Supervised Machine Learning

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

Draft version: February 21, 2020

Contents

1 Introduction **5**

2 Supervised learning: a first approach **7**
2.1 The supervised learning problem 7
2.2 A distance-based method: k-NN 10
2.3 A rule-based method: Decision trees 15
2.A Training a classification tree 19

3 Basic parametric models for regression and classification **25**
3.1 Linear regression 25
3.2 Classification and logistic regression 33
3.A Derivation of the normal equations 41

4 Understanding, evaluating and improving the performance **43**
4.1 Expected new data error E_{new}: performance in production 43
4.2 Estimating E_{new} 45
4.3 The training error–generalization gap decomposition of E_{new} 49
4.4 The bias-variance decomposition of E_{new} 54
4.5 Evaluation for imbalanced and asymmetric classification problems 61

5 Learning parametric models **65**
5.1 Loss functions 65
5.2 Regularization 68
5.3 Parameter optimization 72
5.4 Optimization with large datasets 81
5.5 Further reading 83

6 Neural networks and deep learning **85**
6.1 Neural networks 85
6.2 Convolutional neural networks 92
6.3 Training a neural network 95
6.4 Perspective and further reading 98

7 Ensemble methods: Bagging and boosting **101**
7.1 Bagging 101
7.2 Random forests 106
7.3 Boosting and AdaBoost 108
7.4 Gradient boosting 114

8 Nonlinear input transformations and kernels **117**

9 The Bayesian approach and Gaussian processes **119**

10 User aspects of machine learning **121**
10.1 Defining the machine learning problem 121
10.2 Improving a machine learning model 123

10.3 What if you cannot collect more data? 127
10.4 Practical data issues 130
10.5 Further reading 131

11 Generative models and learning from unlabeled data **133**
11.1 Linear and quadratic discriminant analysis 133
11.2 Unsupervised learning 140

Bibliography **143**

1 Introduction

Chapter to be written.

2 Supervised learning: a first approach

2.1 The supervised learning problem

Supervised machine learning deals with the problem of discovering the relationship between some *input* variable $\mathbf{x}$ and some *output* variable y.[1] Exactly what these variables represent is quite arbitrary and varies from application to application. The starting point, however, is always that: *i)* we have access to the input $\mathbf{x}$ and want to use this information to reason about the output y, and *ii)* we believe that there is some information about y contained in $\mathbf{x}$, i.e., the variables are indeed related.

Learning from labeled data

For most applications of machine learning, the relationship between input and output is difficult to describe explicitly. That is, we assume that it is beyond our ability to write an explicit computer program that takes $\mathbf{x}$ as input and returns y as output. Instead, the approach that we will take is to *learn* the relationship between $\mathbf{x}$ and y from *data*, that is, examples of observed input and output values.

The data which we will use for learning is called *training data*. It consists of several *samples* $(\mathbf{x}_i, y_i)$, in total n of them, and we will compactly write the training data as $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$. 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 word "supervised" refers to the fact that the training data contains *labeled outputs*, in the sense that for each input $\mathbf{x}_i$ in the training data we know the value of the corresponding output y_i. Labeling (or annotating) the training data points is typically done by a domain expert. The entire learning process is thus "supervised" by the domain expert.

Let us have a look at an example.

Example 2.1: Learning a spam filter using supervised machine learning

N.B. This example will be updated!

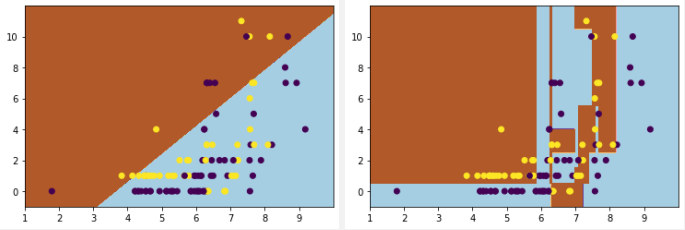
We consider a very simplified spam filter. Hence, the task is to determine whether an email that arrives in the user's inbox is spam or not. The output variable y corresponds to what we want to predict, so it can take two possible values $y \in \{\texttt{spam}, \texttt{legit}\}$. The input $\mathbf{x}$ is the information used to classify the email. In principle it could be the email itself, including the whole content and all meta data, but it could also correspond to some *features* extracted from the email that we believe are informative for determining whether or not it is spam. Which features to use is a design choice. Here we make a simple choice for the purpose of illustration, namely the logarithm of the number of characters in the e-mail (denoted as x_1) and number of hyperlinks in the e-mail (denoted as x_2).

To train the spam filter we also need data. The data should be representative of what the spam filter will see when put in production, so we collect both spam and non-spam emails, in total n of them, from a hypothetical user's inbox. Supervised machine learning requires the data to be labeled, that is each email in the training data set needs to be marked as either spam or not spam. The more data that we collect and label for the training set, the better performance we can expect in general.

In the figure below, data for 210 e-mails collected from various days are illustrated using a scatter plot over the two inputs (features) x_1 and x_2, where each data point is color coded based on its label. Yellow dots correspond to spam emails and blue dots correspond to legit emails. The colored regions in the background

[1] Some common synonyms used for the input variable are feature, attribute, predictor, regressor, covariate, explanatory variable, controlled variable and independent variable, whereas the output are sometimes called response, regressand, label, explained variable, predicted variable or dependent variable.

correspond to two different spam filters, trained on the same data but using different machine learning methods.



In the example above we used both the number of hyperlinks and the length of the email as sources of information regarding whether the email is spam or nor., The input variable in this example is therefore a 2-dimensional vector, $\mathbf{x} = [x_1 \; x_2]^\mathsf{T}$. More generally we assume that the input variable is a p-vector, $\mathbf{x} = [x_1 \; x_2 \; \ldots \; x_p]^\mathsf{T}$, where p is potentially (very) large. For instance, for the skin lesion problem the input consists of all the pixel values of the image, so $p = h \times w$ where h and w denote the height and width of the input image, respectively.[2] 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, however, turns out to be important and is used to distinguish between two subtypes of supervised machine learning problems, as we discuss below.

Numerical and categorical variables

The variables contained in our data can be of two different types: *numerical* or *categorical*. A numerical variable has a natural ordering. We can say that one instance of the 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, e.g. and integer. Categorical variables, on the other hand, are always discrete and they lack the natural ordering associated with numerical values. A few examples to illustrate the concept 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	Uppsala University, Linköping University, KTH	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 $\mathbf{x} = [x_1 \; x_2 \; \ldots \; x_p]^\mathsf{T}$. In particular, the p input variables could be of different types, it is perfectly fine to have a mix of categorical and numerical inputs. In fact, this is often the case in practical application.

The distinction between numerical and categorical is sometimes somewhat arbitrary: one could for instance argue that having no children is something qualitatively different than having children, and use the categorical output "children: yes/no", instead of "0, 1 or 2 children". Consequently, there is sometimes a design choice involved in defining variables as numerical and categorical.

[2] 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.

Classification and regression

It turns out that problems where the output y is numerical have quite different properties, and sometimes require different methods, than problems where y is categorical. For this reason, we distinguish between two types of supervised machine learning problems, referred to as *regression problems* and *classification problems*, respectively.

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

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 we are working with a regression or a classification problem.

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 *labels*. The number of classes M is assumed to be known. 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* mean there is any inherent 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 terms *negative* and *positive* class, to refer to the two labels. The only reason for using a different convention regarding the labels for binary classification is that it results in a more compact mathematical notation when deriving some of the methods studied later in this book. This convention is thus only used for mathematical convenience and has no deeper meaning.

Returning to the spam filter introduced in Example 2.1, we now categorize this as a binary classification problem, because the output variable is $y \in \{\texttt{spam}, \texttt{legit}\}$, that is, a categorical variable with two possible values.

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

So, what is the point of learning a model of the input–output relationship from training data? There are two primary reasons for why this is of interest:

1. To be able to reason about and understand the relationships between input and output variables. For instance, 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 influence your life expectancy?', etc.). Such questions can be addressed by learning a model involving the variables of interest, and carefully reasoning about the chance that the learned relationships between input $\mathbf{x}$ and output y are due only to random effects in the data.

2. To be able to *predict* the output value $y_\star$ for some new, previously unseen input $\mathbf{x}_\star$. This is done by first learning a model of the input–output relationship from the (labeled) training data. We then 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 true (but unknown) output $y_\star$ associated with $\mathbf{x}_\star$. The hat $\widehat{\;}$ indicates that the prediction is an estimate (i.e., an educated guess) of the true output.

These two objectives have sometimes been used to distinguish between classical statistics, which is said to focus more on reasoning and understanding the input–output relationship, and machine learning, which is said to focus more on building predictive models. While this is true to some extent, it is worth emphasizing that there is no clear cut between these two subject areas. Indeed, predictive modeling has

been used extensively in classical statistics as well, and likewise, constructing explainable models is something that has attracted a lot of attention from the machine learning community.

That being said, the primary focus in this book is on predictive modeling, which is the foundation of supervised machine learning. The objective is thus to learn a model from the training data which can be used to predict the output for *test inputs*, that is, previously unseen data points that have not been used to learn the model. The goal is to obtain as accurate predictions as possible (measured in some appropriate way) for a wide range of possible test inputs. We say that the model should *generalize* beyond the training data.

The ability of a predictive model 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 points used to learn the model (we will see an example in the next section). However, if the model is not able to generalize, in the sense that the prediction accuracy drops significantly when the model is applied to new test data points, then the model is of little use in practice. If this is the case we say that the model is *overfitting* to the training data.

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

In this section we develop a first predictive model that can be used for both regression and classification tasks. As discussed above, the starting point is that we have access to labeled training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, which consists of n inputs $\mathbf{x}_i$ with corresponding outputs (i.e., labels) y_i. Given a new test data point $\mathbf{x}_\star$, for which the output value $y_\star$ is unknown, we want to use the information contained in the training data to predict a plausible value for $y_\star$.

The *k*-nearest neighbors method

A natural idea is to assume that if $\mathbf{x}_\star$ is close to some training input $\mathbf{x}_i$, then $y_\star$ should be close to y_i too. We can then construct a prediction method in the following way: First, compute the distance[3] 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 predicted output for $\mathbf{x}_\star$, which we denote by $\widehat{y}(\mathbf{x}_\star)$, be given by $\widehat{y}(\mathbf{x}_\star) = y_j$.

This simple prediction method is referred to as the 1-nearest neighbor method. Unfortunately, for most machine learning applications of interest it is too simplistic. One problem is that data in many cases gives an incomplete picture of reality. Just because we have access to the input value $\mathbf{x}$ we can not say *for certain* what the output value y will be. Mathematically this can be described by viewing y a random variable. We say that data is *noisy*.

For noisy data the 1-nearest neighbor method can be too "erratic". It only considers the closest training data point, even if there are other data points in the vicinity of $\mathbf{x}_\star$ that suggest a different prediction. To make the method more robust to noise, we can extend the idea to make use of the k nearest neighbors, for some user-chosen integer k. Formally, define the set $R_\star = \{i : \mathbf{x}_i \text{ is one of the } k \text{ training data points closest to } \mathbf{x}_\star\}$. We then aggregate the information from the k labels y_j for $j \in R_\star$ to make the prediction, typically as an average for regression problems or a majority vote[4] for classification problems. We summarize the k-nearest neighbors (k-NN) method in Algorithm 1 and explain it further in the example below.

[3] We consider the Euclidean distance for simplicity, but other distance functions can also be used. Categorical input variable can be handled in the same way as discussed in Section 3.1 (p. 31).

[4] 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.

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

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

Example 2.2: Predicting colors with k-NN

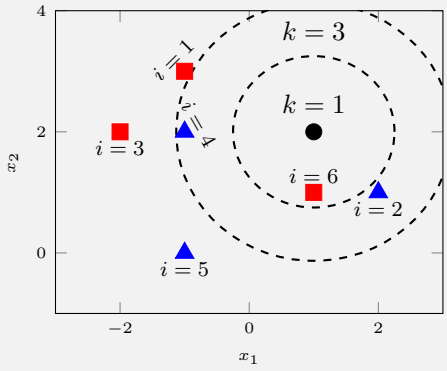
We consider a binary classification problem. We are given a training data set with $n = 6$ observations of $p = 2$ input variables x_1, x_2 and one categorical output y, the color `Red` or `Blue`,

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`



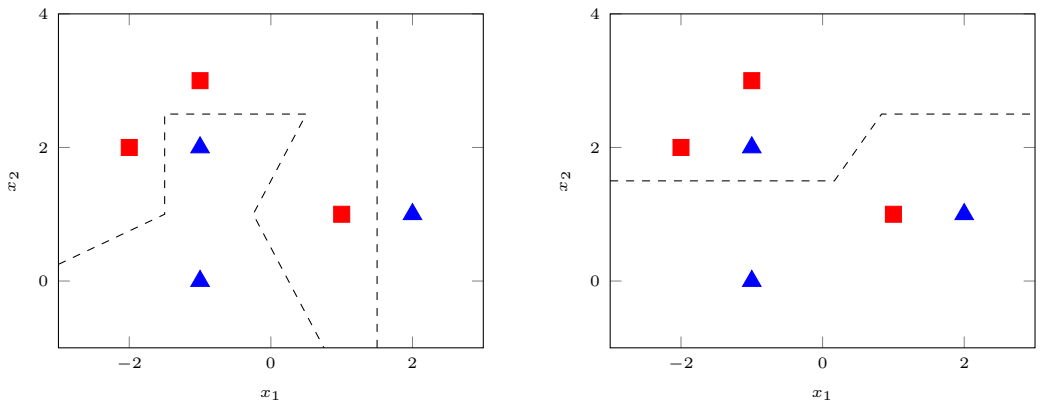
Since the closest training data point to $\mathbf{x}_\star$ is the data point $i = 6$ (`Red`), it means that for k-NN with $k = 1$, we get the prediction $\widehat{y}(\mathbf{x}_\star) = \texttt{Red}$. For $k = 3$, the 3 nearest neighbors are $i = 6$ (`Red`), $i = 2$ (`Blue`), and $i = 4$ (`Blue`). Taking 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.

Methods that explicitly use the training data in the "prediction phase" in this way are referred to as *non-parametric*. This is in contrast with *parametric* models, where the prediction is given by some function, governed by adaptable parameters. The training data is then used to *learn* these parameters in an initial "training phase", but once the model has been learned the training data can in principle be discarded since it is not used explicitly in the "prediction phase". We will discuss parametric models in more detail in Chapter 3.

Decision boundaries for a *k*-NN classifier

In Example 2.2 we only computed a prediction for one single test data point $\mathbf{x}_\star$. If we would shift that test point by one step to the left at $\mathbf{x}_\star^{\text{alt}} = [0\ 2]^\mathsf{T}$ the three closest training data points would still include $i = 6$ and $i = 4$ 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`



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

Figure 2.1: Decision boundaries for the problem in Example 2.2 for the two choices of the parameter k.

or `Blue`[5]. For a classification problem 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 sketch the full decision boundaries for Example 2.2, which are displayed in Figure 2.1.

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. In Figure 2.2 another scenario is illustrated with $p = 2$ input variables, $M = 3$ classes and significantly more training data samples. In the two subfigures the decision boundaries for a k-NN classifier with $k = 1$ and $k = 11$ are illustrated.

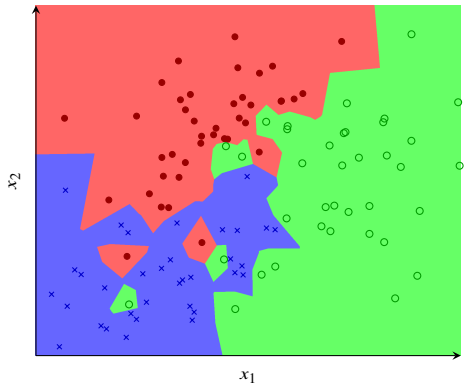
By construction, with $k = 1$ all *training data points* will be correctly classified and the decision boundaries are adapted to the exact locations and labels of the training data. However, this includes also the 'noise' in the data, which gives rise to an "erratic" behavior of the prediction. For instance, there are small red and green regions within the large blue area. However, for the model to generalize well to new test data, it would typically make more sense to predict according to the blue class in the entire lower left region since the vast majority of training data points in this area are blue.

For the k-NN classifier, this can be accomplished by increasing the region of the neighborhood used to compute the prediction, that is increase the parameter k. With the effect of the majority vote that takes place for $k = 11$, some training data points end up in the wrong region. However, the decision boundaries are less adapted to the specific realization of the (noisy) training data and appear to be less erratic. Even though k-NN with $k = 1$ fits all training data points perfectly, the one with $k = 11$ might be preferred. Specifically, it is less prone to *overfit* to the training data and there are good reasons to believe that this model would perform better on test data. A systematic way of choosing k is to use cross-validation, and we will discuss these aspects more in Chapter 4.

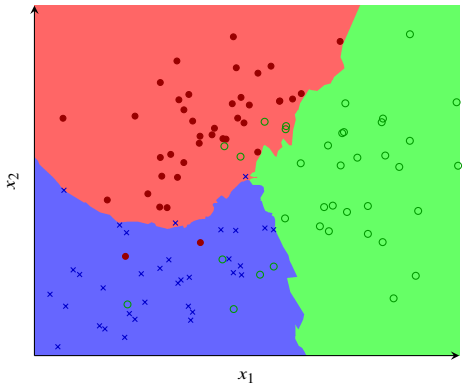
k-NN for regression

We have seen a couple of examples of k-NN applied in classification settings, but as pointed out above the method can be used also for regression problems. The difference is that the prediction $\widehat{y}(\mathbf{x}_\star)$ is computed

[5] In practice this is typically 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, or more systematically by reporting the fractions of `Red` and `Blue` data points in the neighborhood of the test data point to the end used.



(a) Class predictions for k-NN with $k = 1$ for a 3-class problem. Complex, but likely overfitted, decision boundaries.



(b) Class predictions for k-NN with $k = 11$. More rigid and less flexible decision boundaries, less prone to overfitting.

Figure 2.2: Decision boundaries for k-NN.

as an average of the training outputs y_j within the neighborhood $R_\star$,

$$\widehat{y}(\mathbf{x}_\star) = \frac{1}{k} \sum_{j \in R_\star} y_j,$$

instead of as a majority vote. If the output is not a real number but, say, constrained to be an integer, then the prediction can be rounded accordingly. Other output constraints can be handled in similar ways.

> ***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 2.2 where the colored regions correspond to areas of the input space where the prediction is constant according to the color of that region). However, this is true also for a regression problem where the output is numerical. Why does the k-NN method result in a piece-wise constant prediction in such a case?*

Normalization

Finally, one practical aspects crucial for the k-NN worth mentioning is the importance of normalization of the input data. Since k-NN is based on the Euclidean distances between points, it is important that these distances are a valid measure of the closeness between two data points. Imagine a training data set with two input variables $\mathbf{x}_i = [x_{i1},\ x_{i2}]^\mathsf{T}$ where all values of x_{i1} are in the range $[0,\ 100]$ and the values for x_{i2} are in the much smaller range $[0,\ 1]$. It could for example be the case that x_{i1} and x_{i2} represent different physical quantities (where the values can be quite different depending on which unit that is used to measure these quantities). In this case the Euclidean distance between a test point $\mathbf{x}_\star$ and a training data point $\|\mathbf{x}_i - \mathbf{x}_\star\| = \sqrt{(x_{i1} - x_{\star 1})^2 + (x_{i2} - x_{\star 2})^2}$ would almost only depend on the first term $(x_{i1} - x_{\star 1})^2$ and the value of the second input variable x_{i2} would have a small impact.

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

$$x_{ij}^{\text{new}} = \frac{x_{ij} - \min(\mathbf{x}_{:j})}{\max(\mathbf{x}_{:j}) - \min(\mathbf{x}_{:j})}, \qquad \forall j = 1, \ldots, p, \quad i = 1, \ldots, n. \tag{2.1}$$

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

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

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

2.3 A rule-based method: Decision trees

As discussed above, 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 2.2, 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.

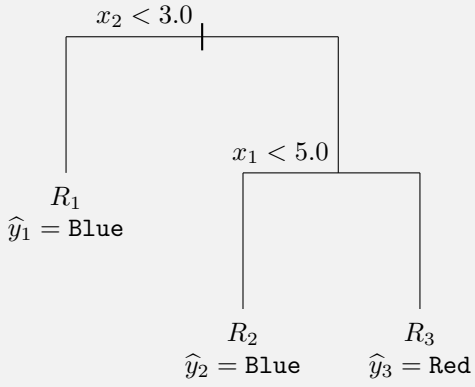
Predicting using a decision tree

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.

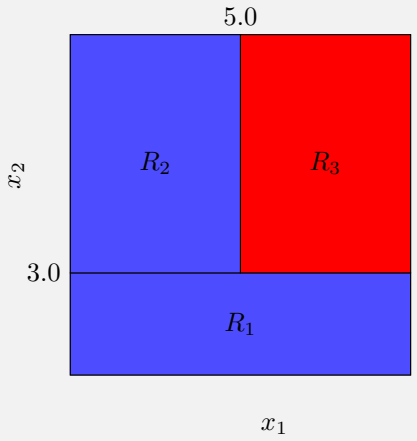
Example 2.3: Predicting colors with a classification tree

We will look at a classification tree applied to a problem with two input variables x_1 and x_2 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 what a final prediction model $\widehat{y}(\mathbf{x}_\star)$ might look like and how it can be illustrated.

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 classify a 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 for the classification tree. Each region corresponds to a leaf node in the tree to the left, and each border between regions correspond to a split in the tree. Each region is marked with the color corresponding to the prediction associated with that region.

The decision tree associate a test input with a certain prediction based on a sequence rules applied to the individual coordinates of the input vector. This corresponds to a partitioning of 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 classifying 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 $x_{\star 1} = 2.5 < 5.0$. Consequently, the for this test point we would predict $\widehat{y}(\mathbf{x}_\star) = \texttt{Blue}$.

To set the terminology, the endpoint of each branch R_1, R_2 and R_3 in Example 2.3 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 2.3 illustrated how a decision tree is used for making a prediction. We will now turn to the question of how the tree can be learnt from training data.

Training a regression tree

The method used to train a decision tree is here presented in the context of a regression problem. Training of a classification tree is conceptually similar, but the details are more involved and are therefore left for the appendix, Section 2.A.

As discussed above, the prediction $\widehat{y}(\mathbf{x}_\star)$ corresponding 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 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:

$$\boxed{\widehat{y}_\ell = \frac{1}{n_\ell} \sum_{i:\mathbf{x}_i \in R_\ell} y_i,}$$

where $n_\ell = |\{i : \mathbf{x}_i \in R_\ell\}| = \sum_{i=1}^n \mathbb{I}\{\mathbf{x}_i \in R_\ell\}$ is the total number of training data points falling in region R_ℓ.

It remains to find the regions R_ℓ, which requires a bit more work. The basic idea is to select the regions so that the predictions given by the model for the training input match the training outputs. Unfortunately, even if we restrict our attention to seemingly simple regions such as the "boxes" used in a decision tree, finding the collection of splitting rules that optimally partitions the input space to fit the training data as well as possible (using some suitable notion of optimality) 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 possibilities is not possible for any reasonably sized tree.

To deal with this limitation, we instead use a *greedy* algorithm for computing the partition known as *recursive binary splitting*. 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,

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

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,

$$\widehat{y}_1(j,s) = \frac{1}{n_1} \sum_{i:\mathbf{x}_i \in R_1(j,s)} y_i \qquad \text{and} \qquad \widehat{y}_2(j,s) = \frac{1}{n_2} \sum_{i:\mathbf{x}_i \in R_2(j,s)} y_i,$$

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

$$\sum_{i:\mathbf{x}_i \in R_1(j,s)} (y_i - \widehat{y}_1(j,s))^2 + \sum_{i:\mathbf{x}_i \in R_2(j,s)} (y_i - \widehat{y}_2(j,s))^2 \,. \tag{2.4}$$

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 (2.4). 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 training procedure.

Classification trees

Trees can also be used for classification. We then use a majority vote instead of an average to compute the prediction associated with each region,

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

Furthermore, 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. We discuss different options in the appendix, Section 2.A. In all other aspects the procedure to train a classification tree is the same as training a regression tree as explained above.

2.A Training a classification tree

This appendix cover the details related to training a classification tree. For this purpose we will use the *maximum likelihood* approach. Readers not familiar with maximum likelihood are therefore encouraged to read Chapter 3 before revisiting this appendix.

To define a maximum likelihood objective we need a probabilistic interpretation of the classification tree. Instead of just predicting a class assignment we construct the model to predict a vector of class probabilities for each input $\mathbf{x}$. For M classes, the output of the model is the vector $[g_1(\mathbf{x}), \ldots, g_M(\mathbf{x})]^\mathsf{T}$, where $g_m(\mathbf{x})$ is interpreted as an estimate of the probability that the output belongs to class m, conditionally on the input being $\mathbf{x}$:

$$\boxed{g_m(\mathbf{x}) \approx p(y = m \,|\, \mathbf{x}).}$$

For this interpretation to make sense we require $g_m(\mathbf{x}) \geq 0$ for each m and $\sum_{m=1}^{M} g_m(\mathbf{x}) = 1$.

We can now write the classification tree as a model for the conditional class probabilities. Specifically, the value $g_m(\mathbf{x})$ is modeled as a constant $c_{\ell m}$ in each region R_ℓ and for each class $m = 1, 2, \ldots, M$:

$$g_m(\mathbf{x}) = \sum_{\ell=1}^{L} c_{\ell m} \mathbb{I}\{\mathbf{x} \in R_\ell\}, \tag{2.5}$$

where, as above, L is the total number of regions (leaf nodes) in the tree. Since the probabilities should sum up to 1 in each region we also have the constraint $\sum_{m=1}^{M} c_{\ell m} = 1$.

In the maximum likelihood formulation, the overall goal in constructing the classification tree based on training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ is to find a tree that makes the observed data as likely as possible. This is done by maximizing the likelihood of the training data w.r.t. the model parameters, where the likelihood is given by

$$p(y_1, \ldots, y_n | x_1, \ldots, x_n) = \prod_{i=1}^{n} p(y_i \,|\, x_i) = \prod_{i=1}^{n} g_{y_i}(\mathbf{x}_i).$$

The first equality follows from the assumption that the training data points are statistically independent, and the second equality follows from the model we use for the conditional class probabilities. Maximizing the likelihood is equivalent to minimizing the negative logarithm of the likelihood, which is often preferred in practice for numerical reasons. If we denote all parameters of the model by $\boldsymbol{\theta} = \{L, \{\{c_{\ell m}\}_{m=1}^M, R_\ell\}_{\ell=1}^L\}$ we can write the negative log-likelihood as as a function of the model parameters as,

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} -\ln g_{y_i}(\mathbf{x}_i; \boldsymbol{\theta}), \tag{2.6}$$

By inserting the tree model (2.5), we get

$$\begin{aligned}
J(\boldsymbol{\theta}) &= -\sum_{i=1}^{n} \ln \left(\sum_{\ell=1}^{L} c_{\ell y_i} \mathbb{I}\{\mathbf{x}_i \in R_\ell\} \right) = -\sum_{i=1}^{n} \sum_{\ell=1}^{L} \ln c_{\ell y_i} \mathbb{I}\{\mathbf{x}_i \in R_\ell\} \\
&= -\sum_{i=1}^{n} \sum_{m=1}^{M} \sum_{\ell=1}^{L} \ln c_{\ell m} \mathbb{I}\{y_i = m\} \mathbb{I}\{\mathbf{x}_i \in R_\ell\} \\
&= -\sum_{\ell=1}^{L} n_\ell \sum_{m=1}^{M} \ln c_{\ell m} \underbrace{\frac{1}{n_\ell} \sum_{i:\mathbf{x}_i \in R_\ell} \mathbb{I}\{y_i = m\}}_{\stackrel{\text{def}}{=} \widehat{\pi}_{\ell m}} \\
&= -\sum_{\ell=1}^{L} n_\ell \sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln c_{\ell m}
\end{aligned} \tag{2.7}$$

Here $\widehat{\pi}_{\ell m}$ is the proportion of training data points in region R_ℓ that are from class m and n_ℓ is the total number of training data points in region ℓ.

