to (3.5) we stack all data points in matrices, where each data point represents one row

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}_1^\mathsf{T} \\ \vdots \\ \mathbf{y}_n^\mathsf{T} \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix}, \quad \widehat{\mathbf{Y}} = \begin{bmatrix} \widehat{\mathbf{y}}_1^\mathsf{T} \\ \vdots \\ \widehat{\mathbf{y}}_n^\mathsf{T} \end{bmatrix}, \quad \text{and} \quad \mathbf{Q} = \begin{bmatrix} \mathbf{q}_1^\mathsf{T} \\ \vdots \\ \mathbf{q}_n^\mathsf{T} \end{bmatrix}. \tag{6.15}$$

We can then conveniently write (6.14) as

$$\mathbf{Q} = h(\mathbf{X}\mathbf{W}^{(1)\mathsf{T}} + \mathbf{b}^{(1)\mathsf{T}}), \tag{6.16a}$$

$$\widehat{\mathbf{Y}} = \mathbf{Q}\mathbf{W}^{(2)\mathsf{T}} + \mathbf{b}^{(2)\mathsf{T}}, \tag{6.16b}$$

where we have also stacked the predicted output and the hidden units in matrices. Note that the transposed offset vectors $\mathbf{b}_1^\mathsf{T}$ and $\mathbf{b}_2^\mathsf{T}$ are added and broadcasted to each row in this notation.

The vectorized equations in (6.16) is also how the model would typically be implemented in languages that support array programming. For the implementation you might want to consider using the transposed version of $\mathbf{W}$ and $\mathbf{b}$ as your weight matrix and offset vector to avoid taking transpose in each layer.

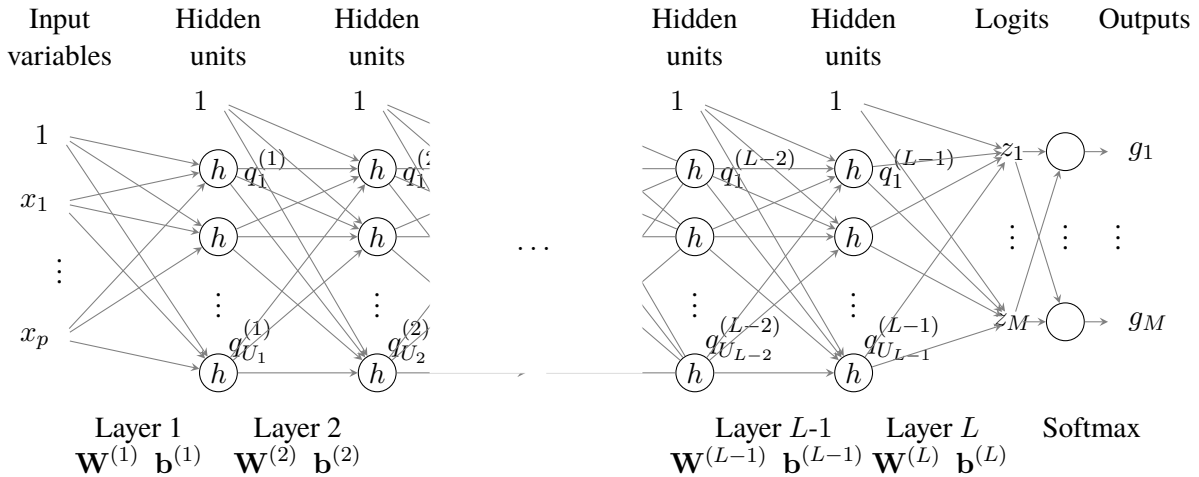


Figure 6.5: A deep neural network with L layers for classification. The only difference to regression (Figure 6.4) is the softmax transformation after layer L.

Neural networks for classification

Neural networks can also be used for classification where we have qualitative outputs $y \in \{1, \ldots, M\}$ instead of quantitative. In Section 3.2 we extended linear regression to logistic regression by simply adding the logistic function to the output. In the same manner we can extend the neural network presented in the previous 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, and more specifically the softmax parametrization given in (3.41), repeated here for convenience

$$\text{softmax}(\mathbf{z}) \triangleq \frac{1}{\sum_{j=1}^{M} e^{z_j}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_M} \end{bmatrix}. \tag{6.17}$$

The softmax function now becomes an additional activation function acting on the final layer of the neural network. In addition to the regression network in (6.11) we add the softmax function at the end of the network as

$$\mathbf{q}^{(1)} = h(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}), \tag{6.18a}$$

$$\vdots$$

$$\mathbf{q}^{(L-1)} = h(\mathbf{W}^{(L-1)}\mathbf{q}^{(L-2)} + \mathbf{b}^{(1)}), \tag{6.18b}$$

$$\mathbf{z} = \mathbf{W}^{(L)}\mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}, \tag{6.18c}$$

$$\mathbf{g} = \text{softmax}(\mathbf{z}). \tag{6.18d}$$

The softmax function maps the output of the last layer $\mathbf{z} = [z_1, \ldots, z_M]^\mathsf{T}$ to $\mathbf{g} = [g_1, \ldots, g_M]^\mathsf{T}$ where g_m is a model of the class probability $p(y = m \mid \mathbf{x}_i)$, see also Figure 6.5 for a graphical illustration. The inputs to the softmax function, i.e. the variables $z_1, \ldots, z_M$, are referred to as *logits*.

Note that the softmax function does not come as a layer with additional parameters, it merely acts as a transformation from the output into the modeled class probabilities. By construction, the outputs of the softmax function will always be in the interval $g_m \in [0, 1]$ and sum to $\sum_{m=1}^{M} g_m = 1$, otherwise they could not be interpreted as probabilities.

	$m=1$	$m=2$	$m=3$
y_{im}	0	**1**	0
$g_{im}(\boldsymbol{\theta}_A)$	0.1	**0.8**	0.1

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

	$m=1$	$m=2$	$m=3$
y_{im}	0	**1**	0
$g_{im}(\boldsymbol{\theta}_B)$	0.8	**0.1**	0.1

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

Figure 6.6: Illustration of the cross-entropy between a data point $\mathbf{y}_i$ and two different prediction outputs $\mathbf{g}_i(\boldsymbol{\theta}_A) = [g_{i1}(\boldsymbol{\theta}_A), g_{i2}(\boldsymbol{\theta}_A), g_{i3}(\boldsymbol{\theta}_A)]$ and $\mathbf{g}_i(\boldsymbol{\theta}_B) = [g_{i1}(\boldsymbol{\theta}_B), g_{i2}(\boldsymbol{\theta}_B), g_{i3}(\boldsymbol{\theta}_B)]$.

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 one-hot encoding scheme for the output $\mathbf{y}_i$. This means that for a problem with M different classes, $\mathbf{y}_i$ consists of M elements $\mathbf{y}_i = \begin{bmatrix} y_{i1} & \dots & y_{iM} \end{bmatrix}^\mathsf{T}$. If a data point i belongs to class m then $y_{im} = 1$ and $y_{ij} = 0$ for all $j \neq m$. See more about the one-hot encoding in Section 3.

For a neural network which has the softmax activation function in 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. (3.44))

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}), \qquad \text{where} \qquad L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) = -\sum_{m=1}^{M} y_{im} \ln p(y = m \,|\, \mathbf{x}_i; \boldsymbol{\theta}). \tag{6.19}$$

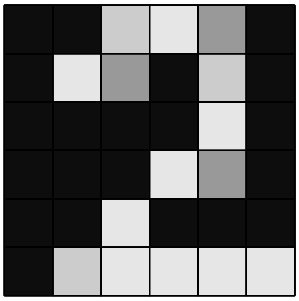
To motivate the use of this cost function, we note that cross-entropy is close to its minimum if the predicted probability $p(m \,|\, \mathbf{x}_i; \boldsymbol{\theta})$ is close to 1 for the class m for which $y_{im} = 1$. For example, if the ith data point belongs to class $m = 2$ out of a total of $M = 3$ classes we have $\mathbf{y}_i = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\mathsf{T}$. Assume that we have a set of parameters for the network that we denote by $\boldsymbol{\theta}_A$, and with these parameters we predict $p(y = 1 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.1$, $p(y = 2 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.8$ and $p(y = 3 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.1$ indicating that we are quite sure that data point i actually belongs to class $m = 2$. This would generate a low cross-entropy $L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_A) = -(0 \cdot \ln 0.1 + 1 \cdot \ln 0.8 + 0 \cdot \ln 0.1) \approx 0.22$. If we instead for another set of parameters $\boldsymbol{\theta}_B$ predict $p(y = 1 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_B) = 0.8$, $p(y = 2 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_B) = 0.1$ and $p(y = 3 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_B) = 0.1$, the cross-entropy would be much higher $L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}_B) = -(0 \cdot \ln 0.8 + 1 \cdot \ln 0.1 + 0 \cdot \ln 0.1) \approx 2.30$. For this case, we would indeed prefer the parameters $\boldsymbol{\theta}_A$ over $\boldsymbol{\theta}_B$. The above reasoning is summarized in Figure 6.6.

Computing the loss function explicitly via the logarithm could lead to numerical problems when $p(y = m \,|\, \mathbf{x}_i; \boldsymbol{\theta})$ is close to zero since $\ln(x) \to -\infty$ as $x \to 0$. This can be avoided since the logarithm in the cross-entropy loss function (6.19) can "undo" the exponential in the softmax function (6.17),

$$\begin{aligned} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}) &= -\sum_{m=1}^{M} y_{im} \ln p(y = m \,|\, \mathbf{x}_i; \boldsymbol{\theta}) = -\sum_{m=1}^{M} y_{im} \ln g_{im} \\ &= -\sum_{m=1}^{M} y_{im} \left(z_{im} - \ln \left\{ \textstyle\sum_{j=1}^{M} e^{z_{ij}} \right\} \right), \qquad (6.20) \\ &= -\sum_{m=1}^{M} y_{im} \left(z_{im} - \max_j z_{ij} - \ln \left\{ \textstyle\sum_{j=1}^{M} e^{z_{ij} - \max_j z_{ij}} \right\} \right), \qquad (6.21) \end{aligned}$$

where z_{im} denote the logits.

Image



Data representation

0.0	0.0	0.8	0.9	0.6	0.0
0.0	0.9	0.6	0.0	0.8	0.0
0.0	0.0	0.0	0.0	0.9	0.0
0.0	0.0	0.0	0.9	0.6	0.0
0.0	0.0	0.9	0.0	0.0	0.0
0.0	0.8	0.9	0.9	0.9	0.9

Input variables

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$

Figure 6.7: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number encoding 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 $\mathbf{X}$).

6.2 Convolutional neural networks

Convolutional neural networks (CNN) are a special kind of neural networks originally 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.

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 6.7 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 6.7.

If we put all input variables representing the image pixels in a long vector, we can use the network architecture presented in Section 6.1 and 6.1. 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 typically 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.

The convolutional layer

Following the input layer, we use a hidden layer with as many hidden units as there are 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 6.8a.

The network layers presented in earlier sections (like the one in Figure 6.3) have been *dense layers*. This means that each input variable is connected to all hidden units in the subsequent layer, and each such connection has a unique parameter W_{jk} associated to it. These layers have empirically been found to 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.

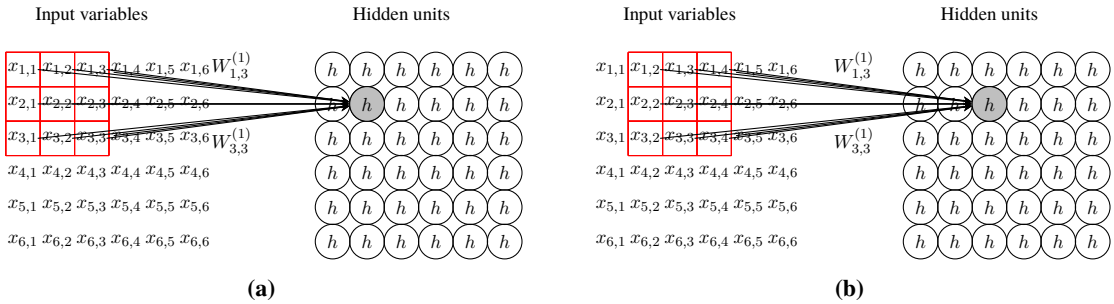


Figure 6.8: An illustration of the interactions in a convolutional layer: Each hidden unit (circle) is only dependent on the pixels in a small region of the image (red boxes), here of size 3×3 pixels. The location of the hidden unit corresponds to the location of the region in the image: if we move to a hidden unit one step to the right, the corresponding region in the image also moves one step to the right, compare Figure 6.8a and Figure 6.8b. Furthermore, the nine parameters $W^{(1)}_{1,1}, W^{(1)}_{1,2}, \ldots, W^{(1)}_{3,3}$ are the *same* for all hidden units in the layer.

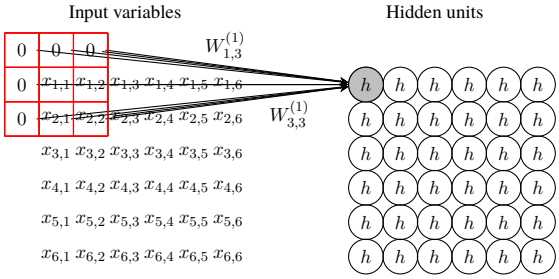


Figure 6.9: An illustration of zero-padding used when the region is partly outside the image. With zero-padding, the size of the image can be preserved in the following layer.

Sparse interactions

With sparse interactions we mean that most of the parameters in a corresponding 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 6.8 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 6.8a and Figure 6.8b. 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 simply replaced with zeros. Zero-padding is illustrated in Figure 6.9.

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 6.8a we use the same set of parameters to map the 3×3 region of pixels to the hidden unit as we do in Figure 6.8b. Instead of learning separate sets of parameters for every position we only learn one 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 *filter*. The mapping between the input variables and the hidden units can be interpreted as a convolution between the input variables and the filter, 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 filter 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 significantly fewer parameters compared to the corresponding dense layer. In Figure 6.8 only $3 \cdot 3 + 1 = 10$ parameters are required (including the

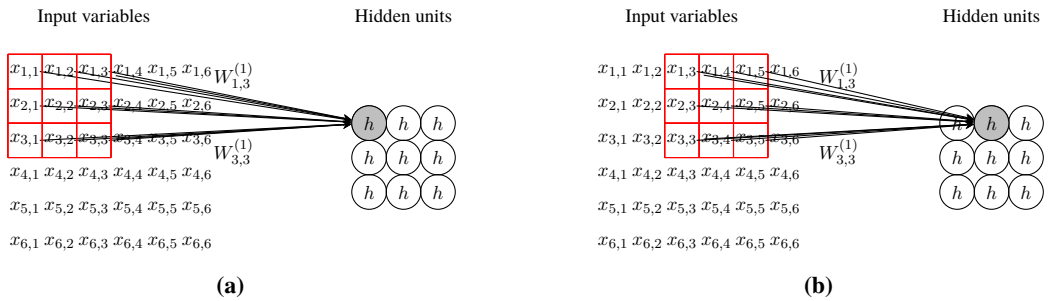


Figure 6.10: A convolutional layer with stride [2,2] and a filter of size 3×3.

offset parameter). If we instead 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.

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 in each layer. One way of doing this is by not applying the filter to every pixel but to say every two pixels. If we apply the filter 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 6.10.

The *stride* controls how many pixels the filter shifts over the image at each step. In Figure 6.8 the stride is [1,1] since the filter moves by one pixel both row- and column-wise. In Figure 6.10 the stride is [2,2] since it moves by two pixels row- and column-wise. Note that the convolutional layer in Figure 6.10 still requires 10 parameters, as the convolutional layer in Figure 6.8 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.

Multiple channels

The networks presented in Figure 6.8 and 6.10 only have 10 parameters each. Even though this parameterization comes with several important advantages, one filter is probably not sufficient to encode all interesting properties of the images in our dataset. To extended the network, we add multiple filters, each with their own set of parameters. Each filter produces its own set of hidden units—a so-called *channel*—using the same convolution operation as explained in Section 6.2. Hence, each layer of hidden units in a CNN is organized into a tensor with the dimensions (rows $\times$ columns $\times$ channels). In Figure 6.11, 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 filter 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 6.11. As a consequence, each filter is a tensor of dimension (filter rows $\times$ filter columns $\times$ input channels). For example, each filter in the second convolutional layer in Figure 6.11 is of size $3 \times 3 \times 4$. If we collect all filter parameters in one weight tensor $\mathbf{W}$, that tensor will be of dimension (filter rows $\times$ filter columns $\times$ input channels $\times$ output channels). In the second convolutional layer in Figure 6.11, the corresponding weight matrix $\mathbf{W}^{(2)}$ is a tensor of dimension $3 \times 3 \times 4 \times 6$. With multiple filters 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.

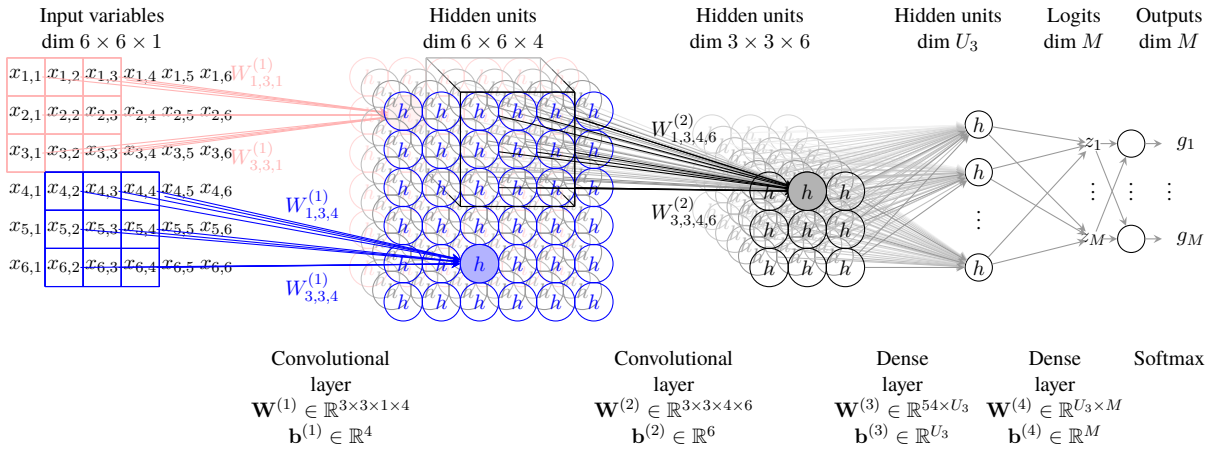


Figure 6.11: A full CNN architecture for classification of grayscale 6×6 images. In the first convolutional layer four filters each of size 3×3 produce a hidden layer with four channels. The first channel (in the back) is visualized in red and the forth channel (in the front) is visualized in blue. We use stride [1,1] which maintains the number of rows and columns. In the second convolutional layer, six filters 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 M logits. The network ends with a softmax function to provide predicted class probabilities as output.

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 the network 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 6.11 a small example of a full CNN architecture is displayed.

6.3 Training a neural network

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

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \qquad \text{where } J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}). \tag{6.22}$$

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 the characteristics of the problem at hand, see e.g. (6.13) for regression and (6.19) for classification.

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

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 $t = 1, 2, \ldots$. (6.23)
3. Terminate when some criterion is fulfilled, and take the last $\boldsymbol{\theta}_t$ as $\widehat{\boldsymbol{\theta}}$.

This problem has two main computational challenges. The first computational challenge is the big data problem. For deep learning application the number of data points n is typically very big making the computation of the cost function and its gradient very costly since it requires a sum over all data points. As a consequence, we cannot afford to compute the exact gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ at each iteration. Instead, we compute an approximation of this gradient by considering a random subset for the training data at each iteration. These so-called stochastic gradient algorithms are further explained in Section 5.4.

The second computational challenge is that the number of parameters $\dim(\boldsymbol{\theta})$, which is also very big for deep learning problems. To efficiently compute the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ we apply the chain rule of calculus and reuse partial derivatives needed to compute this gradient. This is called the back-propagation algorithm and is not further explained here. The intrested reader can for example consult Goodfellow, Bengio, and Courville 2016.

Initialization

Most of the previous optimization problems (such as L^1 regularization and logistic regression) that we have so far 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 to enable the different hidden units to encode different aspects of the data. If the ReLU activation functions are used, the offset elements b_0 are typically initialized to a small positive value such that it operates in the non-negative range of the ReLU.

Dropout

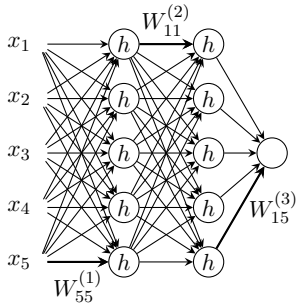
Like all models presented in this course, neural network models can suffer from overfitting if we have a too flexible model in relation to the complexity of the data. Bagging (Section 7.1) is one way to reduce the variance and by that also reducing the risk of overfitting. In bagging we train an entire *ensemble* of models. We train all models (ensemble members) on a different dataset each, which has been bootstrapped (sampled with replacement) from the original training dataset. 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* 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.

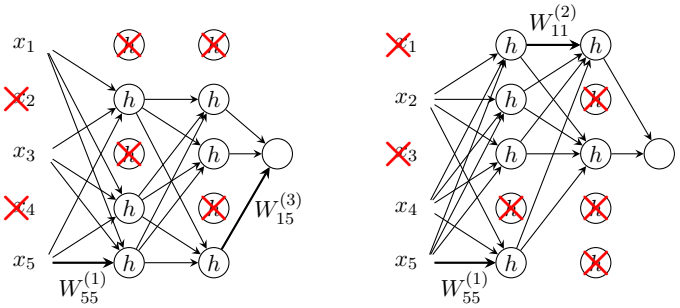
Ensemble of sub-networks

Consider a neural network like the one in Figure 6.12a. 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. Via this process we obtain a sub-network of our original network. Two such sub-networks are displayed in Figure 6.12b. 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, we also remove all of its incoming and outgoing connections. Not only hidden units can be dropped, but also input variables.

Since all sub-networks stem from the very same original network, the different sub-networks share some parameters with each other. For example, in Figure 6.12b the parameter $W_{55}^{(1)}$ 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.



(a) A standard neural network



(b) Two sub-networks

Figure 6.12: 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.

Training with dropout

To train with dropout we use the stochastic gradient algorithm described in Algorithm 10. 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 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

The dropout 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 dataset that has been randomly selected from our training data. However, in bagging we usually do it on a bootstrapped version of the whole dataset 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.

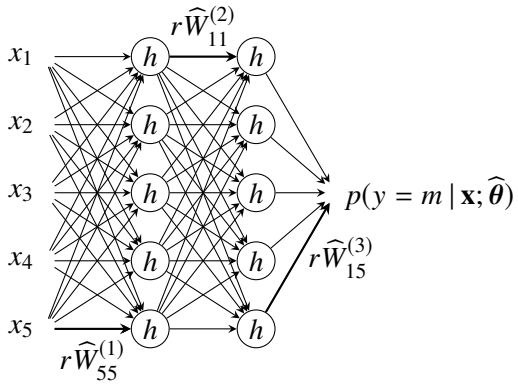


Figure 6.13: The network used for prediction after being trained with dropout. All units and links are present (no dropout) but the weights going out from a certain unit are multiplied with the probability of that unit being included during training. This is to compensate for the fact that some of them were dropped during training. Here all units have been kept with probability r during training (and consequently dropped with probability $1 - r$).

