Machine Learning: The Basics

!! ROUGH DRAFT !!

Alexander Jung

February 5, 2021

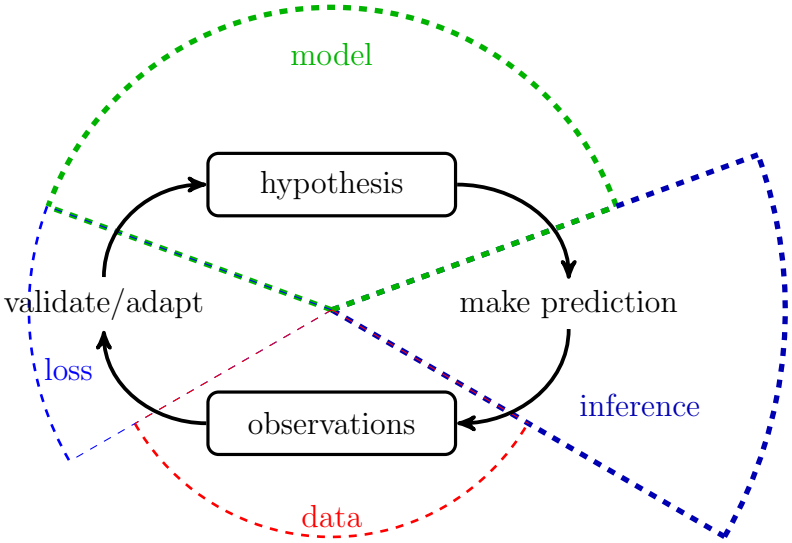


Figure 1: Machine learning implements the scientific principle of "trial and error". Machine learning continuously validates and refines a hypothesis based on a model about a phenomenon that generates observable data.

Preface

Machine learning (ML) has become a commodity in our every-day lives. We routinely ask ML empowered smartphones to suggest lovely food places or to guide us through a strange place. ML methods have also become standard tools in many fields of science and engineering. A plethora of ML applications transform human lives at unprecedented pace and scale.

This book portrays ML as the combination of three basic components: data, model and loss. ML methods combine these three components within computationally efficient implementations of the basic scientific principle "trial and error". This principle consists of the continuous adaptation of a hypothesis about a phenomenon that generates data.

ML methods use a hypothesis to compute predictions for future events. ML methods choose or learn a hypothesis from a (typically very) large set of candidate hypotheses. We refer to this set as candidates as the model of a ML method.

The adaptation or improvement of the hypothesis is based on the discrepancy between predictions and observed data. ML methods use a loss function to quantify this discrepancy.

A plethora of different ML methods is obtained by combining different design choices for the data representation, model and loss. ML methods also differ vastly in their actual implementations which might obscure their unifying basic principles.

Deep learning methods use cloud computing frameworks to train large models on huge datasets. Operating on a much finer granularity for data and computation, linear least squares regression can be implemented on small embedded systems. Nevertheless, deep learning methods and linear regression use the same principle of iteratively updating a model based on the discrepancy between model predictions and actual observed data.

Our three-component picture of ML allows a unified treatment of a wide range of concepts and techniques which seem quite unrelated at first sight. On a low-level, we discuss the regularization effect of early stopping in terms of adjusting the effective model space. On a higher-level, we can interpret privacy-preserving and explainable ML as particular design choices for the model, data and loss.

To make good use of ML tools it is instrumental to understand its underlying principles at different levels of detail. On a lower-level, this tutorial helps ML engineers to choose suitable methods for the application at hand. The book also provides leaders a higher-level view on the development of ML which is required to manage a ML or data analysis team. We believe that thinking about ML as combinations of data, model and loss helps to navigate the steadily growing offer for ready-to-use ML methods.

Acknowledgement

This tutorial is based on lecture notes prepared for the courses CS-E3210 "Machine Learning: Basic Principles", CS-E4800 "Artificial Intelligence", CS-EJ3211 "Machine Learning with Python", CS-EJ3311 "Deep Learning with Python" and CS-C3240 "Machine Learning" offered at Aalto University and within the Finnish university network `fitech.io`. This tutorial is accompanied by practical implementations of ML methods in MATLAB and Python `https://github.com/alexjungaalto/`.