Let us start by considering how to choose $c_{\ell m}$ assuming the regions R_ℓ are fix. We can show[6] that

$$-\sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln c_{\ell m} = \underbrace{\sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln \frac{\pi_{\ell m}}{c_{\ell m}}}_{\geq 0} - \sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln \widehat{\pi}_{\ell m} \geq -\sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln \widehat{\pi}_{\ell m}. \tag{2.8}$$

which is fulfilled with equality if $c_{\ell m} = \widehat{\pi}_{\ell m}$. Hence, to minimize (2.7) we choose $c_{\ell m} = \widehat{\pi}_{\ell m}$. In words, the prediction $g_m(\mathbf{x})$ in leaf node ℓ is the ratio of training data points of class m in that node, $\widehat{\pi}_{\ell m}$.

It remains to find the regions R_ℓ. From above, we have that we want to select the regions such that

$$\sum_{m=1}^{L} n_\ell Q_\ell, \qquad \text{where } Q_\ell = -\sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln \widehat{\pi}_{\ell m}, \tag{2.9}$$

is minimized.[7] This is indeed a direct consequence of our maximum likelihood approach, and we refer to Q_ℓ in (2.9) as the *entropy splitting criteria*.

As in the regression setting, finding the best tree that minimizes (2.9) is a combinatorial problem which is computationally infeasible in all non-trivial cases. However, we can use the same recursive binary splitting algorithm as discussed above to compute an approximate solution to the optimization problem. As above we explain one step of the algorithm by considering the first split a the root of the tree, since all consecutive splits are completely analogous. Thus, we want to split the input space into two half-spaces

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

The corresponding class proportions $\widehat{\pi}_{\ell m}$ are given by

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

From the maximum likelihood objective (2.9) we seek the splitting variable j and cutpoint s that solve

$$\min_{j,s} \left[n_1 \left(-\sum_{m=1}^{M} \widehat{\pi}_{1m}(j, s) \ln \widehat{\pi}_{1m}(j, s) \right) + n_2 \left(-\sum_{m=1}^{M} \widehat{\pi}_{2m}(j, s) \ln \widehat{\pi}_{2m}(j, s) \right) \right]. \tag{2.10}$$

For each input variable we can scan through the finite number of possible splits and pick the pair $(j,\ s)$ which minimizes (2.10). After that, we repeat the process to create new splits by finding the best values $(j,\ s)$ for each of the new branches. We continue the process until some stopping criteria is reached, for example until no region contains more than five training data points.

The tree in Example 2.3 has been constructed based on the methodology outlined above, which we will illustrate in the example below.

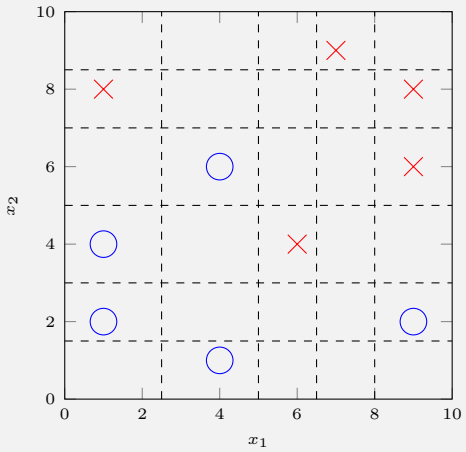
[6] We use the so called log sum inequality and the two constraints $\sum_{m=1}^{M} c_{\ell m} = 1$ and $\sum_{m=1}^{M} \widehat{\pi}_{\ell m} = 1$ for all $\ell = 1, \ldots, L$.

[7] If any $\widehat{\pi}_{\ell m}$ is equal to 0, the term $0 \ln 0$ is taken to be 0, which is consistent with the limit $\lim_{r \to 0^+} r \ln r = 0$.

Example 2.4: Learning a classification tree (continuation of Example 2.3)

We consider the same setup as in Example 2.3 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

$$Q_1 = -\widehat{\pi}_{1\text{B}} \ln(\widehat{\pi}_{1\text{B}}) - \widehat{\pi}_{1\text{R}} \ln(\widehat{\pi}_{1\text{R}}) = -\frac{2}{3}\ln(\frac{2}{3}) - \frac{1}{3}\ln(\frac{1}{3}) = 0.64. \tag{2.11}$$

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

$$Q_2 = -\widehat{\pi}_{2\text{B}} \ln(\widehat{\pi}_{2\text{B}}) - \widehat{\pi}_{2\text{R}} \ln(\widehat{\pi}_{2\text{R}}) = -\frac{3}{7}\ln(\frac{3}{7}) - \frac{4}{7}\ln(\frac{4}{7}) = 0.68 \tag{2.12}$$

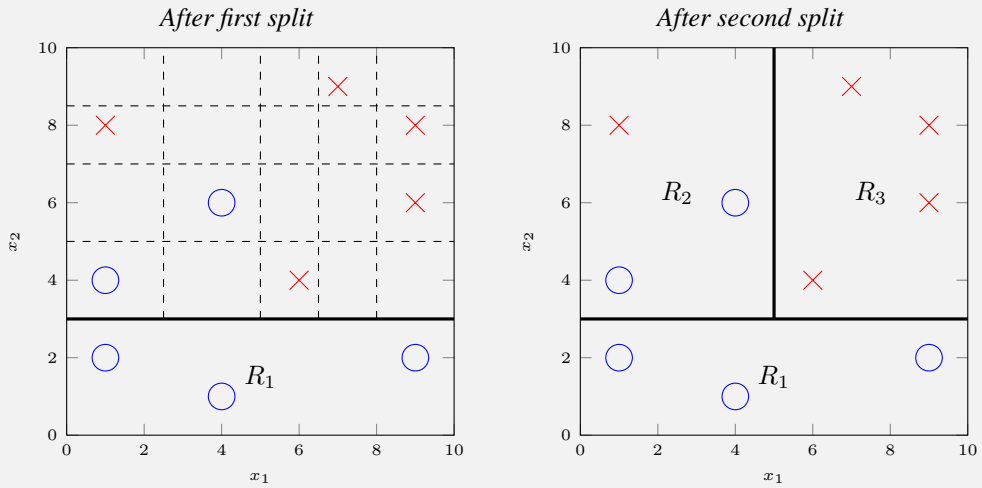
and the total weighted entropy for this split becomes

$$n_1 Q_1 + n_2 Q_2 = 3 \cdot 0.64 + 7 \cdot 0.68 = 6.69. \tag{2.13}$$

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

Split (R_1)	n_1	$\widehat{\pi}_{1\text{B}}$	$\widehat{\pi}_{1\text{R}}$	Q_1	n_2	$\widehat{\pi}_{2\text{B}}$	$\widehat{\pi}_{2\text{R}}$	Q_2	$n_1 Q_1 + n_2 Q_2$
$x_1 < 2.5$	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	$\widehat{\pi}_{2\text{B}}$	$\widehat{\pi}_{2\text{R}}$	Q_2	n_3	$\widehat{\pi}_{3\text{B}}$	$\widehat{\pi}_{3\text{R}}$	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 2.3. None of the three regions has more than five data points. Therefore, we terminate the training.

If we want to use the tree for prediction, we get $p(\texttt{Red}|\mathbf{x}_\star) = \widehat{\pi}_{1\text{R}} = 0$ if $\mathbf{x}_\star$ falls into region R_1, $p(\texttt{Red}|\mathbf{x}_\star) = \widehat{\pi}_{2\text{R}} = 1/3$ if falls into region R_2 or $p(\texttt{Red}|\mathbf{x}_\star) = \widehat{\pi}_{3\text{R}} = 1$ if falls into region R_3, in the same manner as displayed in Example 2.3.

Other splitting criteria

There are other splitting criteria that can be considered instead of the entropy in (2.9). One simple alternative is the *misclassification rate*

$$Q_\ell = 1 - \max_m \widehat{\pi}_{\ell m}, \tag{2.14}$$

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

For example, consider the first split in Example 2.4. If we would use the misclassification rate as splitting criteria, both the split $x_2 < 5.0$ as well as $x_2 < 3.0$ would provide a total misclassification rate of 0.2. However, the split at $x_2 < 3.0$, which the entropy criteria favored, provides a pure node R_1. If we would go with the split $x_2 < 5.0$ the misclassification after the second split would still be 0.2. If we would continue to grow the tree until no data points are misclassified we would need three splits if we used the entropy criteria whereas we would need five splits if we would use the misclassification criteria and started with the split at $x_2 < 5.0$.

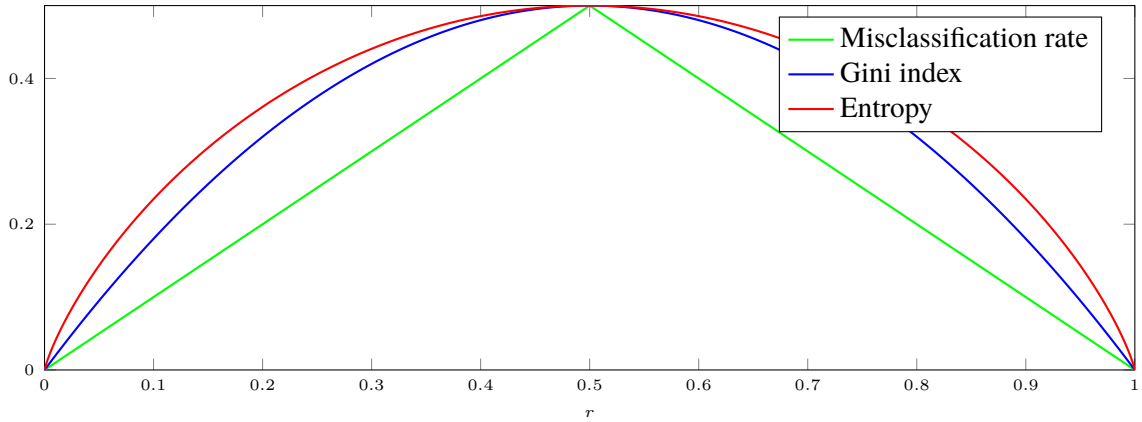


Figure 2.3: Three splitting criteria for classification trees as a function of the proportion in class 2. The entropy criteria has been scaled such that it passes through (0.5,0.5).

Another common splitting criteria is the *Gini index*

$$Q_\ell = \sum_{m=1}^{M} \widehat{\pi}_{\ell m}(1 - \widehat{\pi}_{\ell m}). \tag{2.15}$$

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

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

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

These functions are shown in Figure 2.3. We can se that the entropy and Gini index are quite similar.

3 Basic parametric models for regression and classification

In this chapter we introduce an systematic approach for addressing the supervised learning problem referred to as *parametric* modeling. A parametric model is a function that maps a test input to a predicted output (or a probability distribution over possible outputs). This function is parameterized by some adaptable parameters. Learning the model amounts to finding suitable values for these parameters, so that the resulting input–output relationship matches the observed training data. We introduce the concept of parametric modeling using a linear regression model in Section 3.1 and a linear classification model in Section 3.2.

3.1 Linear regression

Regression is a fundamental task for supervised learning, and one of the two main problem types that we cover in this book (the other one is classification). In this section we discuss the *linear regression* model, which is one (of many) solution to the regression problem. Despite the simplicity of linear regression, it is a surprisingly useful model which also constitutes an important building block for more advanced methods, such as deep learning, Chapter 6.

The regression problem

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

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

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.

To illustrate regression, we will use the dataset introduced in Example 3.1 with car stopping distances. In one sentence, the problem is to learn a regression model that can predict the distance needed for a car to come to a full stop given its current speed.

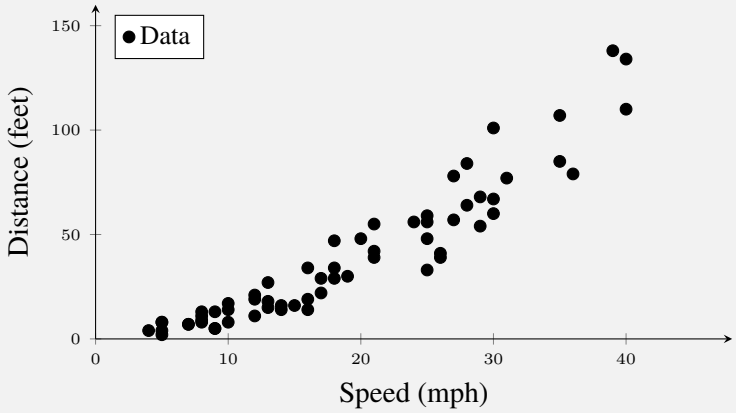
[1] We will start with numerical inputs, and discuss categorical inputs later.

Example 3.1: Car stopping distances

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

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



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

[a] The data is somewhat dated, so the conclusions are perhaps not applicable to modern cars, but it still serves the purpose of being an illustrative example.

The linear regression model

The linear regression model assumes that the output variable y (a scalar) can be described as an affine[2] combination of the input variables $x_1, x_2, \ldots, x_p$ (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 $\theta_0, \theta_1, \ldots \theta_p$ 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 between the data and 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 $\boldsymbol{\theta} = [\theta_0 \;\; \theta_1 \;\; \ldots \;\; \theta_p]^\mathsf{T}$ 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) compactly

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

This notation means that the symbol $\mathbf{x}$ 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 positions. This is only a matter of book-keeping for handling the intercept term θ_0, 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

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

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

Assume that we have learned some parameter values $\widehat{\boldsymbol{\theta}}$ for the linear regression model (how this is done is described below). We can then use the model for making a *prediction* of what the output $y_\star$ will be for a new, previously unseen, test input $\mathbf{x}_\star = \begin{bmatrix} x_{\star 1} & x_{\star 2} & \dots & x_{\star p} \end{bmatrix}^\mathsf{T}$. We assume that the noise term ε is random with zero mean and independent of all observed variables, so the best prediction we can do is

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

This is in fact applicable for all regression models of the type (3.1), where the model used for prediction is

$$\widehat{y}(\mathbf{x}_\star) = f(x_{\star p}). \tag{3.5}$$

That is, since we do not have any information regarding the value of the noise term for a new test input $\mathbf{x}_\star$, our best guess is to predict it to be zero. The noise term ε is often referred to as an *irreducible error* or an *aleatoric*[3] uncertainty in the prediction. We illustrate the predictions made by a linear regression model in Figure 3.1.

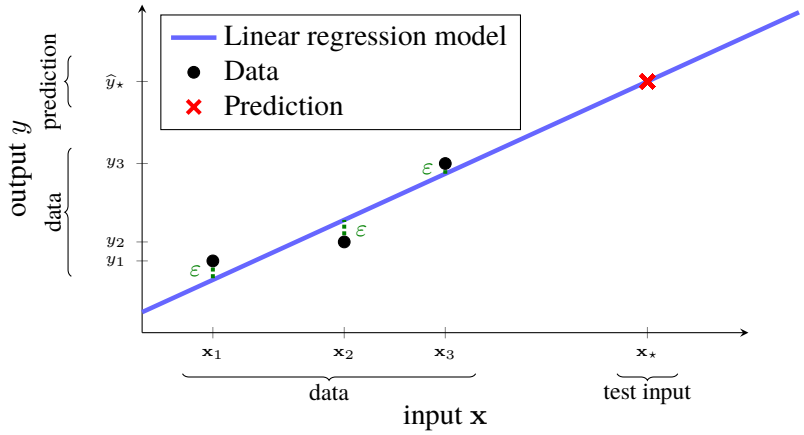


Figure 3.1: Linear regression with $p = 1$: The black dots represent $n = 3$ data samples, from which a linear regression model (blue line) is learned. The model does not fit the data perfectly, but there is a remaining error/noise ε (green). The model can be used to *predict* (red cross) the output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

Learning linear regression from training data

To use the linear regression model, we have to learn the unknown parameters $\boldsymbol{\theta}$ from a training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$. We write the training dataset, which consists of n samples y_i of the output variable y and corresponding n samples $\mathbf{x}_i$ of the input variable $\mathbf{x}$, in the matrix form

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1^\mathsf{T}- \\ -\mathbf{x}_2^\mathsf{T}- \\ \vdots \\ -\mathbf{x}_n^\mathsf{T}- \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \text{where each } \mathbf{x}_i = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix}. \tag{3.6}$$

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

Note that $\mathbf{X}$ is a $n \times (p+1)$ matrix, and $\mathbf{y}$ a n-dimensional vector. Together with the parameter vector,

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix}, \tag{3.7}$$

we can express the relationship between all our n input samples $\mathbf{X}$, parameters $\boldsymbol{\theta}$ and output samples $\mathbf{y}$ according to the linear regression model as a matrix multiplication

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon}, \tag{3.8}$$

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

Learning the unknown parameters $\boldsymbol{\theta}$ amounts to finding values such that *the model fits the data well*. There are multiple ways to define what 'well' actually means, but it somehow amounts to find $\boldsymbol{\theta}$ such that $\boldsymbol{\epsilon}$ is small in some sense. We will approach this by formulating a loss function, which will imply a meaning of 'fitting the data well'. We will also interpret this 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.

Example 3.2: Car stopping distances

We continue Example 3.1 and form the matrices $\mathbf{X}$ and $\mathbf{y}$. Since we only have one input and one output, both x_i and y_i are scalar. We get,

$$\mathbf{X} = \begin{bmatrix} 1 & 4 \\ 1 & 5 \\ 1 & 5 \\ 1 & 5 \\ 1 & 5 \\ 1 & 7 \\ 1 & 7 \\ 1 & 8 \\ \vdots & \vdots \\ 1 & 39 \\ 1 & 39 \\ 1 & 40 \end{bmatrix}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4 \\ 2 \\ 4 \\ 8 \\ 8 \\ 7 \\ 7 \\ 8 \\ \vdots \\ 138 \\ 110 \\ 134 \end{bmatrix}. \tag{3.9}$$

Loss functions and cost functions

To train the linear regression model on data, or equivalently learn $\boldsymbol{\theta}$ from data, we need to have a training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$. Learning $\boldsymbol{\theta}$ from $\mathcal{T}$ then amounts to find a value $\widehat{\boldsymbol{\theta}}$ such that the linear regression model fits the training data $\mathcal{T}$ well. We formulate this mathematically using a loss function $L(\widehat{y}(\mathbf{x};\boldsymbol{\theta}), y)$ and a cost function $J(\boldsymbol{\theta})$ as

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \overbrace{L(\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta}), y_i)}^{\text{loss function}}}_{\text{cost function } J(\boldsymbol{\theta})}. \tag{3.10}$$

The loss function is a function of the prediction (3.4) $\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta})$ 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 them in the notation here.) Furthermore, the cost function is the loss function averaged over the training dataset, and learning $\boldsymbol{\theta}$ amounts to minimize the cost function. 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 (3.10) is general for all cost functions we will consider in this book.

Loss functions for regression

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

$$L(\widehat{y}(\mathbf{x};\boldsymbol{\theta}), y) = \big(\widehat{y}(\mathbf{x};\boldsymbol{\theta}) - y\big)^2. \tag{3.11}$$

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 (3.8) can be written as

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

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 square* cost. We will discuss other loss functions in Chapter 5.

Least squares and the normal equations

When using the squared error loss for training linear regression model on training data $\mathcal{T}$, we need to solve the problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i)^2 = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2, \tag{3.13}$$

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 (Section 3.A) to fulfill

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

Equation (3.14) is often referred to as the *normal equations*, and gives the solution to the least squares problem (3.13). If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, which often is the case, $\widehat{\boldsymbol{\theta}}$ has the closed form

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{3.15}$$

The fact that this closed-form solution exists is important, and is perhaps the reason why the squared error loss is so widely used. Other loss functions, for instance the absolute error (5.3), lead to other optimization problems where no closed-form solution exists.

We now have everything in place for using linear regression, and we summarize by Algorithm 2 and Example 3.3.

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,* (3.15) *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?*

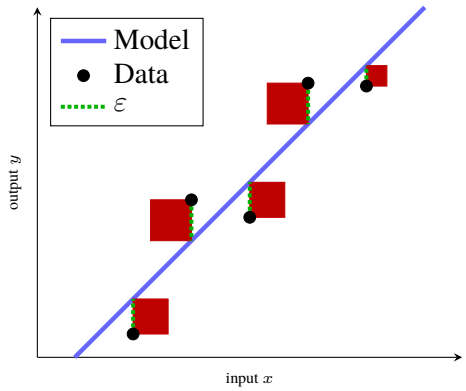


Figure 3.2: A graphical explanation of the squared error loss function: the goal is to choose the model (blue line) such that the sum of the square (red) of each error ε (green) is minimized. That is, the blue line is to be chosen so that the amount of red color is minimized. This motivates the name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

Algorithm 2: Linear regression with squared error loss

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

Learn

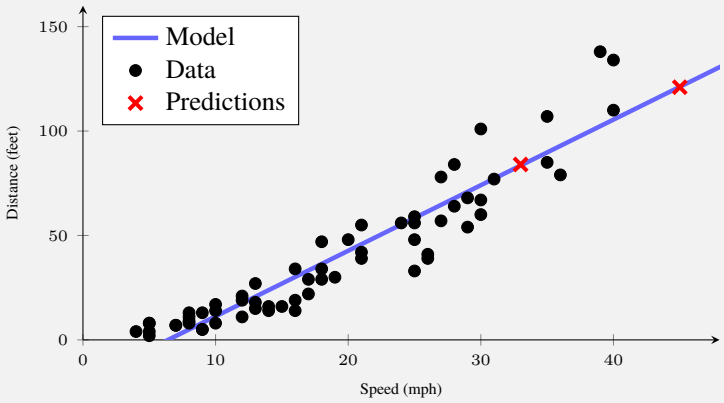
1 Compute $\widehat{\boldsymbol{\theta}}$ by finding a solution to (3.14) (which often is given by (3.15)).

Predict

2 Return prediction $\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T}\mathbf{x}_\star$.

Example 3.3: Car stopping distances

By inserting the matrices (3.9) from Example 3.2 into the normal equations (3.8), we obtain $\widehat{\theta}_0 = -20.1$ and $\widehat{\theta}_1 = 3.1$. If we plot the resulting model, it looks like this:



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

The maximum likelihood perspective

To get another perspective of the absolute and squared error loss functions, we will now interpret them statistically as *maximum likelihood* methods. 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 arbitrary) selecting a loss function, we

start with the problem

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}). \tag{3.16}$$

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 that each ε is independent and has a Gaussian distribution with zero mean and variance σ_ε^2,

$$\varepsilon \sim \mathcal{N}\left(0, \sigma_\varepsilon^2\right). \tag{3.17}$$

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

$$p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}) \tag{3.18}$$

and, together with the linear regression model (3.3), that

$$p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}) = \mathcal{N}\left(y_i; \boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i, \sigma_\varepsilon^2\right) = \frac{1}{\sqrt{2\pi\sigma_\varepsilon^2}} \exp\left(-\frac{1}{2\sigma_\varepsilon^2}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2\right). \tag{3.19}$$

For numerical reasons, it is usually better to consider the logarithm of $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$,

$$\ln p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}). \tag{3.20}$$

Putting (3.19) and (3.20) together, we get

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

Recall that we want to maximize (3.21) w.r.t. $\boldsymbol{\theta}$. Removing terms and factors independent of $\boldsymbol{\theta}$ does not change the maximizing argument, and we see that we can rewrite (3.16) as

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} -\sum_{i=1}^{n}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2 = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n}\frac{1}{n}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2. \tag{3.22}$$

This is indeed linear regression with the least squares cost, the cost function implied by the square error loss function (3.11), and the key to arrive here was the assumption of ε having a Gaussian distribution. Other assumptions on ε would lead 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

$$x = \begin{cases} 0 & \text{if A}, \\ 1 & \text{if B}, \end{cases} \tag{3.23}$$

and use this variable in linear or logistic regression. For linear regression, this effectively gives us a model which looks like

$$y = \theta_0 + \theta_1 x + \varepsilon = \begin{cases} \theta_0 + \varepsilon & \text{if A,} \\ \theta_0 + \theta_1 + \varepsilon & \text{if B,} \end{cases} \tag{3.24}$$

and similarly for logistic regression. The choice is somewhat arbitrary, since A and B can of course be switched. This approach can be generalized to categorical input variables which take more than two values, let us say A, B, C and D. With four different values, we create $3 = 4 - 1$ dummy variables as

$$x_1 = \begin{cases} 1 & \text{if B} \\ 0 & \text{if not B} \end{cases}, \quad x_2 = \begin{cases} 1 & \text{if C} \\ 0 & \text{if not C} \end{cases}, \quad x_3 = \begin{cases} 1 & \text{if D} \\ 0 & \text{if not D} \end{cases} \tag{3.25}$$

which for linear regression altogether gives the model

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \varepsilon = \begin{cases} \theta_0 + \varepsilon & \text{if A,} \\ \theta_0 + \theta_1 + \varepsilon & \text{if B,} \\ \theta_0 + \theta_2 + \varepsilon & \text{if C,} \\ \theta_0 + \theta_3 + \varepsilon & \text{if D.} \end{cases} \tag{3.26}$$

This use of dummy variables is not specific to linear regression, but can be used also with other methods for regression as well as classification.

3.2 Classification and logistic regression

After introducing linear regression as a solution to the regression problem, we now turn our attention to classification. With a modification of the linear regression model, we will be able to apply it to the classification problem, however at the cost of not being able to use the convenient normal equations for learning the parameters. To perform the training, we will need numerical optimization, which we will discuss later in Section 5.3.

The classification problem

Supervised machine learning amounts to predicting the output from the input. In more detail, we understand classification as the problem of predicting the class probabilities

$$p(y = m \,|\, \mathbf{x}), \tag{3.27}$$

where y is the output $(1, 2, \ldots,$ or $M)$ and $\mathbf{x}$ is the input.[4] In words, $p(y = m \,|\, \mathbf{x})$ describes *the probability for class m given that we know the input* $\mathbf{x}$. This probability will be a cornerstone, so we will first spend some effort to understand it well. Talking about $p(y \,|\, \mathbf{x})$ implies that we think about the class label y as a random variable. Why? Because we choose to model the real world, from where the data originates, as involving a certain amount of randomness (cf. the random error term ε in regression). Let us illustrate with an example:

Example 3.4: 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

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

In this way, the probabilities $p(y \,|\, \mathbf{x})$ 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.

[4] Note that we use $p(y \,|\, \mathbf{x})$ to denote probability masses (y categorical) as well as probability densities (y numerical).

We will learn a model which amounts to describing the class probabilities $p(y \mid \mathbf{x})$. That model will be called a *classifier*. More specifically, for binary classification problems ($M = 2$, and y is either 1 or -1), we learn a model $g(\mathbf{x})$ for which

$$\boxed{p(y = 1 \mid \mathbf{x}) \text{ is modeled by } g(\mathbf{x}).} \tag{3.28a}$$

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

$$\boxed{p(y = -1 \mid \mathbf{x}) \text{ is modeled by } 1 - g(\mathbf{x}).} \tag{3.28b}$$

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.