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. 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 the network. This is illustrated in Figure 6.13. 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 (analog to ridge regression and LASSO in Section **??**), early stopping (the training is stopped before the parameters have converged, and thereby the risk of overfitting is reduced) and various sparse representations (for example CNNs can be seen as a regularization method where most parameters are forced to be zero), just to mention a few. Since its invention, dropout has become one of the most popular regularization techniques due to its simplicity, the fact that it is computationally cheap and its good performance. In fact, a good practice of designing a neural network is often to extended the network until it overfits, then extend it a bit more and finally add a regularization like dropout to avoid that overfitting.

6.4 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 believed 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 situation 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 human or even super-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), how to beat the would champion in the board game of Go (Silver et al. 2016) and automatically understand the situation in images for automatic caption generation (Xu et al. 2015).

An accessible introduction and overview of deep learning is provided by LeCun, Bengio, and Hinton (2015), and via the textbook by Goodfellow, Bengio, and Courville (2016).

7 Ensemble methods: Bagging and boosting

In Chapter 3 and 2 we introduced four fundamental models for supervised machine learning. In this chapter we will introduce ensemble methods, a type of *meta-algorithm*, which makes use of multiple copies of some fundamental model. We refer to a set of multiple copies of a fundamental model as an *ensemble of base models*, and the key idea is to train each such base model in a slightly different way. To obtain a prediction from an ensemble, we let each base model make its own prediction and then use a (possibly weighted) average or majority vote to obtain the final prediction. With a carefully constructed ensemble, the prediction obtained in this way is better than the prediction of a single base model.

We start in Section 7.1 by introducing a general technique referred to as bootstrap aggregating, or *bagging* for short. The bagging idea is to first create multiple slightly different "versions" of the training data by, essentially, randomly sample overlapping subsets of the training data (the so-called bootstrap). Thereafter, one base model is trained from each such "version" of the training data. In this way, an ensemble of similar, but not identical, base models is obtained. With this procedure it is possible to *reduce the variance* (without any notable increase in bias) compared to using only a single base model learned from the entire training dataset. In practice this means that by using bagging the risk of overfit decreases, compared to using the base model itself. In Section 7.2 we introduce an extension to bagging only applicable when the base model is a classification or regression tree, which results in a powerful off-the-shelf method called random forests. In random forests, each tree is randomly perturbed in order to obtain additional variance reduction, beyond what is already obtained by the bagging procedure itself.

In Section 7.3-7.4 we introduce another ensemble method known as *boosting*. Boosting is different from bagging and random forests, since its base models are trained sequentially, one after the other, where each model tries to "correct" for the "mistakes" made by the previous ones. On the contrary to bagging, the main effect of boosting is *bias reduction* compared to the base model. Thus, boosting is able to turn an ensemble of "weak" base models (e.g., linear classifiers) into one "strong" ensemble model (e.g., a heavily non-linear classifier), and has also shown very useful in practice.

7.1 Bagging

As discussed already in Chapter 4, 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. Examples of simple yet flexible models are k-NN with a small value of k and a classification tree that is grown deep. Such highly flexible models are sometimes needed for solving real-world machine learning problems, where relationships are far from linear. The downside, however, is the risk of overfitting[1], or equivalently, high model variance. Despite their high variance, those models are not useless. By using them as base models in bootstrap aggregating, or *bagging*, we can

reduce the variance of the base model, without increasing its bias.

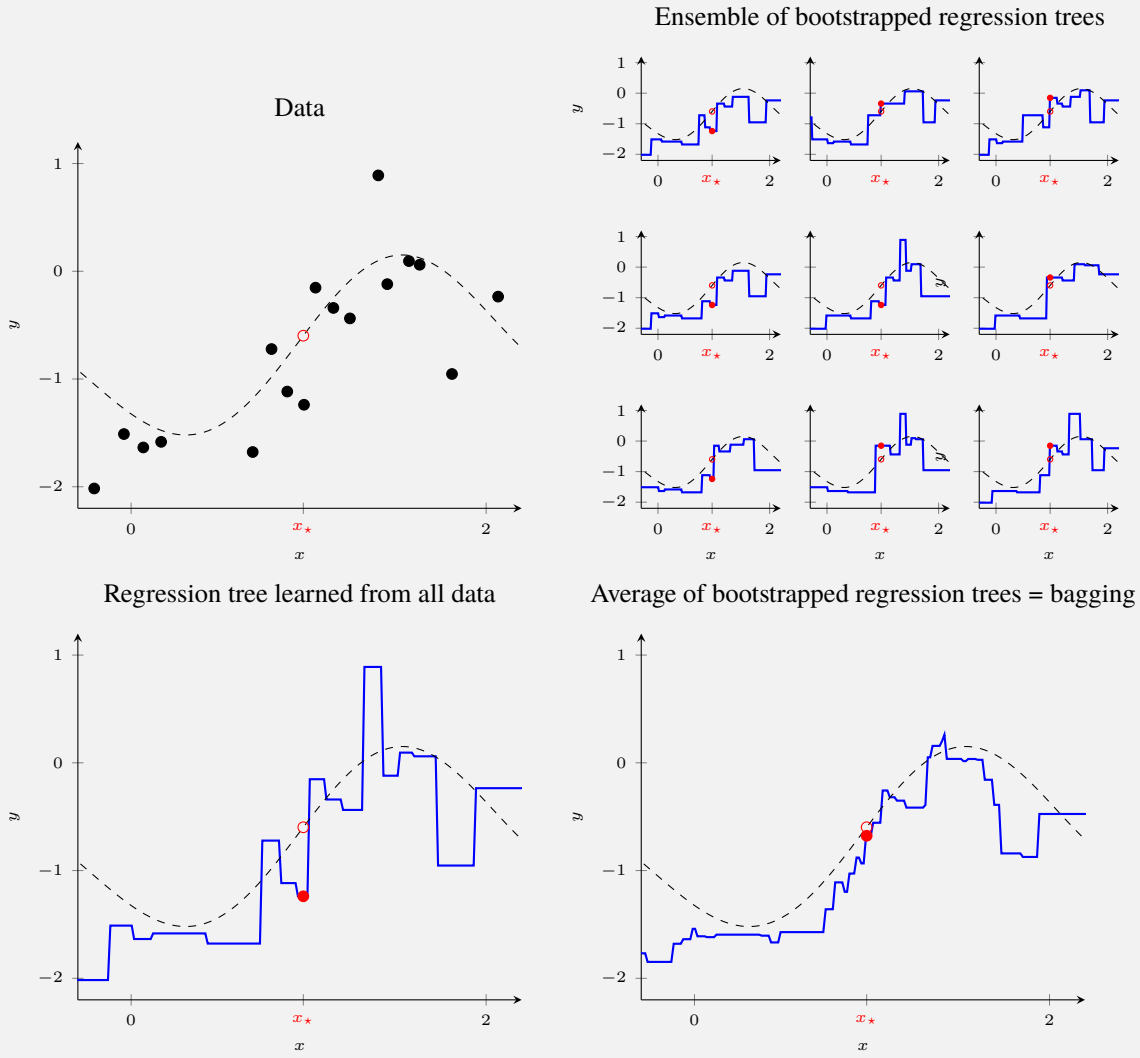
We outline the main idea of bagging with the example below.

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

Example 7.1: Using bagging for a regression problem

Consider the data (black dots) below that are drawn from a function (dashed line) plus noise. As always in supervised machine learning, we want to learn a model from the data which is able to predict new data points well. Being able to predict new data points well means, among other things, that the model should predict the dotted line at $x_\star$ (the red circle) well.

For solving this problem, we could use any regression method. Here, we use a regression tree which is grown until each leaf node only contains one data point, whose prediction is shown to the lower left (blue line and red dot). This is a typical low-bias high-variance model, and the overfit to the training data is apparent from the figure. We could decrease its variance, and hence the overfit, by using a more shallow (less deep) tree, but that would on the other hand increase the bias. Instead, we lower the variance (without increasing the bias much) by using bagging with the regression tree as base model.



The rationale behind bagging goes as follows: Because of the noise in the training data, we may think of the prediction $\hat{y}(x_\star)$ (the red dot) as a random variable. In bagging, we learn an ensemble of base models (upper right panel), where each base model is trained on a different "version" of the training data obtained using the bootstrap. We may therefore think of each base model to be a different realization of the random variable $\hat{y}(x_\star)$. It is well-known that the average of multiple realizations of a random variable has a lower variance than the random variable itself, which means that by taking the average (lower right) of all base models we obtain a prediction with less variance than the base model itself. That is, the bagged regression tree (lower right) has lower variance than a single prediction tree (lower left). Since the base model itself also has low bias, the averaged prediction will have low bias *and* low variance. We can visually confirm that the prediction is better (red dot and circle are closer to each other) for bagging than for the single regression tree.

The bootstrap

As outlined in Example 7.1, the idea of bagging is to average over multiple base models, each learned from a different training dataset. First we therefore have to construct different training datasets. In the best of worlds we would just collect multiple datasets, but most often we cannot do that and instead we have to make the most out of the limited data available. For this purpose, the bootstrap is useful.

The bootstrap is a method for artificially creating multiple datasets (of size n) out of one dataset (also of size n). The traditional usage of the bootstrap is to quantify uncertainties in statistical estimators (such as confidence intervals), but it turns out to be useful also for machine learning. We denote the original dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, and assume that $\mathcal{T}$ provides a good representation of the real-world data generating process, in the sense that *if* we were to collect more training data, these data points would likely be similar to the training data points already contained in $\mathcal{T}$. We can thus argue that randomly picking data points from $\mathcal{T}$ is a reasonable way to simulate a "new" training dataset. 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.

The bootstrap is stated in Algorithm 11 and illustrated in Example 7.2 below. Note that the sampling is done with replacement, meaning that the resulting bootstrapped dataset may contain multiple copies of some of the original training data points, whereas other data points are not included at all.

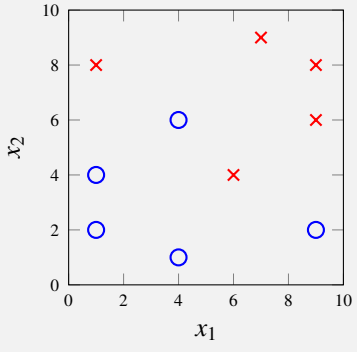
Time to reflect 7.1: *What would happen if the sampling was done without replacement in the bootstrap?*

Example 7.2: The bootstrap

We have a small training dataset with $n = 10$ data points with a two-dimensional input $\mathbf{x} = [x_1 \ x_2]$ and a binary output $y \in \{\text{Blue}, \text{Red}\}$.

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

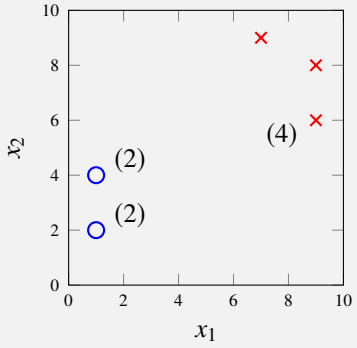
Index	x_1	x_2	y
1	9.0	2.0	Blue
2	1.0	4.0	Blue
3	4.0	6.0	Blue
4	4.0	1.0	Blue
5	1.0	2.0	Blue
6	1.0	8.0	Red
7	6.0	4.0	Red
8	7.0	9.0	Red
9	9.0	8.0	Red
10	9.0	6.0	Red



To generate a bootstrapped dataset $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^{10}$ 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 dataset, 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



Algorithm 11: The bootstrap.

Data: Training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
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

Algorithm 12: Bagging.

Data: Training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
Result: A prediction $\widehat{y}_{\text{bag}}(x_\star)$ or $\mathbf{g}_{\text{bag}}(x_\star)$

for $b = 1, \ldots, B$ **do**
  Run Algorithm 11 to obtain a bootstrapped training dataset $\widetilde{\mathcal{T}}^{(b)}$
  Learn a base model from $\widetilde{\mathcal{T}}^{(b)}$
  Use the base model to predict $\widetilde{y}^b(x_\star)$
end
Obtain $\widehat{y}_{\text{bag}}(x_\star)$ or $\mathbf{g}_{\text{bag}}(x_\star)$ by averaging (7.1).

Variance reduction by averaging

By running the bootstrap (Algorithm 11) repeatedly B times we obtain B *identically distributed* bootstrapped datasets $\widetilde{\mathcal{T}}^1, \ldots, \widetilde{\mathcal{T}}^B$. We can then use those bootstrapped datasets to train an ensemble of B base models. We thereafter average their predictions

$$\widehat{y}_{\text{bag}}(x_\star) = \frac{1}{B}\sum_{b=1}^{B} \widetilde{y}^b(x_\star) \quad \text{or} \quad \mathbf{g}_{\text{bag}}(x_\star) = \frac{1}{B}\sum_{b=1}^{B} \widetilde{\mathbf{g}}^b(x_\star), \tag{7.1}$$

depending on whether we are concerned with regression (predicting an output value $\widehat{y}_{\text{bag}}(x_\star)$) or classification (predicting class probabilities $\mathbf{g}_{\text{bag}}(x_\star)$). In (7.1), $\widetilde{y}^1(x_\star), \ldots, \widetilde{y}^B(x_\star)$ and $\widetilde{\mathbf{g}}^1(x_\star), \ldots, \widetilde{\mathbf{g}}^B(x_\star)$ denote the predictions from the individual ensemble members. The averaged prediction, denoted $\widehat{y}_{\text{bag}}(x_\star)$ or $\mathbf{g}_{\text{bag}}(x_\star)$, is the final prediction obtained from bagging. We summarize this by Algorithm 12. (For classification, the prediction could alternatively be decided by majority vote among the ensemble members, but that typically degrades the performance slightly compared to averaging the predicted class probabilities.)

We will now give some more details on the variance reduction that happens in (7.1), which is the entire point of bagging. We focus on regression, but the intuition works also for classification.

Let us point out 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, computing the mean and the variance of *the average* $\sum_{b=1}^{B} z_b$ of these variables we get

$$\mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \mu, \tag{7.2a}$$

$$\mathrm{Var}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2. \tag{7.2b}$$

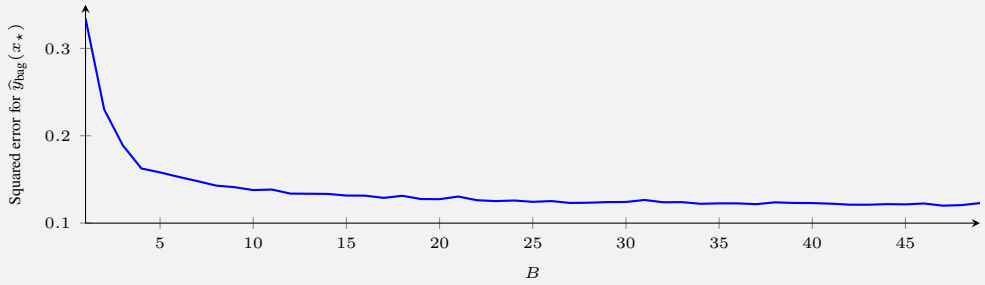
The first equation (7.2a) tells us that the mean is unaltered by averaging a number of identically distributed random variables. Furthermore, the second equation (7.2b) tells us that the variance is reduced by averaging if the correlation $\rho < 1$. The first term in the variance expression (7.2b) can be made arbitrarily small by increasing B, whereas the second term is only determined by the correlation ρ.

To make a connection between bagging and (7.2), consider the predictions $\widehat{y}^b(x_\star)$ from the base models as random variables (in other words, imagine z_b being the red dots in Example 7.1). All base models, and hence their predictions, originate from the same data $\mathcal{T}$ (via the bootstrap), and $\widehat{y}^b(x_\star)$ are therefore identically distributed but correlated. By averaging the predictions we decrease the variance, according to (7.2b). If we only chose B large enough, the achieved variance reduction will be limited by the correlation ρ. Experience has shown that ρ is often small enough such that the computational complexity of bagging (compared to only using the base model itself) pays off well in terms of decreased variance. To summarize, by averaging the identically distributed predictions from several base models as in (7.1), each with a low bias, the *bias remains low*[3] (cf. (7.2a)) and *the variance is reduced* (cf. (7.2b)).

At first glance, one might *think* that a bagging model (7.1) becomes more "complex" as the number of ensemble members B increase, and that we therefore run a risk of overfitting if we use many ensemble members B. However, there is nothing in (7.2) which indicate any such problem (bias remains low, variance decreases), and we confirm this by Example 7.3.

Example 7.3: Bagging for regression (cont.)

We consider the problem from Example 7.1 again, and explore how the number of base models B affects the result. We measure the squared error between the "true" function value at $x_\star$ and the predicted $\widehat{y}^{\mathrm{bag}}(x_\star)$ when using different B. (Because of the bootstrap, there is a certain amount of randomness in the bagging algorithm itself. To avoid that "noise", we average the result over multiple runs of the bagging algorithm.)



What we see here is that the squared error eventually reaches a plateau as $B \to \infty$. Had there been an overfit issue with $B \to \infty$, the squared error would have started do increase again for some large value of B.

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

[3] Strictly speaking, (7.2a) implies that the bias is identical for a single ensemble member and the ensemble average. The use of the bootstrap might however affect the bias, in that a base model trained on the original data might have a smaller bias than a base model trained on a bootstrapped version of the training data. Most often, this is not an issue in practice.

Despite the fact that the number of parameters in the model increases as B increases, the lack of overfit as $B \to \infty$ according to Example 7.3 is the expected (and intended) behavior. It is important to understand that by the construction of bagging, *more ensemble members does not make the resulting model more flexible*, but only reduces the variance. 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.

Remark 7.1 *Bagging can still overfit, we cannot prevent from that. The only claim we have is that the problem never gets worse when* $B \to \infty$.

Out-of-bag error estimation

When using bagging (or random forests), it turns out that there is a way to estimate the expected new data error E_{new} *without* using cross-validation. The first observation we have to make is that not all data points from the original dataset $\mathcal{T}$ will have been used for training all ensemble members. It is actually possible to show that with the bootstrap, on average only 63% of the original training data points in $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ will be present in a bootstrapped training dataset $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^n$. Roughly speaking, this means that for any given $\{\mathbf{x}_i, y_i\}$ in $\mathcal{T}$, one third of the ensemble members will not have seen that data point yet. We refer to these roughly $B/3$ ensemble members as being out-of-bag for sample i, and we let them form their own ensemble, the out-of-bag-ensemble i. Note that the out-of-bag-ensemble is different for each data point $\{\mathbf{x}_i, y_i\}$.

The next key insight is that for the out-of-bag-ensemble i, the data point $\{\mathbf{x}_i, y_i\}$ can actually act as a test data point since it has not yet been seen by any of its ensemble members. By computing the squared or misclassification error when the out-of-bag-ensemble i predicts $\{\mathbf{x}_i, y_i\}$, we thus get an estimate of E_{new} for this out-of-bag-ensemble, which we denote $E_{\text{OOB}}^{(i)}$. Since $E_{\text{OOB}}^{(i)}$ is based on only one data point, it will be a fairly poor estimate of E_{new}. If we however repeat this for all data points $\{\mathbf{x}_i, y_i\}$ in the training data $\mathcal{T}$ and average $E_{\text{OOB}} = \frac{1}{n}\sum_{i=1}^n E_{\text{OOB}}^{(i)}$, we get a better estimate of E_{new}. Indeed, E_{OOB} will be an estimate of E_{new} for an ensemble with only $B/3$ (and not B) members, but as we have seen (Example 7.3), the performance of bagging plateaus after a certain number of ensemble members. Hence, if B is large enough so that ensembles with B and $B/3$ members perform roughly equally, E_{OOB} provides an estimate of E_{new} which can be at least as good as the estimate $E_{k\text{-fold}}$ from k-fold cross-validation. Most importantly, however, E_{OOB} comes almost for free in bagging, whereas $E_{k\text{-fold}}$ requires much more computations when re-training k times.

7.2 Random forests

In bagging we reduce the variance by averaging over an ensemble of models. That reduction, however, is limited by the correlation between the individual ensemble members (cf. the role of ρ in (7.2b)). Using a simple trick, it turns out to be possible to reduce that correlation further, beyond what is achieved by using the bootstrap. This is known as random forests.

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 classification or regression trees. The idea is to inject additional randomness when constructing each tree, in order to further reduce the correlation among the base models. 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.

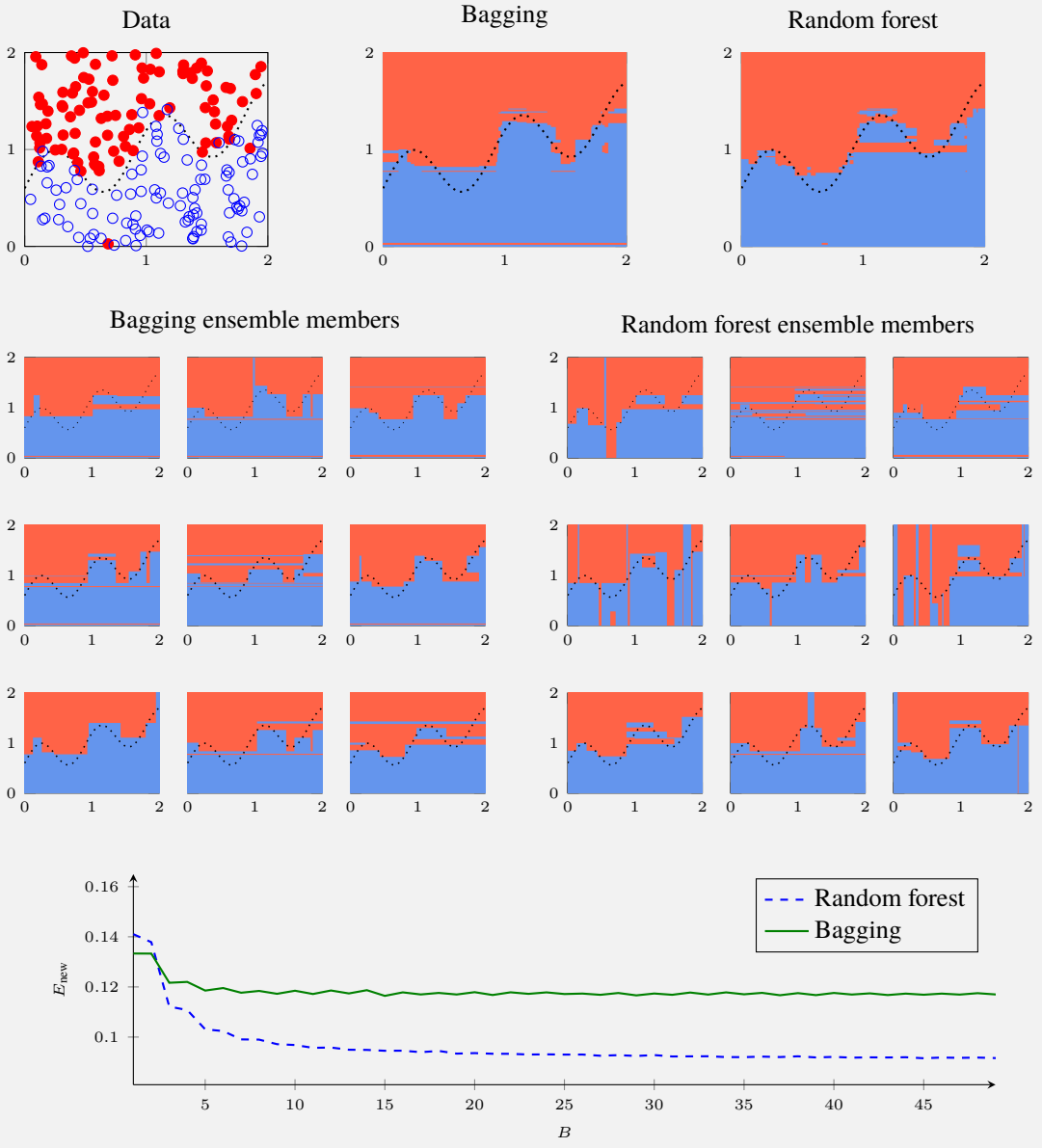
Let $\widetilde{\mathcal{T}}^{(b)}$ be one of the B bootstrapped datasets in bagging. To train a classification or regression tree on this data we proceed as usual (see Section 2.3), but with one difference. Throughout the training, whenever we are about to split a node we do not consider all possible input variables $x_1, \ldots, x_p$ as splitting variables. Instead, we pick a random subset consisting of $q \leq p$ inputs, and only consider these q variables as possible splitting variables. At the next splitting point we draw a new random subset of q 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 additional random constraint when training is what turns bagging into *random forests*. 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 the variance[4] of each *individual tree*. In the notation of Equation (7.2b), random forests decreases ρ (good) but increases σ^2 (bad) compared to bagging. Experience has however shown that the reduction in correlation is the dominant effect, so that the averaged prediction variance is often reduced. We illustrate this in Example 7.4 below.