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

$$\boxed{\begin{bmatrix} p(y = 1 \mid \mathbf{x}) \\ p(y = 2 \mid \mathbf{x}) \\ \vdots \\ p(y = M \mid \mathbf{x}) \end{bmatrix} \text{ is modeled by } \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ \vdots \\ g_M(\mathbf{x}) \end{bmatrix} = \mathbf{g}(\mathbf{x}).} \tag{3.29}$$

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 $\mathbf{x}$.

Logistic regression

We will now introduce one possible way of modeling the conditional class probabilities, referred to as the logistic regression model. Logistic regression can be viewed as an modification of the linear regression model so that it fits the classification setting. We start with binary classification, that is we want to find a function $g(\mathbf{x})$ that approximates the conditional probability of the positive class. From the previous part of this chapter we know that the linear regression model, without the noise term, is given by

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p = \boldsymbol{\theta}^\mathsf{T} \mathbf{x}. \tag{3.30}$$

The input to this function is $\mathbf{x}$, and the output, here denoted by z, takes values on the entire real line. In classification, $g(\mathbf{x})$ should indeed be a function of $\mathbf{x}$, but only take values on the interval $[0, 1]$. The key idea underlying logistic regression is thus to 'squeeze' the output from linear regression z to the interval $[0, 1]$ by using the logistic function $h(z) = \frac{e^z}{1+e^z}$. See Figure 3.3. Since the logistic function is limited to take values between 0 and 1 we obtain a function of $\mathbf{x}$

$$g(\mathbf{x}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}, \tag{3.31a}$$

that we can interpret as a probability. This is the logistic regression model for $p(y = 1 \mid \mathbf{x})$. Note that this implicitly also gives a model for $p(y = -1 \mid \mathbf{x})$,

$$1 - g(\mathbf{x}) = 1 - \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}. \tag{3.31b}$$

We now have a model $g(\mathbf{x})$ which contains unknown parameters $\boldsymbol{\theta}$. As for linear regression, the parameters can be learned from training data. That is, we have constructed a parametric binary classifier, called *logistic regression*.

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

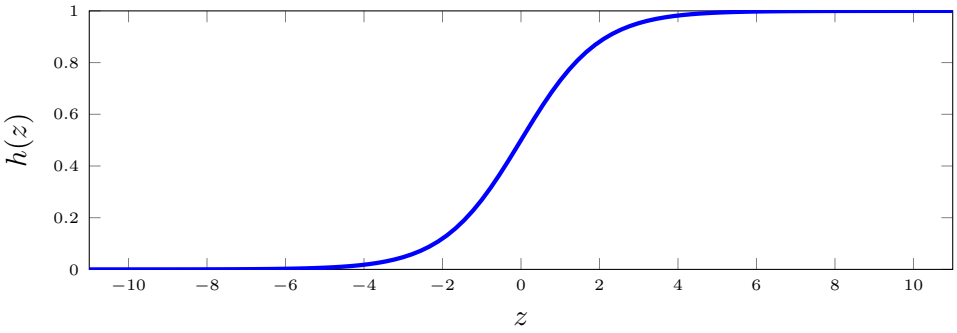


Figure 3.3: The logistic function $h(z) = \frac{e^z}{1+e^z}$.

Training logistic regression: loss functions for classification

By using the logistic function, we have transformed linear regression (a regression model) into logistic regression (a classification model). 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). Starting with the maximum likelihood approach, we will now derive a loss function for how to learn $\boldsymbol{\theta}$ from training data $\mathcal{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. The maximum likelihood problem amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \,|\, \mathbf{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \ln p(y_i \,|\, \mathbf{x}_i; \boldsymbol{\theta}), \tag{3.32}$$

where we, similarly to linear regression (3.20), 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 $p(y = 1 \,|\, \mathbf{x}; \boldsymbol{\theta})$ is $g(\mathbf{x}; \boldsymbol{\theta})$, which gives

$$\ln p(y_i \,|\, \mathbf{x}_i; \boldsymbol{\theta}) = \begin{cases} \ln g(\mathbf{x}_i; \boldsymbol{\theta}) & \text{if } y_i = 1, \\ \ln \left(1 - g(\mathbf{x}_i; \boldsymbol{\theta})\right) & \text{if } y_i = -1. \end{cases} \tag{3.33}$$

It is common to turn the maximization problem (3.32) into an equivalent minimization problem by using the negative log-likelihood as cost function, $J(\boldsymbol{\theta}) = -\frac{1}{n} \sum \ln p(y_i \,|\, \mathbf{x}_i; \boldsymbol{\theta})$, that is

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\begin{cases} -\ln g(\mathbf{x}_i; \boldsymbol{\theta}) & \text{if } y_i = 1, \\ -\ln \left(1 - g(\mathbf{x}_i; \boldsymbol{\theta})\right) & \text{if } y_i = -1. \end{cases}}_{\text{Binary cross-entropy loss } L(g(\mathbf{x}_i;\boldsymbol{\theta}),y_i)}. \tag{3.34}$$

This loss function is called the *cross-entropy loss* or *negative log-likelihood loss*. It is not specific to logistic regression, but can be used for any classifier that predict class probabilities $g(\mathbf{x}; \boldsymbol{\theta})$.

However, considering specifically the logistic regression model we can write out the cost function in more detail. In doing so, the particular choice of labeling $\{-1, 1\}$ turns out to be convenient. Specifically, note that for $y_i = 1$ we can write

$$g(\mathbf{x}_i; \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}, \tag{3.35a}$$

and for $y_i = -1$

$$1 - g(\mathbf{x}_i; \boldsymbol{\theta}) = \frac{e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}. \tag{3.35b}$$

. Since we get the same expression in both cases, we write (3.34) compactly as

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} -\ln \frac{e^{y_i \boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1+e^{y_i \boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{1}{n}\sum_{i=1}^{n} -\ln \frac{1}{1+e^{-y_i \boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{1}{n}\sum_{i=1}^{n} \underbrace{\ln\left(1+\exp\left(-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i\right)\right)}_{\text{Logistic loss } L(\mathbf{x}_i, y_i, \boldsymbol{\theta})}. \tag{3.36}$$

We have now derived a loss function $L(\mathbf{x}, y_i, \boldsymbol{\theta})$ which is the cross-entropy loss for logistic regression. This particular loss function is called the *logistic loss* (or sometimes binomial deviance). Learning a logistic regression model thus amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \ln\left(1+e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}\right). \tag{3.37}$$

Contrary to linear regression, (3.37) has no closed-form solution, so we have to use numerical optimization instead. We will come back to this later in Section 5.3.

Remark 3.2 *Note that whereas the cross-entropy loss* (3.34) *is a function of* $g(\mathbf{x};\boldsymbol{\theta})$*, the logistic loss* (3.36) *is rather a function of* $\boldsymbol{\theta}$*. For logistic regression they are equivalent, but when we later will introduce other classifiers we will have to be aware of this difference.*

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, and we will follow one path which also will be useful later in deep learning (Chapter 6).

For the binary problem, we used the logistic function to design a model $g(\mathbf{x})$ (a scalar-valued function) representing $p(y = 1 \,|\, \mathbf{x})$. 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 (3.30), each denoted z_m and each with a different set parameters $\boldsymbol{\theta}_m$. We stack all z_m into $\mathbf{z} = [z_1\ z_2\ \ldots\ z_M]^\mathsf{T}$ and use the *softmax function* as a vector-valued generalization of the logistic function,

$$\text{softmax}(\mathbf{z}) \triangleq \frac{1}{\sum_{m=1}^{M} e^{z_m}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_M} \end{bmatrix}. \tag{3.38}$$

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 (3.31), we have now combined linear regression and the softmax function to model the class probabilities,

$$\mathbf{g}(\mathbf{x}) = \text{softmax}(\mathbf{z}), \qquad \text{where } \mathbf{z} = \begin{bmatrix} \boldsymbol{\theta}_1^\mathsf{T}\mathbf{x}_i \\ \boldsymbol{\theta}_2^\mathsf{T}\mathbf{x}_i \\ \vdots \\ \boldsymbol{\theta}_M^\mathsf{T}\mathbf{x}_i \end{bmatrix}, \tag{3.39}$$

or equivalently

$$g_m(\mathbf{x}) = \frac{e^{\boldsymbol{\theta}_m^\mathsf{T}\mathbf{x}_i}}{\sum_{j=1}^{M} e^{\boldsymbol{\theta}_j^\mathsf{T}\mathbf{x}_i}}. \tag{3.40}$$

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 idea. We use $\boldsymbol{\theta}$ to denote *all* parameters in $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M$. Since $g_m(\mathbf{x}; \boldsymbol{\theta})$ is our model for $p(y_i = m \,|\, \mathbf{x}_i; \boldsymbol{\theta})$, the cost function for the cross-entropy (or negative log likelihood) loss for the multiclass problem is

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} -\ln p(y_i \,|\, \mathbf{x}_i; \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{-\ln g_{y_i}(\mathbf{x}_i; \boldsymbol{\theta})}_{\substack{\text{Multi-class cross-entropy} \\ \text{loss } L(g(\mathbf{x}_i;\boldsymbol{\theta}),y_i)}} . \tag{3.41}$$

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 3.5.

Inserting (3.40) into the multiclass cross-entropy loss (3.41) gives the multiclass softmax version of the logistic loss. The cost function becomes,

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(-\boldsymbol{\theta}_{y_i}^{\mathsf{T}} \mathbf{x}_i + \ln \sum_{j=1}^{M} e^{\boldsymbol{\theta}_j^{\mathsf{T}} \mathbf{x}_i} \right) . \tag{3.42}$$

Example 3.5: The cros-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:

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

We use logistic regression with softmax, and train the classifier using the cross-entropy (equivalently, negative log-likelihood) loss (3.41). For any $\mathbf{x}$ and $\boldsymbol{\theta}$, the model will return a 3-dimensional probability vector $\mathbf{g}(\mathbf{x}; \boldsymbol{\theta})$. 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 cross-entropy loss now simply amounts to compute the negative average of all circled elements, where the element that we have circled in each row is given by the training labels $\mathbf{y}$. Training the model amounts to finding $\boldsymbol{\theta}$ such that this negative average of these elements is minimized.

Time to reflect 3.3: *Can you derive* (3.34) *as a special case of* (3.41)*?*
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 not a problem in practice. Compare the multiclass model* (3.39) *for the case* $M = 2$ *with binary logistic regression* (3.31) *and see if you can spot it!*

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 $p(y = 1 \,|\, \mathbf{x}_\star)$ in binary classification) or $\mathbf{g}(\mathbf{x}_\star)$ (our model for $p(y = m \,|\, \mathbf{x}_\star)$ in multiclass classification). If we actually want to make a "hard" prediction for the test input $\mathbf{x}_\star$, i.e., deciding whether we believe $y_\star = -1$ or $y_\star = 1$ in binary classification, we have to add a final step in which we turn the predicted probabilities into a class prediction. The most common approach is to *let the class prediction* $\widehat{y}(\mathbf{x}_\star)$ *be the most probable class according to the predicted probabilities*. For binary classification, we can express this[5] as

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} 1 & \text{if } g(\mathbf{x}_\star) > 0.5 \\ -1 & \text{if } g(\mathbf{x}_\star) \leq 0.5 \end{cases}, \tag{3.43}$$

which for binary logistic regression simplifies to

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_\star). \tag{3.44}$$

This is illustrated in Figure 3.4 for a two-dimensional input $\mathbf{x}$, and we now have everything in place for summarizing binary logistic regression in Algorithm 3. For the multiclass problem, we express it as

$$\widehat{y}(\mathbf{x}_\star) = \arg\max_m g_m(\mathbf{x}_\star). \tag{3.45}$$

Now $\widehat{y}(\mathbf{x}_\star)$ gives us a predicted class for the test input $\mathbf{x}_\star$. (Sometimes, there might be reasons to construct the prediction $\widehat{y}(\mathbf{x}_\star)$ differently than (3.43) or (3.45), which we discuss later in this chapter.) If the prediction $\widehat{y}(\mathbf{x}_\star)$ is different from the true class label $y_\star$, it is a referred to as a *misclassification*.

A classifier associates all points in the space of possible test inputs $\mathbf{x}_\star$ to a prediction $\widehat{y}(\mathbf{x}_\star)$. Most often, the classifier forms regions where all points have the same prediction. The boundary between those regions, that is, the curve in the $\mathbf{x}$-space which separates different class predictions from each other, is called the *decision boundary*.

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

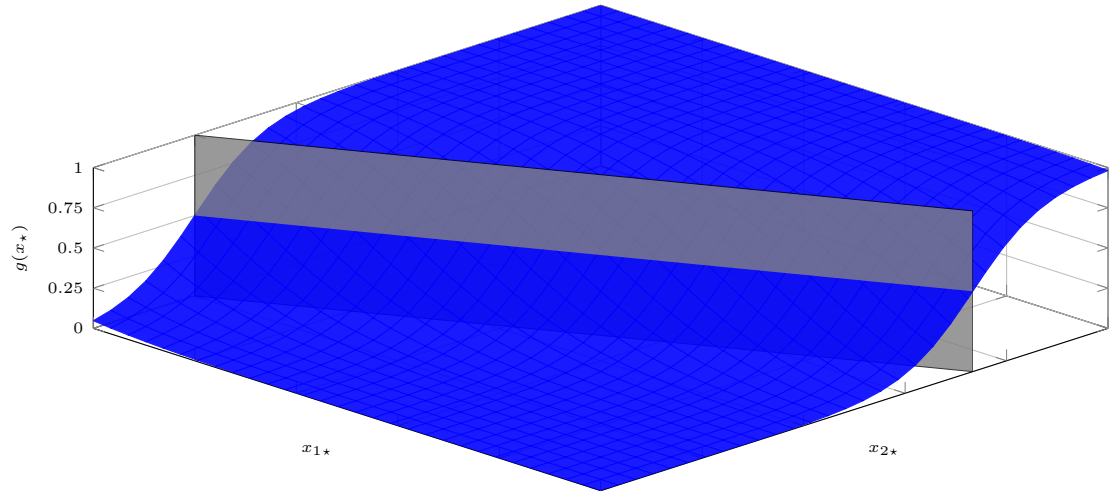
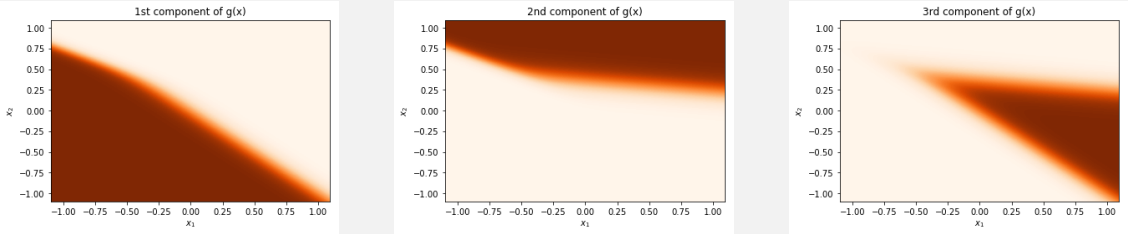


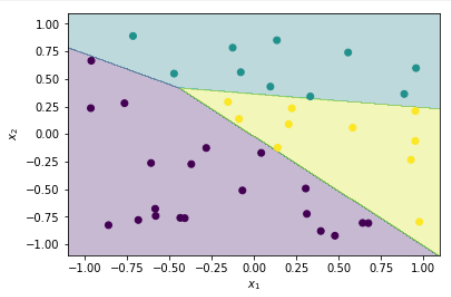
Figure 3.4: Consider binary classification ($y = -1$ or 1) when the input $\mathbf{x}$ is two-dimensional. Once $\boldsymbol{\theta}$ is learned from training data (not shown), logistic regression gives us a model $g(\mathbf{x}_\star)$ (blue surface), which is a prediction of $p(y = 1 \,|\, \mathbf{x}_\star)$. This implicitly also gives a prediction for $p(y = -1 \,|\, \mathbf{x}_\star)$ 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 is taken as the prediction, as in Equation (3.43). The point(s) where the prediction changes from from one class to another is called the *decision boundary* (gray plane).

Example 3.6: Multiclass predictions

We extend the data set from Example 3.5 with a few more data samples, and train a logistic regression classifier, which becomes a function from $\mathbb{R}^2$ (the space where $\mathbf{x}$ lives) to $[0,1]^3$ (the space in which $\mathbf{g}(\mathbf{x})$ lives). We can illustrate this using three plots, each one showing a different component $g_m(\mathbf{x})$ of $\mathbf{g}(\mathbf{x})$.



The more intense color, the higher value of $g_m(\mathbf{x})$. The figures hence show the probability, according to the logistic regression model, for one class each. If we are to turn these probabilities into class prediction according to (3.45), we simply pick the one which has the most intense orange color, and thereby obtain the predictions.



We have also plotted the training data in the same figure. The boundaries that we obtain between the different regions are the decision boundaries.

Algorithm 3: 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 (3.37) numerically.

Predict

2 Compute $g(\mathbf{x}_\star)$ (3.31a).
3 If $g(\mathbf{x}_\star) > 0.5$, return $\widehat{y}(\mathbf{x}_\star) = 1$, otherwise return $\widehat{y}(\mathbf{x}_\star) = -1$.

Explicit decision boundaries for logistic regression

For binary classification in general, we can compute the decision boundary by solving the equation

$$g(\mathbf{x}) = 1 - g(\mathbf{x}). \tag{3.46}$$

Solutions to this equation gives us points in the input space for which the two classes are equally probable (according to our model). These points therefore lie on the decision boundary. For binary logistic regression, this means

$$\frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1+e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1+e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} \Leftrightarrow e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}} = 1 \Leftrightarrow \boldsymbol{\theta}^\mathsf{T}\mathbf{x} = 0. \tag{3.47}$$

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, and logistic regression is therefore referred to as a *linear classifier*. 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.

Linear and nonlinear classifiers

We distinguish between different types of classifiers by the shape of their decision boundaries. A classifier whose decision boundaries are constructed using linear hyperplanes is called a *linear classifier*. As we saw above, logistic regression is a linear classified. A *nonlinear classifier* can possibly have nonlinear decision boundaries.

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

Tuning the decision threshold

For binary classification, we have said that we convert the predicted probability $g(\mathbf{x})$ to a class prediction as

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > r \\ -1 & \text{if } g(\mathbf{x}) \leq r \end{cases}, \tag{3.48}$$

with the decision threshold $r = 0.5$. The value $r = 0.5$ is a sensible default choice, but depending on the problem at hand it is not necessarily the best one.

It can be shown that *if* $g(\mathbf{x}) = p(y = 1 \,|\, \mathbf{x})$, that is, the model provides a correct description of the true class probabilities, then the choice $r = 0.5$ will give the smallest possible number of misclassification on average. In other words, $r = 0.5$ will minimize the so-called *misclassification rate*. There are, however, two important points to make here: First, in practice, it is usually not true that $g(\mathbf{x})$ is exactly equal to $p(y = 1 \,|\, \mathbf{x})$ (since $g(\mathbf{x})$ is only a model learned from a limited set of training data), and there is consequently no guarantee that $r = 0.5$ is the choice that will minimize the misclassification rate.

Second, and perhaps more important, is that the misclassification rate is not always the most important aspect of a binary classifier. Many binary classification problems are asymmetric or imbalanced, meaning that the classes are of different importance or occur with very different frequency. A typical example is a health-care application where the two classes represent the presence (positive class) or absence (negative class) of a certain medical disorder. Such a problem can be asymmetric in the sense that falsely predicting a sick patient as healthy might have much more severe consequences than the opposite. This asymmetry suggests that we should pay more attention to so-called false negatives (patients misclassified as healthy) than the false positives (patients misclassified as sick), and not just minimize the plain misclassification rate. Furthermore, the problem would be imbalanced if the disorder is very rare, meaning that the vast majority of the data samples belong to the negative class. By only considering the misclassification rate, 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 handle this and choose r more systematically in Section 4.5. In the end, however, the decision threshold r is a choice the user has to make.

3.A Derivation of the normal equations

The normal equations (3.14)

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

can be derived from (3.13) (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 (3.14) 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 (3.14) has infinitely many solutions $\widehat{\boldsymbol{\theta}}$, which all are equally good solutions to the problem (3.13).

A calculus approach

Let

$$V(\boldsymbol{\theta}) = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^\mathsf{T}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y} - 2\mathbf{y}^\mathsf{T}\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta}, \tag{3.49}$$

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

$$\frac{\partial}{\partial \boldsymbol{\theta}} V(\boldsymbol{\theta}) = -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta}. \tag{3.50}$$

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

$$\frac{\partial}{\partial \boldsymbol{\theta}} V(\widehat{\boldsymbol{\theta}}) = 0 \Leftrightarrow -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta} = 0 \Leftrightarrow \mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{3.51}$$

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

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\theta} - (\mathbf{y}_\perp + \mathbf{y}_\parallel)\|_2^2 = \|(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\parallel) - \mathbf{y}_\perp\|_2^2 \geq \|\mathbf{y}_\perp\|_2^2, \tag{3.52}$$

and the triangle inequality also gives us

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\perp - \mathbf{y}_\parallel\|_2^2 \leq \|\mathbf{y}_\perp\|_2^2 + \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\parallel\|_2^2. \tag{3.53}$$

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.,

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\theta}})^\mathsf{T} c_j = 0, j = 1, \ldots, p+1 \tag{3.54}$$

(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

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\theta}})^\mathsf{T}\mathbf{X} = 0, \tag{3.55}$$

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/`

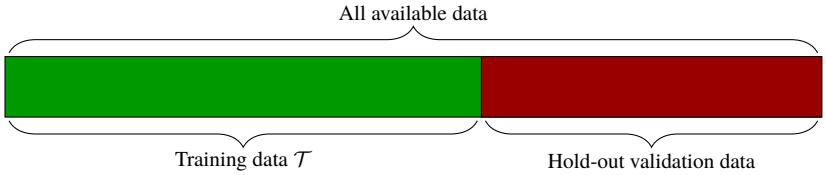


Figure 4.1: The hold-out validation dataset approach: If we split the available data in two sets and train the model on the training set, we can compute $E_{\text{hold-out}}$ using the hold-out validation set. The more data that are in the hold-out validation dataset, the less variance (better estimate) 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.*

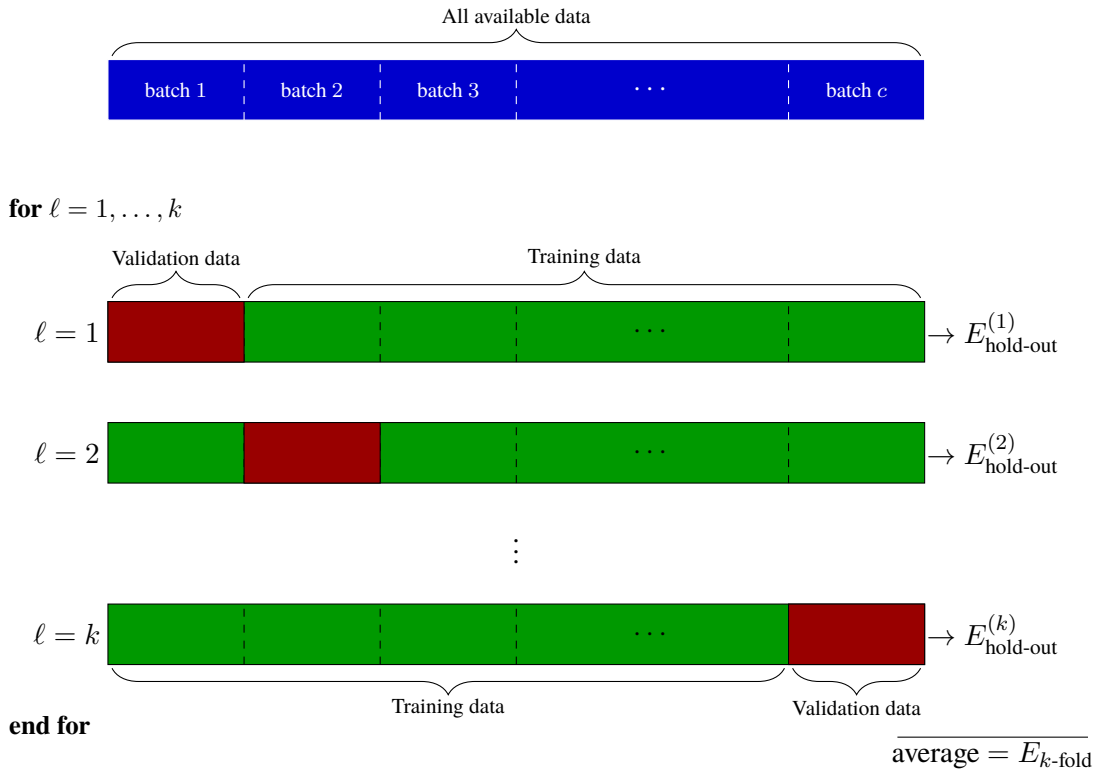


Figure 4.2: Illustration of k-fold cross-validation. The data is split in k batches of similar sizes. When looping over $\ell = 1, 2, \ldots, k$, batch ℓ is held out as validation data, and the model is trained on the remaining $k-1$ data batches. Each time, the trained model is used to compute the average 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.

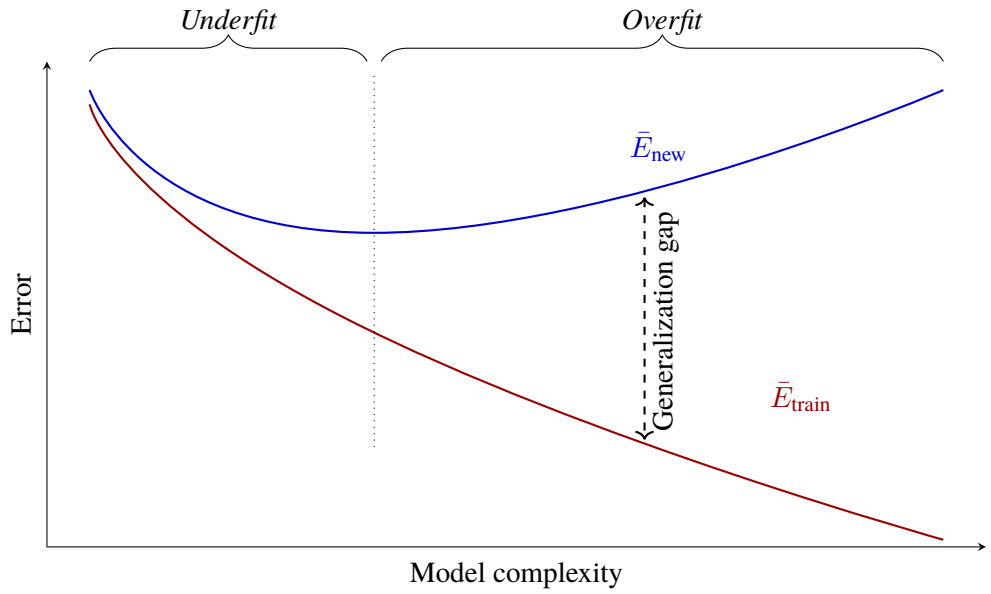


Figure 4.3: Behavior of $\bar{E}_{\text{train}}$ and $\bar{E}_{\text{new}}$ for many supervised machine learning methods, as a function of model complexity. We have not made a formal definition of complexity, but a rough proxy is the number of parameters that are learned from the data. The difference 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.