[4] And possibly also the bias, in a similar manner as the bootstrap might increase the bias, see Footnote 3, page 107.

Example 7.4: Random forests and bagging for a binary classification problem

Consider the binary classification with $p = 2$ using the data given below. The different classes are the blue circles and the red dots. The input values were randomly sampled from $[0, 2] \times [0, 2]$, and labeled red with probability 0.98 if above the dotted line, and vice versa. We use two different classifiers: bagging with classification trees (which is equivalent to a random forest with $q = p = 2$) and a random forest with $q = 1$, each with $B = 9$ ensemble members. Below we plot the decision boundary for each ensemble member as well as the majority-voted final decision boundary.

The most apparent difference is that the variation among the ensemble members of the random forest than those of bagging. Roughly half of the random forest ensemble members have been forced to make the first split along the horizontal axis, which has lead to increased variance and decreased correlation compared to bagging where all ensemble members made the first split along the vertical axis.



While it is hard to visually compare the final decision boundaries for bagging and random forest (top right), we also compute E_{new} for different numbers of ensemble members B. Since the learning itself has a certain amount of randomness, we average over multiple learned models to not be confused by that random effect. Indeed we see that the random forest performs better than bagging, except for very small B, and we conclude that the positive effect of the reduced correlation between the ensemble members outweighs the negative effect of additional variance. The poor performance of random forest with only one ensemble member is expected, since this lonely model has higher variance and no averaging is taking place when $B = 1$.

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 plain 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, some of the ensemble members will not even have access to this dominant variable at the first split, since it most likely will not be present in the random subset of q inputs selected at the first split for some of the ensemble members. This will force those members to split according to some other variable. While there is 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.

User aspects

Since random forest is a bagging method, the tools and properties from Section 7.1 applies also to random forests. One such example is the out-of-bag error estimation, which is applicable also to random forests. Also for random forests, $B \to \infty$ does not lead to overfit. Hence, there is no reason to choose B small, other than the computational load. Compared to using a single tree, a random forest requires approximately B times as much computations. Since all trees are identically distributed, it is however possible to parallelize the random forest learning over multiple nodes, where each node learns a few ensemble members.

The choice of q is a tuning parameter, where for $q = p$ we recover the basic bagging method described previously. As a rule-of-thumb we can set $q = \sqrt{p}$ for classification problems and $q = p/3$ for regression problems (values rounded down to closest integer). A more systematic way of selecting q is to use out-of-bag error estimation or cross-validation and select q such that E_{OOB} or $E_{k\text{-fold}}$ is minimized.

7.3 Boosting and AdaBoost

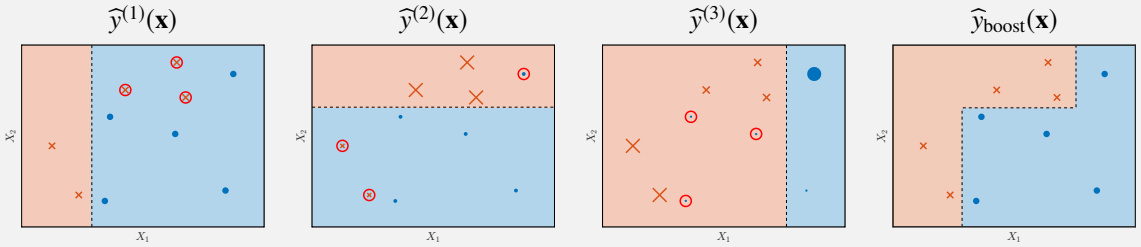
Whereas bagging primarily is an ensemble method for reducing variance in high-variance base models, boosting is rather an ensemble method for reducing bias in high-bias base models. A typical example of a simple (or, in other words, weak) high-bias model is a classification tree of depth one (sometimes called a classification stump). Boosting is built on the idea that even a weak high-bias 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 *reduce the bias* by turning an ensemble of weak models into one strong model.

Boosting shares some similarities with bagging. 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 main difference is how the base models are learned. In bagging we learn B identically distributed models in parallel. 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 dataset at each iteration in order to put more emphasis on the data points for which the model (so far) has performed poorly. The final prediction is obtained from a weighted average or a weighted majority vote among the models. We look at a simple example to illustrate this idea.

Example 7.5: Boosting illustration

We consider a binary classification problem with a two-dimensional input $\mathbf{x} = [x_1 \ x_2]$. 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.



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 the figure above, 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 obtained as a majority vote of the three decision stumps.

The decision boundary of the boosted classifier is shown in the right panel. Note that this decision boundary is nonlinear, whereas the decision boundary for each ensemble member is linear. This illustrates the concept of turning an ensemble of three weak (high-bias) base models into a stronger (low-bias) model.

This example illustrates the idea of boosting, but there are several important details left to specify in order to have a useful off-the-shelf algorithm. We will soon have a look at a specific boosting algorithm called AdaBoost, and thereafter consider the more general framework of gradient boosting. We will restrict our attention to binary classification ($K = 2$, with $y \in \{+1, -1\}$), but boosting is possible to apply also for the multiclass problem and for regression.

AdaBoost

What we have discussed so far is a general idea, but there are still a few technical design choices left. Let us now derive an actual boosting method, the AdaBoost (Adaptive Boosting) algorithm for binary classification. AdaBoost was the first successful practical implementation of the boosting idea and lead the way for its popularity.

As we outlined in Example 7.5 boosting attempts to construct a sequence of B (weak) binary classifiers $\widehat{y}^{(1)}(\mathbf{x}), \widehat{y}^{(2)}(\mathbf{x}), \ldots, \widehat{y}^{(B)}(\mathbf{x})$. We will in this procedure only consider the final 'hard' prediction $\widehat{y}(\mathbf{x})$ from the base models, and not their predicted class probabilities $g(\mathbf{x})$. Any classification model can in principle be used as base classifier—shallow classification trees are common in practice. The individual predictions of the B ensemble members are then combined into a final prediction. Unlike bagging, 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

$$\widehat{y}^{(B)}_{\text{boost}}(\mathbf{x}) = \operatorname{sign}\left\{\sum_{b=1}^{B} \alpha^{(b)} \widehat{y}^{(b)}(\mathbf{x})\right\}. \tag{7.3}$$

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

In AdaBoost, the ensemble members and their coefficients $\alpha^{(b)}$ are trained greedily by minimizing the exponential loss of the boosted classifier at each iteration. That is, one ensemble member is added iteratively at a time. When member b is added, it is trained such that the exponential loss (5.12) of the entire ensemble of b members (that is, the boosted classifier we have so far) is minimized. The reason for choosing the exponential loss is that the problem becomes easier to solve (much like the squared error loss in linear regression), which we will now see when we derive a mathematical expression for this prodecure.

Let us write the boosted classifier after b iterations as $\widehat{y}^{(b)}_{\text{boost}}(\mathbf{x}) = \operatorname{sign}\{c^{(b)}(\mathbf{x})\}$ where $c^{(b)}(\mathbf{x}) = \sum_{j=1}^{b} \alpha^{(j)} \widehat{y}^{(j)}(\mathbf{x})$. Since $c^{(b)}(\mathbf{x})$ is a sum (cf. (7.3)), we can express it iteratively as

$$c^{(b)}(\mathbf{x}) = c^{(b-1)}(\mathbf{x}) + \alpha^{(b)} \widehat{y}^{(b)}(\mathbf{x}), \tag{7.4}$$

initialized with $c^0(\mathbf{x}) = 0$. The ensemble members are constructed sequentially, meaning that at iteration b of the procedure the function $c^{(b-1)}(\mathbf{x})$ is known and fixed. This is what makes this construction "greedy". What remains to be learned at iteration b is the ensemble member $\widehat{y}^{(b)}(\mathbf{x})$ and its coefficient $\alpha^{(b)}$. We do this by minimizing the exponential loss of the training data,

$$\begin{aligned}
(\alpha^{(b)}, \widehat{y}^{(b)}) &= \arg\min_{(\alpha, \widehat{y})} \sum_{i=1}^{n} L(y_i \cdot c^{(b)}(\mathbf{x}_i)) && \text{(7.5a)} \\
&= \arg\min_{(\alpha, \widehat{y})} \sum_{i=1}^{n} \exp\left(-y_i \left(c^{(b-1)}(\mathbf{x}_i) + \alpha \widehat{y}(\mathbf{x}_i)\right)\right) && \text{(7.5b)} \\
&= \arg\min_{(\alpha, \widehat{y})} \sum_{i=1}^{n} \underbrace{\exp\left(-y_i c^{(b-1)}(\mathbf{x}_i)\right)}_{= w_i^{(b)}} \exp\left(-y_i \alpha \widehat{y}(\mathbf{x}_i)\right), && \text{(7.5c)}
\end{aligned}$$

where for the first equality we have used the definition of the exponential loss function (5.12) and the sequential structure of the boosted classifier (7.4). The last equality is where the convenience of the exponential loss appears, namely the fact that $\exp(a + b) = \exp(a)\exp(b)$. This allows us to define the quantities

$$w_i^{(b)} \overset{\text{def}}{=} \exp\left(-y_i c^{(b-1)}(\mathbf{x}_i)\right), \tag{7.6}$$

which can be interpreted as *weights* for the individual data points in the training dataset. Note that the weights $w_i^{(b)}$ are independent of α and $\widehat{y}$. That is, when learning $\widehat{y}^{(b)}(\mathbf{x})$ and its coefficient $\alpha^{(b)}$ by solving (7.5c) we can regard $\{w_i^{(b)}\}_{i=1}^n$ as constants.

To solve (7.5) we start by rewriting the objective function as

$$\sum_{i=1}^{n} w_i^{(b)} \exp\left(-y_i \alpha \widehat{y}(\mathbf{x}_i)\right) = e^{-\alpha} \underbrace{\sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{y_i = \widehat{y}(\mathbf{x}_i)\}}_{= W_c} + e^{\alpha} \underbrace{\sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{y_i \neq \widehat{y}(\mathbf{x}_i)\}}_{= W_e}, \tag{7.7}$$

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 points erroneously classified by $\widehat{y}$. (Remember that $\widehat{y}$ is the ensemble member we are to learn at this step.) 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 (7.7) 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 $\alpha > 0$, another convenient effect of using the exponential loss function. To see this, note that we can write the objective function (7.7) as

$$e^{-\alpha} W + (e^{\alpha} - e^{-\alpha}) W_e. \tag{7.8}$$

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,

$$\widehat{y}^{(b)} = \arg\min_{\widehat{y}} \sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{y_i \neq \widehat{y}(\mathbf{x}_i)\}. \tag{7.9}$$

In words, the b^{th} 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.

> ***Time to reflect 7.2:*** *In AdaBoost, we use the exponential loss for training the boosting ensemble. How come that we end up training the individual ensemble members using a weighted misclassification loss (and not the unweighted exponential loss) then?*

How the problem (7.9) 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, solving (7.9) is almost our standard classification problem, except for the weights $w_i^{(b)}$. Training the ensemble member b on a *weighted* classification problem is, for most base classifiers, straightforward. Since most classifiers are trained by minimizing some cost function, this simply boils down to weighting the individual terms of the cost function and solve that slightly modified problem instead.

When the b^{th} ensemble member, $\widehat{y}^{(b)}(\mathbf{x})$, has been trained for solving the weighted classification problem (7.9) it remains to learn its coefficient $\alpha^{(b)}$. This is done by solving (7.5), which amounts to minimizing (7.8) once $\widehat{y}$ has been trained. By differentiating (7.8) w.r.t. α and setting the derivative to zero we get the equation

$$-\alpha e^{-\alpha} W + \alpha \left(e^{\alpha} + e^{-\alpha}\right) W_e = 0 \Leftrightarrow W = \left(e^{2\alpha} + 1\right) W_e \Leftrightarrow \alpha = \frac{1}{2} \ln\left(\frac{W}{W_e} - 1\right).$$

Thus, by defining

$$E_{\text{train}}^{(b)} \stackrel{\text{def}}{=} \frac{W_e}{W} = \sum_{i=1}^{n} \frac{w_i^{(b)}}{\sum_{j=1}^{n} w_j^{(b)}} \mathbb{I}\{y_i \neq \widehat{y}^{(b)}(\mathbf{x}_i)\} \tag{7.10}$$

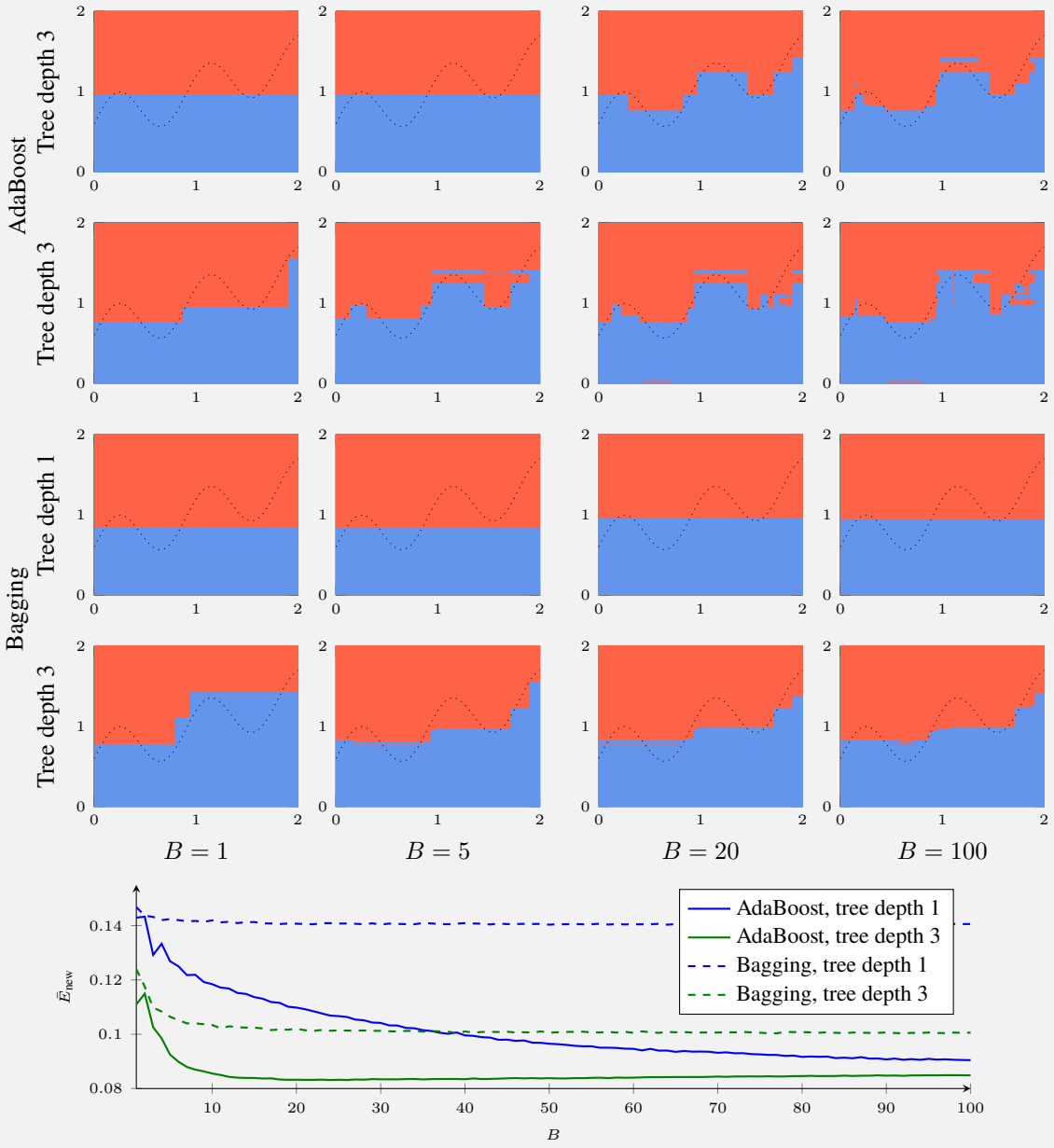
to be the weighted misclassification error for the b^{th} classifier, we can express the optimal value for its coefficient as

$$\alpha^{(b)} = \frac{1}{2} \ln\left(\frac{1 - E_{\text{train}}^{(b)}}{E_{\text{train}}^{(b)}}\right). \tag{7.11}$$

This completes the derivation of the AdaBoost algorithm, which is summarized in Algorithm 13. In the algorithm we exploit the fact that the weights (7.6) can be computed recursively by using the expression (7.4) in line 5. Furthermore, we have added an explicit weight normalization (line 6) which is convenient in practice and which does not affect the derivation of the method above.

Example 7.6: AdaBoost and bagging for a binary classification example

Consider the same binary classification problem as in Example 7.4. We now compare how AdaBoost and bagging performs on this problem, when using trees of depth one (decision stumps) and three, respectively. The decision boundaries for each method with $B = 1, 5, 20$ and 100 ensemble members are shown below. Despite using quite weak ensemble members (a shallow tree has high bias), AdaBoost adapts quite well to the data (Example 7.4). This is in contrast to bagging, where the decision boundary does not become much more flexible despite using many ensemble members. In other words, AdaBoost reduces the bias of the base model, whereas bagging only has minor effect on the bias.



We also numerically compute $\bar{E}_{\text{new}}$ for this problem, as a function of B, which is shown above. Remember that $\bar{E}_{\text{new}}$ depends on both the bias and the variance. As discussed, the main effect of bagging is variance reduction, but that does not help much since the base model already is quite low-variance (but high-bias). Boosting, on the other hand, reduces bias, which has a much bigger effect in this case. Furthermore, bagging does not overfit as $B \to \infty$, but that is *not* the case for boosting! We can indeed see that for trees of depth 3, the smallest $\bar{E}_{\text{new}}$ is obtained for $B \approx 25$, and there is actually a slight increase in $\bar{E}_{\text{new}}$ for larger values of B. Hence, AdaBoost with depth-3 trees suffers from a (minor) overfit as $B \gtrsim 25$ in this problem.

Algorithm 13: AdaBoost

Data: Training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
Result: A prediction $\widehat{y}^{\text{boost}}(x_\star)$

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

Remark 7.2 *The derivation of AdaBoost 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* (7.11)*, note that the function* $\ln((1-x)/x)$ *is positive for any* $0 < x < 0.5$*. Thus,* $\alpha^{(b)}$ *will be positive as long as the weighted training error for the* b^{th} *classifier,* $E_{train}^{(b)}$*, 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* $E_{train}^{(b)}$ *is the* training *error). (Indeed, if* $E_{train}^{(b)} > 0.5$*, then we could simply flip the sign of all predictions made by* $\widehat{y}^{(b)}(\mathbf{x})$ *to reduce the error below* 0.5*.)*

User aspects of AdaBoost

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 7.5). This choice is guided by the fact that boosting reduces bias efficiently, and can thereby learn good models despite using a very weak (high-bias) base model. Since shallow trees can be trained quickly, they are a good default choice. Practical experience suggests that trees with roughly 6 terminal nodes often works well as base models, but trees of depth one (only $M = 2$ terminal nodes in binary classification) are perhaps even more commonly used. In fact, using deep classification trees (high-variance models) as base classifiers typically deteriorates performance.

The base models are learned sequentially in boosting, each iteration introduces a new base model aiming at reducing the errors made by the current model. As an effect, the boosting model becomes more and more flexible as the number of iterations B increases and using too many base models can result in overfitting (in contrast to bagging, where increased B cannot lead to overfit). 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 early stopping during training, similarly to how it is used for neural networks (recall Section 6.3). Another unfortunate aspect of the sequential nature of boosting it that it is not possible to parallelize the learning.

In the method discussed above we have assumed that each base classifier outputs a class prediction, $\widehat{y}^{(b)}(\mathbf{x}) \in \{-1, 1\}$. However, many classification models outputs $g(\mathbf{x})$, which is an estimate of the class probability $p(y = 1 \mid \mathbf{x})$. In AdaBoost it is possible to use the predicted probabilities $g(\mathbf{x})$ (instead of the binary prediction $\widehat{y}(\mathbf{x})$) when constructing the prediction, however at the cost of a more complicated expression than (7.3). This extension of Algorithm 13 is referred to as Real AdaBoost.

7.4 Gradient boosting

It has been seen in practice that AdaBoost performs well if there is little noise (few mislabeled data points) in the training data. If the training data has more noise and hence contains more outliers, the performance of AdaBoost typically deteriorates. That is not an artifact of the boosting idea, but of the exponential loss function that we discussed in Section 5.1. It is possible to construct more robust boosting algorithms by choosing another loss function, but the training becomes more computationally involved, as we will see.

A first gradient boosting algorithm

If the squared error loss in linear regression is replaced with, say, absolute error loss, the simple analytical solution (the normal equations) is lost. We can, however, still learn the model by using numerical optimization. The situation is quite similar for boosting. The simplicity of AdaBoost is that the boosting problem is turned into a sequence of weighted classification problems, which we can solve (at least approximately) by almost any off-the-shelf classification method. If the exponential loss in (7.5a) is replaced, the problem does not separate into weighted classification problems anymore, and the situation becomes more intricate.

It turns out to be possible to approximately solve (7.5a) for rather general loss functions using a method reminiscent of gradient descent ((5.25) in Chapter 5). The resulting method is referred to as *gradient boosting*.

Like AdaBoost (7.4), we construct the classifier sequentially as

$$c^{(b)}(\mathbf{x}) = c^{(b-1)}(\mathbf{x}) + \alpha^{(b)}\widehat{y}^{(b)}(\mathbf{x}), \tag{7.12}$$

for $b = 1, \ldots, B$, and the goal is to sequentially select $\{\alpha^{(b)}, \widehat{y}^{(b)}(\mathbf{x})\}$ such that the final $c^{(B)}(\mathbf{x})$ minimizes

$$J(c) = \frac{1}{n}\sum_{i=1}^{n} L(y_i \cdot c(\mathbf{x}_i)) \tag{7.13}$$

for any arbitrary differentiable loss function L, such as the logistic loss.

The idea of gradient boosting is to think of

$$\underbrace{\begin{bmatrix} c^{(B)}(\mathbf{x}_1) \\ \vdots \\ c^{(B)}(\mathbf{x}_n) \end{bmatrix}}_{c^{(B)}(\mathbf{X})}$$

as an n-dimensional optimization variable, which by construction is build up in the additive fashion (7.12). We then observe that the boosted model (7.12) actually looks like a gradient descent update (recall (5.25) in Chapter 5), if (7.13) would be the objective function $J(c(\mathbf{X}))$ and $\widehat{y}^{(b)}(\mathbf{x})$ its gradient $\nabla_c J(c^{(b-1)}(\mathbf{X}))$.

With this inspiration from gradient descent, we would like to choose $\widehat{y}^{(b)}(\mathbf{x})$ in (7.12) such that

$$\begin{bmatrix} \widehat{y}^{(b)}(\mathbf{x}_1) \\ \vdots \\ \widehat{y}^{(b)}(\mathbf{x}_n) \end{bmatrix} \text{ is close to } \underbrace{\begin{bmatrix} \frac{\partial L(y_1 \cdot c^{(b-1)}(\mathbf{x}_1))}{\partial c} \\ \vdots \\ \frac{\partial L(y_n \cdot c^{(b-1)}(\mathbf{x}_n))}{\partial c} \end{bmatrix}}_{\nabla_c J(c^{(b-1)}(\mathbf{X}))}. \tag{7.14}$$

By the arguments of gradient descent this would as $b \to \infty$ lead to a $c^{(B)}(\mathbf{x})$ which is, at least approximately, a local minimizer of (7.13).

We have now outlined the idea of gradient boosting, but it is not yet a concrete algorithm. Specifically we have to be more precise about $\widehat{y}^{(b)}$ in (7.14). In gradient boosting, we simply handle (7.14) as a

regression problem where $\widehat{y}^{(b)}$ is learned with input $\mathbf{x}_i$ and output $\frac{\partial L(y_i, c^{(b-1)}(\mathbf{x}_i))}{\partial c}$ ($i = 1, \ldots, n$). (We have assumed that the loss function L is differentiable, meaning that we can always compute the necessary derivatives.) That is, gradient boosting builds a classifier by using regression. The type of base regression model $\widehat{y}$ is in principle arbitrary, but regression trees are often used in practice.

We have not yet discussed how to choose $\alpha^{(b)}$, which corresponds to γ (the step size or learning rate) in gradient descent. In the simplest version of gradient descent it is considered a tuning choice left to the user, but in can also be formulated as a line-search optimization problem itself to choose the optimal value at each iteration. For gradient boosting, it is most often handled in the latter way. When using trees as base models optimizing $\alpha^{(b)}$ can be done jointly with learning $\widehat{y}$, resulting in a more efficient implementation. If multiplying the optimal $\alpha^{(b)}$ with a parameter < 1, a regularizing effect is obtained which has proven useful in practice.

We summarize gradient boosting by Algorithm 14.

Algorithm 14: 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 $\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}) = \text{sign}\{c^{(B)}(\mathbf{x})\}$.

While presented for classification in Algorithm 14, gradient boosting can also be used for regression with minor modifications. 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 logistic loss (see Figure 5.2) is in this respect a "safe choice" which is infinitely differentiable and strongly convex, while still enjoying good statistical properties. As a consequence, the logistic loss is one of the most commonly used loss functions in practice.

8 Nonlinear input transformations and kernels

In this chapter we will continue to develop the idea from Chapter 3 of creating new input features by using nonlinear transformations $\boldsymbol{\phi}(\mathbf{x})$. It turns out that by the so-called kernel trick, we can have *infinitely* many such nonlinear transformations and we can extend our basic methods, such as linear regression and k-NN, into more versatile and flexible ones. When we also change the loss function of linear regression, we achieve support vector machines, another powerful off-the-shelf machine learning method. The concept of kernels are important also to the next chapter (Chapter 9), where a Bayesian perspective of linear regression and kernels will lead to the Gaussian process.

8.1 Creating features by nonlinear input transformations

The reason for the word 'linear' in the name 'linear regression' is that the output is modelled as a *linear* combination of the inputs. We have, however, not made a clear definition of what an input is: if the speed is an input in Example 2.2, then why could not also the kinetic energy—the square of the speed—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 in any model, including linear regression. If we, for example, only have a one-dimensional input x, the vanilla linear regression model is

$$y = \theta_0 + \theta_1 x + \varepsilon. \tag{8.1}$$

However, we can extend the model with $x^2, x^3, \ldots, x^{d-1}$ as inputs (d is a user-choice), and thus obtain a linear regression model which is a polynomial in x,

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_{d-1} x^{d-1} + \varepsilon = \boldsymbol{\theta}^\mathsf{T} \boldsymbol{\phi}(x). \tag{8.2}$$

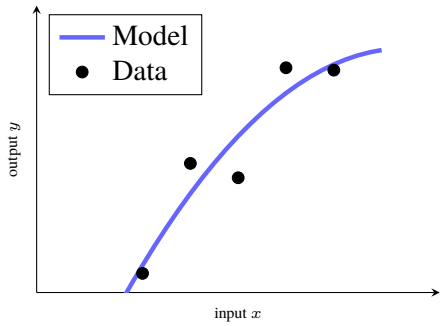
Since x is known, we can compute $x^2, \ldots, x^{d-1}$. Note that this is still a linear regression model *since the parameters $\boldsymbol{\theta}$ appear in a linear fashion* with $\boldsymbol{\phi}(x) = [1 \; x \; x^2 \; \ldots \; x^{d-1}]^\mathsf{T}$ as a new input vector. We refer to a transformation of $\mathbf{x}$ as a[1] *feature* and the vector of transformed inputs $\boldsymbol{\phi}(\mathbf{x})$, a vector of dimension $d \times 1$, as a *feature vector*. The parameters $\widehat{\boldsymbol{\theta}}$ are still learned the same way, but we

$$\text{replace the original } \mathbf{X} = \underbrace{\begin{bmatrix} -\mathbf{x}_1^\mathsf{T}- \\ -\mathbf{x}_2^\mathsf{T}- \\ \vdots \\ -\mathbf{x}_n^\mathsf{T}- \end{bmatrix}}_{n \times p+1} \text{ with the transformed } \boldsymbol{\Phi}(\mathbf{X}) = \underbrace{\begin{bmatrix} -\boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}- \\ -\boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}- \\ \vdots \\ -\boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}- \end{bmatrix}}_{n \times d} \tag{8.3}$$

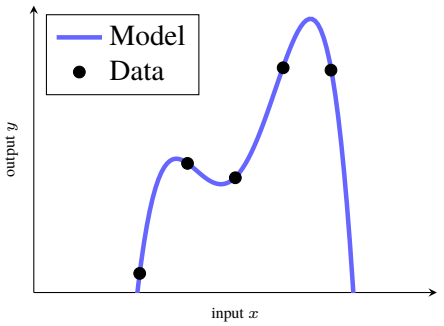
in the cost function. For linear regression, this means that we can learn the parameters by doing the substitution (8.3) directly in the normal equations (3.13).

The idea of nonlinear input transformations is not unique to linear regression, but any arbitrary choice of nonlinear transformation $\boldsymbol{\phi}(\cdot)$ can be used with any supervised machine learning methods: the nonlinear transformation is first applied to the input, like a pre-processing step, and the transformed input is thereafter used when training, evaluating and using the model. We illustrate this by Example Example 8.1.

[1] Sometimes also the original input $\mathbf{x}$ is called a feature.



(a) A linear regression model with a 2nd order polynomial, trained with squared error loss. The line is no longer straight (cf. Figure 3.1), but this is merely an artifact 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 model.



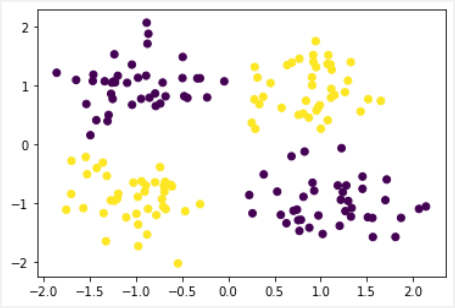
(b) A linear regression model with a 4th order polynomial, trained with squared error loss. Note that a 4th order polynomial implies 5 unknown parameters, which roughly means that we can expect the learned model to fit 5 data points exactly, a typical case of overfit.

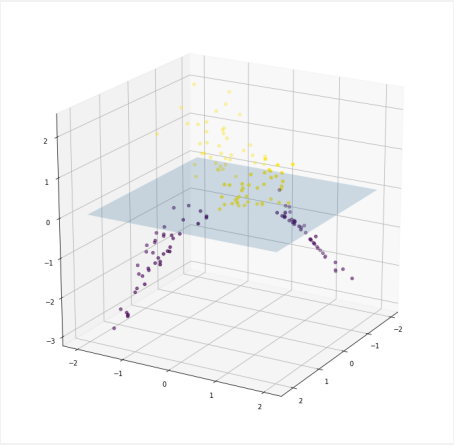
Figure 8.1: A linear regression model with 2nd and 4th order polynomials in the input x, as shown in (8.2).

Time to reflect 8.1: *Figure 8.1 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?*

Example 8.1: Nonlinear feature transformations for classification

This example is not yet complete.





Polynomials is only one out of (infinitely) many possible choices of features $\boldsymbol{\phi}(\mathbf{x})$. One should take care when using polynomials higher than second order in practice, because of their behavior outside the range where the data is observed (such as Figure 8.1b). Instead, there are several alternatives that are more useful in practice, such as Forier series, meaning (for scalar x) something like $\boldsymbol{\phi}(x) = [1 \ \sin(x) \ \cos(x) \ \sin(2x) \ \cos(2x) \ \cdots]^\mathsf{T}$, step functions, regression splines, etc. The use of nonlinear input transformations $\boldsymbol{\phi}(\mathbf{x})$ arguably makes simple models more flexible and applicable to real-world problems with nonlinear characteristics. In order to obtain a good performance, it is important to chose $\boldsymbol{\phi}(\mathbf{x})$ such that enough flexibility is obtained, but overfit avoided. With a very careful choice of $\boldsymbol{\phi}(\mathbf{x})$ good performance can be obtained for certain problems, but that choice is problem-specific and perhaps more of an art than a science. Let us instead explore the conceptual idea of letting the number of features $d \to \infty$, and combine it with regularization. That will lead us to the family of kernel methods, which are a set of powerful off-the-shelf machine learning tools.

8.2 Using kernels in regression: kernel ridge regression and support vector regression

A carefully engineered transformation $\boldsymbol{\phi}(x)$ in linear regression, or any other method for that matter, may indeed perform well for a specific machine learning problem. However, we would like $\boldsymbol{\phi}(x)$ to contain "all" transformations that could possibly be of interest for most problems, in order to obtain a general off-the-shelf method. We will therefore explore choosing d really big, much bigger than the number of data points n, and eventually even $d \to \infty$. We will make the derivation and reasoning using linear regression, but we will later see that the idea is applicable also to other models.

Re-formulating linear regression

First of all we have to use some kind of regularization if we are going to increase d in linear regression, in order to avoid overfit when $d \gg n$. For reasons that we will discuss later we chose to use L^2 regularization. We repeat the equation for L^2-regularized linear regression,

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \Big(\underbrace{\boldsymbol{\theta}^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_i)}_{\widehat{y}(\mathbf{x}_i)} - y_i \Big)^2 + \lambda \|\boldsymbol{\theta}\|_2^2 = (\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X}) + n\lambda \mathbf{I})^{-1} \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \mathbf{y}, \tag{8.4a}$$

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_\star) \tag{8.4b}$$

We have not fixed the nonlinear transformations $\boldsymbol{\phi}(\mathbf{x})$ to anything specific yet, but we are preparing for choosing $d \gg n$ of them. The downside of choosing d, the dimension of $\boldsymbol{\phi}(\mathbf{x})$, large is that we also have to learn d parameters when training. In linear regression, we usually first learn and store the d-dimensional vector $\widehat{\boldsymbol{\theta}}$, and thereafter use it for computing a prediction. To be able to choose d really large, perhaps even $d \to \infty$, we have to re-formulate the model such that there are no computations or storage demands that scales with d. The first step is to realize that the prediction $\widehat{y}(\mathbf{x}_\star)$ can be re-written as

$$\begin{aligned} \widehat{y}(\mathbf{x}_\star) = \underbrace{\widehat{\boldsymbol{\theta}}^\mathsf{T}}_{1\times d} \underbrace{\boldsymbol{\phi}(\mathbf{x}_\star)}_{d\times 1} &= \Big((\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X}) + n\lambda \mathbf{I})^{-1} \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \mathbf{y} \Big)^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_\star) \\ &= \underbrace{\underbrace{\mathbf{y}^\mathsf{T}}_{1\times n} \underbrace{\boldsymbol{\Phi}(\mathbf{X})}_{n\times d} \underbrace{(\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X}) + n\lambda \mathbf{I})^{-1}}_{d\times d} \underbrace{\boldsymbol{\phi}(\mathbf{x}_\star)}_{d\times 1}}_{n\times 1} \end{aligned} \tag{8.5}$$

(where the underbraces emphasizes the size of the vectors and matrices). This expression for $\widehat{y}(\mathbf{x}_\star)$ suggests that instead of computing and storing the d-dimensional $\widehat{\boldsymbol{\theta}}$ once (independently of $\mathbf{x}_\star$) we could, for each prediction input $\mathbf{x}_\star$, compute the n-dimensional vector $\boldsymbol{\Phi}(\mathbf{X})(\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X}) + n\lambda \mathbf{I})^{-1} \boldsymbol{\phi}(\mathbf{x}_\star)$. By doing so, we avoid storing a d-dimensional vector. However, this would still require the inversion of a $d \times d$ matrix, so we have to sort out some more details before we have a practically useful method where we can select d really large.

The push-through matrix identity says that $\mathbf{A}(\mathbf{A}^\mathsf{T}\mathbf{A} + \mathbf{I})^{-1} = (\mathbf{A}\mathbf{A}^\mathsf{T} + \mathbf{I})^{-1}\mathbf{A}$ holds for any matrix $\mathbf{A}$. By using it in (8.5), we can write $\widehat{y}(\mathbf{x}_\star)$ as

$$\widehat{y}(\mathbf{x}_\star) = \underbrace{\mathbf{y}^\mathsf{T}}_{1\times n} \underbrace{(\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} + n\lambda \mathbf{I})^{-1}}_{n\times n} \underbrace{\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star)}_{n\times 1}. \tag{8.6}$$

It now seems as if we can compute $\widehat{y}(\mathbf{x}_\star)$ *without* having to deal with any d-dimensional vectors or matrices. That requires, however, that the matrix multiplication $\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}$ and $\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star)$ in (8.6) can

be computed. Let us therefore have a closer look at it,

$$\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Phi}(\mathbf{X})^{\mathsf{T}} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_1)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_2) & \dots & \boldsymbol{\phi}(\mathbf{x}_1)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_n) \\ \boldsymbol{\phi}(\mathbf{x}_2)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_2)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_2) & \dots & \boldsymbol{\phi}(\mathbf{x}_2)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_n) \\ \vdots & & \ddots & \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_n)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_2) & \dots & \boldsymbol{\phi}(\mathbf{x}_n)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_n) \end{bmatrix} \tag{8.7}$$

$$\text{and } \boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star) = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_\star) \\ \boldsymbol{\phi}(\mathbf{x}_2)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_\star) \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}_\star) \end{bmatrix}. \tag{8.8}$$

Remember that $\boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}')$ is an inner product between the two d-dimensional vectors $\boldsymbol{\phi}(\mathbf{x})$ and $\boldsymbol{\phi}(\mathbf{x}')$. The key insight here is to note that the transformed inputs $\boldsymbol{\phi}(\mathbf{x})$ only enters into (8.6) as inner products $\boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}')$. That is, if we are able to compute the inner product $\boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}')$ directly, without explicitly computing $\boldsymbol{\phi}(\mathbf{x})$ first, we are safe.

As a concrete illustration, let us use polynomials since they are familiar and have no complicated expressions. With $p = 1$, meaning $\mathbf{x}$ is a scalar x, and $\boldsymbol{\phi}(x)$ is a third order polynomial ($d = 4$) with the second and third term scaled[2] by $\sqrt{3}$, we have

$$\boldsymbol{\phi}(x)^{\mathsf{T}}\boldsymbol{\phi}(x') = \begin{bmatrix} 1 & \sqrt{3}x & \sqrt{3}x^2 & x^3 \end{bmatrix} \begin{bmatrix} 1 \\ \sqrt{3}x' \\ \sqrt{3}x'^2 \\ x'^3 \end{bmatrix} = 1 + 3xx' + 3x^2x'^2 + x^3x'^3 = (1 + xx')^3. \tag{8.9}$$

One can generally show that if $\boldsymbol{\phi}(x)$ is a (scaled) polynomial of order $d-1$, then $\boldsymbol{\phi}(x)^{\mathsf{T}}\boldsymbol{\phi}(x') = (1 + xx')^{d-1}$. The point here is that instead of first computing the two d-dimensional vectors $\boldsymbol{\phi}(x)$ and $\boldsymbol{\phi}(x')$ and thereafter computing their inner product, we could just evaluate the expression $(1 + xx')^{d-1}$ directly instead. With a second or third order polynomial this might not make much of a difference, but imagine a situation where it is of interest to use d in the hundreds or thousands.

The point we are getting at is that *if we only make the choice of $\boldsymbol{\phi}(\mathbf{x})$ such that the inner product $\boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}')$ can be computed without first computing $\boldsymbol{\phi}(\mathbf{x})$, we can let d be arbitrary big.* Since it is possible to define inner products also between infinite-dimensional vectors, there is nothing preventing us from letting $d \to \infty$.

We have now derived a version of L^2-regularized linear regression that we can use in practice also with an unbounded number of features d in $\boldsymbol{\phi}(\mathbf{x})$, if we only restrict ourselves to $\boldsymbol{\phi}(\mathbf{x})$ such that its inner product $\boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}')$ has a closed-form expression (or can, at least, be computed n such a way that it does not scale with d). This might appear to be of rather limited interest for a machine learning engineer, since one still has to come up with a nonlinear transformation $\boldsymbol{\phi}(\mathbf{x})$, choose d (possibly ∞) and thereafter make a pen-and-paper derivation (like (8.9)) of $\boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}')$. To this end, let us therefore introduce the definition of a *kernel*.

Introducing kernels

A kernel $\kappa(\mathbf{x}, \mathbf{x}')$ is (in this book) any function that takes two arguments $\mathbf{x}$ and $\mathbf{x}'$ from the same space, and returns a scalar. Throughout this book, we will limit ourselves to kernels that are real-valued and symmetric, meaning $\kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}', \mathbf{x}) \in \mathbb{R}$ for all $\mathbf{x}$ and $\mathbf{x}'$. Importantly, the inner product between two nonlinear input transformations is an example of a kernel

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathsf{T}}\boldsymbol{\phi}(\mathbf{x}'). \tag{8.10}$$

[2]The scaling $\sqrt{3}$ has no big relevance, and can be compensated with an inverse scaling of the second and third element in $\boldsymbol{\theta}$.

Equation (8.9), for example, is such a kernel. The key insight now is that since $\boldsymbol{\phi}(\mathbf{x})$ only appears in the linear regression model (8.6) via inner products, we do not have to engineer a d-dimensional vector $\boldsymbol{\phi}(\mathbf{x})$ and derive its inner product. Instead, we can just chose a kernel function $\kappa(\mathbf{x}, \mathbf{x}')$ directly. This is known as the *kernel trick*,

If $\mathbf{x}$ only enters the model as $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$, we can choose a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of chosing $\boldsymbol{\phi}(\mathbf{x})$.

To be clear on what this means in practice, we re-write (8.6) using the kernel (8.10),

$$\widehat{y}(\mathbf{x}_\star) = \underbrace{\mathbf{y}^\mathsf{T}}_{1\times n}\underbrace{(\boldsymbol{K}(\mathbf{X},\mathbf{X}) + n\lambda\mathbf{I})^{-1}}_{n\times n}\underbrace{\boldsymbol{K}(\mathbf{X},\mathbf{x}_\star)}_{n\times 1}, \tag{8.11a}$$

$$\boldsymbol{K}(\mathbf{X},\mathbf{X}) = \begin{bmatrix} \kappa(\mathbf{x}_1,\mathbf{x}_1) & \kappa(\mathbf{x}_1,\mathbf{x}_2) & \dots & \kappa(\mathbf{x}_1,\mathbf{x}_n) \\ \kappa(\mathbf{x}_2,\mathbf{x}_1) & \kappa(\mathbf{x}_2,\mathbf{x}_2) & \dots & \kappa(\mathbf{x}_2,\mathbf{x}_n) \\ \vdots & & \ddots & \vdots \\ \kappa(\mathbf{x}_n,\mathbf{x}_1) & \kappa(\mathbf{x}_n,\mathbf{x}_2) & \dots & \kappa(\mathbf{x}_n,\mathbf{x}_n) \end{bmatrix}, \tag{8.11b}$$

$$\boldsymbol{K}(\mathbf{X},\mathbf{x}_\star) = \begin{bmatrix} \kappa(\mathbf{x}_1,\mathbf{x}_\star) \\ \kappa(\mathbf{x}_2,\mathbf{x}_\star) \\ \vdots \\ \kappa(\mathbf{x}_n,\mathbf{x}_\star) \end{bmatrix}. \tag{8.11c}$$

These equations now describes linear regression with L^2 regularization using a kernel $\kappa(\mathbf{x}, \mathbf{x}')$. Since L^2 regularization also is called ridge regression, we refer to (8.11) as kernel ridge regression. We initially argued that linear regression with a possibly infinite-dimensional nonlinear transformation vector $\boldsymbol{\phi}(\mathbf{x})$ could be an interesting model, and (8.11) is (for certain choices of $\boldsymbol{\phi}(\mathbf{x})$ and $\kappa(\mathbf{x}, \mathbf{x}')$) equivalent to that. The design choice for the user is, however, now to select a kernel function $\kappa(\mathbf{x}, \mathbf{x}')$ instead of $\boldsymbol{\phi}(\mathbf{x})$. In practice, choosing $\kappa(\mathbf{x}, \mathbf{x}')$ is a much less tedious problem than choosing $\boldsymbol{\phi}(\mathbf{x})$.

As users we may in principle choose the kernel function $\kappa(\mathbf{x}, \mathbf{x}')$ arbitrarily, as long as we can compute (8.11a). That requires that the inverse of $(\boldsymbol{K}(\mathbf{X},\mathbf{X}) + n\lambda\mathbf{I})$ exists. We are therefore on the safe side if we restrict ourselves to kernels for which the matrix $\boldsymbol{K}(\mathbf{X},\mathbf{X})$ (8.11b) always is positive semidefinite. Such kernel functions are called[3] positive semidefinite kernels.

The user of kernel ridge regression chooses a positive semidefinite kernel, and does neither have to select nor compute $\boldsymbol{\phi}(\mathbf{x})$. However, a corresponding $\boldsymbol{\phi}(\mathbf{x})$ always exists for any positive semidefinite kernel as we will discuss in Section 8.3.

There is a number of positive semidefinite kernels that a user can choose between. A commonly used positive semidefinite kernel is the squared exponential kernel

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\ell^2}\right), \tag{8.12}$$

where the hyperparameter $\ell > 0$ is a design choice left to the user, for example to be chosen using cross validation. Another example of a positive semidefinite kernel mentioned earlier is the polynomial kernel $\kappa(\mathbf{x}, \mathbf{x}') = (c + \mathbf{x}^\mathsf{T}\mathbf{x}')^{d-1}$. A special case thereof is the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$. We will give more examples later.