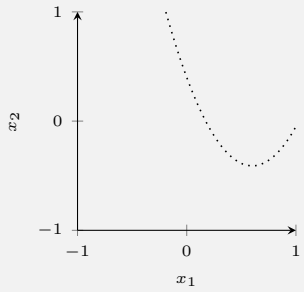
Typically, *higher model complexity implies larger generalization gap*. Furthermore, $\bar{E}_{\text{train}}$ decreases as the model complexity increases, whereas $\bar{E}_{\text{new}}$ typically attains a minimum for some intermediate model complexity value: too small *and* too high model complexity 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*

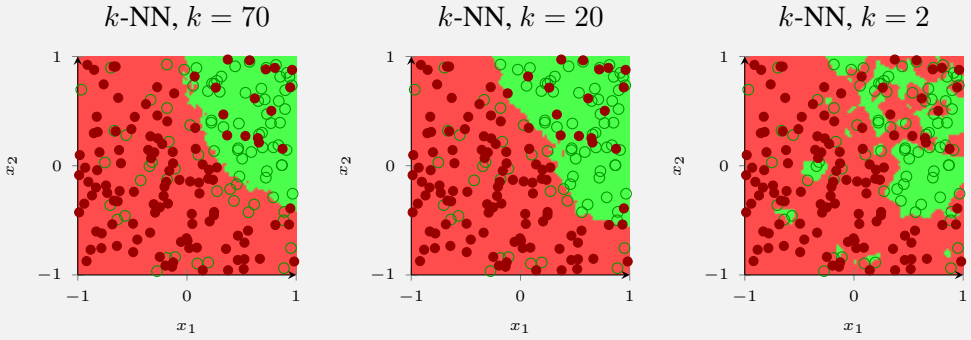
counter-examples may exist. One should also keep in mind that claims about $\bar{E}_{train}$ *and* $\bar{E}_{new}$ *are about the* average *behavior, which hopefully is clear in Example 4.1.*

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:

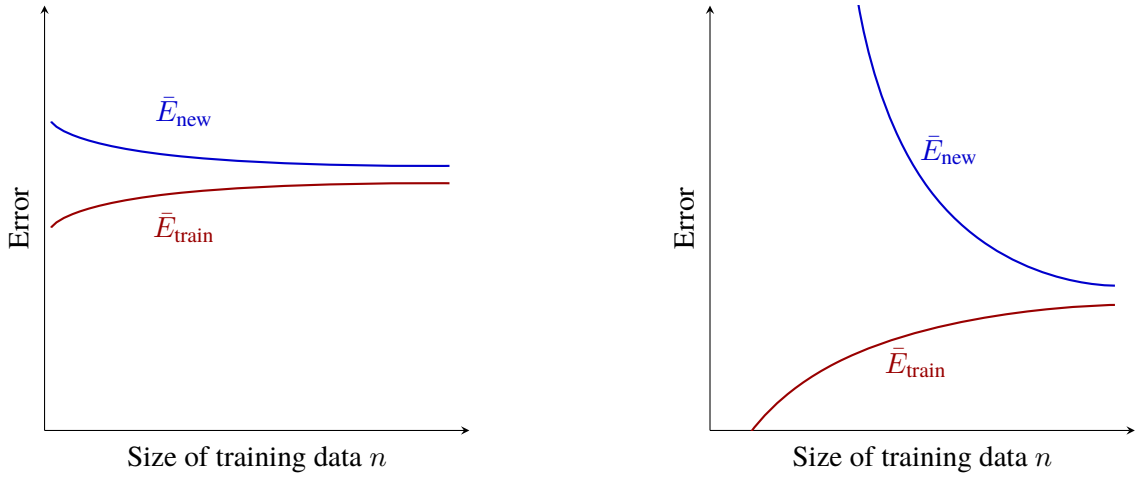


Intuitively we see that $k = 2$ (right) adapts too well to the data. With $k = 70$, on the other hand, the model is rigid enough not to adapt to the noise, but appears to possibly be too inflexible to adapt well to the true dotted line above.

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 $E_{\text{new}} = E_{\text{train}} +$ 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.

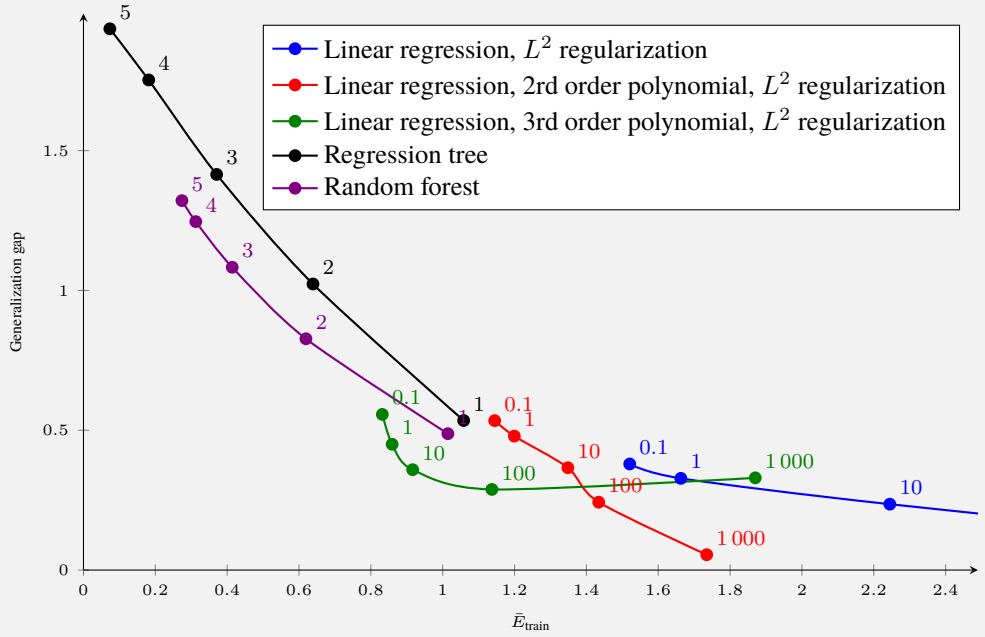
It is also important to realize that the one-dimensional model complexity scale in Figure 4.3 does not make justice for the space of all supervised machine learning methods. For a given problem, one method can have a smaller generalization gap than another method *without* having a larger training error. 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

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

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

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,

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

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,

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

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 $\bar{z}$,

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

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

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

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

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

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

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.

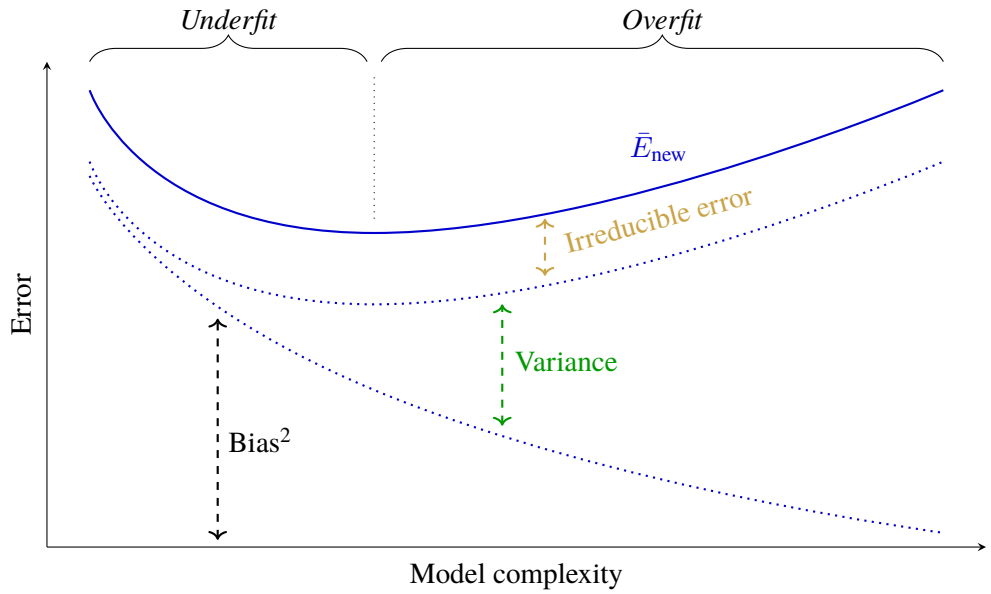


Figure 4.5: The bias-variance decomposition of $\bar{E}_{\text{new}}$ (cf. Figure 4.3). Low model complexity means high bias. The more complicated the model is, the more it adapts to training data, and the higher variance. The irreducible error is always constant. The problem of achieving a small E_{new}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

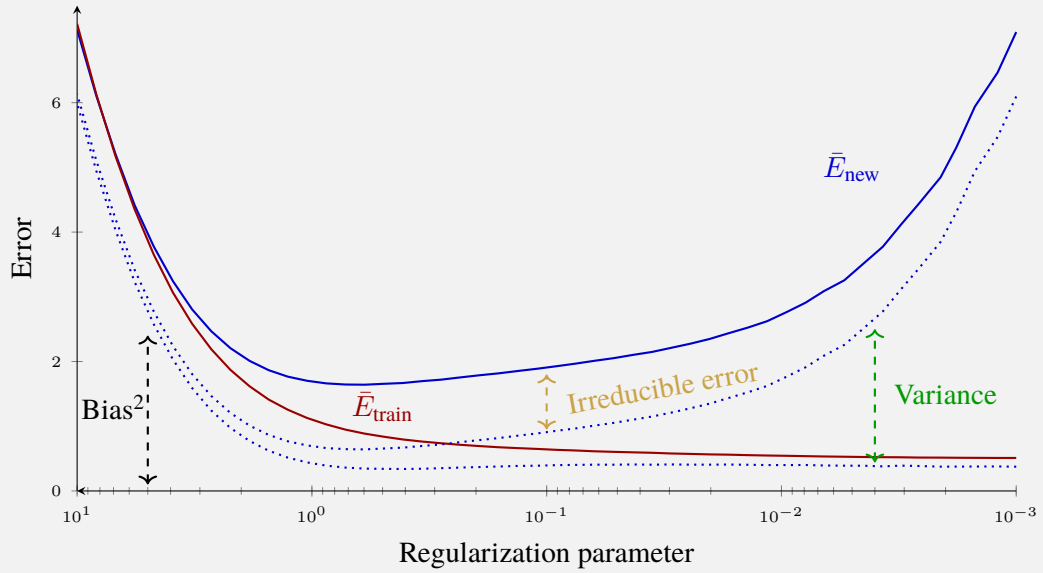
$$y = 5 - 2\mathbf{x} + \mathbf{x}^3 + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1). \tag{4.20}$$

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.



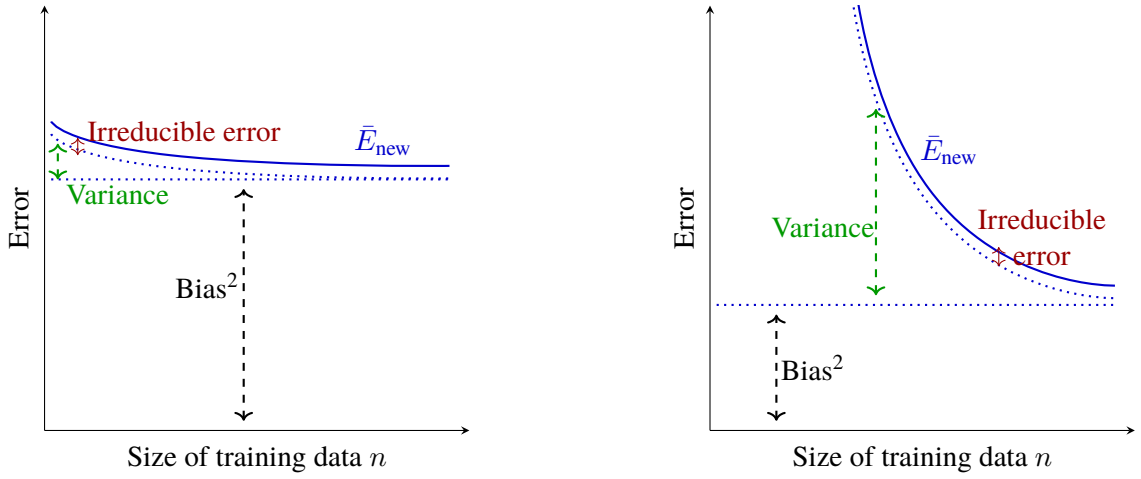
Bias, variance and its relation to the size n of training data

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

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

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

$$\text{generalization gap} \gtrapprox \text{variance}, \quad (4.23a)$$

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

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.

[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)

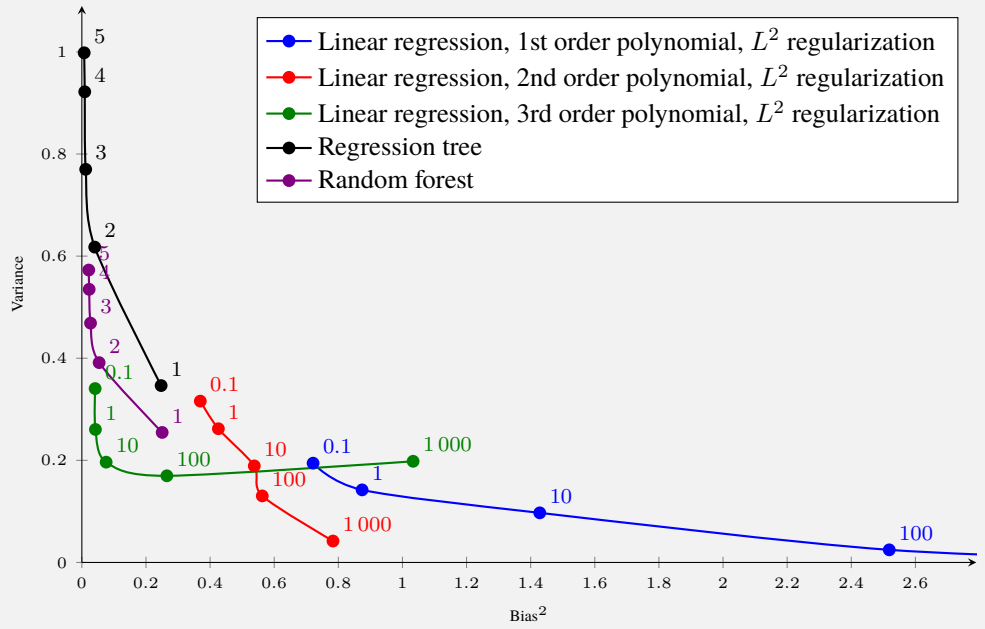
A much more useful way of dealing with bias and variance is the meta-method called bagging, Chapter 7. It makes use of several copies (an ensemble) of a base model, each trained on a slightly different version of the training dataset. Since bagging averages over many base models, it reduces the variance,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

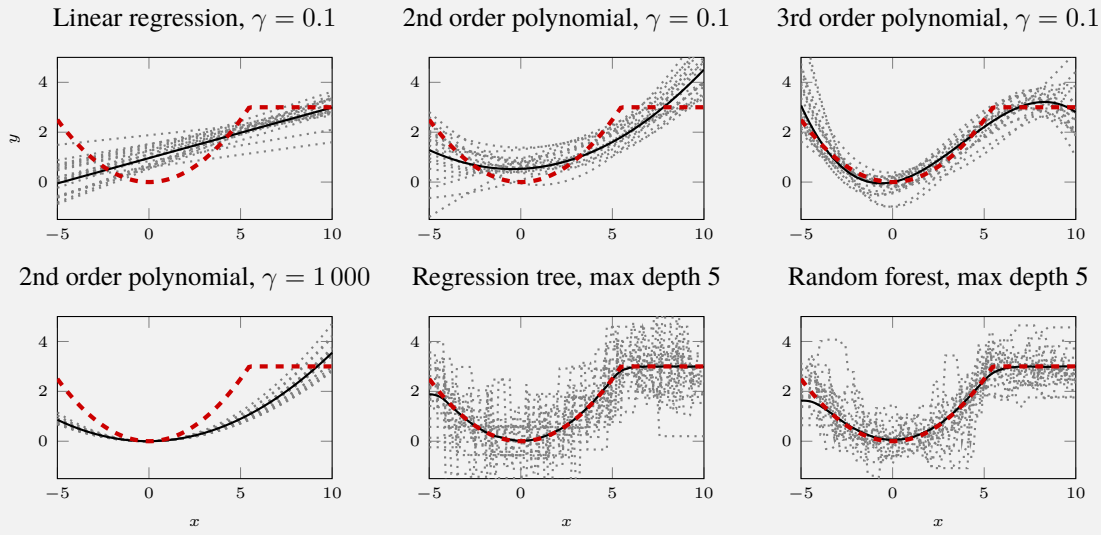
Example 4.4: Bias and variance for a regression problem

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 Section 3.2.

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 $\{\texttt{normal}, \texttt{hyperthyroid}, \texttt{hypothyroid}\}$, which we convert into the binary problem with the output classes $\{\texttt{normal}, \texttt{not normal}\}$. 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 (3.48)). We obtain the confusion matrix

	$y = \texttt{normal}$	$y = \texttt{not normal}$
$\widehat{y}(\mathbf{x}) = \texttt{normal}$	3177	237
$\widehat{y}(\mathbf{x}) = \texttt{not normal}$	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.)

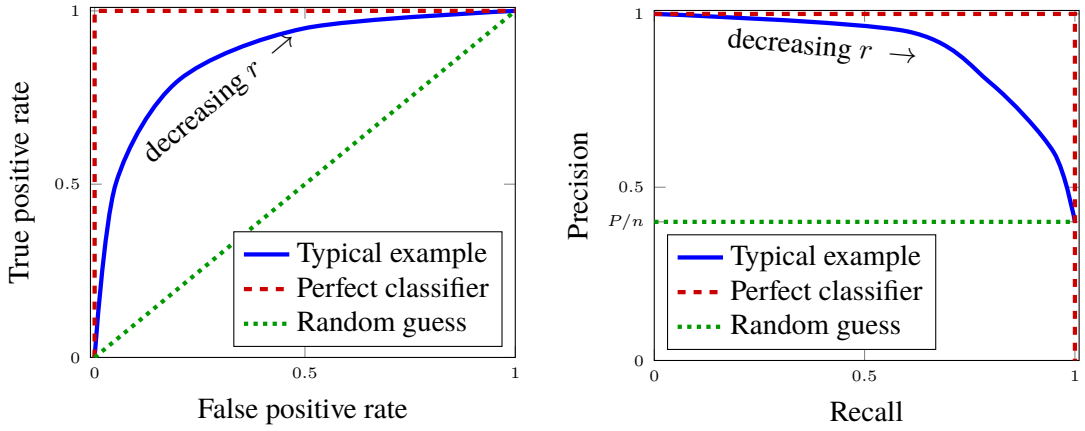
To change the picture, we change the threshold to $r = 0.15$, and obtain new predictions with the following confusion matrix instead:

	$y = \texttt{normal}$	$y = \texttt{not normal}$
$\widehat{y} = \texttt{normal}$	3067	165
$\widehat{y} = \texttt{not normal}$	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 (3.48) 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. (3.43), 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

5.1 Loss functions

As we discussed in Chapter 3, many regression and classification models are trained by minimizing a cost function $J(\boldsymbol{\theta})$, which is the average value of the loss function[1] L evaluated on the training data.

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

The choice of loss function is, in the end, a user choice. Certain combinations of models and loss functions have proven particularly fruitful, but we will in this section have a general discussion about different loss functions and their properties without connecting them to a specific model. We will later make use of some of the loss functions introduced here when we come to boosting (Chapter 7) and support vector machines (Chapter 8).

Different loss functions give different solutions $\widehat{\boldsymbol{\theta}}$, and the choice of loss function is one of the choices the machine learning engineer has to make. Different loss functions give different characteristics to the learned model.

An important property for a loss function is the *population minimizer*. The population minimizer describes what function minimizes the loss function when $n \to \infty$. For example is the population minimizer for the binary cross-entropy loss (3.34) indeed[2] $g(\mathbf{x}, \theta) = p(y = 1 \,|\, \mathbf{x})$. In other words, the binary cross-entropy loss trains the model such that $g(\mathbf{x}, \theta)$ indeed becomes (if possible) the sought conditional class probabilities. The population minimizer is important for understanding the behavior of models that are learned from it.

Another important concept for loss function is *robustness*. We do not give a formal definition of robustness, but we informally say that a loss function is robust if outliers in the training data (spurious data points that do not describe the relationship we are interested in modeling) only have a small impact on the trained model. Conversely do outliers have a major impact on the learned if the loss function is not robust. Robustness is however not a binary property, but loss functions can be robust to a greater or smaller degree. Indeed is a robust loss function desired in applications where outliers occur in training data, but some of the common "default" loss-functions, such as squared error loss, are unfortunately not particularly robust.

Loss functions for regression

In Chapter 3 we introduced the squared error loss

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

Another common loss function is the absolute error loss,

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

[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 some loss functions are formulated in terms of the prediction $\widehat{y}$, whereas other loss functions are formulated in terms of the classifier margin, $g(\mathbf{x})$, etc.

[2] This assumes that the model is flexible enough such that it can describe $p(y = 1 \,|\, \mathbf{x})$.

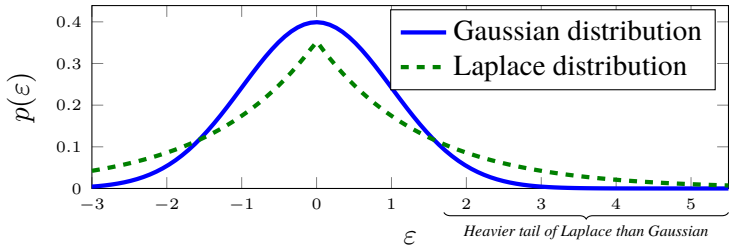


Figure 5.1: From a maximum likelihood perspective, the squared error loss corresponds to assuming a Gaussian distribution (solid blue line) for the noise ε. The absolute error loss, on the other hand, corresponds to a Laplace distribution (dashed green line) for the noise ε. The variance for both distributions is 1 in this figure. The Laplace distribution, however, has a heavier tail than the Gaussian, meaning that it assigns a higher probability to ε far from zero. Data samples in regression that deviates considerably from the other data, so-called outliers, correspond to ε having a distribution with heavy tails. With this perspective, it comes at no surprise that the absolute error loss is more robust (less sensitive to outliers) than squared error loss, since it corresponds to the statistical assumption that ε indeed has a distribution with a heavier tail.

The absolute error loss (5.3) is more robust than the squared error loss (5.2), which is a desired property. For linear regression, however, has the squared error loss the advantage that if $\widehat{\boldsymbol{\theta}}$ is unique, the optimization problem (3.10) has a closed-form solution (the normal equations, (3.15)), whereas the absolute error loss requires numerical optimization.

We have already introduced the maximum likelihood motivation of the squared error loss in Chapter 3, 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 assuming ε has a Laplace distribution,

$$\varepsilon \sim \mathcal{L}(0, b_\varepsilon). \tag{5.4}$$

This statistical maximum likelihood perspective can give some insights into why the absolute error loss is more robust (less sensitive to outliers) than the squared error loss. As illustrated by Figure 5.1 the Laplace distribution has a thicker tail than the Gaussian distribution, and hence describes outliers as more probable. In other words, sporadic large noise values (i.e., outliers) are more probable under the Laplace assumption.

An 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 parameter. This loss places a "tolerance space" of width 2ϵ around the true $\mathbf{y}$, and behaves like the absolute error loss outside this band. Thanks to the similarities to the absolute error, the ϵ-insensitive loss also enjoys the same robustness properties. It is, however, less natural to interpret from a maximum likelihood perspective. It will nevertheless turn out to be an useful loss function for support vector regression in Chapter 8.

Loss functions for binary classification

A natural loss function for binary classification is the misclassification loss,

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

Even though a small misclassification loss often is the ultimate goal when predicting new, previously unseen, data, the misclassification loss is rarely used in practice when training models. The reasons are twofold. The most apparent reason in that the cost function becomes a piecewise constant function, which is a poor optimization objective since the gradient is everywhere zero, except when it is undefined, so

gradient descent cannot be used. Furthermore, a more subtle reason is that for many models the final prediction $\widehat{y}$ does not reveal all the aspects of the classifier. Thinking in terms of decision boundaries, we may prefer to not have the decision boundary close up to the training data points, but (if possible) to push the boundary further away to have some "safety leeway". It is therefore of interest to consider also other types of loss functions for binary classification.

If learning a function $g(\mathbf{x})$ for binary classification, like in logistic regression, the maximum likelihood/cross entropy loss function is as introduced in Chapter 3

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

Unlike regression, there are no choices left in the cross entropy/maximum likelihood loss (other than what model to use for $g(\mathbf{x})$).

The cross entropy loss is however not the only loss function. To introduce an entire family of loss functions for logistic regression, let us first introduce the concept of margin.

Most 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

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

In this formulation logistic regression, for example, simply corresponds to $c(\mathbf{x}) = \theta^{\mathsf{T}}\mathbf{x}$. The decision boundary is then given by the values of $\mathbf{x}$ for which $c(\mathbf{x}) = 0$. For simplicity of presentation we will assume that no data points fall exactly on the decision boundary (which always gives rise to an ambiguity), so that we can assume that $\widehat{y}(\mathbf{x})$ as defined above is always either -1 or $+1$.

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

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

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

That is, learning θ in $c(\mathbf{x}) = \theta^{\mathsf{T}}\mathbf{x}$ by minimizing the loss function (5.10a), and make predictions as (5.8), is equivalent to logistic regression (except for the fact that we have lost the notion of a class probability estimate $g(\mathbf{x})$ in the notation). Furthermore, we can also formulate the misclassification loss in terms of margins as

$$L(y \cdot c(\mathbf{x})) = \begin{cases} 1 & \text{if } y \cdot c(\mathbf{x}) < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5.10b}$$

The benefit of formulating loss functions in terms of the margin, is that it allows us to naturally introduce other loss functions as well. Let us now introduce the *exponential loss* defined as

$$L(y \cdot c(\mathbf{x})) = \exp(-y \cdot c(\mathbf{x})), \tag{5.10c}$$

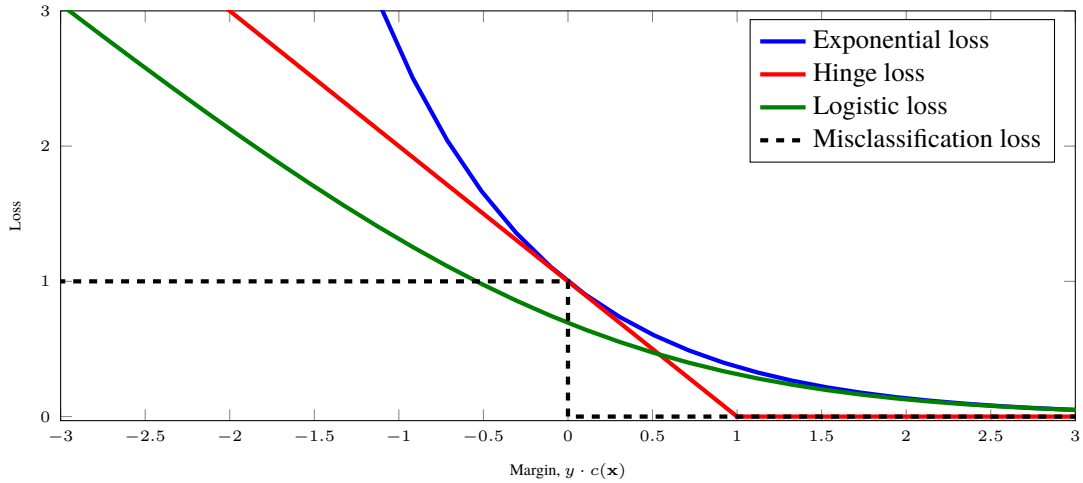


Figure 5.2: Comparison of some common loss functions for classification, plotted as a function of the margin.