From the formulation (8.11), it may seem as if we have to compute the inverse of $(\boldsymbol{K}(\mathbf{X},\mathbf{X}) + n\lambda\mathbf{I})$ every time we want to make a prediction. That is, however, not necessary since it does not depend on the test input $\mathbf{x}_\star$. It is therefore wise to introduce the n-dimensional vector

$$\boldsymbol{\alpha} = \begin{bmatrix} \widehat{\alpha}_1 \\ \widehat{\alpha}_2 \\ \vdots \\ \widehat{\alpha}_n \end{bmatrix} \tag{8.13a}$$

[3] Confusingly enough, such kernels are sometimes called positive definite in other texts.

which allows us to re-write kernel ridge regression (8.11) as

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\alpha}}^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star), \tag{8.13b}$$

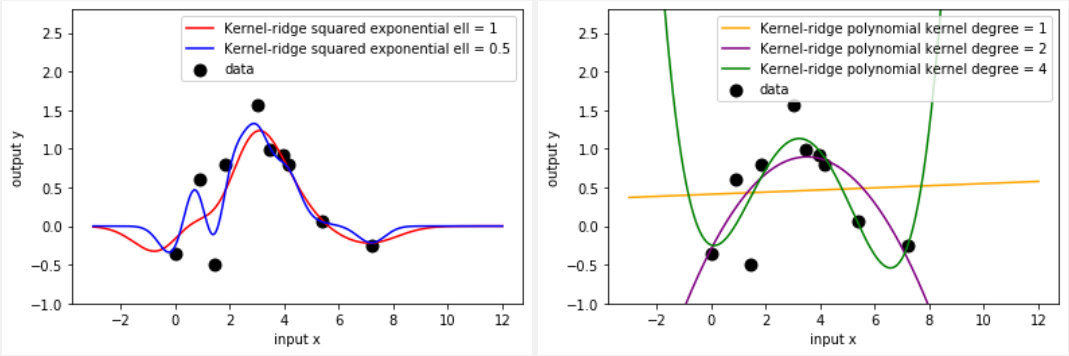
$$\widehat{\boldsymbol{\alpha}} = \mathbf{y}^\mathsf{T}(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda \mathbf{I})^{-1}. \tag{8.13c}$$

That is, instead of computing and storing a d-dimensional vector $\widehat{\boldsymbol{\theta}}$ as we intended initially, we compute and store an n-dimensional vector $\widehat{\boldsymbol{\alpha}}$. However, we also need to store $\mathbf{X}$, since we also have to compute $\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)$ for every prediction.

We summarize kernel ridge regression by Algorithm 15. Kernel ridge regression is in itself a practically useful method, but we will next take a step back and discuss what we have derived, in order to prepare for more kernel methods. We will also come back to kernel ridge regression in Chapter 9, where we will derive the Gaussian process model from it.

Example 8.2: Linear regression with kernels

This example is not yet complete.



Time to reflect 8.2: *Verify that you retrieve L^2-regularized linear regression, without any nonlinear transformations, by using the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}$ in* (8.11).

Algorithm 15: Kernel ridge regression

1 To be completed

Preparing for more kernel methods: the representer theorem

The formulation (8.13) is not only practical for implementation, but also important for theoretical understanding. In fact, we may understand (8.13) as being a *dual formulation* of linear regression, where we have the *dual parameters* $\boldsymbol{\alpha}$ instead of the primal formulation (8.4) with primal parameters $\boldsymbol{\theta}$. Remember that d, the (possibly infinite) number of primal parameters in $\boldsymbol{\theta}$, is a user design choice, whereas n, the number of dual parameters in $\boldsymbol{\alpha}$, is the number of data points.

By comparing (8.5) and (8.4b), we have that

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star) = \widehat{\boldsymbol{\alpha}}^\mathsf{T} \underbrace{\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star)}_{\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)} \tag{8.14}$$

for all $\mathbf{x}_\star$, which suggests that

$$\widehat{\boldsymbol{\theta}} = \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\widehat{\boldsymbol{\alpha}}. \tag{8.15}$$

This relationship between the primal parameters $\boldsymbol{\theta}$ and the dual parameters $\boldsymbol{\alpha}$ is not specific for kernel ridge regression, but (8.15) a fairly general result called *the representer theorem*.

In essence the representer theorem says that equation (8.15) holds for the model $\widehat{y}(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$ learned using (almost) any loss functions and L^2 regularization in $\boldsymbol{\theta}$. A full treatment of it is beyond the scope of this chapter, but we give a complete statement of it in Section 8.A. An implication of the representer theorem is that L^2 regularization is crucial in order to obtain kernel ridge regression (8.13), and we could not have achieved it using, say, L^1 regularization instead. The representer theorem is a cornerstone for kernel methods, since it tells us that *we can express some models in terms of dual parameters $\boldsymbol{\alpha}$ (of finite length n) and a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of the primal parameters $\boldsymbol{\theta}$ (possibly of infinite length d) and a nonlinear feature transformation $\boldsymbol{\phi}(\mathbf{x})$*, just like we did with linear regression in (8.13).

Support vector regression

We will now introduce another off-the-shelf kernel method, namely support vector regression. From a model perspective the only difference to kernel ridge regression is a change of loss function. That different loss function, however, has an interesting effect in that the dual parameter vector $\widehat{\boldsymbol{\alpha}}$ in support vector regression becomes *sparse*, meaning that several elements of $\widehat{\boldsymbol{\alpha}}$ are exactly zero. The training data points corresponding to the non-zero elements of $\widehat{\boldsymbol{\alpha}}$ are referred to as *support vectors*, since $\widehat{y}(\mathbf{x}_\star)$ will depend only on those (in contrast to kernel ridge regression (8.11), where all training data points are needed to compute $\widehat{y}(\mathbf{x}_\star)$). This makes support vector regression an example of a so-called *support vector machine*, a family of methods with sparse dual parameter vectors.

The loss function we will use for support vector regression is the ϵ-insensitive loss

$$L(\widehat{y}(\mathbf{x}_\star), y_\star) = \begin{cases} 0 & \text{if } |\widehat{y}(\mathbf{x}_\star) - y_\star| < \epsilon \\ |\widehat{y}(\mathbf{x}_\star) - y_\star| - \epsilon & \text{otherwise} \end{cases}, \tag{8.16}$$

which we introduced in Section 5.1. The parameter ϵ is a user design choice. In its primal formulation, support vector regression also makes use of the linear regression model

$$\widehat{y}(\mathbf{x}_\star) = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star), \tag{8.17a}$$

but instead of the least square cost function in (8.4), we have

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \max\{0, |\widehat{y}(\mathbf{x}_i) - y_i| - \epsilon\} + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{8.17b}$$

As with kernel ridge regression, we reformulate the primal formulation (8.17) into a dual formulation with $\boldsymbol{\alpha}$ instead of $\boldsymbol{\theta}$ and use the kernel-trick. For the dual formulation, we cannot repeat the convenient closed-form derivation like (8.4)-(8.13) since there is no closed-form solution for $\widehat{\boldsymbol{\theta}}$. Instead we have to use optimization theory and introduce slack variables and construct the Lagrangian of (8.17b)). We do not give the full derivation here, but as it turns out the dual formulation becomes

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\alpha}}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star) \tag{8.18a}$$

where $\widehat{\boldsymbol{\alpha}}$ is the solution to the optimization problem

$$\widehat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \ \frac{1}{2}\boldsymbol{\alpha}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{X})\boldsymbol{\alpha} - \boldsymbol{\alpha}^\mathsf{T}\mathbf{y} + \epsilon\|\boldsymbol{\alpha}\|_1 \tag{8.18b}$$

$$\text{subject to } \ |\alpha_i| \leq \frac{2}{n\lambda} \tag{8.18c}$$

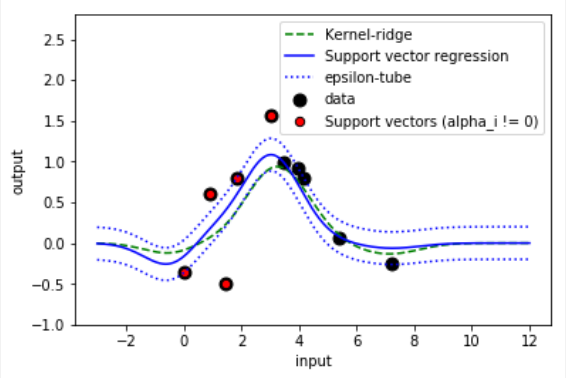
Note that (8.18a) is identical to the corresponding expression for kernel ridge regression (8.13b). This is a consequence of the representer theorem. The only difference to kernel ridge regression is how the dual parameters $\boldsymbol{\alpha}$ are learned; by numerically solving the optimization problem (8.18b) instead of using the closed-form solution (8.13c).

The ϵ-insensitive loss function could be use for any regression model but it is particularly interesting in combination in this kernel context since the dual parameter vector $\boldsymbol{\alpha}$ becomes sparse (meaning that only

some elements are non-zero). Remember that $\boldsymbol{\alpha}$ has one entry per training data point. This implies that the prediction (8.18a) depends only on *some* of the training data points, namely those whose corresponding α_i is non-zero. All training data is indeed used at training time (solving (8.18b)), but when making predictions (using (8.18a)) only the data points with non-zero α_i contributes, which reduces the computational burden. These data points are called support vectors. The larger ϵ chosen by the user, the fewer support vectors and the fewer computations needed for making predictions. It can also be argued that ϵ has a regularizing effect, in the sense that the more/fewer support vectors that are used the more/less complicated model. We illustrate support vector regression with Example 8.3, and summarize it by Algorithm 16.

Example 8.3: support vector regression and kernel ridge regression

This example is not yet complete.



Algorithm 16: Support vector regression

1 To be completed

The ϵ-insensitive loss indeed makes the dual parameter vector $\boldsymbol{\alpha}$ sparse. Note however that it does *not* mean that the corresponding primal parameter vector $\boldsymbol{\theta}$ is sparse (their relationship is given by (8.15)). Also note that (8.18b) is a *constrained* optimization problem (there is a constraint given by (8.18c)), and more theory than what we presented in Section 5.3 is needed to implement a good solver.

Remark 8.1 *The nonlinear feature vector* $\boldsymbol{\Phi}(\mathbf{x})$ *corresponding to some kernels, such as the squared exponential kernel* (8.12)*, does not have a constant/offset term. Therefore an additional* θ_0 *is sometimes included in* (8.18a) *for support vector regression, which gives the additional constraint* $\sum_i \alpha_i = 0$ *to the optimization problem* (8.18b)*. The same addition could be made also to kernel ridge regression* (8.13b)*, but that would break the closed-form calculation of* $\boldsymbol{\alpha}$ (8.13b)*.*

Conclusions on using kernels for regression

In this part of the chapter we have actually been dealing with the interplay of three different concepts, each of them interesting on its own. To make them clear, we repeat them in an ordered list:

1. We have considered the **primal and dual formulation** of a model. The primal formulation expresses the model in terms of $\boldsymbol{\theta}$ (fixed size d), whereas the dual formulation uses $\boldsymbol{\alpha}$ (one α_i per data sample i, hence size n no matter what d is). Both formulations are mathematically equivalent, but more or less useful in practice depending on whether $d > n$ or $n > d$.

2. We have introduced **kernels** κ, which allows us to let $d \to \infty$. This idea is practically useful only when using the dual formulation with $\boldsymbol{\alpha}$, since d is the dimension of $\boldsymbol{\theta}$.

3. We have used **different loss functions**. Kernel ridge regression uses squared error loss, whereas support vector regression uses the ϵ-insensitive loss. The ϵ-insensitive loss is particularly interesting

in the dual formulation, since it gives *sparse* $\boldsymbol{\alpha}$. (We will later also use the hinge loss for support vector classification in Section 8.4, which has a similar effect.)

We will now spend some additional effort on understanding the kernel concept in Section 8.3, and thereafter introduce support vector classification (SVC) in Section 8.4.

8.3 Kernel theory

We have defined a kernel as being any function taking two arguments from the same space, and returning a scalar. We have also suggested that we should (at least sometimes) restrict ourselves to positive semidefinite kernels, and presented two practically useful algorithms—kernel ridge regression and support vector regression. We will now discuss what a kernel means in practice, and also give a flavor of the available theory behind it. To make the discussion more concrete, let us start by introducing the kernel version of k-NN.

Introducing kernel *k*-NN

As you know from Chapter 2, k-NN constructs the prediction for $\mathbf{x}_\star$ by taking the average or a majority vote among the k nearest neighbors to $\mathbf{x}_\star$. In its standard formulation, "nearest" is defined by the Euclidian distance. Since the Euclidian distance always is positive, it is equivalent to instead consider the squared Euclidian distance, which can be written in terms of the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$ as

$$\|\mathbf{x} - \mathbf{x}'\|_2^2 = (\mathbf{x} - \mathbf{x}')^\mathsf{T}(\mathbf{x} - \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x} + \mathbf{x}'^\mathsf{T}\mathbf{x}' - 2\mathbf{x}^\mathsf{T}\mathbf{x}' = \kappa(\mathbf{x}, \mathbf{x}) + \kappa(\mathbf{x}', \mathbf{x}') - 2\kappa(\mathbf{x}, \mathbf{x}'). \tag{8.19}$$

To generalize the k-NN algorithm to use kernels, we allow the linear kernel to be replaced by any, say, positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$ in (8.19). *Kernel k-NN* thus works as standard k-NN, but determines proximity between the data points using the right hand side of (8.19) (with a user-chosen kernel $\kappa(\mathbf{x}, \mathbf{x}')$) instead of the left hand side of (8.19).

For many (but not all) kernels $\kappa(\mathbf{x}, \mathbf{x}) = \kappa(\mathbf{x}', \mathbf{x}')$ for all $\mathbf{x}$ and $\mathbf{x}'$, suggesting that the most interesting part of the right hand side of (8.19) is the term $-2\kappa(\mathbf{x}, \mathbf{x}')$. Thus, if $\kappa(\mathbf{x}, \mathbf{x}')$ takes a large value the two data points $\mathbf{x}$ and $\mathbf{x}'$ are considered to be close, and vice versa if $\kappa(\mathbf{x}, \mathbf{x}')$ takes a small value. That is, *the kernel defines how close any two data points are*.

Furthermore, kernel k-NN allows us to use k-NN also for data where the Euclidian distance has no natural meaning. As long as we have a kernel which acts on the input space, we can apply kernel k-NN no matter if the Euclidian distance is defined for that input type or not. We can thereby apply kernel k-NN to input data that is neither numerical nor categorical, such as text snippets, as illustrated by Example 8.4.

Example 8.4: Kernel k-NN for interpreting words

This example illustrates how kernel k-NN can be applied to text data, where the Euclidian distance has no meaning and standard k-NN therefore can not be applied. In this example, the input is single words (or more technically: character strings) and we use the so-called Levenshtein distance to construct a kernel. The Levenshtein distance is the number of single-character edits needed to transform one word (string) into another. It takes two strings and returns a non-negative integer, which is zero only if the two strings are equivalent. We can thereby use it to construct a kernel for example as $\kappa(x, x') = \frac{1}{1+\mathrm{LD}(x,x')}$ (where LD is the Levenshtein distance). This defines a positive semidefinite kernel since it will always yield real-valued symmetric and diagonal-dominant (and thereby positive semidefinite) matrices $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$.

In this very small example, we consider a training data set of 10 adjectives shown below (x_i), each labeled `Positive` or `Negative`, according to their meaning. We will now use kernel k-NN (with the kernel defined above) to predict whether the word "horrendous" ($x_\star$) is a positive or negative word. In the third column below we have therefore computed the Levenshtein distance (LD) between each known word (x_i) and "horrendous" ($x_\star$). The rightmost column shows the value of the right hand side of (8.19), which is the value that kernel k-NN uses to determine how close two data points are.

Draft (*June* 17, 2020) *of Supervised Machine Learning. Feedback and exercise problems*: `http://smlbook.org`

Word, x_i	Meaning, y_i	Levenshtein dist. from x_i to $x_\star$	$\kappa(x_i, x_i) + \kappa(x_\star, x_\star) - 2\kappa(x_i, x_\star)$
Awesome	`Positive`	8	1.79
Excellent	`Positive`	10	1.81
Spotless	`Positive`	9	1.80
Terrific	`Positive`	8	1.79
Tremendous	`Positive`	4	1.60
Awful	`Negative`	9	1.80
Dreadful	`Negative`	6	1.71
Horrific	`Negative`	6	1.71
Outrageous	`Negative`	6	1.71
Terrible	`Negative`	8	1.79

According to the rightmost column, the closest word to horrendous is the positive word tremendous.Thus, if we use $k = 1$ the conclusion would be that horrendous is a positive word. However, the second, third and fourth closest words are all negative (dreadful, horrific, outrageous), and with $k = 3$ or $k = 4$ the conclusion thereby becomes that horrendous is a negative word (which happens to be the correct conclusion, for this particular example).

The purpose of this example is to illustrate how a kernel allows a basic method such as k-NN to be used for a problem where the input has a more intricate structure than just being numerical. For the particular application of predicting word semantics, more elaborate machine learning methods exist.

The meaning of a kernel

From kernel k-NN we got (at least) two lessons about kernels that are applicable in general to all supervised machine learning methods that use kernels:

- The kernel defines how close/similar any two data points are. If, say, $\kappa(\mathbf{x}_i, \mathbf{x}_\star) > \kappa(\mathbf{x}_j, \mathbf{x}_\star)$, then $\mathbf{x}_\star$ is considered to be more similar to $\mathbf{x}_i$ than $\mathbf{x}_j$. Intuitively speaking, for most methods the prediction $\widehat{y}(\mathbf{x}_\star)$ is most influenced by the training data points that are closest/most similar to $\mathbf{x}_\star$. The kernel thereby plays an important role in determining the individual influence of each training data point when making a prediction.

- Even though we started by introducing kernels from the inner product $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$, the inner product does not have to defined for the space in which $\mathbf{x}$ lives itself! As we saw in Example 8.4, we can apply a positive semidefinite kernel method also to text string without bothering about inner products, as long as we have a kernel for that type of data.

In addition to this, the kernel also plays a somewhat more subtle role in methods that builds on the representer theorem (such as kernel ridge regression, support vector regression and support vector classification, but not kernel k-NN). Remember that the primal formulation of those methods, by virtue of the representer theorem, contains the L^2 regularization term $\lambda\|\boldsymbol{\theta}\|_2^2$. Even though we do not solve the primal formulation explicitly when using kernels (we solve the dual instead), it is nevertheless an equivalent representation, and we may ask what impact the regularization $\lambda\|\boldsymbol{\theta}\|_2^2$ has to the solution?

The L^2 regularization means that primal parameter values $\boldsymbol{\theta}$ close to zero are favored. Besides the regularization term, $\boldsymbol{\theta}$ only appear in the expression $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$. The solution $\widehat{\boldsymbol{\theta}}$ to the primal problem is therefore an interplay between the feature vector $\boldsymbol{\phi}(\mathbf{x})$ and the L^2 regularization. Consider two different choices of feature vectors $\boldsymbol{\phi}_1(\mathbf{x})$ and $\boldsymbol{\phi}_2(\mathbf{x})$. If they both span the same space of functions, there exists $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ such that $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) = \boldsymbol{\theta}_1^\mathsf{T}\boldsymbol{\phi}_2(\mathbf{x})$ for all $\mathbf{x}$, and it might appear irrelevant which feature vector is being used. However, the L^2 regularization complicates the situation because it acts directly on $\boldsymbol{\theta}$, and it therefore matters whether $\boldsymbol{\phi}_1(\mathbf{x})$ or $\boldsymbol{\phi}_2(\mathbf{x})$ is being used. In the dual formulation we choose the kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of feature vector $\boldsymbol{\phi}(\mathbf{x})$, but since that choice implicitly corresponds to a feature vector the effect is still present, and we may add one more bullet point about the meaning of a kernel:

- The choice of kernel corresponds to a choice of a regularization functional. That is, the kernel implies a preference for certain functions in the space of all functions that are spanned by the feature

vector. For example does the squared exponential kernel enforce smooth solutions.

Using a kernel makes a method quite flexible, and one could perhaps expect it to suffer heavily from overfit. However, the regularizing role of the kernel explains why that rarely is the case in practice.

All three bullet points above are central to understand the usefulness and versatility of kernel methods. They also highlight the importance for the machine learning engineer to choose the kernel wisely, and not only resort to "default" choices.

Valid choices of kernels

We introduced kernels as a way to compactly work with nonlinear feature transformations as (8.10). As we thereby only have to consider $\kappa(\mathbf{x}, \mathbf{x}')$, and not $\boldsymbol{\phi}(\mathbf{x})$, it is natural to ask whether any arbitrary kernel function $\kappa(\mathbf{x}, \mathbf{x}')$ actually corresponds to a feature transformation $\boldsymbol{\phi}(\mathbf{x})$, such that it can be written as the inner product

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}')? \tag{8.20}$$

Before answering the question, we have to be aware that this question is merely of theoretical nature. As long as we can use $\kappa(\mathbf{x}, \mathbf{x}')$ when computing predictions it serves its purpose, no matter if it admits the factorization (8.20) or not. The specific requirements on $\kappa(\mathbf{x}, \mathbf{x}')$ is different for different methods, for example is the inverse $(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I})^{-1}$ needed for kernel ridge regression but not support vector regression. Furthermore, whether a kernel admits the factorization (8.20) or not has no direct correspondence on how well it performs in terms of, say, prediction accuracy. For any practical machine learning problem the performance still has to be evaluated using cross validation or similarly.

That being said, we will now have a closer look at the important family of positive semidefinite kernels. A kernel is said to be positive semidefinite if the matrix $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ (8.11b) is positive semidefinite (has no negative eigenvalues) for any choice of $\mathbf{X}$.

First, it holds that any kernel $\kappa(\mathbf{x}, \mathbf{x}')$ that is defined as an inner product between feature vectors $\boldsymbol{\phi}(\mathbf{x})$, as (8.20), is positive semidefinite. It can, for example, be shown by using Cauchy-Schwartz inequality which implies that $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ is diagonally dominant. Since $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ also is real-valued, symmetric and has no negative diagonal entries, it follows that $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ always is positive semidefinite.

Less trivially the other direction also holds, that is, for any positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$ there always exists a feature vector $\boldsymbol{\phi}(\mathbf{x})$ such that $\kappa(\mathbf{x}, \mathbf{x}')$ can be written as an inner product (8.20). Technically it can be shown that for any positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$ it is possible to construct a function space, more specifically a Hilbert space, that is spanned by a feature vector $\boldsymbol{\phi}(\mathbf{x})$ for which (8.20) holds. The dimensionality of the Hilbert space, and thereby also the dimension of $\boldsymbol{\phi}(\mathbf{x})$, can however be infinite.

Give a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ there are multiple ways to construct a Hilbert space spanned by $\boldsymbol{\phi}(\mathbf{x})$, and we will only mention some directions here. One alternative is to consider the so-called reproducing kernel map. The reproducing map is obtained by consider one argument, say the latter, to $\kappa(\mathbf{x}, \mathbf{x}')$ fix and let $\kappa(\cdot, \mathbf{x}')$ span the Hilbert space and define the inner product $\langle \cdot, \cdot \rangle$ such that $\langle \kappa(\cdot, \mathbf{x}), \kappa(\cdot, \mathbf{x}') \rangle = \kappa(\mathbf{x}, \mathbf{x}')$. This inner product has a so-called reproducing property, and is the main building block for the so-called reproducing kernel Hilbert space. Another alternative is to use the so-called Mercer kernel map, which constructs the Hilbert space using eigenfunctions to an integral operator which is related to the kernel.

A given Hilbert space uniquely defines a kernel, but for a given kernel there exists multiple Hilbert spaces which corresponds to it. In practice this means that given a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ the corresponding feature vector $\boldsymbol{\phi}(\mathbf{x})$ is not unique, in fact not even its dimensionality. As a simple example consider the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$, which can either be expressed as an inner product between $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$ (one dimensional $\boldsymbol{\phi}(\mathbf{x})$) or as an inner product between $\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \frac{1}{\sqrt{2}}\mathbf{x} & \frac{1}{\sqrt{2}}\mathbf{x} \end{bmatrix}^\mathsf{T}$ (two dimensional $\boldsymbol{\phi}(\mathbf{x})$).

Examples of kernels

We will now give a list of some of the commonly used kernels, of which we already have introduced some. These examples are only for the case when $\mathbf{x}$ is a numeric type of variable. For other types of input

variables (such as Example 8.4), one has to resort to the more application specific literature. We start with some positive semidefinite kernels, of which the *linear kernel* might be the simplest one

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'. \tag{8.21}$$

A generalization thereof, still positive semidefinite, is the *polynomial kernel*

$$\kappa(\mathbf{x}, \mathbf{x}') = (c + \mathbf{x}^\mathsf{T}\mathbf{x}')^{d-1} \tag{8.22}$$

with hyperparameter $c \geq 0$ and polynomial order $d - 1$ (integer). The polynomial kernel corresponds to a finite-dimensional feature vector $\boldsymbol{\phi}(\mathbf{x})$ of monomials up to order $d - 1$. The polynomial kernel does therefore not conceptually enable anything that could not be achieved by instead implementing the primal formulation and the finite-dimensional $\boldsymbol{\phi}(\mathbf{x})$ explicitly. The other positive semidefinite kernels below, on the other hand, all corresponds to infinite-dimensional feature vectors $\boldsymbol{\phi}(\mathbf{x})$.

We have previously also mentioned the[4] *squared exponential kernel*,

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\ell^2}\right), \tag{8.23}$$

with hyperparameter $\ell > 0$. As we saw in Example 8.2 and 8.3, this kernel has more of a "local" nature compared to the polynomial since $\kappa(\mathbf{x}, \mathbf{x}') \to 0$ as $\|\mathbf{x} - \mathbf{x}'\| \to \infty$. This property makes sense in many problems, and is perhaps the reason why this might be the most commonly used kernel.

Somewhat related to the squared exponential is the family of *Matérn kernels*

$$\kappa(\mathbf{x}, \mathbf{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right)^\nu \zeta_\nu\left(\tfrac{\sqrt{2\nu}\|\mathbf{x}-\mathbf{x}'\|}{\ell}\right), \tag{8.24}$$

with hyperparameters $\ell > 0$ and $\nu > 0$. Here Γ is the Gamma function and ζ_ν is a modified Bessel function. The Met'ern kernel i positive semidefinite. Of particular interest are the cases $\nu = \frac{1}{2}, \frac{3}{2}$ and $\frac{5}{2}$, when (8.24) simplifies to

$$\nu = \frac{1}{2}: \qquad \kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right), \tag{8.25}$$

$$\nu = \frac{3}{2}: \qquad \kappa(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\sqrt{3}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right)\exp\left(-\frac{\sqrt{3}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right), \tag{8.26}$$

$$\nu = \frac{5}{2}: \qquad \kappa(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\sqrt{5}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell} + \frac{5\|\mathbf{x} - \mathbf{x}'\|_2^2}{3\ell^2}\right)\exp\left(-\frac{\sqrt{5}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right). \tag{8.27}$$

It can also be shown that the Matérn kernel (8.24) equals the squared exponential (8.23) when $\nu \to \infty$.

A last positive semidefinite kernel that we will mention is the *rational quadratic kernel*

$$\kappa(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2a}\right)^{-a} \tag{8.28}$$

with hyperparameters $\ell > 0$ and $a > 0$. The squared exponential, Matérn and rational quadratic kernel are *isotropic* kernels, since they are a function only of $\|\mathbf{x} - \mathbf{x}'\_2|$.

Going back to the discussion in connection to (8.20), positive semidefinite kernels is a subset of all kernels, for which we know that certain theoretical properties holds. In practice, however, a kernel is potentially useful as long as we can compute a prediction using it, no matter its theoretical properties. One popular kernel for SVM methods which is not positive semidefinite is the *Sigmoid kernel*

$$\kappa(\mathbf{x}, \mathbf{x}') = \tanh\left(a\mathbf{x}^\mathsf{T}\mathbf{x}' + b\right), \tag{8.29}$$

[4] Also known as Gaussian radial basis function kernel

where $a > 0$ and $b < 0$ are hyperparameters. The fact that it is not positive semidefinite can, for example, be seen by computing the eigenvalues of $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ with $a = 1$, $b = -1$ and $\mathbf{X} = [1\ 2]^\mathsf{T}$. Since this kernel is not positive semidefinite the inverse $(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I})^{-1}$ does not always exists, and it is therefore not suitable for kernel ridge regression. It can, however, be used in support vector regression and classification, where that inverse is not needed. For certain values of b it can be shown to be a so-called conditional positive semidefinite kernel (a weaker property than positive semidefinite).

It is possible to construct "new" kernels by modifying or combining existing ones. There are in particular a set of operations that retains the property of positive semidefiniteness: If $\kappa(\mathbf{x}, \mathbf{x}')$ is a positive semidefinite kernel, then so is $a\kappa(\mathbf{x}, \mathbf{x}')$ if $a > 0$ (scaling). Furthermore if both $\kappa_1(\mathbf{x}, \mathbf{x}')$ and $\kappa_2(\mathbf{x}, \mathbf{x}')$ are positive semidefinite kernels, then so are $\kappa_1(\mathbf{x}, \mathbf{x}') + \kappa_2(\mathbf{x}, \mathbf{x}')$ (addition) as well as $\kappa_1(\mathbf{x}, \mathbf{x}')\kappa_2(\mathbf{x}, \mathbf{x}')$ (multiplication).

Most kernels contains a few hyperparameters that are left to the user to chose. Much like cross validation can give valuable help in choosing between different kernels, cross validation can also help choosing hyperparameters (and regularization parameter λ) with grid search.

8.4 Kernels for classification

We have so far spent most time deriving two kernel versions of linear regression: kernel ridge regression and support vector regression. We will now focus on classification. Unfortunately the derivations are now more technically intricate than for regression, and we have therefore put the details in the chapter appendix. The intuition carries, however, over, as well as the main ideas of the dual formulation, the kernel trick and the change of loss function.

It is possible to derive a kernel version of logistic regression with L^2 regularization. The derivation can be made by first replacing $\mathbf{x}$ with $\boldsymbol{\phi}(\mathbf{x})$, and then use the representer theorem to derive its dual formulation and apply the kernel trick. Kernel logistic regression is the a classification counterpart to kernel ridge regression. It appears, however, to be little used in practice, and we will therefore go straight to support vector classification instead. Support vector classification is the classification counterpart to support vector regression. Both support vector regression and classification are referred to as support vector machines (SVM) since they both have sparse dual parameter vectors.

Support vector classification

We consider the binary classification problem $y \in \{-1, 1\}$. Just like (kernel) logistic regression, the support vector classifier is on the format

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}\left\{\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})\right\} \tag{8.30}$$

(cf. (5.8)). If we were to learn $\boldsymbol{\theta}$ using the logistic loss (5.10), we would obtain logistic regression with a nonlinear feature transformation $\boldsymbol{\phi}(\mathbf{x})$, from which kernel logistic regression eventually would follow. However, in analogy to support vector regression, we instead use the hinge loss function (5.13),

$$L(\mathbf{x}, y, \boldsymbol{\theta}) = \begin{cases} 1 - y\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) & \text{if } y\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) < 1 \\ 0, & \text{otherwise} \end{cases}. \tag{8.31}$$

From Figure 5.2, it is not immediate clear what advantages the hinge loss has over the logistic loss. However, as with the ϵ-insensitive loss, the main advantage of the hinge loss comes when we look at the dual formulation using $\boldsymbol{\alpha}$ instead of the primal formulation with $\boldsymbol{\theta}$.

Before introducing a dual formulation we first have to spell out the primal one. Since the representer theorem is lurking behind all this, we have to consider L^2 regularization, meaning that the primal formulation of our model is

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y_i\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)\right\} + \lambda\|\boldsymbol{\theta}\|_2^2 \tag{8.32}$$

The primal formulation does not allow for the kernel trick, since the feature vector does not show up as $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$. By using optimization theory and constructing the Lagrangian (the details are found in Appendix 8.A), we can arrive at the dual formulation[5] of (8.32),

$$\widehat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \ \frac{1}{2}\boldsymbol{\alpha}^\mathsf{T}\boldsymbol{K}(\mathbf{X},\mathbf{X})\boldsymbol{\alpha} - \boldsymbol{\alpha}^\mathsf{T}\mathbf{y} \tag{8.33a}$$

$$\text{subject to } \ |\alpha_i| \leq \frac{2}{n\lambda} \text{ and } 0 \leq \alpha_i y_i. \tag{8.33b}$$

with

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}\left(\widehat{\boldsymbol{\alpha}}\boldsymbol{K}(\mathbf{X},\mathbf{x}_\star)\right). \tag{8.33c}$$

instead of (8.30). Note that we here also have made use of the kernel trick by replacing what would be $\boldsymbol{\phi}(\mathbf{x})\boldsymbol{\phi}(\mathbf{x}')$ with $\kappa(\mathbf{x},\mathbf{x}')$ in $\boldsymbol{K}(\mathbf{X},\mathbf{X})$ in (8.33a) and $\boldsymbol{K}(\mathbf{X},\mathbf{x}_\star)$ in (8.33c).

Because of the representer theorem, the formulation (8.33c) should come as no surprise to us, since it simply is (8.15) inserted into (8.30). The representer theorem, however, only tells us that this dual formulation exists, but (8.33a)-(8.33b) does not follow automatically from the representer theorem but requires its own derivation.

The perhaps most interesting property of the constrained optimization problem (8.33) is that the solution $\widehat{\boldsymbol{\alpha}}$ turns out to be sparse. This is exactly the same phenomena as with support vector regression, and implies that also (8.33) is a support vector machine (SVM) type of method. More specifically, (8.33) is called support vector classification. The strength of this method, like support vector regression, is that the model has the full flexibility of being a kernel method, and yet the prediction (8.33c) only depends explicitly on some of the training data points (the support vectors). (Remember, however, that all training data points still are needed when solving (8.33a).)

The support vector property is achieved by the fact that the loss function is exactly equal to zero when the margin (see Section 5.1) is > 1. In the dual formulation, the α_i becomes nonzero only if the margin for datapoint i is < 1, which makes $\widehat{\boldsymbol{\alpha}}$ sparse.

As a consequence of using the hinge loss, as we discussed in Section 5.1, support vector classification does not provide probability estimates $g(\mathbf{x})$ but only a "hard" classification $\widehat{y}(\mathbf{x}_\star)$. The margin, in this case $y\widehat{\boldsymbol{\alpha}}\boldsymbol{K}(\mathbf{X},\mathbf{x}_\star)$, is not possible to interpret as a class probability estimate, because of the asymptotic minimizer of the hinge loss. As an alternative, however, it is possible to instead use the squared hinge loss or the Huberized squared hinge loss which allows for a probability interpretation of the margin. Since all these loss functions are exactly zero for margins > 1, they retain the support vector property with a sparse $\widehat{\boldsymbol{\alpha}}$. However, using another loss function than the hinge loss, implies another optimization problem than (8.33a)-(8.33b), since the derivation would no longer start in (8.32).

It is not *necessary* to make use of nonlinear transformations $\boldsymbol{\phi}(\mathbf{x})$, and thereby kernels, in support vector classification. Or, put differently, it is perfectly possible to use the linear kernel $\kappa(\mathbf{x},\mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$ (or any other kernel corresponding to a finite dimensional $\boldsymbol{\phi}(\mathbf{x})$) in (8.33). When using the linear kernel support vector classification (8.33) becomes a linear classifier, like logistic regression. In that case, we could (if we prefer) implement and solve the the primal formulation, since $\boldsymbol{\theta}$ is of finite dimension, and ignore the dual formulation. The support vector property would still be present, but much less visible. If only using the primal formulation and no kernels the representer theorem is not needed (it is only a matter of using the hinge loss to a linear classifier), and one could thereby also replace the L^2 regularization with any other regularization method.

Example 8.5:

ex:svc This example is not yet complete.

[5] In other texts, it is common to let the dual variables instead be $\alpha' = (\alpha_i y_i)$ (which happens to be the Lagrange multipliers). It is mathematically equivalent, but this formulation better highlights the similarities to the other methods and the importance of the representer theorem.

Algorithm 17: Support vector classification

1 To be completed

The support vector classifier is most often formulated as a solution to the binary classification problem. The generalization to the multiclass problem is unfortunately not straightforward, since it requires a multiclass generalization of the loss function, as discussed in Section 5.1. In practice it is common to construct a multiclass classifier from multiple binary ones using either the one-versus-rest or the one-versus-one strategy (see page 71).

8.5 Further reading

8.A The representer theorem and the derivation of support vector classification

9 The Bayesian approach and Gaussian processes

Chapter to be written.

10 User aspects of machine learning

Dealing with supervised machine learning problems in practice is to a great extent an engineering discipline where many practical issues have to be considered and where the scarce resource is in the end the available man-hours. To use this resource efficiently, we need to have a well-structured procedure for how to develop and improve the model. Multiple actions can potentially be taken. How do we know which action to take and is it really worth spending the time implementing them? Is it for example worth spending an extra week collecting and labeling more training data or should we do something else? These issues will be addressed in this chapter.

10.1 Defining the machine learning problem

Solving a machine learning problem in practice is an iterative process. We train the model, evaluate the model, and from there suggest an action for improvement and train the model again, and so on. To do this efficiently, we need to be able to tell whether a new model is an improvement over the previous model or not. One way to evaluate the model after each iteration would be to put it in production (for example running a traffic-sign classifier in a self-driving car for a few hours). Besides the obvious safety issues, this would be very time inefficient as well as inaccurate since it could still be hard to tell whether the proposed change was an actual improvement or not.

A better strategy is to automate this evaluation procedure without the need to put the model in production each time we want to evaluate its performance. We do this by putting aside a *validation* dataset and *test* dataset and evaluate the performance using a single number *evaluation metric*. This will define the machine learning problem that we are solving.

Training, validation and test data

In Chapter 4 we introduced the strategy of splitting the available data into training data, validation data and test data.

- **Training data** is used for training the model.
- **Hold-out validation data** is used for comparing different model structures, choosing hyperparameters of the model, feature selection, and so on.
- **Test data** is used to evaluate the performance of the final model.

If the amount of available data is small, it is possible to perform k-fold cross-validation instead of putting aside hold-out validation data, the idea of how to use it in the iterative procedure is unchanged. To get a final estimate of the performance, test data is always needed.

In the iterative procedure, the hold-out validation data (or k-fold cross-validation) is used to judge if the new model is an improvement over the previous model. This step is called *validation*. During the

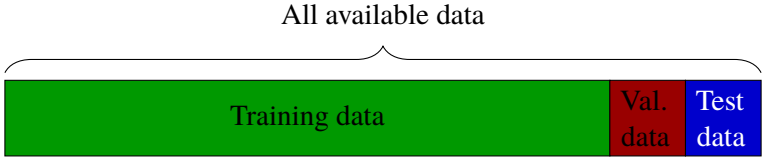


Figure 10.1: Splitting the data into training data, hold-out validation data and test data.

validation stage, we can also choose to train several new models and not just one new model. For example, if we have a neural network model and are interested in the number of hidden units to use in a certain layer, we can train several models, each with a different choice of hidden units. Afterwards, we pick the one that performs best on the validation data. In the next iteration we can use validation to choose the learning rate (by training multiple models using different learning rates), and so on.

Eventually, we will effectively have used the validation data to compare many models. Depending on the size of the validation data, we might risk picking a model that do particularly well on the validation data in comparison to completely unseen data. To detect this and to get a fair estimate of the actual performance of a model, we use the test data, which has neither been used during training nor validation. If the performance on the test data is substantially better than the performance on the validation data, we have overfitted on the validation data. The easiest solution in that case would be to extend the size of the validation data, if possible.

It is important that both the validation data and the test data always come from the same data distribution, namely the data distribution that we are expecting to see when we put the model into production. If they do not stem from the same distribution, we are validating and improving our model towards something that is not represented in the test data and hence "aiming for the wrong target". Preferably, also the training data should come from the same data distribution, but this requirement can be relaxed if we have good reasons to do so, more about this in Section 10.2.

Size of validation and test data

How much data should we set aside as hold-out validation data and test data, or should we even avoid setting aside hold-out validation data and use k-fold cross-validation instead? This depends on how much data we have available, which performance difference we plan to detect, and how many models we plan to compare. For example, if we have a classification model with a 99.8% accuracy and want to know if a new model is even better, a validation dataset of 100 samples will not be able to tell that difference. Also, if we plan to compare many (say, hundreds or more) different hyperparameter values and model structures using 100 validation data points, we will most likely overfit to that validation data.

If we have, say, 500 data points, a reasonable split would be 60%-20%-20% (i.e. 300-100-100 samples) for training-validation-test. With such a small validation dataset, we cannot afford to compare several hyperparameter values and model structures or detecting an improvement in accuracy of 0.1%. In this situation, we are probably better off using k-fold cross-validation to decrease the risk of overfitting to the validation data. Be aware, however, that the risk of overfitting to the data still exists even with k-fold cross-validation. We also still need to set aside test data if we want a final unbiased estimate of the performance.

Many machine learning problems have substantially larger datasets. Assume we have a dataset of 1 000 000 data points. In this scenario it would be enough to use a split of 98%-1%-1%, i.e. leaving 10 000 data points for validation and test, respectively, unless we really care about the very last decimals in performance. In this scenario it is also of less use with k-fold cross-validation, since having all 99%=98%+1% (training+validation) available for training would make a small difference in comparison to using “only” 98%. Also the price for training k models (instead of only one) with this amount of data would be much higher.

Another advantage of having a hold-out validation dataset is that we can allow the training data to come from a slightly different distribution than the validation and test dataset, for example if that would enable us to find a much larger training dataset. We will discuss this more in Section 10.2.

Single number evaluation metric

In Section 4.5 we introduced additional metrics, besides the misclassification rate such as precision, recall and F1-score, for evaluating binary classifiers. There is no unique answer to which metric is the most appropriate for your. What metric to pick is rather a part of the problem definition. To improve you model quickly and in a more automated fashion, it is advisable to agree on a single number evaluation metric,

especially if you are a larger team of engineers working on the problem. Such a metric, together with the validation and test data, defines the problem, and without a proper problem definition it is difficult to solve that problem and to improve its solution. If we during the development process realize that the metric does not favor the properties we want a good model to have, we can always change that metric later.

The single number evaluation metric together with the validation data defines the supervised machine learning problem. Having an efficient procedure in place where we can evaluate the model on the hold-out validation data (or by k-fold cross-validation) using the metric, allows us to speed up the iterations since we quickly can see if a proposed change to the model improves the performance or not. This is important in order to manage an efficient workflow of trying out and accepting or rejecting new models.

10.2 Improving a machine learning model

As already mentioned, solving a machine learning problem is an iterative procedure where we train, evaluate, and suggest action for improvement, for instance by changing some hyperparameters or trying another methods. How do we start this iterative procedure?

Try simple things first

A good strategy is to *try simple things first*, for example k-NN or linear/logistic regression. Trying a simple thing first allows us to start early with the iterative procedure of finding a good model. This is important, since it might reveal important aspects of the problem formulation that we need to re-think before it makes sense to proceed with more complicated models. It also reduces the risk of ending up with a too complicated model when a much simpler model would be just as good (or even better).

There are many actions that could be taken to improve the model, for example changing the type of model, increasing/decreasing model complexity, changing input variables, collecting more data, starting correcting mislabeled data (if there are any) etc. What should we do next? Two possible strategies for guiding us to meaningful actions to improve the solution are by trading *training error and generalization gap*, or by applying *error analysis*.

Training error vs generalization gap

With the notation from Chapter 4, E_{train} is the performance of the model on training data and $E_{\text{hold-out}}$ the performance on hold-out validation data. In the validation step, we are interested in changing the model such that $E_{\text{hold-out}}$ is minimized. We can write the hold-out validation error as a sum of the training error and the generalization gap as

$$E_{\text{hold-out}} = E_{\text{train}} + \underbrace{(E_{\text{hold-out}} - E_{\text{train}})}_{\approx \text{ generalization gap}}. \tag{10.1}$$

In words, the generalization gap is the difference between the validation error $E_{\text{hold-out}}$ and the training error E_{train}. [1]

Once the validation step is completed ($E_{\text{hold-out}}$ or $E_{k\text{-fold}}$ computed), we can easily compute the training error E_{train} and the generalization gap $E_{\text{hold-out}} - E_{\text{train}}$. By computing these quantities, we can actually get good guidance for what changes we may consider for the next iteration.

As we discussed in Chapter 4, if the training error is small and the generalization gap is big (E_{train} small, $E_{\text{hold-out}}$ big), we have typically overfitted the model. The opposite situation, big training error and small generalization gap (both E_{train} and $E_{\text{hold-out}}$ big), typically indicates underfitting.

If we want to reduce the generalization gap $E_{\text{hold-out}} - E_{\text{train}}$ (reduce overfitting), the following actions can be explored:

[1] This can be related to (4.11), if approximating $\bar{E}_{\text{train}} \approx E_{\text{train}}$ and $\bar{E}_{\text{new}} \approx E_{\text{hold-out}}$. If using k-fold cross validation, $\bar{E}_{\text{new}} \approx E_{k\text{-fold}}$ would be an equally good approximation when computing the generalization gap.

- *Use a less flexible model.* If we have a very flexible model we might start overfitting to the training data, i.e. that E_{train} is much smaller than $E_{\text{hold-out}}$. If we use a less flexible model, we also reduce this gap.
- *Use more regularization.* Using more regularization will reduce the flexibility of the model, and hence also reduce the generalization gap.
- *Use bagging*, or use more ensemble members if we already are using it. Bagging is a method for reducing the variance of the model, which typically also means that we reduce the generalization gap.
- *Collect more training data.* If we collect more training data, the model is less prone to overfit that extended training dataset and is forced to only focus on aspects which generalizes to the validation data.

If we want to reduce the training error E_{train} (reduce underfitting), the following actions can be considered:

- *Use a more flexible model* that is able to fit the training data better. This can be to change a hyperparameter in the model we are considering, for example decreasing k in k-NN or changing the model to a more flexible one, for example change the linear regression model to deep neural network.
- *Extend the set of input variables.* If we suspect that there are more input variables that carry information, we might want to extend the data with these input variables.
- *Use less regularization.* Can of course only be applied if regularization is used at all.
- *Train the model longer* (only for models that are trained iteratively and aborted before reaching the minimum, like a neural network).

We summarize the above discussion in Figure 10.2. Fortunately, evaluating E_{train} and $E_{\text{hold-out}}$ is cheap. We only have to evaluate the model on the training data and the validation data, respectively. Yet, it gives us good advice on what actions to take next. Besides suggesting what action to explore next, this procedure also tells us what *not* to do: If the training error E_{train} is large and the generalization gap small ($E_{\text{hold-out}}$ also large), collecting more training data will most likely not help. Furthermore, if we have a large generalization gap and a small training error ($E_{\text{hold-out}} \gg E_{\text{train}} \approx 0$) a more flexible model will most likely not help.

Error analysis

Another strategy to identify actions that can improve the model is to perform *error analysis*. Below we only describe error analysis for classification problems, but the same strategy can be applied to regression problems as well.

In error analysis we manually look at a subset, say 100 data points, of the validation data that the model classified incorrectly. Such an analysis does not take much time, but might give valuable clues as to what type of data the model is struggling with and how much improvement we can expect if by fixing these issues. We illustrate the procedure with an example.

Example 10.1: Error analysis applied to vehicle detection

Consider a classification problem of detecting cars, bicycles and pedestrians in an image. The model takes an image as input and one of the four classes `car`, `bike`, `pedestrian`, or `other` as output. Assume that the model has a classification accuracy of 90% on validation data.