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

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

The hinge loss is also more robust against outliers than the exponential loss. Figure 5.2 illustrates all these losses as a function of the margin.

Remark 5.1 *There is* no *universal correspondence between the margin $c(\mathbf{x})$ and the predicted class probability $g(\mathbf{x})$* (3.28)*. A consequence is that some of the classifiers trained using a "margin-based" loss function, including boosting (Chapter 7) and support vector classification (Chapter 8), only delivers a "hard" prediction $\widehat{y}(\mathbf{x})$ and no probability estimate $g(\mathbf{x})$.*

5.2 Regularization

We will now have a look at regularization, which is a useful tool for tuning the model flexibility (or complexity). Finding the right level of flexibility, and thereby avoiding so-called overfit, can be crucial for how good predictions a model is able to make. Regularization is applicable not only to linear and logistic regression, but also to some of the more advanced methods that we will discuss in later chapters.

The need for regularization

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. In fact, if $n = p + 1$ the linear regression model is able to always find a solution such that the cost function is zero, meaning the model fits the training data perfectly.

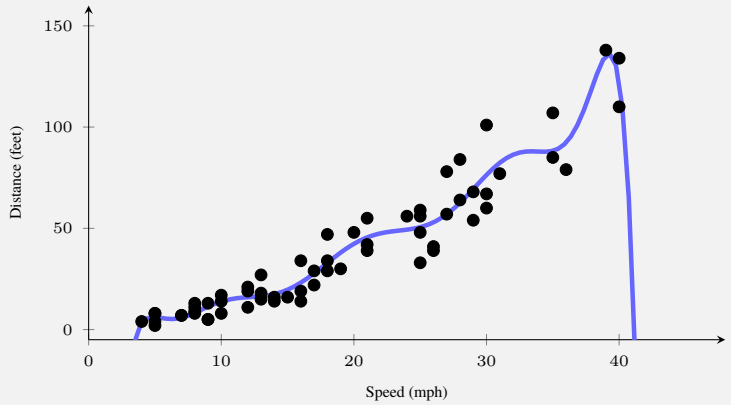
A common way of (somewhat artificially) increasing the input dimension in linear and logistic regression, which we will discuss more thoroughly in Chapter 8, is to make a nonlinear transformation of the input. A simple nonlinear transformation is to replace a one-dimensional input x with itself raised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

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \varepsilon, \tag{5.11}$$

and similarly for 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. Even though $n = p + 1$ is an extreme case, the conceptual problem with overfitting is present also in less extreme situations, as we will discuss thoroughly in Chapter 4.

Example 5.1: Overfit in car stopping distances

We consider again the regression problem with car stopping distances. Let us assume that we found the model used in Example 3.3 not to be flexible enough, but we want to try a more flexible model. Instead, we now use a linear regression model with a tenth order polynomial (blue line). Even though $n > p$, this becomes a typical example of *overfit* where the model adapts (too) well to the data, and the behavior in between the data samples and outside the data range is nonsense.



For linear (or logistic) regression with nonlinear input transformations, one could deal with the overfitting problem by a very careful selection of the nonlinear transformations. Such a careful selection can be quite hard to perform in practice, and the overfitting problem can occur also in models and situations when there is no such explicit choice to make. A more useful approach in practice is instead 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 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'. There are several ways to implement this idea mathematically, which leads to different regularization methods. We will have a closer look at the so-called L^2 and L^1 regularization.

L^2 regularization

Using 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. The purpose of the penalty term is to 'keep $\widehat{\boldsymbol{\theta}}$ small', whereas the original cost function rewards a good fit to training data. Here $\|\cdot\|_2$ is the Euclidean vector norm (or 2-norm) $\|\boldsymbol{\theta}\|_2 = \sqrt{\theta_0^2 + \theta_1^2 + \cdots + \theta_p^2}$, and $\|\cdot\|_2^2$ its square. That is, linear regression with square error lost (3.13) with L^2 regularization is[3]

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{5.12}$$

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

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

made. In the setting $\lambda = 0$ we recover the original least squares problem (3.13), 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. A common approach is to consider λ as a *hyperparameter* that can either be set manually, or in a more systematic fashion using cross-validation (Chapter 4). In its simplest fashion, cross-validation amounts to holding out a section of the training data, called validation data, and choose λ such that the predictions for the validation data are as good as possible.

It is actually possible to derive a version of the normal equations for (5.12), namely

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{5.13}$$

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

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{5.14}$$

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 (3.14) have no unique solution $\widehat{\boldsymbol{\theta}}$, whereas the L^2-regularized version always has the unique solution (5.14) if $\lambda > 0$.

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. The L^2-regularized version of logistic regression is

$$\widehat{\boldsymbol{\theta}} = \frac{1}{n}\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i\right)\right) + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{5.15}$$

It is common in practice to train logistic regression using (5.15) instead of (3.31). One reason is indeed to decrease possible problems with overfitting. The other reason is that for the non-regularized cost function (3.31), 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 training, that is, solving optimization problem (3.31), diverges with some datasets. The L^2 regularized logistic regression with $\lambda > 0$, however, always has a finite minimum and does not cause convergence for that reason.

L^1 regularization

With L^1 regularization (also called *LASSO*, an abbreviation for Least Absolute Shrinkage and Selection Operator), a penalty term $\|\boldsymbol{\theta}\|_1$ is instead 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 cost function for linear regression (with squared error loss) (3.13) then becomes

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1. \tag{5.16}$$

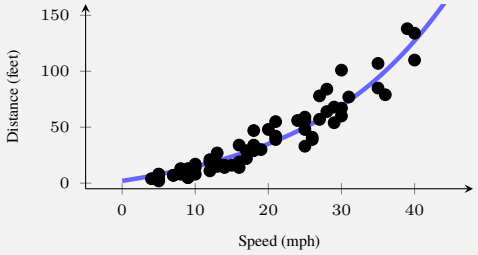
Contrary to linear regression with L^2 regularization (5.12), there is no closed-form solution available for (5.16). However, as we will see in Section 5.3, it is possible to design an efficient numerical optimization algorithm for solving (5.16).

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.

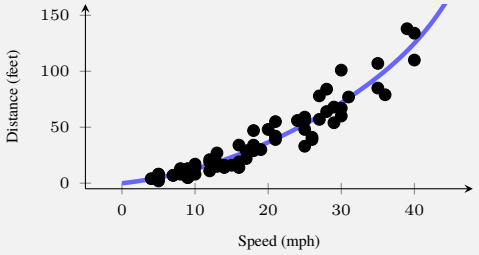
Example 5.2: Regularization for car stopping distance

Consider again the regression problem with the car stopping distances. To reduce the overfit problem, we can use L^2 or L^1 regularization. With a manually chosen λ, we obtain the following models

L^2 regularization



L^1 regularization



Both models suffer less from overfit than the non-regularized model in Example 5.1. These models are, however, not identical. Whereas all parameters are relatively small but non-zero in the L^2-regularized model, only 4 (out of 11) parameters are non-zero in the L^1-regularized model. 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. You have probably already figured out the general regularization scheme

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})}_{\text{(i)}} + \underbrace{\lambda}_{\text{(iii)}} \underbrace{R(\boldsymbol{\theta})}_{\text{(ii)}}. \tag{5.17}$$

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 will discuss 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.

Regularization of trees

Implicit regularization

Any supervised machine learning method that is trained by minimizing a cost function can be regularized as (5.17). 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, for some model early stopping has shown to be a practically useful way of counteracting overfit.

5.3 Parameter optimization

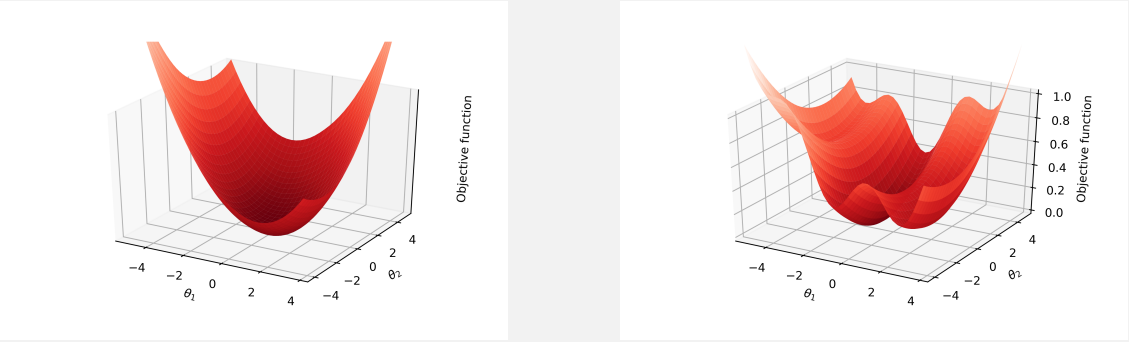
Many supervised machine learning methods, including linear and logistic regression, involves one (or more) optimization problems, such as (3.13), (3.37) or (5.16). It is therefore good to be familiar with some of the main strategies for how to solve optimization problems fast. Starting in the optimization 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.

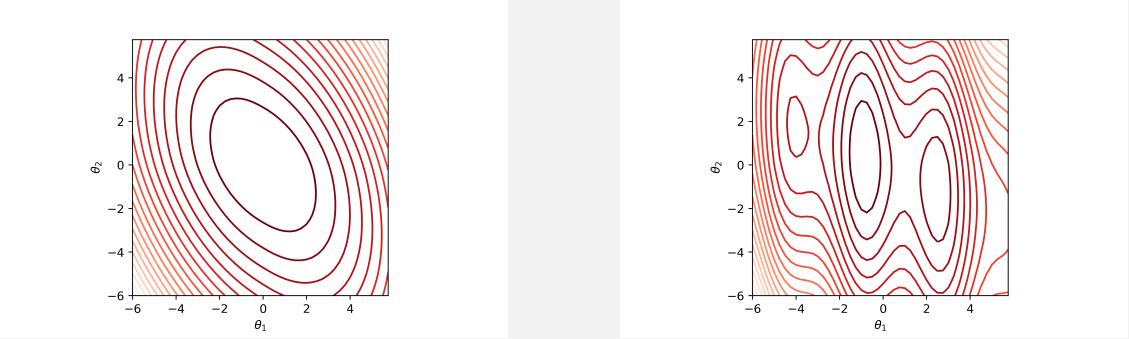
An important class of objective functions are *convex* functions. Optimization is often easier to carry out for convex objective functions, and it is a general advise to spend some extra effort to consider whether a non-convex optimization problem can be re-formulated into a convex problem (which sometimes, 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[4], 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 5.3 below.

Example 5.3: 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.



[4]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 (3.13)

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

As we have discussed, and also prove in Appendix 3.A, the solution (3.15) 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 (3.15), 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 (5.16)

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

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 (5.16). 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

$$\arg\min_{\theta_j} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1 = \text{sign}(t)(|t| - \lambda), \text{ where } t = \sum_{i=1}^{n} x_{ij}(y_i - \sum_{k\neq j} x_{ik}\theta_k). \tag{5.18}$$

It turns out that making repeated "sweeps" through the vector $\boldsymbol{\theta}$ and updating one-parameter-at-a-time according to (5.18) is a good way to solve (5.16). This type of algorithm, where we update one-parameter-at-a-time, is referred to as *coordinate descent*, and we illustrate it in Example 5.4

It can be shown that the cost function in (5.16) 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 (5.16) 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 (5.16). The keys are (i) that (5.18) 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 5.4: Coordinate descent

We apply coordinate descent to the objective functions from Example 5.3. For coordinate descent to be an efficient alternative in practice, closed-form solutions for updating one-parameter-at-a-time, similar to (5.18), 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 (5.16). 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 5.5.

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 4, where we use it for determining a regularization parameter λ.

Algorithm 4: 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

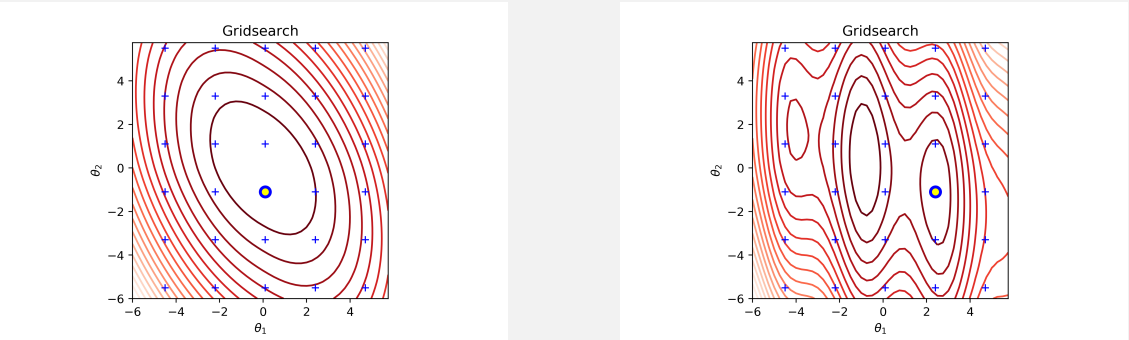
  Compute error on validation data $E_{\text{val}}(\lambda) \leftarrow \sum_{j=1}^{n_v} (\widehat{y}(\mathbf{x}_j; \widehat{\boldsymbol{\theta}}) - y_j)^2$

end

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

Example 5.5: Grid search

We apply grid search to the objective functions from Example 5.3, 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 4 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

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{5.19}$$

(even though gradient descent possibly can be used for hyperparameters as well). We will assume[5] 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 (3.36) is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{1+e^{y_i\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}_i}}\right) y_i\mathbf{x}_i. \tag{5.20}$$

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,

$$J\big(\boldsymbol{\theta} - \gamma\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\big) \leq J\big(\boldsymbol{\theta}\big) \tag{5.21}$$

for some (possibly very small) $\gamma > 0$. If $J(\boldsymbol{\theta})$ is convex, the inequality in (5.21) 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

$$\boxed{\text{update } \boldsymbol{\theta}^{(t+1)} \text{ as } \boldsymbol{\theta}^{(t)} - \gamma\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)}).} \tag{5.22}$$

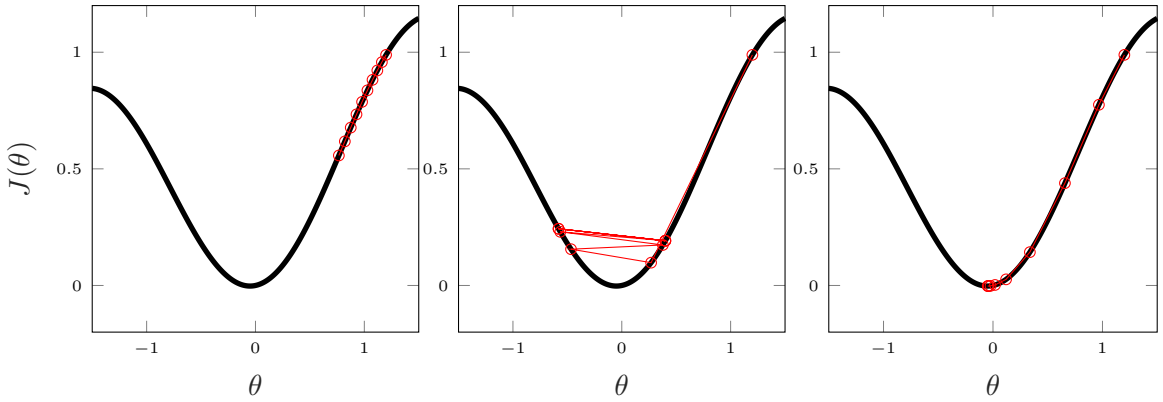
Repeating (5.22) gives the gradient descent Algorithm 5.

Algorithm 5: 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 $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

[5] 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 5 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. (5.21)) 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 5.6. 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.

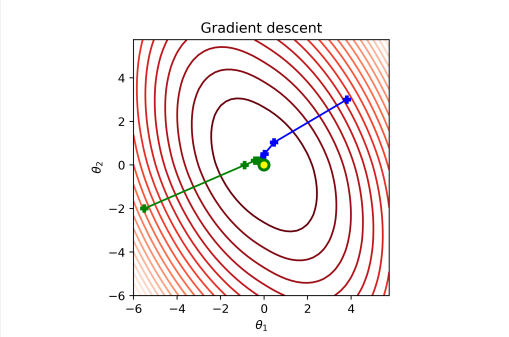
For convex problems there is only one stationary point, which also is the global minimum. Hence, the initialization for convex problem can be done arbitrarily. However, by *warm-starting* the optimization with a good initial guess, we may still save valuable computational time. Sometimes, such as when 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 5 with the parameters learned for the previous model.

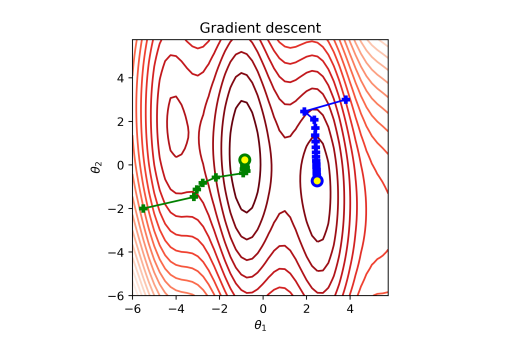
For learning logistic regression (3.37), gradient descent can be used. If we remember that its cost function is convex, we know that once gradient descent has converged to a minimum, it has reached the global minimum and we are done. However, for logistic regression, there are more advanced alternatives that usually performs even better.

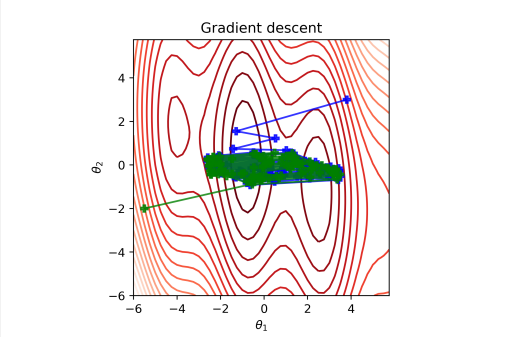
Example 5.6: Gradient descent

We first consider the convex objective function from Example 5.3, 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 5.3 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,

$$J(\boldsymbol{\theta}+\mathbf{v}) \approx \underbrace{J(\boldsymbol{\theta}) + \mathbf{v}^{\mathsf{T}}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})] + \frac{1}{2}\mathbf{v}^{\mathsf{T}}[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]\mathbf{v}}_{\triangleq s(\boldsymbol{\theta},\mathbf{v})}, \tag{5.23}$$

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,

$$\frac{\partial}{\partial \mathbf{v}} s(\boldsymbol{\theta}, \mathbf{v}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]\mathbf{v} = 0 \Leftrightarrow \mathbf{v} = -[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]. \tag{5.24}$$

This suggests to update

$$\boldsymbol{\theta}^{(t+1)} \text{ as } \boldsymbol{\theta}^{(t)} - [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})], \tag{5.25}$$

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 (5.23) is exact and Newton's method (5.25) will find the optimum in only one iteration! Quadratic objective functions are, however, rare in machine learning[6]. 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 (5.25). 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 (5.23) 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 (5.23) 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 (5.25) 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

$$\boxed{\text{update } \boldsymbol{\theta}^{(t+1)} \text{ as } \boldsymbol{\theta}^{(t)} - \eta[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})],} \tag{5.26}$$

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 6. 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 (5.25). 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.

[6] 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 $\boldsymbol{\theta}$.

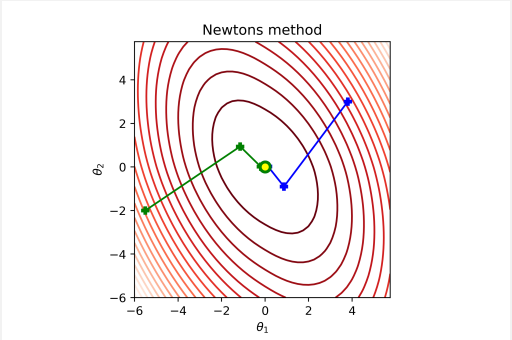
Algorithm 6: Truncated Newton's method

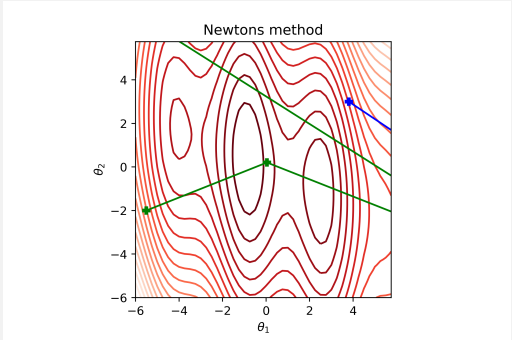
Input: Objective function $J(\boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, trust region radius D
Result: $\widehat{\boldsymbol{\theta}}$

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 $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

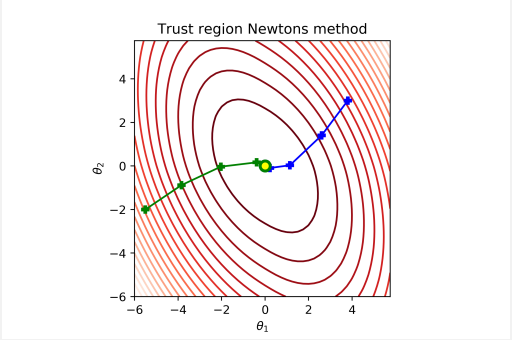
Example 5.7: Newton's method

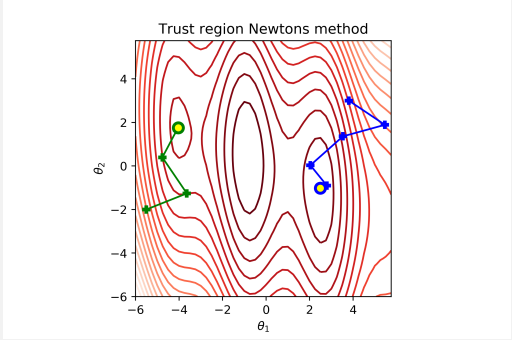
We first apply Newton's method to the cost functions from Example 5.3. Since the convex cost (left) function also happens to be close to a quadratic function, the Newton's method works well and finds, for both initializations, the minimum in only two iterations. For the non-convex problem (right), Newton's method diverges for both initializations, since the second order Taylor expansion (5.23) 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

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}) \tag{5.27}$$

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 5, and then compute the gradient for the new parameters based on the second half of the training data,

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \frac{1}{n/2} \sum_{i=1}^{\frac{n}{2}} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}^{(t)}), \tag{5.28a}$$

$$\boldsymbol{\theta}^{(t+2)} = \boldsymbol{\theta}^{(t+1)} - \gamma \frac{1}{n/2} \sum_{i=\frac{n}{2}+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}^{(t+1)}). \tag{5.28b}$$

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 7.

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 7: 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 ← 0
while Convergence criteria not met do
    for i = 1, 2, ..., E do
        Randomly shuffle the training data {x_i, y_i}_{i=1}^n
        for j = 1, 2, ..., n/n_b do
            Approximate the gradient using the mini-batch {(x_i, y_i)}_{i=(j-1)n_b+1}^{jn_b},
              d̂^(t) = (1/n_b) Σ_{i=(j-1)n_b+1}^{jn_b} ∇_θ L(x_i, y_i, θ^(t)).
            Update θ^(t+1) ← θ^(t) − γ^(t) d^(t)
            Update t ← t + 1
        end
    end
end
return θ̂ ← θ^(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 $\gamma^{(t)} = \frac{1}{t}$.

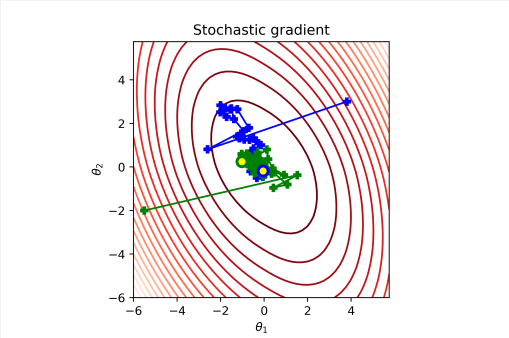
For many machine learning problems, however, it has been found that better performance in practice often is obtained if not letting $\gamma^{(t)} \to 0$, but to cap it at some small value $\gamma_{\text{min}} > 0$. This will cause stochastic gradient not to exactly converge, but the algorithm will in 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

$$\gamma^{(t)} = \gamma_{\text{min}} + (\gamma_{\text{max}} - \gamma_{\text{min}}) e^{-\frac{t}{\tau}}. \tag{5.29}$$

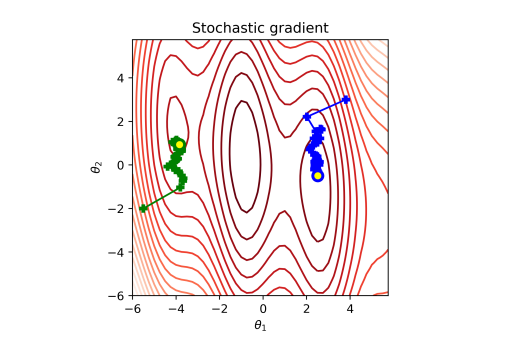
Now the learning rate $\gamma^{(t)}$ starts at γ_{max} and goes to γ_{min} as $t \to \infty$. How to pick the parameters γ_{min}, γ_{max} and τ is more of an art than science. As a rule of thumb γ_{min} can be chosen approximately as 1% of γ_{max}. The parameter τ depends on the size of the dataset and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

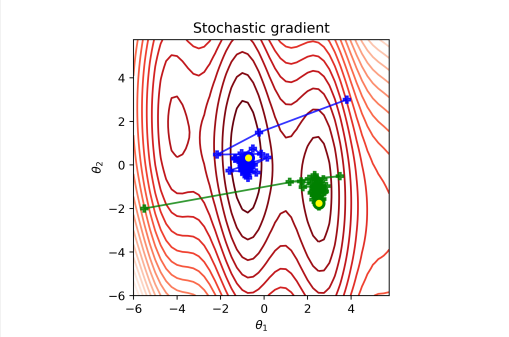
Example 5.8: Optimization

We apply the stochastic gradient method to the objective functions from Example 5.3. 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

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

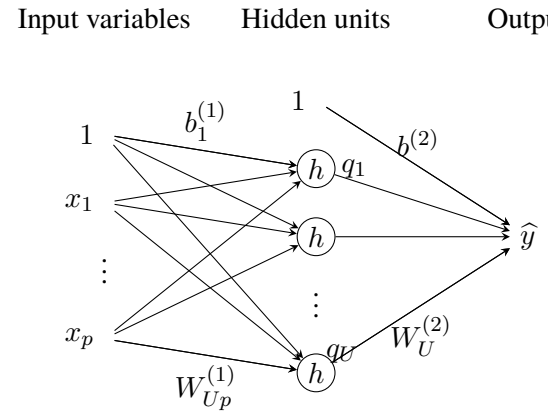


Figure 6.3: A two-layer neural network, or equivalently, a neural network with one intermediate layer of hidden units.