When looking at a subset of 100 images that were misclassified in the validation data, we make the following observations:

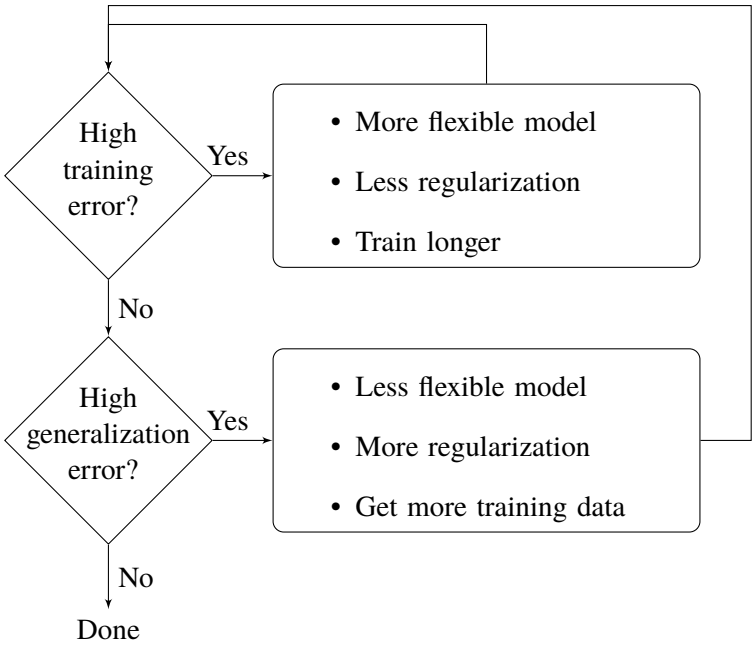


Figure 10.2: The iterative procedure of improving the model based on the decomposition into on the training error and generalization gap .

- All 10 images of class `pedestrian` that were incorrectly classified as `bike` contained a baby carrier.
- 30 images were substantially tilted.
- 15 images were mislabeled.

From this observation we can conclude:

- If we launch a project for improving the model to classify pedestrians with baby carriers as `pedestrian` and not incorrectly as `bike`, an improvement of at most a $\sim 1\%$ (a tenth of the 10% classification error rate) can be expected.
- If we improve the performance on tilted images, an improvement of at most $\sim 3\%$ can be expacted.
- If we correct all mislabeled data, an improvement of at most $\sim 1.5\%$ can be expected.

Following the example, we get an indication on what improvement we can expect by tackling these three issue. These numbers should be considered as the maximal possible improvement. To prioritize which aspect to focus on, we should also consider what strategies are available for improving them, how much progress we expect to make applying these strategies and how much effort we have to invest fixing these issues.

For example, to improve the performance on tilted images we could try to extend the training data by augmenting it with more tilted images. This strategy could be investigated without too much extra effort by augmenting the training data with tilted versions of the training data point that we already have. Since, this could be applied fairly quickly and have a maximal performance increase of 3%, it seems to be a good thing to try out.

To improve the performance on the images with baby carriers, one approach would be to collect more images of pedestrians with baby carriers. This obviously requires some more manual work and can be questioned if it is worth the effort since it would only give performance improvement of at most 1%.

Regarding the mislabeled data, the obvious actions to take to improve on this issue is to manually go through the data and correct these labels. In the example above we may say it is not quite worth the effort of getting an improvement of 1.5%. However, assume that we have improved the model with other actions to an accuracy of 98.0% on validation data and that still 1.5% of the total error is due to mislabeled data,

this issue is now quite relevant to address if we want to improve the model further. Remember, the purpose of the validation data is to choose between different models. This purpose is degraded when the majority of the reported error on validation is due to incorrectly labeled data rather than the actual performance of the model.

There are three levels of ambitions for correcting the labels:

1. Go through the data points in the validation/test data that were misclassified by the best algorithm and correct these labels.

2. Go through all data points in the validation/test data and correct the labels.

3. Go through all data points, including the training data and correct the labels.

Approach 1 is in general not recommended since there could be mislabeled data points that the algorithm also classified to belong to that mislabeled class. These data points would then not be corrected, resulting in a too optimistic estimate of the performance on validation and test data. Therefore, it is safer to go through all data points in the validation and test data as suggested in approach 2. The advantage of approach 1, in comparison to approach 2, is the less amount of work it requires. If we have a model with 98% accuracy, there are 50 times less data to process in comparison to approach 2. Also, note that correcting labels in test and validation data and test data only does not necessarily increase the performance of model in production, but it will give us a more fair estimate of the actual performance of the model.

An even more ambitious approach is to go through all mislabeled data in the training data as well, as suggested in approach 3. This is also substantially more labor intensive. Assume, for example, that we have made a $98\% - 1\% - 1\%$ split of training-validation-test data. Then it is yet another factor 50 more time consuming in comparison with approach 2. Unless the mislabeling is systematic, correcting the labels in the training data does not necessarily pay off.

Applying the data cleaning to validation and test data only, as suggested in approach 2, will result in the training data coming from a slightly different distribution than the validation and test data. However, if we are eager to correct the mislabeled data in the training data as well, a good recommendation would still be to start correcting validation and test data only and then use the techniques in the following section to see how much extra performance we can expect by cleaning data in the training data as well before lunching that substantially more labor intensive data cleaning project.

Mismatched training and validation/test data

As already pointed out in Chapter 4, we should strive for letting the training data come from the same distribution as the validation and test data. However, there are situations where we, for different reasons, can accept the training data to come from a slightly different distribution than the validation and test data. One reason was presented in the previous section where we choose to correct mislabeled data in validation and test data, but not necessarily invest the time to do the same correction to the training data.

Another reason is that we might have access to another substantially larger data set which comes from a slightly different distribution than the data we care about, but similar enough that the advantage of having a larger training data overcomes the disadvantage of that data mismatch. This scenario is further elaborated in Section 10.3.

If we have a mismatch between training data and validation/test data, that mismatch contributes to yet another error source of the final validation error $E_{\text{hold-out}}$ that we care about. We want to estimate the magnitude of that error source. This can be done by revising the training-validation-test data split. From the training data we carve out a separate *training-validation* dataset, see Figure 10.3. That dataset is neither used for training nor for validation. However, we do evaluate the performance of our model on that dataset as well. As before, the remaining part of the training data is used for training, the validation data is used for comparing different model structures, and test data is used for evaluating the final performance of the model.

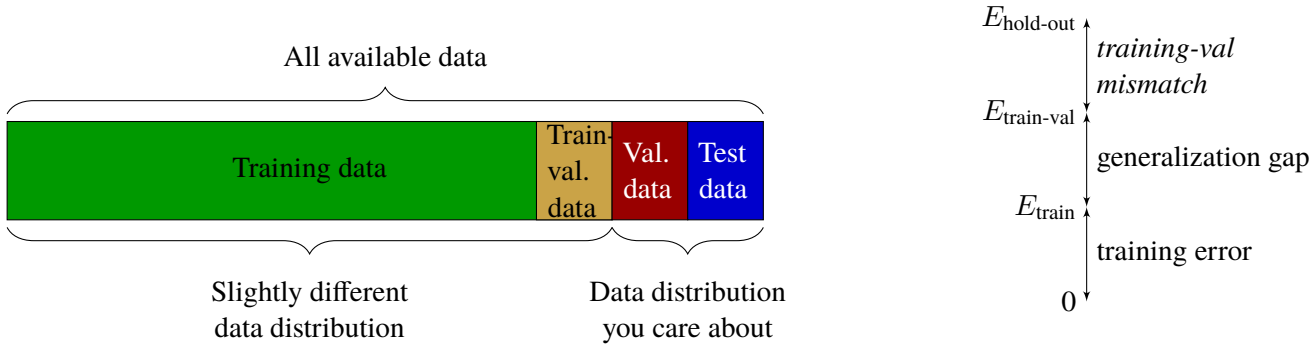


Figure 10.3: Revising the training-validation-test data split by carving out a separate train-validation data from the training data.

This revised data split also allows us to revise the decomposition in (10.1)

$$E_{\text{hold-out}} = E_{\text{train}} + \underbrace{(E_{\text{train-val}} - E_{\text{train}})}_{\approx \text{ generalization gap}} + \underbrace{(E_{\text{hold-out}} - E_{\text{train-val}})}_{\approx \text{ Train-val mismatch}}, \tag{10.2}$$

where $E_{\text{train-val}}$ is the performance of the model on the new training-validation data and where, as before, $E_{\text{hold-out}}$ and E_{train} are the performances on validation and training data, respectively. With this new decomposition, the term $E_{\text{train-val}} - E_{\text{train}}$ is an approximation of the generalization gap, i.e. how well the model generalizes to unseen data *of the same distribution* as the training data, whereas the term $E_{\text{hold-out}} - E_{\text{train-val}}$ is the error related to the training-validation data mismatch. If the term $E_{\text{hold-out}} - E_{\text{train-val}}$ is small in comparison to the other two terms, it seems likely that the training-validation data mismatch is not a big problem and that it is better to focus on techniques reducing the other training error and the generalization gap as we talked about earlier. On the other hand, if $E_{\text{hold-out}} - E_{\text{train-val}}$ is significant, the data mismatch does have an impact and it might be worth investing time reducing that term. For example, if the mismatch is caused by the fact that we only corrected labels in the validation and test data, we might want to consider correcting labels for the training data as well.

10.3 What if we cannot collect more data?

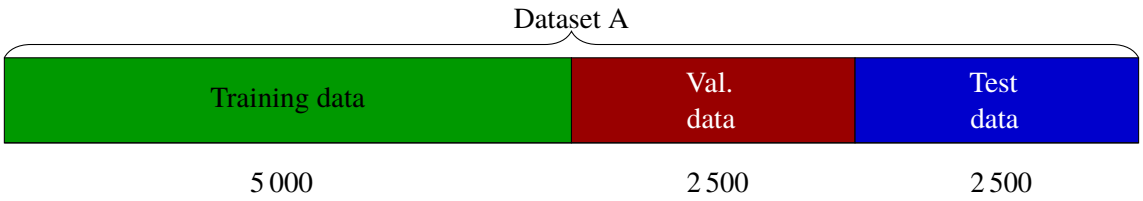
We have seen in Section 10.2 that collecting more data is a good strategy to reduce the generalization gap and hence reduce overfitting. However, collecting labeled data is usually expensive and sometimes not even possible. What can we do if we cannot afford to collect more data but still want to benefit from the advantages that a larger dataset would give? In this section a few approaches are presented.

Extending the training data with slightly different data

As already mentioned, there are situations where we can accept the training data to come from a slightly different distribution than the validation and test data. One reason to accept this is if we then would have access to a substantially larger training data set.

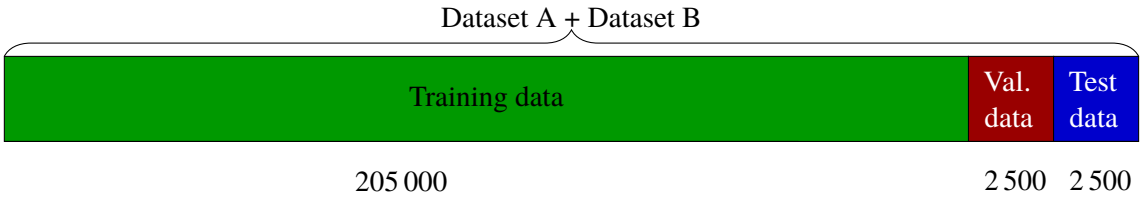
Consider a problem with 10 000 data points representing the data that we also would expect to get when the model is deployed in production. We call this dataset A. We also have another dataset with 200 000 data points that come from a slightly different distribution, but which is similar enough that we think exploiting information from that data can improve the model. We call this dataset B. Some options to proceed would be the following:

- **Option 1** Use only dataset A and split it into training, validation, and test data.



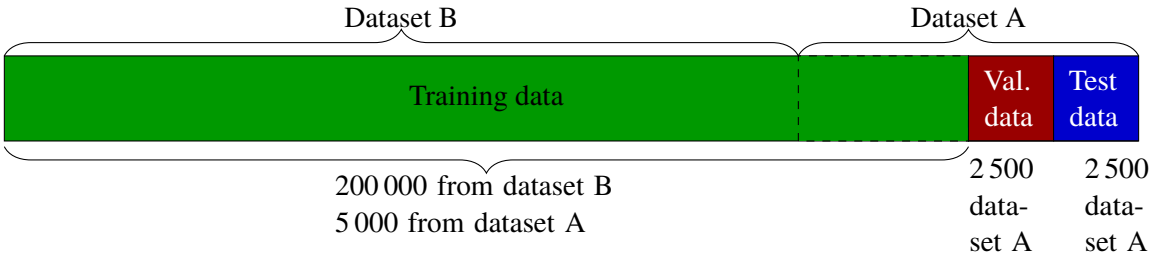
The advantage of this option is that we only train, validate, and evaluate on dataset A, which is also the type of data that we want our model to perform well on. The disadvantage is that we have quite few data points and we do not exploit potentially useful information in the larger dataset B.

- **Option 2** Use both dataset A and dataset B. Randomly shuffle the data and split it in training, validation and test data.



The advantage over option 1 is that we have a lot more data available for training. However, the disadvantage is that we mainly evaluate the model on data from dataset B, whereas we want our model to perform well on data from dataset A.

- **Option 3** Use both dataset A and dataset B. Use data points from dataset A for validation data and test data and some in the training data. Dataset B only goes into the training data.



Similar to option 2, the advantage is that we have more training data in comparison to option 1 and in contrast to option 2 we now evaluate the model on data from dataset A, which is the data we want our model to perform well on. However, one disadvantage is that the training data no longer have the same distribution as the validation and test data.

From these three options we would strongly recommend option 3. We do exploit the information available in the much larger dataset B, but evaluate only on the data where we want the model to perform well on (dataset A). The main disadvantage with option 3 is that the training data no longer come from the same distribution as the validation data and test data. In order to quantify how big impact this mismatch has on the final performance, the techniques described in Section 10.2 can be used. To push the model to do better on data from dataset A during training, we can also consider giving data from dataset A higher weight in the cost function than data from dataset B.

Artificially extending the training data set

Data augmentation is another approach to extend the dataset without the need to collect more data. In data augmentation we construct new data points duplicating the existing data by applying invariant transformations. This is especially common for images where such invariant transformations can be scaling, rotation, and vertical flipping of the images. For example, if we vertically flip an image of a cat, it still displays a cat, see the examples Figure 10.4.

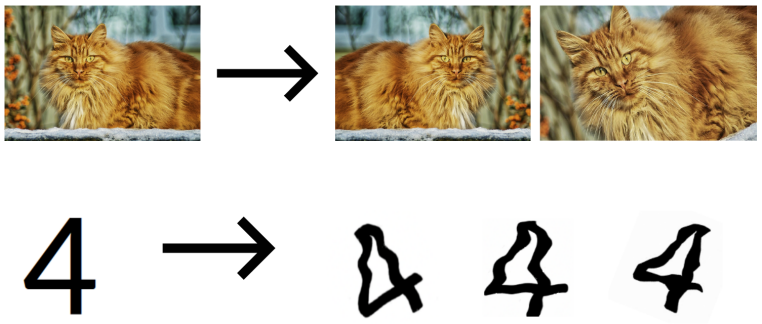


Figure 10.4: Different examples of data augmentation applied to images. Above: An image of a cat has been reproduced by tilting and vertical flipping. Below: An image of a digit has been reproduced by tilting and blurring.

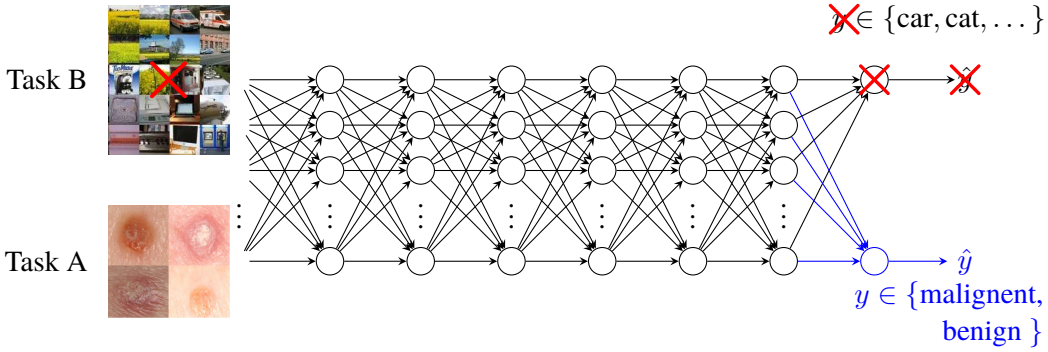


Figure 10.5: In transfer learning we reuse models that have been trained on a different task than the one we are interested in. Here we reuse a model which has been trained on images displaying all sorts of classes, such as cars, cats, computers and later train only the last few layers on the skin cancer data which is the task we are interested in.

Transfer learning

Yet another technique to effectively use more data than the dataset we have is by applying transfer learning. In transfer learning we use the knowledge from a model that has been trained on a different task with a different dataset and then apply that model in solving a different, but slightly related, problem.

Transfer learning is especially common for sequential model structures such as the neural network models introduced in Chapter 6. Consider an application where we want to detect whether a certain type of skin cancer is malignant or benign, and for this task we have 100 000 labeled images of skin cancer. We call this task A. Instead of training the full neural network from scratch on this data, we can reuse an already pre-trained network from another image classification task (task B), which preferably has been trained on a much larger dataset, not necessarily containing images even resembling skin cancer tumors. By using the weights from the model trained in task B and only train the last few layers on the data in task A, we can get a better model than if the whole model had been trained on just dataset A. The procedure is also displayed in Figure 10.5. The intuition is that the layers closer to the input accomplish tasks that are generic for all types of images, such as extracting lines, edges and corners in the image, whereas the layers closer to the output are more specific to the particular problem.

In order for transfer learning to be applicable, we need the two tasks to have the same type of input (in the example above, images of the same dimension). Further, for transfer learning to be an attractive option, the task that we transfer from should have been trained on substantially more data than the task we transfer to.

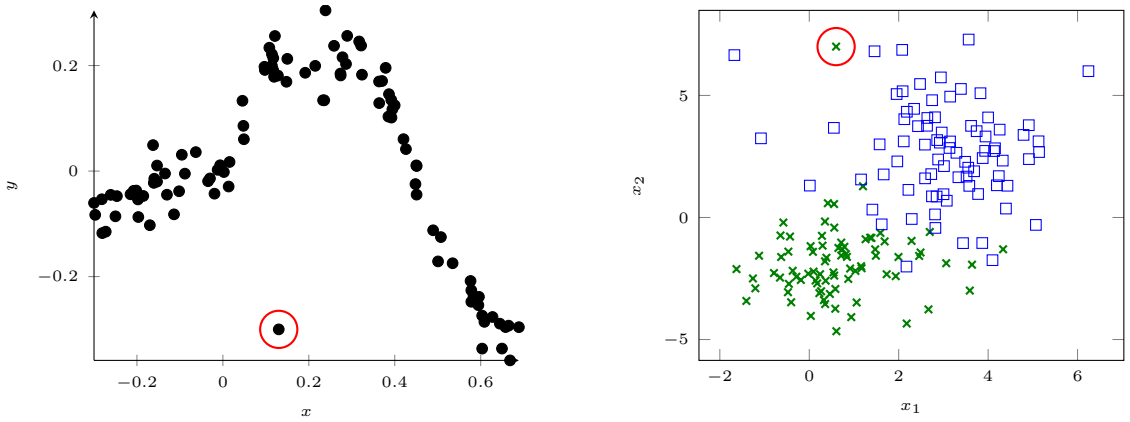


Figure 10.6: Two typical examples of outliers (marked with red circle) in regression (left) and classification (right), respectively.

10.4 Practical data issues

Outliers

In some applications, a common issue is *outliers*, meaning data points whose outputs do not follow the overall pattern. Two typical examples of outliers are sketched in Figure 10.6. Even though the situation in Figure 10.6 looks simple, it can be quite hard to find outliers in data where the input dimension is higher. The error analysis discussed in Section 10.2, which amounts to inspecting misclassified samples in the validation data, is a systematic way to discover outliers.

When facing a problem with outliers, the first question to ask is whether the outliers are meant to be captured by the model or not. Do the encircled data points in Figure 10.6 describe an interesting phenomena that we would like to predict, or are they irrelevant noise (possibly originating from a poor data collection process)? The answer to this question depends on the actual problem and ambition. Since outliers by definition (no matter their origin) do not follow the overall pattern, they are typically hard to predict.

If the outliers are not of any interest, there are basically two approaches one could take. The first approach is to simply delete (or replace) the outliers in the data. Unfortunately this means that one has to first find the outliers, which can be hard, but sometimes some thresholding and manual inspection (that is, look at all data points whose output value is smaller/larger than some value) can help. Once the outliers are removed from the data, one can proceed as usual. The second approach is to instead make sure that the learning algorithm is robust against outliers, if applicable, for example by using a robust loss function such as absolute error instead of squared error loss (see Chapter 5 for more details). Making a model more robust amounts, to some extent, to making it less flexible. However, robustness amounts to making the model less flexible in a particular way, namely by putting less emphasis on the data points whose predictions are severely wrong.

If the outliers are of interest to the prediction, they are not really any problem. We just have to use a model that is flexible enough to capture the behavior (small bias). This has to be done with care, since very flexible models have a high risk of overfitting also to noise. If it turns out that the outliers in a classification problem indeed are interesting and in fact are from an underrepresented class, we are rather facing an imbalanced problem, which was discussed in Section 4.5.

Missing data

A common practical issue is that certain values are sporadically missing in the data. As always in this book, the data consists of input-output pairs $\{\mathbf{x}_i, y_i\}_{i=1}^n$, and *missing data* refers to the situation where some (or a few) values from either the input $\mathbf{x}_i$ or the output y_i, for some i, is missing. It is a common practice to denote missing data in a computer with `NaN` (not a number), but less obvious codings also exists, such

as 0. Reasons for missing data could for instance be a malfunctioning sensor or similar issues at data collection time, or that certain values for some reason have been discarded during the data processing.

Much like outliers, there is no universal solution for how to handle missing data. There is, however, some common practice which can serve as a guideline. First of all, if the output y_i would be missing, the data point is useless for supervised machine learning,[2] and can be discarded. In the following, we assume that the missing values are only in the input $\mathbf{x}_i$.

The easiest way to handle missing data is to discard the entire data points ("rows in $\mathbf{X}$") where data is missing. That is, if some feature is missing in $\mathbf{x}_i$, the entire input sample $\mathbf{x}_i$ and its corresponding output value y_i is discarded from the data, and we are left with a smaller dataset. If the dataset that remains after this procedure still contains enough data, this approach can work well. However, if this would lead to a too small dataset, it is of course problematic. More subtle, but also important, is if the data is missing in a systematic fashion, such that missing data is more common for a certain class. In such a situation, discarding data points with missing data would lead to a dataset that no longer would represent the reality well and thereby degrade the performance of the learned model.

If missing data are common, but only for certain features, another easy option is to not use those features ("columns in $\mathbf{X}$") where data is missing. It depends on the situation whether this is a fruitful approach or not.

10.5 Further reading

User aspects of machine learning is a fairly under-explored area both in academic research publications and in standard textbooks on machine learning. Two exceptions are Ng (2019) and Burkov (2020) from which parts of this chapter has been inspired. Connected to data augmentation, see further in Shorten and Khoshgoftaar (2019) for a review of different data augmenting techniques for images.

[2] The "partly labeled data" problem is a semi-supervised problem, and hence not part of this book. There are methods where also data points with missing y_i can contribute to the learning, however under assumptions that might be hard to verify in practice.

11 Generative models and learning from unlabeled data

The models introduced so far in this book are so-called *discriminative* models, meaning (in the context of supervised learning) that they are designed with the aim of predicting new outputs. We will in this chapter introduce model of another type, namely so-called *generative* models. Generative models belongs to another type of modeling paradigm. They are indeed also learned from data, but their scope is wider than, and predictions are merely obtained as a by-product from a generative model. Generative modeling is therefore a natural way to take us beyond supervised learning, which we will do in this chapter.

A generative model aims to describe the distribution $p(\mathbf{x}, y)$, that is, how the data (both its input and output) is generated. To make predictions with a generative model, the expression for $p(y \mid \mathbf{x})$ has to be derived from $p(\mathbf{x}, y)$ using probability theory. We will make this idea concrete by considering the rather simple, yet useful, generative Gaussian mixture model. The Gaussian mixture model can, as any generative model, be used for different purposes, and when used for classification it results in the classifiers traditionally named linear and quadratic discriminant analysis (LDA and QDA, respectively). We will thereafter see how the Gaussian mixture model can be used for semi-supervised (where labels y are partly missing) and unsupervised learning (where no labels are present at all; there are only $\mathbf{x}$ and no y) as well. In the latter case, the Gaussian mixture model can be used for solving the so-called clustering problem where similar $\mathbf{x}$ are to be grouped together.

Generative models bridge the gap between supervised and unsupervised machine learning, but not all methods for unsupervised learning comes from generative models. We therefore finally also review two other popular unsupervised methods (not derived from generative models), namely k-means and principal component analysis (PCA). k-means is another clustering method, whereas the purpose of PCA is to infer which dimensions of $\mathbf{x}$ that are most (and least) informative, so-called dimensionality reduction.

11.1 The Gaussian mixture model and the LDA & QDA classifiers

We will now introduce a generative model, the Gaussian mixture model, from which we will derive several methods for different purposes. The Gaussian mixture model attempts to model $p(\mathbf{x}, y)$, that is, the *joint* distribution between inputs $\mathbf{x}$ and outputs y. (The discriminative models in previous chapters only attempts to model the the conditional distribution $p(y \mid \mathbf{x})$, a less ambitious problem since $p(y \mid \mathbf{x})$ can be derived from $p(\mathbf{x}, y)$ but not vice versa.) For the Gaussian mixture model, we assume that $\mathbf{x}$ is a numerical and y a categorical variable.

The Gaussian mixture model

The Gaussian mixture model makes use of the factorization

$$p(\mathbf{x}, y) = p(\mathbf{x} \mid y)p(y), \tag{11.1a}$$

where $p(\mathbf{x} \mid y)$ is *assumed* to be a Gaussian distribution

$$p(\mathbf{x} \mid y) = \mathcal{N}\left(\mathbf{x} \mid \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y\right) \tag{11.1b}$$

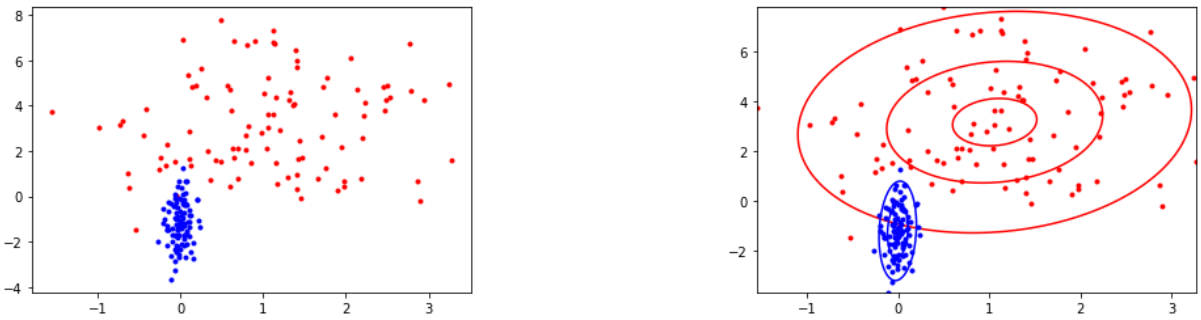


Figure 11.1: The Gaussian mixture model is a generative model, from which several useful machine learning methods can be derived. In this model we think about the input variables $\mathbf{x}$ as random and *assume* that they have a certain distribution. The Gaussian mixture model assumes that $p(\mathbf{x} \mid y)$ has a Gaussian distribution for each y. In the figure above, y is either red or blue, and for each value of y the contour lines of $p(\mathbf{x} \mid y)$ are shown. However, when using a method derived from this model (such as QDA), the collected data $\mathbf{x}$ may not actually have a Gaussian distribution; the data always is what it is, no matter what assumption are made by the model.

where $\boldsymbol{\mu}_y$ and $\boldsymbol{\Sigma}_y$ depends on y. Since y is categorical, and thereby takes values $1, \ldots, M$, the distribution $p(y)$ is modeled with M parameters $\{\pi_m\}_{m=1}^M$ as

$$p(y = 1) = \pi_1, \tag{11.1c}$$

$$\vdots$$

$$p(y = M) = \pi_M. \tag{11.1d}$$

Altogether, (11.1) is a generative model which describes an *assumption* on how data $\{\mathbf{x}, y\}$ is generated.

As it turns out, the Gaussian mixture model (11.1) will lead us to classifiers (for supervised learning) which are easy to learn (requires no numerical optimization, in contrast to logistic regression) and are useful in practice also when the data do not obey the Gaussian assumption (11.1b) perfectly. The classifiers are for historical reasons called linear and quadratic discriminant analysis (LDA[1] and QDA, respectively). In addition it will also lead us to a method for unsupervised learning as well as being useful also for the in-between problem when the labels y are missing for some of the data points.

Supervised learning of the Gaussian mixture model

Like other machine learning models, also the Gaussian mixture model (11.1) can be learned from training data. The unknown parameters in (11.1) are $\boldsymbol{\theta} = \{\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m, \pi_m\}_{m=1}^M$. We start by the supervised case, meaning that the training data does contain inputs $\mathbf{x}$ and outputs y as $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ (which has been the case for all other methods in this book so far). We will, however, later see how we can learn the Gaussian mixture model also when the output y is missing.

Mathematically, we learn the Gaussian mixture model by maximizing the likelihood

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathcal{T} \mid \boldsymbol{\theta}). \tag{11.2}$$

When $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, this optimization problem turns out to have the closed form solution

$$\widehat{\pi}_m = \frac{n_m}{n}, \tag{11.3a}$$

where n_m is the number of training data samples in class m. (Consequently, all n_m must sum to n, and thereby $\sum_m \widehat{\pi}_m = 1$.) Furthermore, the mean vector $\boldsymbol{\mu}_m$ of each class is learned as

$$\widehat{\boldsymbol{\mu}}_m = \frac{1}{n_m} \sum_{i:y_i=m} \mathbf{x}_i, \tag{11.3b}$$

[1] Note to be confused with Latent Dirichlet Allocation, also abbreviated LDA, which is a completely different method.

the empirical mean among all training samples of class m, and the covariance matrix $\boldsymbol{\Sigma}_m$ of each class $m = 1, \ldots, M$, is learned as

$$\widehat{\boldsymbol{\Sigma}}_m = \frac{1}{n_m} \sum_{i:y_i=m} (\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^\mathsf{T}. \tag{11.3c}$$

Note that we can compute the learned parameters $\widehat{\boldsymbol{\theta}}$ no matter if the data comes from a distribution where $p(\mathbf{x} \mid y)$ is Gaussian or not. In fact (11.3b)-(11.3c) learns a Gaussian distribution to $\mathbf{x}$ for each class such that the mean and covariance fits the data (a so-called moment-matching).

Predicting output labels for new inputs

We have so far described the generative Gaussian mixture model $p(\mathbf{x}, y)$ for numerical $\mathbf{x}$ and categorical y, and how to learn the unknown parameters in $p(\mathbf{x}, y)$ from training data. We will now discuss how this can be used as a classifier for supervised machine learning.

The key insight for using a generative model $p(\mathbf{x}, y)$ to make predictions is to realize that prediction "only" amounts to compute $p(y \mid \mathbf{x})$. From probability theory we have

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} = \frac{p(\mathbf{x}, y)}{\sum_{m=1}^{M} p(\mathbf{x}, m)}. \tag{11.4}$$

The left hand side $p(y \mid \mathbf{x})$ is the prediction whereas all expressions on the right hand side are defined by the generative Gaussian mixture model (11.1). We therefore get the classifier

$$p(y = m \mid \mathbf{x}_\star) = \frac{\widehat{\pi}_m \mathcal{N}\left(\mathbf{x}_\star \mid \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\right)}{\sum_{j=1}^{M} \widehat{\pi}_j \mathcal{N}\left(\mathbf{x}_\star \mid \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}_j\right)}, \tag{11.5}$$

which historically has been called quadratic discriminant analysis (QDA). As usual, we can obtain "hard" predictions $\widehat{y}_\star$ by selecting the class which is predicted to be the most probable,

$$\widehat{y}_\star = \arg\max_m p(y = m \mid \mathbf{x}_\star), \tag{11.6}$$

and compute corresponding decision boundaries. It turns out that the decision boundary for QDA is always a quadratic function (hence its name). We summarize this by Algorithm 18 and Figure 11.2, and in Figure 11.3 we show the decision boundary when the Gaussian mixture model from Figure 11.1 is turned into a QDA classifier.

Algorithm 18: Quadratic Discriminant Analysis, QDA

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

Learn
for $k = 1, \ldots, M$ **do**
  Compute $\widehat{\pi}_m$ (11.3a), $\widehat{\boldsymbol{\mu}}_m$ (11.3b) and $\widehat{\boldsymbol{\Sigma}}_m$ (11.3c)
end
Predict
for $k = 1, \ldots, M$ **do**
  Compute $p(y = m \mid \mathbf{x}_\star)$ (11.5)
end
Find largest $p(y = m \mid \mathbf{x}_\star)$ and set $\widehat{y}_\star$ to that k

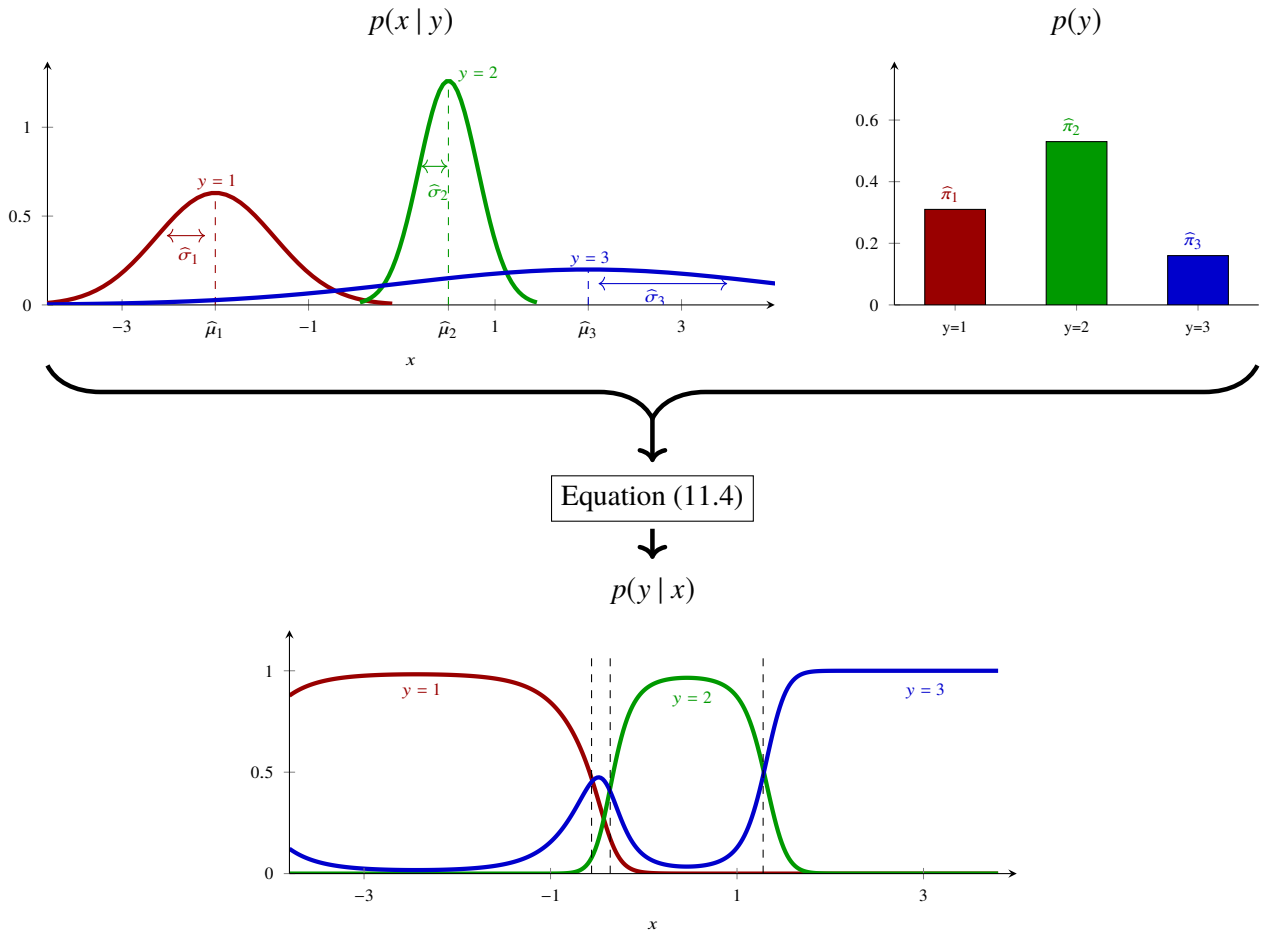


Figure 11.2: An illustration of QDA for $M = 3$ classes, with dimension $p = 1$ of the input $\mathbf{x}$. In the top is the generative Gaussian mixture model shown. To the left is the Gaussian model of $p(\mathbf{x} \mid k)$, parameterized by $\widehat{\boldsymbol{\mu}}_m$ and $\widehat{\boldsymbol{\Sigma}}_m$. To the right is the model of $p(y)$ shown, parameterized by $\widehat{\pi}_m$. All parameters are learned from training data, not shown in the figure. (Since $p = 1$, we only have a scalar variance $\boldsymbol{\Sigma}_m^2$, instead of a covariance matrix $\boldsymbol{\Sigma}_m$). By (11.4) is the generative model "warped" into $p(y = m \mid x)$, shown in the bottom. The decision boundaries are shown as vertical dotted lines in the bottom plot.

It is possible to make the restriction that the covariance matrix is equal for all classes, $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \cdots = \boldsymbol{\Sigma}_M = \boldsymbol{\Sigma}$ in (11.1b). With that restriction (11.3c) is replaced with

$$\widehat{\boldsymbol{\Sigma}} = \frac{1}{n - M} \sum_{m=1}^{M} \sum_{i:y_i=m} (\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^{\mathsf{T}}. \tag{11.7}$$

Using this in (11.5) leads to the so-called linear discriminant analysis (LDA), a classifier with linear decision boundaries. Note that LDA is obtained by replacing (11.3c) with (11.7) in Algorithm 18. We compare LDA and QDA in Figure 11.4.

> ***Time to reflect 11.1:*** *In the Gaussian mixture model it was assumed that $p(\mathbf{x} \mid y)$ is Gaussian. When applying LDA or QDA "out of the box" for a supervised problem, is there any check that the Gaussian assumption actually holds? If yes, what? If no, is that a problem?*

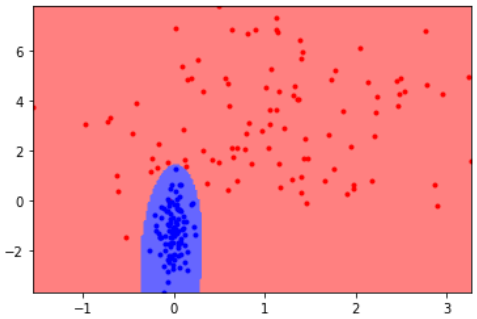
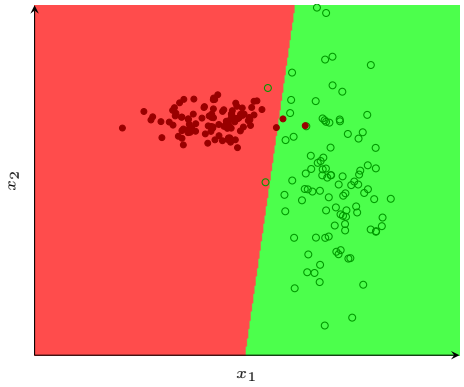
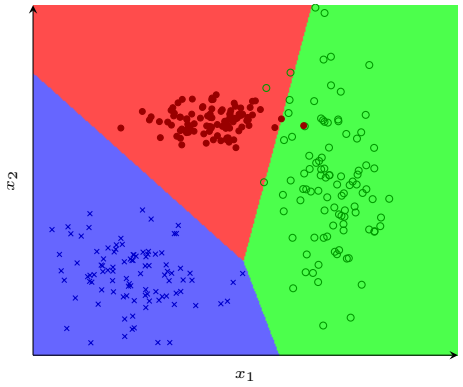


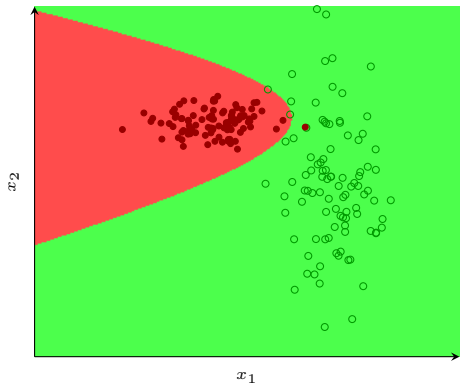
Figure 11.3: The decision boundary for the QDA classifier which is obtained by (11.5) and (11.6) from the Gaussian mixture model in Figure 11.1.



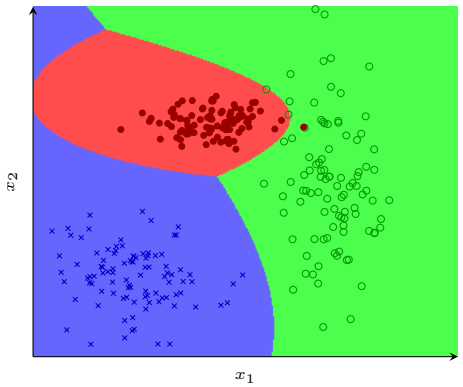
(a) LDA for $M = 2$ classes always gives a linear decision boundary. The red dots and green circles are training data from different classes, and the intersection between the red and green fields is the decision boundary obtained for an LDA classifier learned from the training data.



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

Figure 11.4: Examples of decision boundaries for LDA and QDA, respectively. TODO: Replace with music example.

11.2 The Gaussian mixture model when some or all labels are missing

Semi-supervised learning of the Gaussian mixture model

Unsupervised learning of the Gaussian mixture model

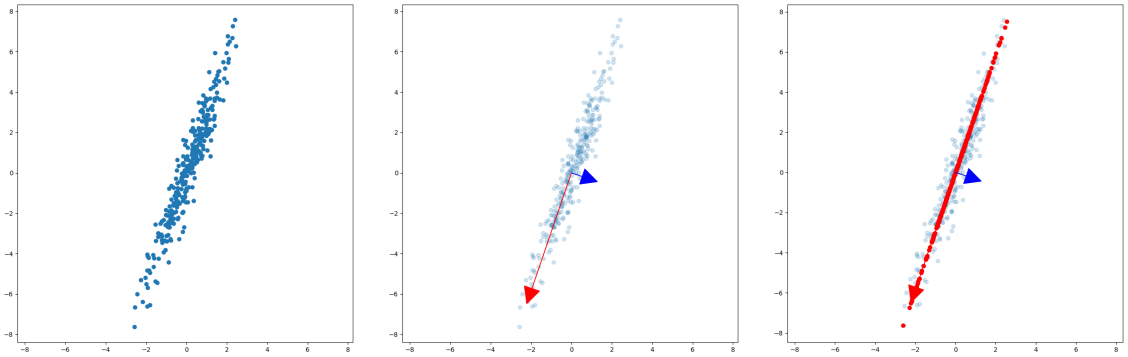


Figure 11.5: PCA

11.3 More unsupervised methods: *k*-means and PCA

k-means clustering

Principal component analysis

Starting from a dataset in $\mathbb{R}^D$ consisting of n data points $\{x_i\}_{i=1}^n$, linear dimensionality reduction amounts to somehow projecting the data onto a space of dimension $M < D$ in such a way that we keep as much information from the original dataset as possible. The most commonly used method corresponds to finding the projection that maximizes the variance of the projected data. The result is referred to as principal component analysis (PCA) and this idea is further motivated in Figure 11.5.

For simplicity we assume that $M = 1$ and let the direction of this first principal component to be denoted by $u_1 \in \mathbb{R}^D$. Since we are only interested in the direction we can without loss of any generality assume that the vector u_1 is normalized according to

$$u_1^\mathsf{T} u_1 = 1. \tag{11.8}$$

Let us continue by noting that we can project any data point x_i onto the first principal component u_1 according to

$$p = \frac{u_1^\mathsf{T} x_i}{\|u_1\|^2} u_1 = u_1^\mathsf{T} x_i u_1. \tag{11.9}$$

Hence, each entry in our dataset can now be projected onto a scalar value $u_1^\mathsf{T} x_i$. The mean of the projected data is $u_1^\mathsf{T} \bar{x}$, where $\bar{x}$ is given by

$$\bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n \tag{11.10}$$

and its variance is given by

$$\begin{aligned} \mathbb{E}\left[u_1^\mathsf{T} x_n - \mathbb{E}\left[u_1^\mathsf{T} x_n\right]\right]^2 &\approx \frac{1}{N} \sum_{n=1}^{N} \left(u_1^\mathsf{T} x_n - u_1^\mathsf{T} \bar{x}\right)^2 = \frac{1}{N} \sum_{n=1}^{N} \left(u_1^\mathsf{T} x_n x_n^\mathsf{T} u_1 - 2 u_1^\mathsf{T} x_n \bar{x}^\mathsf{T} u_1 + u_1^\mathsf{T} \bar{x} \bar{x}^\mathsf{T} u_1\right) \\ &= u_1^\mathsf{T} \underbrace{\left(\frac{1}{N} \sum_{n=1}^{N} (x_n - \bar{x})(x_n - \bar{x})^\mathsf{T}\right)}_{S} u_1 = u_1^\mathsf{T} S u_1. \end{aligned} \tag{11.11}$$

We have now arrived at an expression for the projected variance as a function of our principal component, which can now be found by solving the following optimization problem

$$\max_{u_1} \; u_1^\mathsf{T} S u_1 \tag{11.12}$$

subject to the constraint that $u_1^\mathsf{T} u_1 = 1$. Note that the constraint is important for this optimization problem to make sense. By introducing a Lagrange multiplier λ_1 we can now rewrite the above optimization problem as the following unconstrained problem

$$\max_{u_1, \lambda_1} \underbrace{u_1^\mathsf{T} S u_1 + \lambda_1 (1 - u_1^\mathsf{T} u_1)}_{L(u_1, \lambda_1)}, \tag{11.13}$$

which is a quadratic problem that we can solve simply by computing the derivative $\partial L / \partial u_1 = 1/2 S u_1 - 1/2 \lambda_1 u_1$ and setting it to zero, resulting in the following equation

$$S u_1 = \lambda_1 u_1. \tag{11.14}$$

The above relationship is actually the eigenvalue equation for the matrix S of the dataset. By multiplying (11.14) from the left with u_1^T we obtain the following expression for the variance of the projected data

$$u_1^\mathsf{T} S u_1 = u_1^\mathsf{T} \lambda_1 u_1 = \lambda_1. \tag{11.15}$$

Draft (*June* 17, 2020) *of Supervised Machine Learning. Feedback and exercise problems*: `http://smlbook.org`