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

[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.

parameters in the model, which can be written as

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

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, $\ldots$ 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, $\ldots$ 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

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

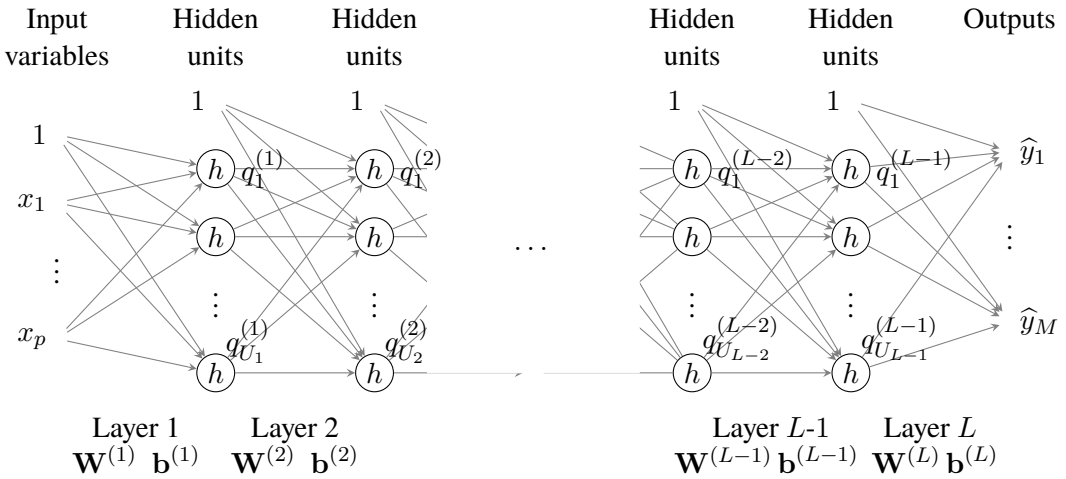


Figure 6.4: A deep neural network with L layers. Each layer l is parameterized by $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$.

The wider and deeper the network is, the more parameters there are. Practical deep neural networks can easily have in the order of millions of parameters and these models are therefore also extremely flexible. Hence, some mechanism to avoid overfitting is needed. Regularization such as ridge regression is common (cf. Section 5.2), but there are also other techniques specific to deep learning; see further Section 6.3. Furthermore, the more parameters there are, the more computational power is needed to train the model. As before, the training data $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ consists of n samples of the input $\mathbf{x}$ and the output $\mathbf{y}$.

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

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

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 (3.6) 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.

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 3.2, and more specifically the softmax parametrization given in (3.38), repeated here for convenience

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

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

$$\begin{aligned} \mathbf{q}^{(1)} &= h(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}), && \text{(6.18a)} \\ &\vdots \\ \mathbf{q}^{(L-1)} &= h(\mathbf{W}^{(L-1)}\mathbf{q}^{(L-2)} + \mathbf{b}^{(1)}), && \text{(6.18b)} \\ \mathbf{z} &= \mathbf{W}^{(L)}\mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}, && \text{(6.18c)} \\ \mathbf{g} &= \mathrm{softmax}(\mathbf{z}). && \text{(6.18d)} \end{aligned}$$

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 $p(y = m \,|\, \mathbf{x}_i)$, 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.

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 $\mathbf{y}_i = \begin{bmatrix} y_{i1} & \ldots & y_{iM} \end{bmatrix}^\mathsf{T}$. 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 3.2.

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. (3.41))

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

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

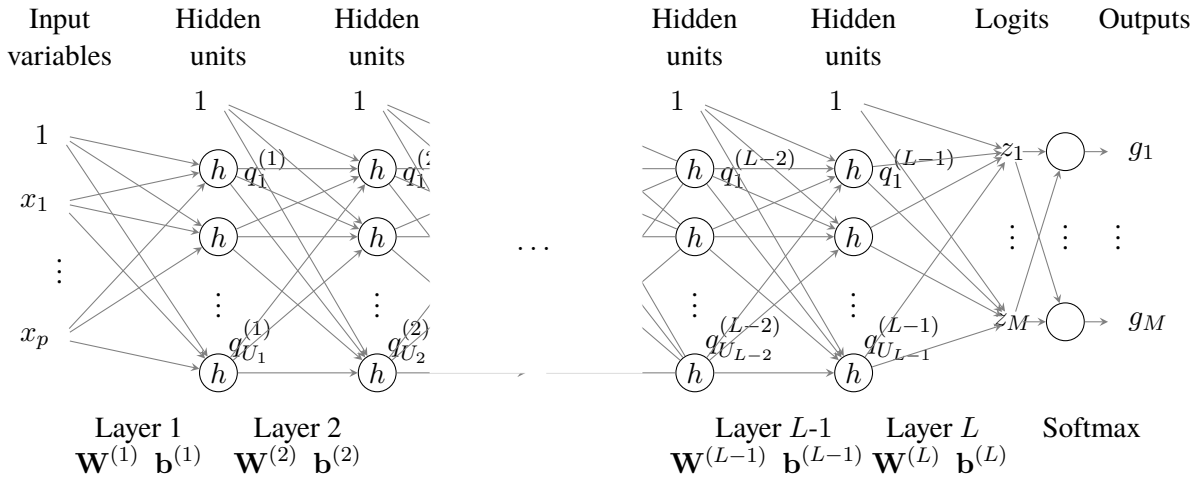


Figure 6.5: A deep neural network with L layers for classification. The only difference to regression (Figure 6.4) is the softmax transformation after layer L.

	$m=1$	$m=2$	$m=3$
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 $\mathbf{g}_i(\boldsymbol{\theta}_B) = [g_{i1}(\boldsymbol{\theta}_B), g_{i2}(\boldsymbol{\theta}_B), g_{i3}(\boldsymbol{\theta}_B)]$.

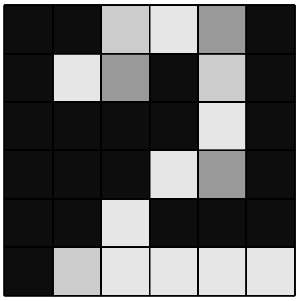
belongs to class $m = 2$ out of a total of $M = 3$ classes we have $\mathbf{y}_i = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\mathsf{T}$. Assume that we have a set of parameters for the network that we denote by $\boldsymbol{\theta}_A$, and with these parameters we predict $p(y = 1 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.1$, $p(y = 2 \,|\, \mathbf{x}_i; \boldsymbol{\theta}_A) = 0.8$ 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.

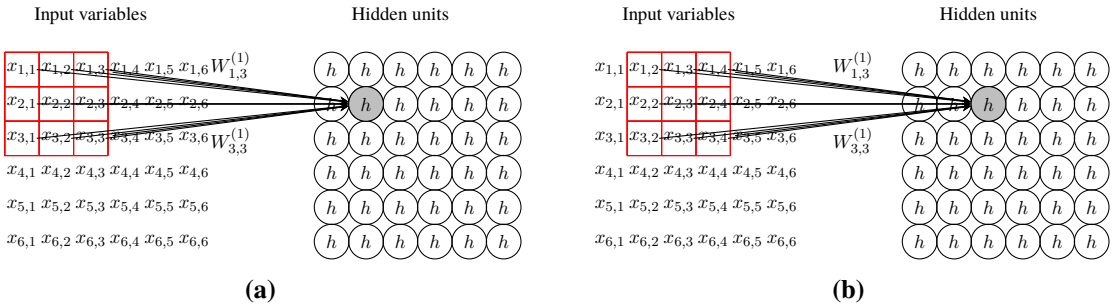


Figure 6.8: An illustration of the interactions in a convolutional layer: Each hidden unit (circle) is only dependent on the pixels in a small region of the image (red boxes), here of size 3×3 pixels. The location of the hidden unit corresponds to the location of the region in the image: if we move to 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 $W_{1,1}^{(1)}, W_{1,2}^{(1)}, \ldots, W_{3,3}^{(1)}$ are the *same* for all hidden units in the layer.

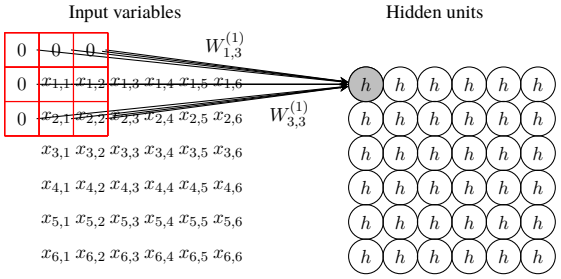


Figure 6.9: An illustration of zero-padding used when the region is partly outside the image. With zero-padding, the size of the image can be preserved in the following layer.

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

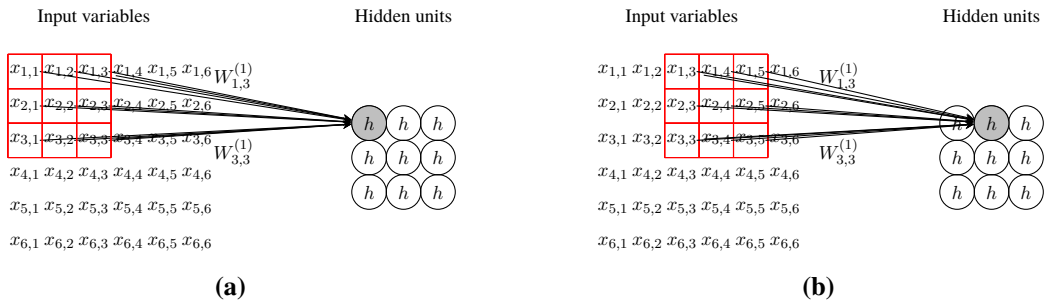


Figure 6.10: A convolutional layer with stride [2,2] and a filter of size 3×3.

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.

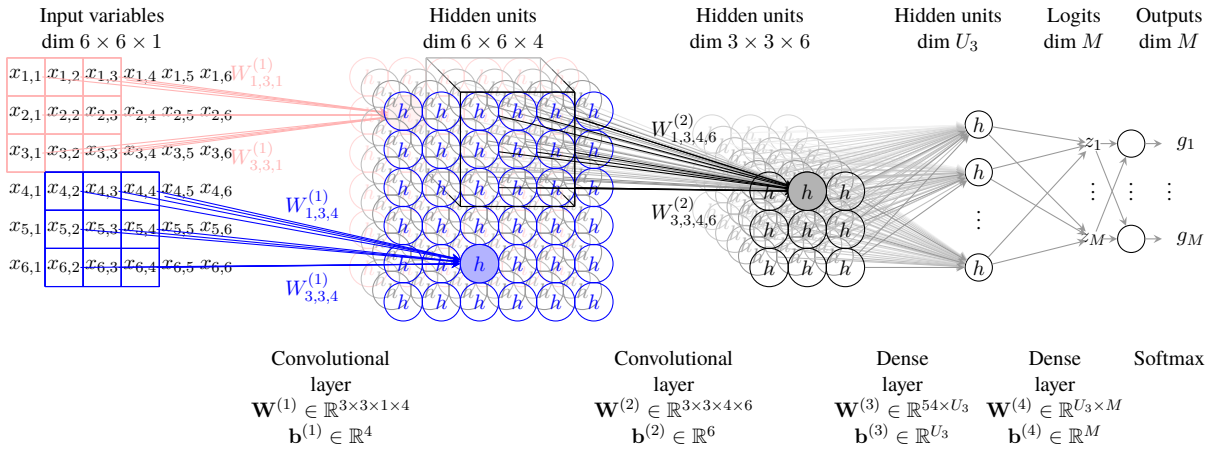


Figure 6.11: A full CNN architecture for classification of grayscale 6×6 images. In the first convolutional layer four filters each of size 3×3 produce a hidden layer with four channels. The first channel (in the back) is visualized in red and the forth channel (in the front) is visualized in blue. We 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

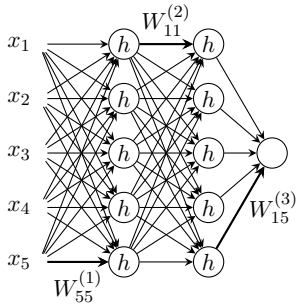
Like all models presented in this course, neural network models can suffer from overfitting if we have a too flexible model in relation to the complexity of the data. Bagging (Section 7.1) is one way to reduce the variance and by that also reducing the risk of overfitting. In bagging we train an entire *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.

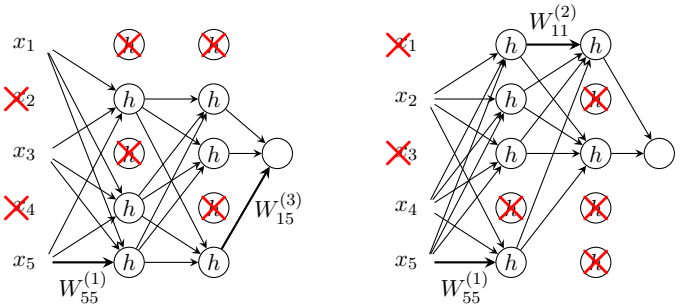
Ensemble of sub-networks

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 7. 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.

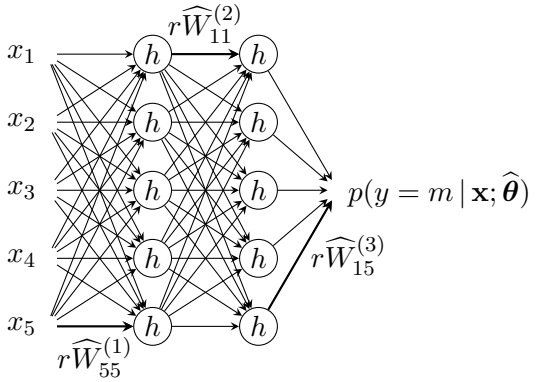


Figure 6.13: The network used for prediction after being trained with dropout. All units and links are present (no dropout) but the weights going out from a certain unit are multiplied with the probability of that unit being included during training. This is to compensate for the fact that some of them 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 5.2), 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.

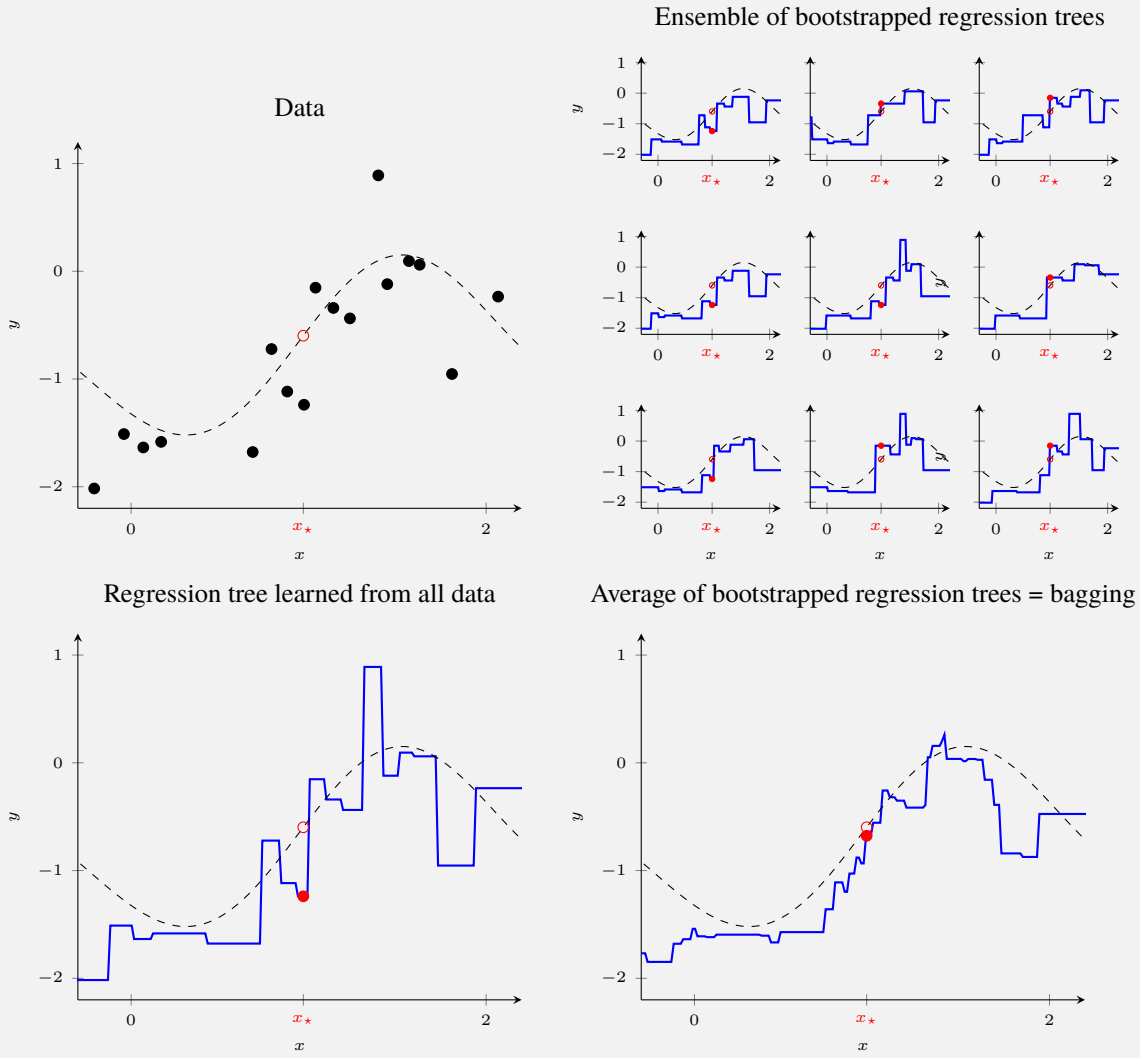
We outline the main idea of bagging with the example below.

[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 8 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.

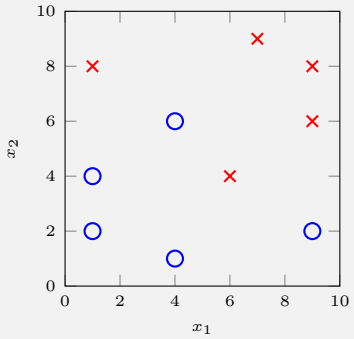
Time to reflect 7.1: *What would happen if the sampling was done without replacement in the bootstrap?*

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 $y \in \{\text{Blue, Red}\}$.

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

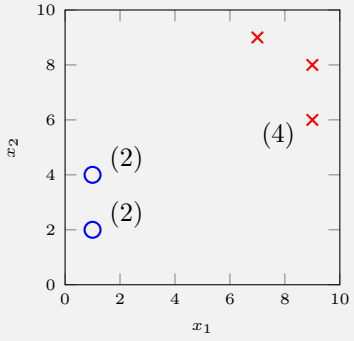
Index	x_1	x_2	y
1	9.0	2.0	Blue
2	1.0	4.0	Blue
3	4.0	6.0	Blue
4	4.0	1.0	Blue
5	1.0	2.0	Blue
6	1.0	8.0	Red
7	6.0	4.0	Red
8	7.0	9.0	Red
9	9.0	8.0	Red
10	9.0	6.0	Red



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 8: 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 9: 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 8 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 8) 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 $\widetilde{\mathbf{g}}^1(x_\star), \ldots, \widetilde{\mathbf{g}}^B(x_\star)$ denoted the predictions from the individual ensemble members. The averaged prediction, $\widehat{y}_{\text{bag}}(x_\star)$ or $\mathbf{g}_{\text{bag}}(x_\star)$, is the final prediction obtained from bagging. We summarize this by Algorithm 9. (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 $\text{Var}[z_b] = \sigma^2$ 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

$$\mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \mu, \tag{7.2a}$$

$$\text{Var}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2. \tag{7.2b}$$

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

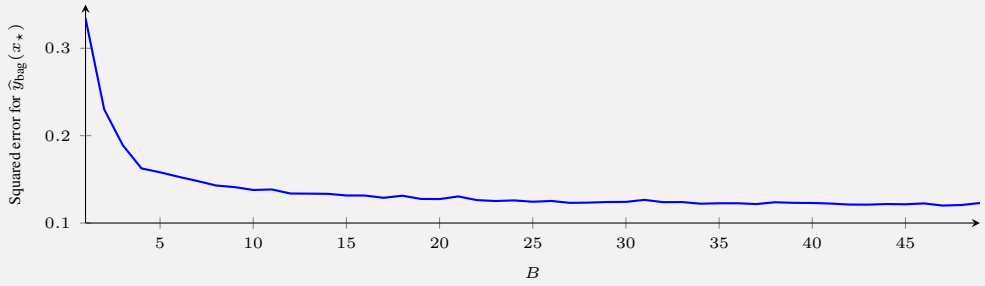
The first equation (7.2a) tells us that the mean is unaltered by averaging a number of identically distributed random variables. Furthermore, the second equation (7.2b) tells us that the variance is reduced by averaging if the correlation $\rho < 1$. The first term in the variance expression (7.2b) can 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 $\widetilde{y}^b(x_\star)$ 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 $\widetilde{y}^b(x_\star)$ 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 $\widehat{y}^{\text{bag}}(x_\star)$ 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.

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* $B \to \infty$.

[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.

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 the average only about 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.

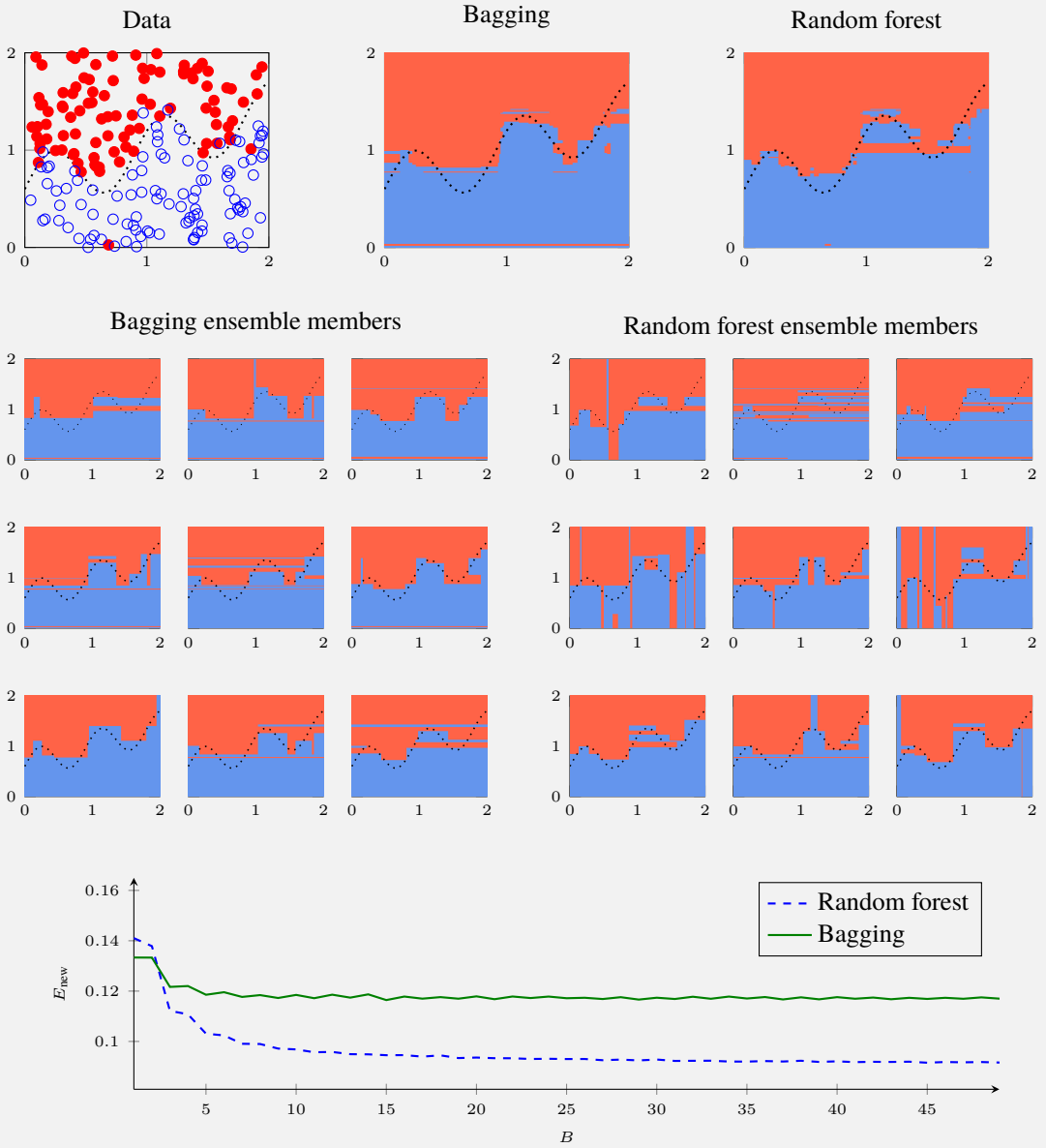
Let $\widetilde{\mathcal{T}}^{(b)}$ be one of the B bootstrapped datasets in bagging. To train a classification or regression tree on this data we proceed as usual (see Section 2.3), but with one difference. Throughout the training, whenever we are about to split a node we do not consider all possible 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 105.

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

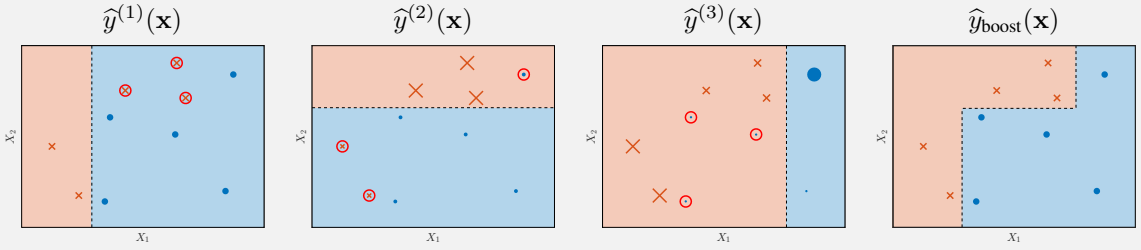
Whereas bagging primarily is an ensemble method for reducing variance in high-variance base models, boosting is rather an ensemble method for reducing bias in high-bias base models. A typical example of a simple (or, in other words, weak) high-bias model is a classification tree of depth one (sometimes 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

$$\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}) = \operatorname{sign}\left\{\sum_{b=1}^{B} \alpha^{(b)} \widehat{y}^{(b)}(\mathbf{x})\right\}. \tag{7.3}$$

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 (5.10c) 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 $\widehat{y}_{\text{boost}}^{(b)}(\mathbf{x}) = \operatorname{sign}\{c^{(b)}(\mathbf{x})\}$ 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 (5.10c) 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

$$E_{\text{train}}^{(b)} \overset{\text{def}}{=} \frac{W_e}{W} = \sum_{i=1}^{n} \frac{w_i^{(b)}}{\sum_{j=1}^{n} w_j^{(b)}} \mathbb{I}\{y_i \neq \widehat{y}^{(b)}(\mathbf{x}_i)\} \tag{7.10}$$

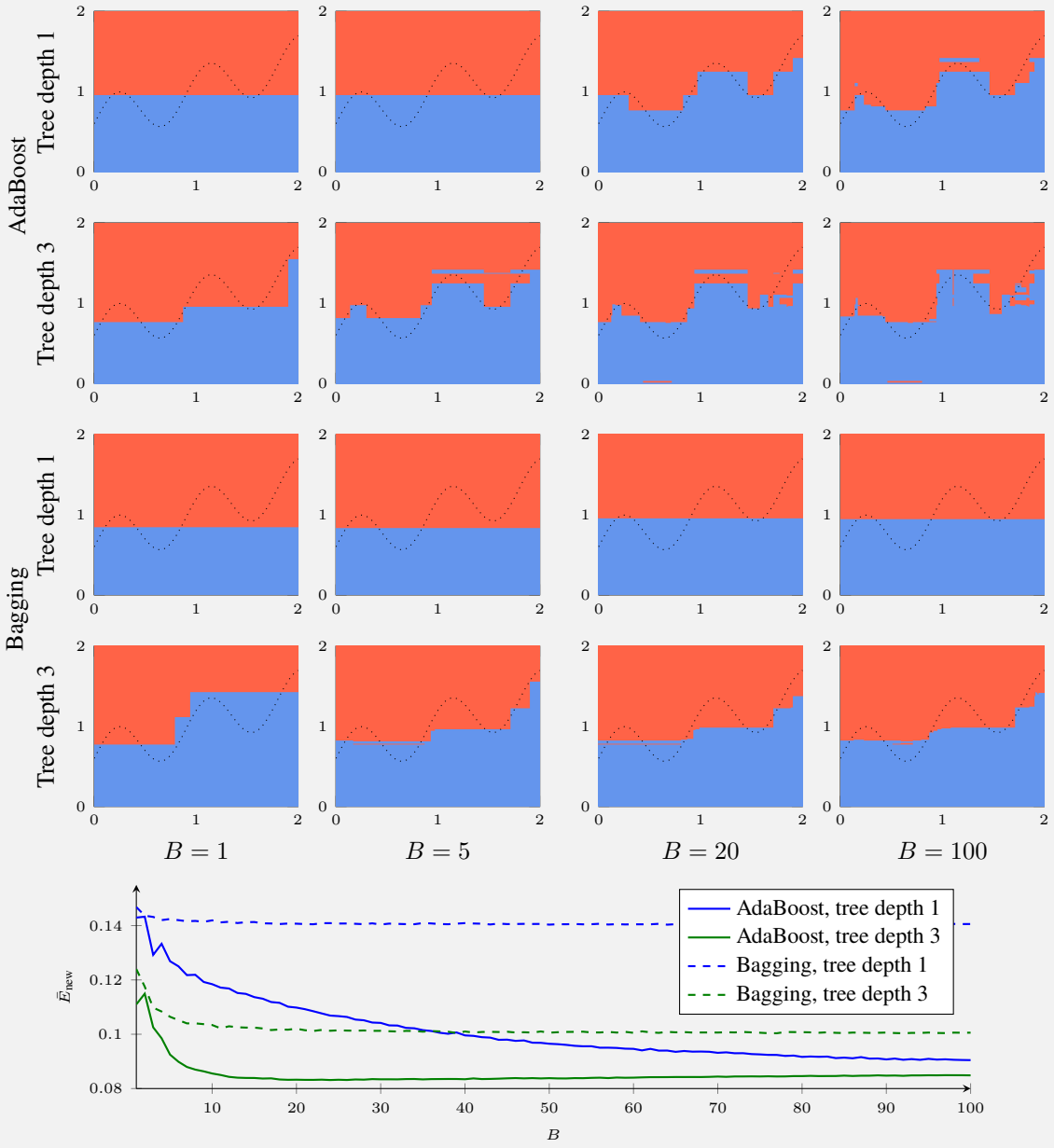
to be the weighted misclassification error for the b^{th} classifier, we can express the optimal value for its coefficient as

$$\alpha^{(b)} = \frac{1}{2} \ln\left(\frac{1 - E_{\text{train}}^{(b)}}{E_{\text{train}}^{(b)}}\right). \tag{7.11}$$

This completes the derivation of the AdaBoost algorithm, which is summarized in Algorithm 10. 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 10: 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 $p(y = 1 \,|\, \mathbf{x})$. 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 10 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 in Chapter 5. 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 replacing the exponential loss in (7.5a), the problem does not separate into weighted classification problems anymore, but the situation becomes more intricate.

It turns out to be possible to approximately solve (7.5a) for rather general loss function using a method reminiscent of gradient descent ((5.22) 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 ((5.22) 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 specific 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 $\frac{\partial L(y_i \cdot c^{(b-1)}(\mathbf{x}_i))}{\partial c}$ $(i = 1, \ldots, n)$. (We have assumed that the loss function L is differentiable, meaning that we can always compute the derivative needed.) 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 11.

Algorithm 11: 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 11, 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

Chapter to be written.

9 The Bayesian approach and Gaussian processes

Chapter to be written.

10 User aspects of machine learning

Dealing with supervised machine learning problem in practice is to a great extent an engineering discipline where many practical issues have to be considered and the scare resource is in the end man-hours. To use this resource efficiently one needs to have a well-structured procedure for how to develop and improve your model. Multiple actions can potentially be taken to improve your model. How do you know which action to take and is it worth spending the time implementing them? Is it for example worth spending an extra week collecting and labeling more training data or should one do something else? These issues we will try to address 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 be able to do this efficiently, we need to be able to tell whether a new model is an improvement over the previous model. One way to evaluate the model after each iteration would be to but it into production (for example running your traffic-sign classifier in your self-driving the car for a few hours). Besides the obvious security 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.

A better strategy is to automate this evaluation procedure without the need to put the model into production each time you 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 final performance of the model.

If the amount of data available 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*. In validation

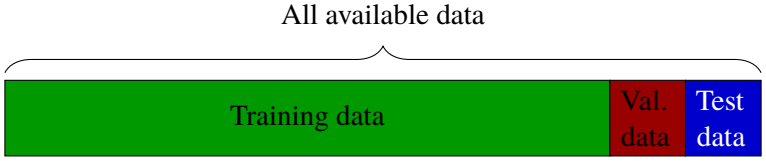


Figure 10.1: Splitting your data intro training data, hold-out validation data and test data.

we can also choose to train not only one new model but many models. For example, if we have a neural network model and are interested in the number of hidden units in a certain layer we can train not only one model but many models, each with a different choice of hidden units, and pick the one that performance the 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 between many models. Depending on the size of the validation data we might risk picking 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 come from the same distribution, we are validating and improving our model towards something which 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 you set aside as hold-out validation data and test data, or should you even avoid setting aside hold-out validation data and use k-fold cross-validation instead? This depends on how much data you have available, which performance difference you plan to detect, and how many models you plan to compare between. For example, if you 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 you plan to compare many (say, hundreds or more) different hyperparameter values and model structures using 100 validation data points, you will most likely overfit to that validation data.

If you have, say, 500 data points, a reasonable split would perhaps be 60%-20%-20% (i.e. 300-100-100 samples) for training-validation-test. With such small validation data you cannot afford comparing between a large number of hyperparameter values and model structures or detecting an improvement of 0.1%. In this situation, you are probably better of using k-fold cross-validation, to decrease the risk of overfitting to validation data. Be aware, however, that the risk of overfitting to the data still exists even with k-fold cross-validation. You also still need to set aside test data if you want a final unbiased estimate of the performance.

Many machine learning problems have substantially larger datasets. Assume you 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 you 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 introduce additional metrics, besides the misclassification rate such as precision, recall and F1-score, for evaluating binary classifiers. There is no unique answer which metric that is the most appropriate. Which metric to pick is rather a part of the problem definition and there is no unique answer to which metric is the best. However, to choose a single number evaluation metric is important since that

defines your problem, and without a proper problem definition it is difficult to solve the problem, and to improve that solution. If you later realize that your metric does not favor the properties you want a good model to have, you 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 you to speed up your iterations since you 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 accept or reject 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 you 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 you 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 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 you do next? Two possible strategies for guiding you to meaningful actions to improving your 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, 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}. (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.)

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 $E_{\text{hold-out}} - E_{\text{train}}$ (or $E_{k\text{-fold}} - E_{\text{train}}$). By computing these quantities and have a look at them, we can actually get some good guidance for what changes that we may consider for the next iteration.

As we discussed in Chapter 4, if your training error is small and your generalization gap is big (E_{train} small, $E_{\text{hold-out}}$ big), you have typically overfitted your model. The opposite situation, big training error and small generalization gap (both E_{train} and $E_{\text{hold-out}}$ big), typically indicates underfitting.

If you want to reduce the generalization gap $E_{\text{hold-out}} - E_{\text{train}}$ (reduce overfitting) you can explore the following actions:

- Use a less flexible model. If we have a very flexible model we can 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 you 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 you want to reduce the training error E_{train} (reduce underfitting) you can consider the following actions:

- Use a more flexible model that is able to fit your training data better. This can be to change a hyperparameter in the model you are considering, for example decreasing k in k-NN or change the model to a more flexible one, for example change the linear regression model to multi-layer neural network.
- Extend the set of input variables. If you suspect that there are more input variables that carry information, you might want to extend you data with these input variables.
- Use less regularization in your model, if you use regularization.
- Train your model longer (only for models that are trained iteratively and aborted before reaching the minimum, like a neural network).

We summarize this by Figure 10.2. Luckily, evaluating E_{train} and $E_{\text{hold-out}}$ is typically cheap. Yet, it gives you good advice on what actions to take next. Besides suggesting what action to explore next, this procedure also tells you what *not* to do: If your training error E_{train} is large and your generalization gap small ($E_{\text{hold-out}}$ also large), collecting more training data will most likely not help. Further, if you have a large generalization gap and 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 your model is to do *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 you manually look at a subset (for example 100 data points) of the data points in the validation dataset that your model classified incorrectly. Such an analysis does not take much time, but might give you valuable clues on what type of data that your model is struggling with and how much improvement you can expect if you fix these issues. We illustrate the procedure with an example.

Example 10.1: 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 your model has a classification accuracy of 90% on validation data.

When looking at subset of 100 images that were misclassified in the validation data, you make the following observations:

- All 10 images of class `pedestrian` that were incorrectly classified as `bike` contained a baby carrier.

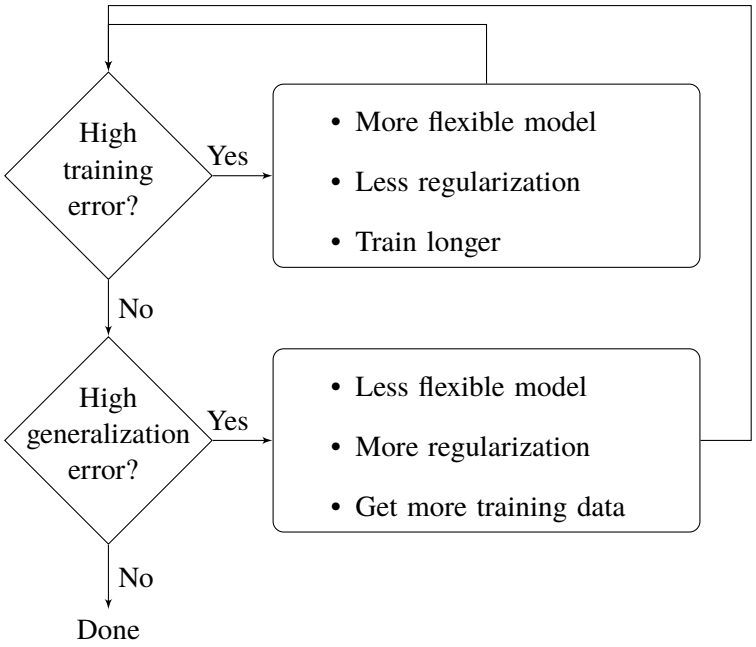


Figure 10.2: The iterative procedure of improving you model based on the training error–generalization gap decomposition.

- 30 images were substantially tilted.
- 15 images were mislabeled.

From this observation you can conclude:

- If you launch a project for improving the model to classify pedestrians with baby carriers as `pedestrian` and not incorrectly as `bike`, you can expect an improvement of at most a $\sim 1\%$ (a tenth of the 10% classification error rate).
- If you improve the performance on tilted images you might expect an improvement of at most $\sim 3\%$.
- If you correct all mislabeled data you can expect an improvement of at most $\sim 1.5\%$.

Following the example, you get an indication on what improvement you can expect by tackling these three issue. These numbers should be considered as the maximal possible improvement. To prioritize which aspect to focus on you should also consider what strategies you have available for improving them, how much progress you expect to make applying these strategies and how much effort you have to put in fix these issues.

For example, to improve the performance on tilted images you could try to extend your training data by augmenting it with tilted more 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, a likely approach to take is 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 increase 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 you may say it is not quite worth the effort of getting an improvement of 1.5%. However, assume that you have improved the model with other actions to an accuracy of 98.0% accuracy 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 you 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 only through the data points in the validation/test data that were misclassified by your 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 data points that both your algorithm and the original label made the same mistake on and these would not be corrected, resulting in a too optimistic estimate of the performance on validation and test data. Therefore, is it 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 that it is less labor intensive. If you have a model with a 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 do give you 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 you 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.

Nevertheless, 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. This can of course be a non-negligible error source. Look further in the following section how to estimate the size this training-validation/test data mismatch error source. If you 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 you 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

There are situations where you for different reasons can accept your training data to come from a slightly different distribution than your 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 you might have access to another substantially larger data set which comes from a slightly different distribution than the data you care about, but similar enough that you suspect the advantage of having a larger training data overcomes the disadvantage of that data mismatch. See further in Section 10.3 about extending your training data with data from a different source.

Note that it is not an end in itself letting the training data come from a different distribution than the validation and test data. In contrast, as pointed out already in Chapter 4, you should strive for letting your training data come from the same distribution as validation and test data, for example by doing the data split randomly. However, in some cases the (possibly minor) disadvantage of that data mismatch might be acceptable in comparison to other possible advantages this might bring.

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

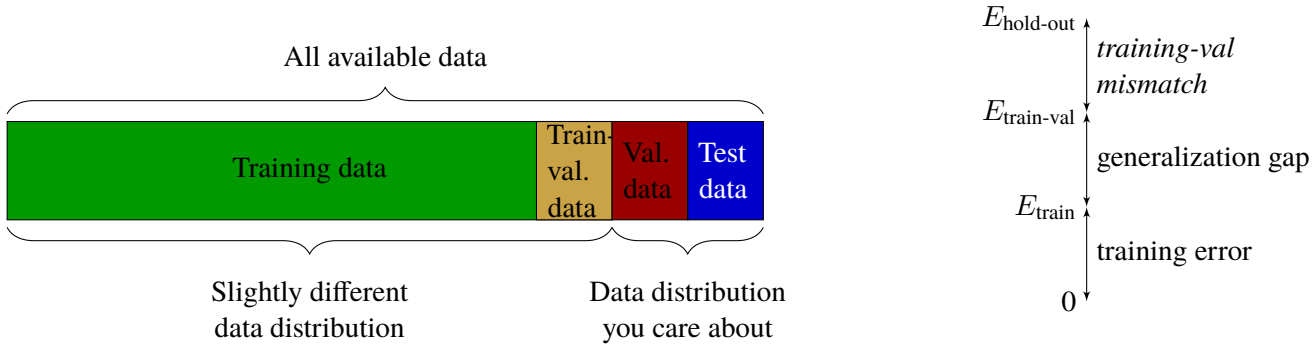


Figure 10.3: Revising the training-validation-test data split by carving out a separate train-validation data from the training data.

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.

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} and are the performance 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 in Section 10.2. 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 doing it for the training data as well.

10.3 What if you cannot collect more data?

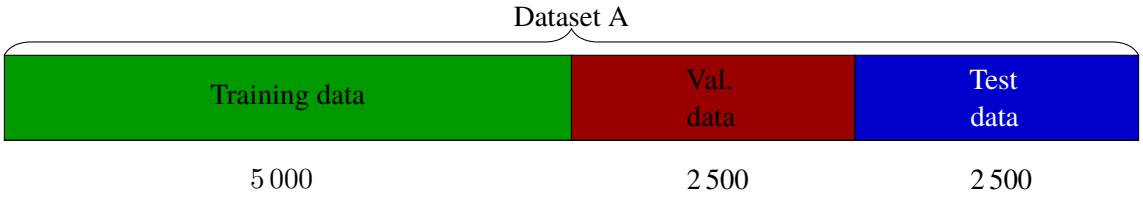
We have seen in Section 10.2 that collecting more data is a good strategy to reduce the generalization gap and hence reduce overfitting. However, collecting labeled data is usually expensive and sometimes not even possible. What can you do if you cannot afford to collect more data but still want to benefit from the advantages that a larger dataset would give? In this section a few approaches are presented.

Extending your training data with slightly different data

As already mentioned in 10.2, there are situations where you can accept your training data to come from a slightly different distribution than your validation and test data. One reason to accept this is if you then would have access to a substantially larger training data set.

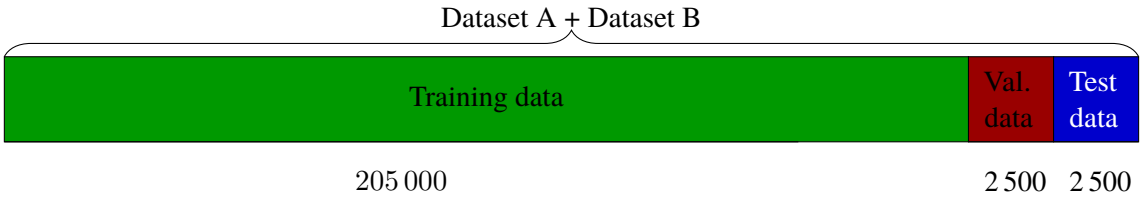
Consider a problem where you have 10000 data points representing the data that you also would expect to get when you set your model into production. We call this dataset A. You also have another dataset with 200 000 data points which come from a slightly different distribution, but that is similar enough that you think expoiting information from that data can improve your model. We call this dataset B. Some options to proceed would be the following:

- **Option 1** Use only dataset A and split it up between training, validation and test data.



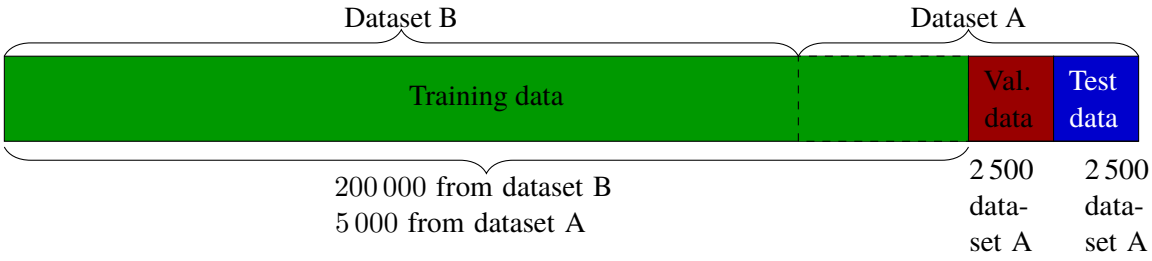
The advantage of this option is that we only train, validate and evaluate on dataset A which is also the type of data that we want our model to perform well on. The disadvantage is that we have quite few data points and we do not exploit information in dataset B.

- **Option 2** Use both dataset A and dataset B. Randomly shuffle your data and split it up between training, validation and test data.



The advantage over option 1 is that we have a lot more 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 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. This makes sure that you use all data you have available but evaluate on the data that you want to do well on (dataset A) and still exploit the information available in the much larger dataset B. The main disadvantage with option 3 is that your training data no longer come from the same distribution as your validation data and test data. In order to quantify how big impact this mismatch has on your final performance, you can use the techniques described in Section 10.2. To push your model to do better on data from dataset A during training you can also consider giving data from dataset A higher weight in the cost function than data from dataset B.

Artificially extending your training data set

Another approach to extend your dataset without the need to collect more data for your task is to use *data augmentation*. In data augmentation you 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 you vertically flip and image of a

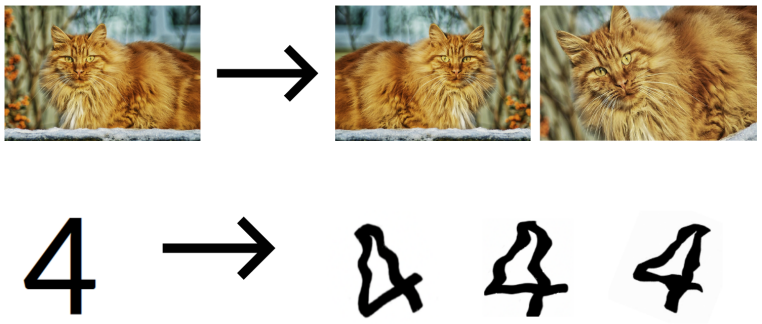


Figure 10.4: Different examples of data augmentation applied to images. Above: An image of a cat has been reproduced by tilting and vertical flipping. Below: An image if a digit has been reproduced by tilting and blurring.

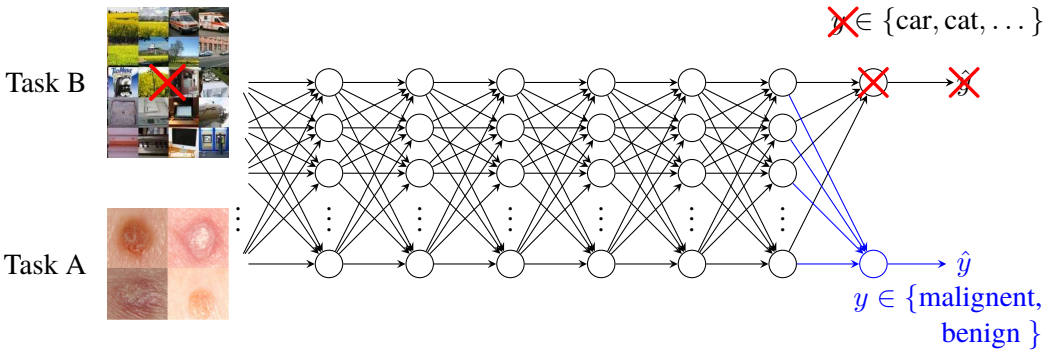


Figure 10.5: In transfer learning we reuse models that have been trained on a different task than the one we are interested in. Here we reuse a model which has been trained on images displaying all sorts of classes, such as cars, cats, computers and later train only the last few layers on the skin cancer data which is the task we are interested in.

cat, it still displays a cat, see Figure 10.4. See further in Shorten and Khoshgoftaar 2019 of a review of different data augmenting techniques for images.

Transfer learning

Yet another technique to effectively use more data than the dataset you have is to use transfer learning. In transfer learning you use the knowledge from a model that has been trained on a different task with a different dataset and then apply that knowledge to different but slightly related problem.

Transfer learning is especially common for sequential model structures such as 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 we 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 you had trained the whole model on only dataset A. The procedure is also displayed in Figure 10.5. The intuition is that the layers closer to the input accomplish task which is 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.

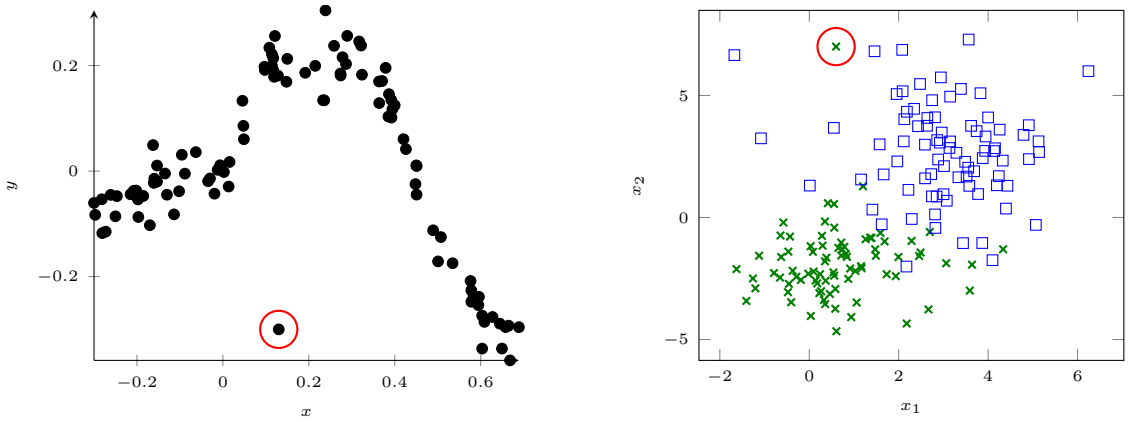


Figure 10.6: Two typical pictures of outliers (marked with red circle) in regression (left) and classification (right), respectively.

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 pictures 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. 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 your 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 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 model amounts, to some extent, to make 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 predicted, they are not really any problem, but we just have to use a model that is flexible enough to capture the behavior (small bias). This has, however, 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 is discussed more in 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,[1] 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 data set would no longer 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 feature ("column in $\mathbf{X}$") where data is missing. It depends on the situation whether this is a fruitful approach.

Example 10.2: Missing data

$$\mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ \texttt{NaN} \\ -1 \\ -1 \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} -0.3 & 0.2 & 0.2 & \texttt{NaN} & -1.1 \\ -0.4 & 0.5 & -0.8 & 3.2 & -1.3 \\ 2.3 & -1.2 & 0.5 & -0.2 & -1.9 \\ -3.1 & 1.3 & -.9 & 1.1 & -1.3 \\ -0.9 & 1.6 & 0.5 & \texttt{NaN} & 1.4 \\ \texttt{NaN} & 1.1 & 0.2 & \texttt{NaN} & -2.5 \\ -0.7 & 0.3 & 0.4 & \texttt{NaN} & 1.3 \end{bmatrix}$$

10.5 Further reading

User aspects of machine learning is a fairly underexplored area both in academic research publications and in standard text books on machine learning. One exception is Ng 2019 from which parts of this chapter has been inspired.

[1] 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 another model design paradigm, namely so-called *generative* models. The scope of generative modeling is wider than for discriminative models, and predictions are obtained almost as a by-product from a generative model. Generative modeling is also a natural way to go beyond the plain supervised problem, and also introduce semi-supervised and unsupervised learning.

A generative model aims to describe the distribution $p(\mathbf{x}, y)$, that is, how the data (both its input and output) is generated. For supervised machine learning predictions (that is, $p(y \mid \mathbf{x})$) can be derived from the generative model, using the laws of probability distributions. We will make this idea concrete by considering a rather simple, yet useful, generative model which gives rise to two classifiers named linear and quadratic discriminant analysis (LDA and QDA, respectively). We will thereafter see how generative modeling can take us outside the world of supervised learning by using the same generative model as in QDA for another purpose, namely mixture modeling.

11.1 Linear and quadratic discriminant analysis

We will now introduce two classifiers which can be derived in the generative framework, namely linear and quadratic discriminant analysis (LDA[1] and QDA, respectively). LDA and QDA builds on a generative model, that is a model for $p(\mathbf{x}, y)$. It is assumed that $\mathbf{x}$ is a numerical variable, and y a categorical variable. In more detail, LDA and QDA 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

$$p(\mathbf{x} \mid y) = \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{11.1b}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ may depend on y. Since we are considering classification with y taking values $1, \ldots, M$, the distribution $p(y)$ is naturally modeled with M parameters $\{\pi_m\}_{m=1}^M$ as

$$p(y = 1) = \pi_1, \tag{11.1c}$$

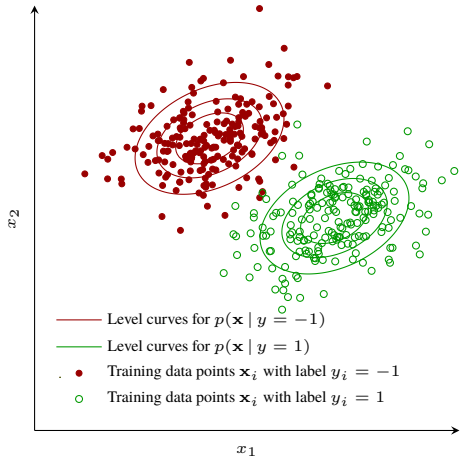
$$\vdots$$

$$p(y = M) = \pi_M. \tag{11.1d}$$

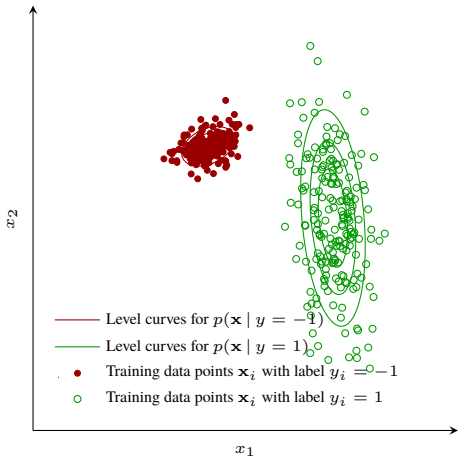
Altogether, (11.1) is a generative model which describes a user assumption on how data $\{\mathbf{x}, y\}$ is generated. As it turns out, from the model (11.1) a classifier follows which is easy to learn (requires no numerical optimization, in contrast to logistic regression), and is useful in practice also when the data do not obey the assumption (11.1) (in particular (11.1b)) perfectly.

It still remains to specify how the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ depends on y in (11.1b). We will assume that $\boldsymbol{\mu}$ is different for each class y, whereas we consider two alternatives for $\boldsymbol{\Sigma}$. We first

[1] Note to be confused with Latent Dirichlet Allocation, also abbreviated LDA, which is a completely different method.



(a) The generative model behind LDA assumes that the input $\mathbf{x}$, for a given output y, is distributed as a Gaussian distribution. The mean is different for each class y, but the covariance is the same for all classes. This figure shows what the data looks like, according to the assumptions of the generative model.



(b) Also in the generative model behind QDA it is assumed that the input $\mathbf{x}$, for a given output y, is distributed as a Gaussian distribution. However, both the mean and the covariance is different for each class. This figure shows what the data looks like, according to the assumptions of the generative model.

Figure 11.1: The classifiers LDA and QDA are derived from a generative model, which assumes that $p(\mathbf{x} \mid y)$ has a Gaussian distribution. This means that we think about the input variables as random and *assume* that they have a certain distribution. In the model for LDA (left panel, a), we assume that the covariance of the input distribution (shape of the level curves) is the same for all classes, and they only differ in locations. In the model for QDA (right panel, b) we assume that the covariance can be different for different classes. In fact, when using LDA and QDA in practice, these assumptions on how the inputs $\mathbf{x}$ are distributed do not have to be satisfied, but this is nevertheless the assumptions made by their underlying generative models.

assume that $\boldsymbol{\Sigma}$ is the same for all classes y, which will lead to the LDA classifier. We thereafter generalize and assume that $\boldsymbol{\Sigma}$ is different for each class, leading to the QDA classifier. In equations,

$$p(\mathbf{x} \mid y = m) = \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_m, \boldsymbol{\Sigma}) \text{ (LDA)} \tag{11.2a}$$
$$p(\mathbf{x} \mid y = m) = \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m) \text{ (QDA).} \tag{11.2b}$$

We summarize these generative models by Figure 11.1.

Learning the parameters

Also generative models are learned from training data. For the model (11.1), the parameters π_m, $\boldsymbol{\mu}_m$ and $\boldsymbol{\Sigma}$ (for (11.2a), the model behind LDA) or $\boldsymbol{\Sigma}_m$ (for (11.2b), the model behind QDA) have, for each $m = 1, \ldots, M$, to be learned from the training data. The perhaps most straightforward parameter to learn is $\widehat{\pi}_m$, the relative occurrence of class m in the training data,

$$\widehat{\pi}_m = \frac{n_m}{n}, \tag{11.3a}$$

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

$$\widehat{\boldsymbol{\mu}}_m = \frac{1}{n_m} \sum_{i:y_i=m} \mathbf{x}_i, \tag{11.3b}$$

the empirical mean among all training samples of class m. For (11.2a), the common covariance matrix $\boldsymbol{\Sigma}$ for all classes is usually learned as

$$\widehat{\boldsymbol{\Sigma}} = \frac{1}{n-M}\sum_{m=1}^{M}\sum_{i:y_i=m}(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^{\mathsf{T}}. \tag{11.3c}$$

For (11.2b), one covariance matrix $\boldsymbol{\Sigma}_m$ has to be learned for each class $m = 1, \ldots, M$, usually as

$$\widehat{\boldsymbol{\Sigma}}_m = \frac{1}{n_m - 1}\sum_{i:y_i=m}(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^{\mathsf{T}}. \tag{11.3d}$$

Obtaining predictions

We have so far described two generative models $p(\mathbf{x}, y)$ where $\mathbf{x}$ is numerical and y categorical. We also know how to learn the unknown parameters in $p(\mathbf{x}, y)$ from training data, but it remains to discuss how they can be used for supervised machine learning by making predictions.

The key insight for using a generative model $p(\mathbf{x}, y)$ for prediction, is to realize that prediction "only" amounts to compute $p(y \,|\, \mathbf{x})$. From probability theory, we have

$$p(y \,|\, \mathbf{x}) = \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} = \frac{p(\mathbf{x}, y)}{\sum_{m=1}^{M} p(\mathbf{x}, m)}. \tag{11.4}$$

For predictions, the left hand side $p(y \,|\, \mathbf{x})$ is what we are interested in, whereas the right hand side is given by the generative model. Thus, with (11.1) and (11.2a), we get $p(y \,|\, \mathbf{x})$ as

$$p(y = m \,|\, \mathbf{x}_\star) = \frac{\widehat{\pi}_m \mathcal{N}\left(\mathbf{x}_\star \,|\, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}\right)}{\sum_{j=1}^{M} \widehat{\pi}_j \mathcal{N}\left(\mathbf{x}_\star \,|\, \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}\right)}, \tag{11.5}$$

for $m = 1, 2, \ldots, M$. The classifier obtained from (11.5) is commonly called linear discriminant analysis (LDA). In full analogy we obtain quadratic discriminant analysis (QDA) as

$$p(y = m \,|\, \mathbf{x}_\star) = \frac{\widehat{\pi}_m \mathcal{N}\left(\mathbf{x}_\star \,|\, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\right)}{\sum_{j=1}^{M} \widehat{\pi}_j \mathcal{N}\left(\mathbf{x}_\star \,|\, \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}_j\right)}, \tag{11.6}$$

the only difference is the covariance matrix $\widehat{\boldsymbol{\Sigma}}_m$. As usual, we can obtain "hard" predictions $\widehat{y}_\star$ by selecting the class which is predicted to be the most probable as

$$\widehat{y}_\star = \arg\max_m p(y = m \,|\, \mathbf{x}_\star), \tag{11.7}$$

and compute corresponding decision boundaries. We summarize this by Algorithm 12 and 13, and illustrate by Figure 11.2 and 11.3. It turns out that for binary classification problems, the decision boundary for LDA is always a linear function, whereas the decision boundary for QDA is always a quadratic function (hence their names), as shown in Figure 11.4.

Remark 11.1 *In the derivation of the generative model behind LDA and QDA, it was assumed that $p(\mathbf{x} \,|\, y)$ 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?*

Algorithm 12: Linear Discriminant Analysis, LDA

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) and $\widehat{\boldsymbol{\mu}}_m$ (11.3b)
end
Compute $\widehat{\boldsymbol{\Sigma}}$ (11.3c)

Predict

for $k = 1, \ldots, M$ **do**
  Compute $p(y = m \,|\, \mathbf{x}_\star)$ (11.5)
end
Find largest $p(y = m \,|\, \mathbf{x}_\star)$ and set $\widehat{y}_\star$ to that k

Algorithm 13: 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 $\widehat{\boldsymbol{\Sigma}}_m$ (11.3d)
end

Predict

for $k = 1, \ldots, M$ **do**
  Compute $p(y = m \,|\, \mathbf{x}_\star)$ (11.6)
end
Find largest $p(y = m \,|\, \mathbf{x}_\star)$ and set $\widehat{y}_\star$ to that k

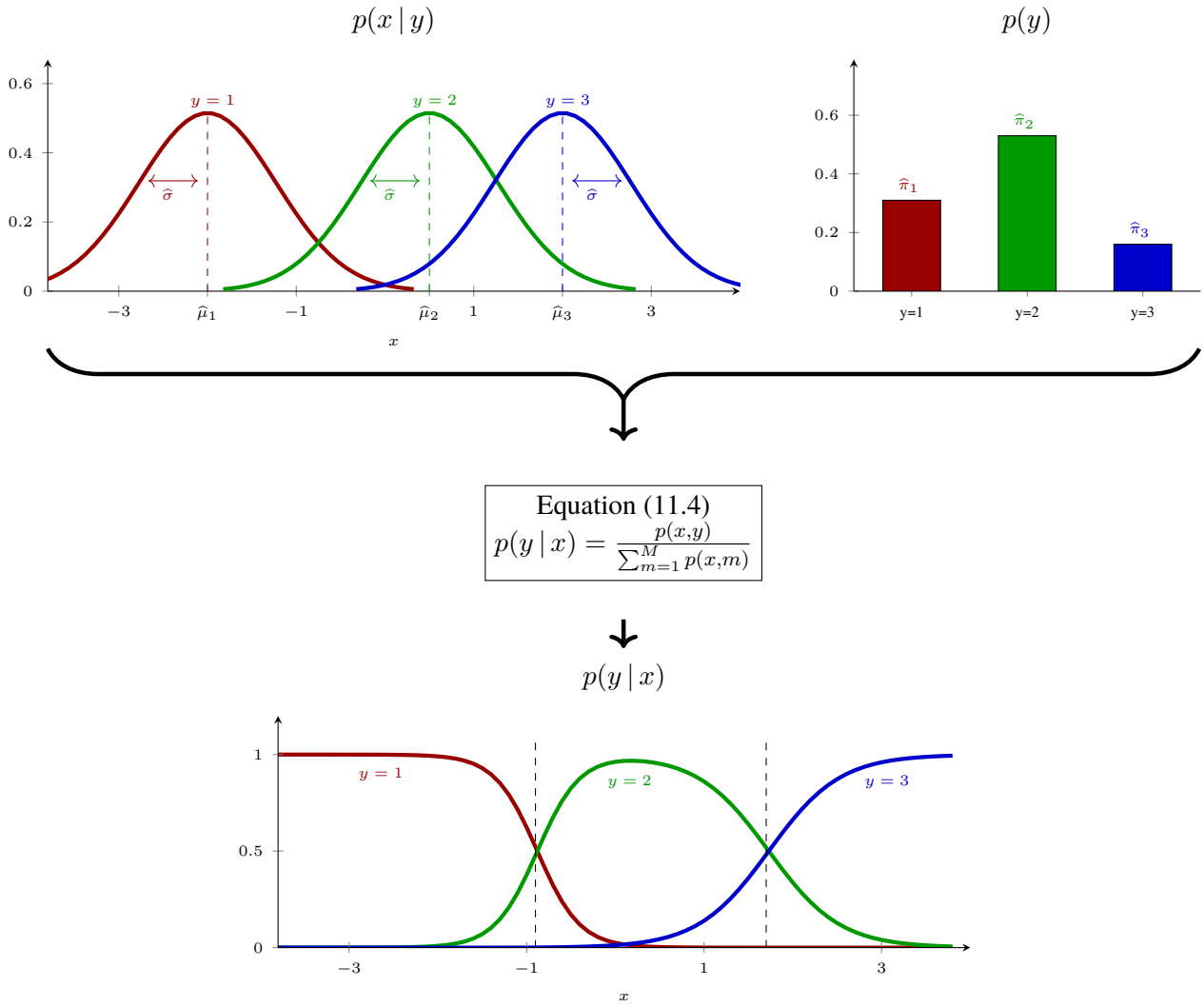


Figure 11.2: An illustration of LDA for $M = 3$ classes, with dimension $p = 1$ of the input $\mathbf{x}$. In the top is the generative model shown. To the left is the Gaussian model of $p(\mathbf{x} \mid k)$, parameterized by $\widehat{\boldsymbol{\mu}}_m$ and $\widehat{\boldsymbol{\Sigma}}$. To the 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 $\boldsymbol{\Sigma}^2$, instead of a covariance matrix $\boldsymbol{\Sigma}$). By (11.4) is the generative model "warped" into $p(k \mid x)$, shown in the bottom. The decision boundaries are shown as vertical dotted lines in the bottom plot.

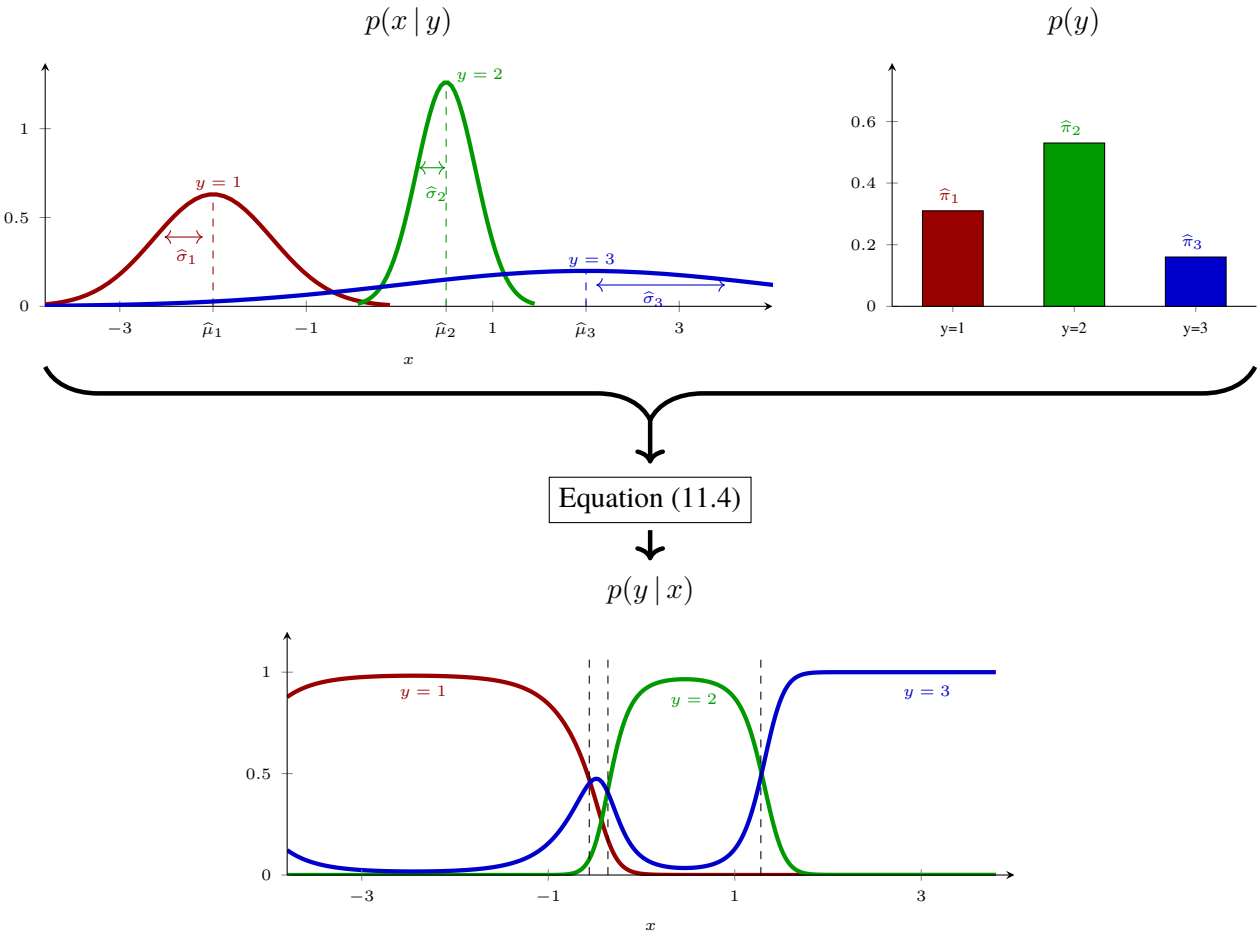
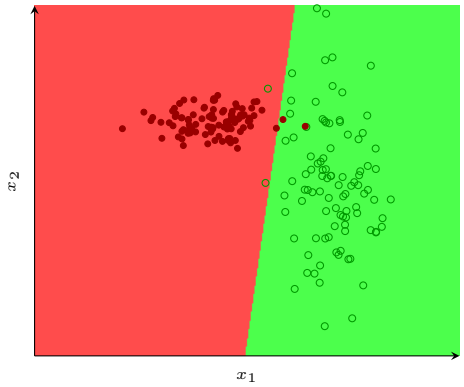
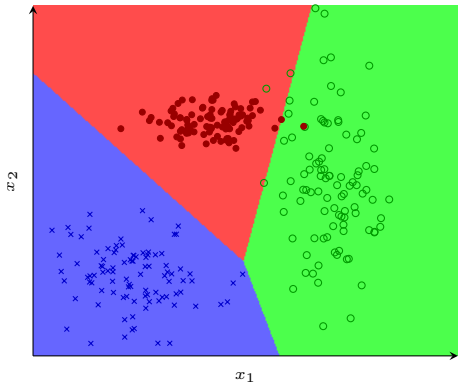


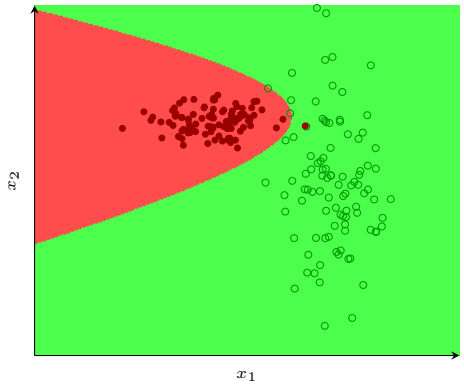
Figure 11.3: An illustration of QDA for $M = 3$ classes, in the same fashion as Figure 11.2. However, in contrast to LDA in Figure 11.2, is the learned variance $\widehat{\boldsymbol{\Sigma}}_m$ of $p(\mathbf{x} \mid m)$ different for different m in the generative model (upper left panel). For this reason can the resulting decision boundaries (bottom panel) be more complicated than for LDA, note for instance the small slice of $\widehat{y} = 3$ (blue) inbetween $\widehat{y} = 1$ (red) and $\widehat{y} = 2$ (green) around -0.5.



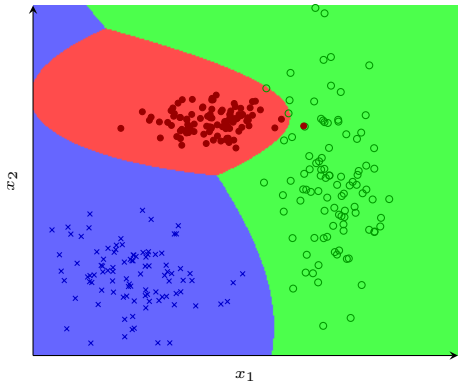
(a) LDA for $M = 2$ classes always gives a linear decision boundary. The red dots and green circles are training data from different classes, and the intersection between the red and green fields is the decision boundary obtained for an LDA classifier learned from the training data.



(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.

11.2 Unsupervised learning

Unsupervised learning deals with unlabeled data $\{x_n\}_{n=1}^N$. The name stems from the fact that there are no outputs to "supervise" the learning. The unsupervised learning algorithms will thus have to find some structure hidden only in the input data without the insights provided by the outputs. The aim in unsupervised learning is to model the unconditional distribution $p(x_n)$, as opposed to conditional distribution $p(y_n \mid x_n)$ that we are searching for in supervised learning. Importantly, unsupervised learning is more widely applicable compared to supervised learning, since it does not require a human expert to manually label the data.

Based on their functionality we can broadly divide the unsupervised learning algorithms into the following three main groups; 1. Density estimation, where the aim it to empirically describe the distribution from which the data originates. 2. Clustering, divide the data into groups of related examples. 3. Dimensionality reduction, finding a manifold that the data lies near.

Kernel density estimation

Kernel density estimation is a practical tool that allows you to estimate a smooth function from a finite set of measurements. It can be thought of as a continuous replacement for the discrete histogram and it is a useful tool for visualizing the data. More precisely, kernel density estimation offers a non-parametric method for estimating the probability density function of a random variable. The form of this estimator is

$$\widehat{p}(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{h}\kappa\left(\frac{x - x_i}{h}\right), \tag{11.8}$$

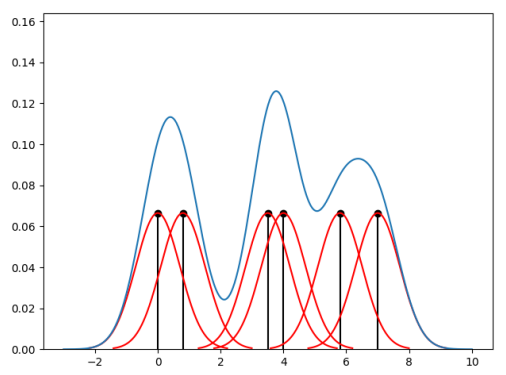
where κ denotes the kernel—a non-negative function—and h denotes the so-called bandwidth controlling the smoothness of the resulting estimate. An illustration of the idea is provided in Example 11.1.

Example 11.1: Kernel density estimation – the idea

Let the data be given by $x_1 = 0, x_2 = 0.8, x_3 = 3.5, x_4 = 4, x_5 = 5.8, x_6 = 7$, illustrated via the black vertical lines in the figure below. By centering the Gaussian kernel

$$\kappa(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right) \tag{11.9}$$

around each of these data points (illustrated via the red curves) we obtain the resulting kernel density estimate (shown as the blue curve) of the form (11.8).



Schematic illustration of the use of a kernel density estimator (11.9) to compute a continuous probability density function estimator (blue curve) according to (11.8). The underlying data is shown as black vertical lines and we centered a Gaussian kernel with a bandwidth of $h = 1$ around each data point.

In this context a kernel refers to a non-negative ($\kappa(x) > 0$), symmetric ($\kappa(-x) = \kappa(x)$) and integrable function κ with the following additional properties: $\int \kappa(x)dx = 1$, $\int x\kappa(x)dx = 0$ and $\int x^2\kappa(x)dx > 0$. In selecting the bandwidth parameter h we effectively make a trade-off between the bias and variance of the resulting estimator.

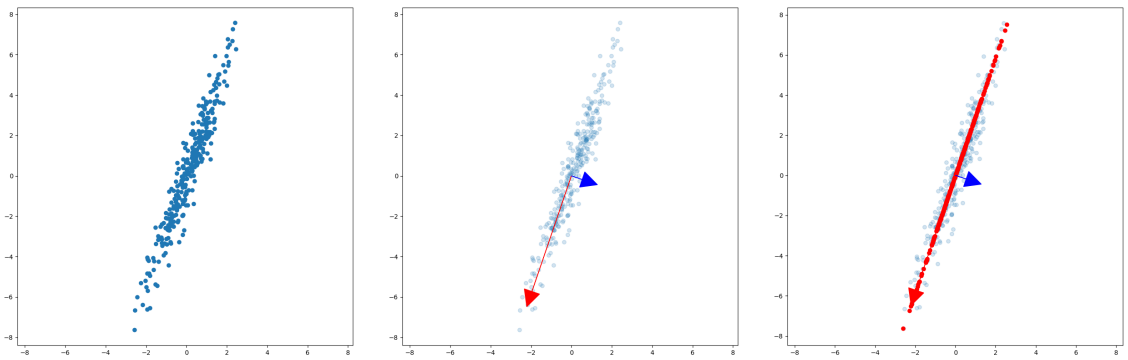


Figure 11.5: PCA

Linear dimensionality reduction

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

$$u_1^\mathsf{T} u_1 = 1. \tag{11.10}$$

Let us continue by noting that we can project any data point x_i onto the first principal component u_1 according to

$$p = \frac{u_1^\mathsf{T} x_i}{\|u_1\|^2} u_1 = u_1^\mathsf{T} x_i u_1. \tag{11.11}$$

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

$$\bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n \tag{11.12}$$

and its variance is given by

$$\begin{aligned}\mathbb{E}\left[u_1^\mathsf{T} x_n - \mathbb{E}\left[u_1^\mathsf{T} x_n\right]\right]^2 &\approx \frac{1}{N}\sum_{n=1}^{N}\left(u_1^\mathsf{T} x_n - u_1^\mathsf{T}\bar{x}\right)^2 = \frac{1}{N}\sum_{n=1}^{N}\left(u_1^\mathsf{T} x_n x_n^\mathsf{T} u_1 - 2u_1^\mathsf{T} x_n \bar{x}^\mathsf{T} u_1 + u_1^\mathsf{T}\bar{x}\bar{x}^\mathsf{T} u_1\right) \\ &= u_1^\mathsf{T} \underbrace{\left(\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})(x_n - \bar{x})^\mathsf{T}\right)}_{S} u_1 = u_1^\mathsf{T} S u_1.\end{aligned} \tag{11.13}$$

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

$$\max_{u_1} \; u_1^\mathsf{T} S u_1 \tag{11.14}$$

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

$$\max_{u_1, \lambda_1} \underbrace{u_1^\mathsf{T} S u_1 + \lambda_1 (1 - u_1^\mathsf{T} u_1)}_{L(u_1, \lambda_1)}, \tag{11.15}$$

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

$$S u_1 = \lambda_1 u_1. \tag{11.16}$$

The above relationship is actually the eigenvalue equation for the matrix S of the dataset. By multiplying (11.16) from the left with u_1^T we obtain the following expression for the variance of the projected data

$$u_1^\mathsf{T} S u_1 = u_1^\mathsf{T} \lambda_1 u_1 = \lambda_1. \tag{11.17}$$

Bibliography

Dheeru, Dua and Efi Karra Taniskidou (2017). *UCI Machine Learning Repository*. URL: http://archive.ics.uci.edu/ml.

Ezekiel, Mordecai and Karl A. Fox (1959). *Methods of Correlation and Regression Analysis*. John Wiley & Sons, Inc.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521, pp. 436–444.

LeCun, Yann, Bernhard Boser, et al. (1990). "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 396–404.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Ng, Andrew (2019). *Machine learning yearning*. In press. URL: http://www.mlyearning.org/.

Shorten, Connor and Taghi M. Khoshgoftaar (2019). "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1, p. 60.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587, p. 484.

Xu, Kelvin et al. (2015). "Show, attend and tell: Neural image caption generation with visual attention". In: *Proceedings of the International Conference on Learning representations (ICML)*.