This text benefited from the numerous feedback of the students within the courses that have been (co-)taught by the author. The author is indebted to Shamsiiat Abdurakhmanova, Tomi Janhunen, Yu Tian, Natalia Vesselinova, Ekaterina Voskoboinik, Buse Atli, Stefan Mojsilovic for carefully reviewing early drafts of this tutorial. Some of the figures have been generated with the help of Eric Bach. The author is grateful for the feedback received from Jukka Suomela, Väinö Mehtola, Oleg Vlasovetc, Anni Niskanen, Georgios Karakasidis, Joni Pääkkö, Harri Wallenius and Satu Korhonen.

Contents

Chapter 1

Introduction

Consider waking up some morning during winter in Finland and looking outside the window (see Figure 1.1). It seems to become a nice sunny day which is ideal for a ski trip. To choose the right gear (clothing, wax) it is vital to have some idea for the maximum daytime temperature which is typically reached around early afternoon. If we expect a maximum daytime temperature of around plus 10 degrees, we might not put on the extra warm jacket but rather take only some extra shirt for change.

Figure 1.1: Looking outside the window during the morning of a winter day in Finland.

How can we predict the maximum daytime temperature for the specific day depicted in Figure 1.1? Let us now show how this can be done via ML. In a nutshell, ML methods are computational implementations of a simple (scientific) principle.

> Find a good hypothesis based on **a model** for the phenomenon of interest by using **observed data** in order to minimize **a loss function**.

This principle contains three components: data, a model and a loss function. Any ML

method, including linear regression and deep reinforcement learning, combines these three components.

We illustrate the (rather abstract) concepts behind the main components of ML with the above problem of predicting the maximum daytime temperature during some day in Finland (see Figure 1.1). The prediction shall be based solely on the minimum daytime temperature observed in the morning of that day.

The **Finnish Meteorological Institute (FMI)** offers data on historic weather observations. We can download historic recordings of minimum and maximum daytime temperature recorded by some FMI weather station. Let us denote the resulting dataset by

$$\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}. \tag{1.1}$$

Each datapoint $\mathbf{z}^{(i)} = \big(x^{(i)}, y^{(i)}\big)$, for $i = 1, \ldots, m$, represents some previous day for which the minimum and maximum daytime temperature $x^{(i)}$ and $y^{(i)}$ has been recorded at some FMI station.

We depict the data (1.1) in Figure 1.2. Each dot in Figure 1.2 represents a particular day which is characterized by the minimum daytime temperature x and the maximum daytime temperature y.

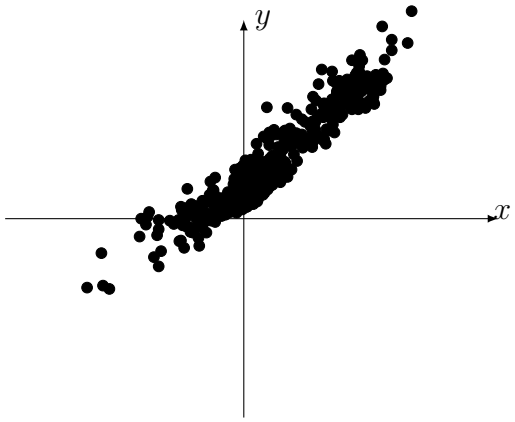


Figure 1.2: Each dot represents a day that is characterized by its minimum daytime temperature x and its maximum daytime temperature y measured at some weather station in Finland.

ML methods allow to learn a predictor map $h(x)$, reading in the minimum temperature

x and delivering a prediction (forecast or approximation) $\hat{y} = h(x)$ for the actual maximum daytime temperature y. We base this prediction on a simple hypothesis for how the minimum and maximum daytime temperature during some day are related. We assume that they are related approximately by

$$y \approx w_1 x + w_0 \text{ with } w_1 \geq 0. \tag{1.2}$$

This hypothesis reflects the intuition that the maximum daytime temperature y should be higher for days with a higher minimum daytime temperature x.

Given our initial hypothesis (1.2), it seems reasonable to restrict the ML method to only consider linear predictor maps

$$h(x) = w_1 x + w_0 \text{ with some weights } w_1 \in \mathbb{R}_+, w_0 \in \mathbb{R}. \tag{1.3}$$

The map (1.3) is monotonically increasing since $w_1 \geq 0$.

Note that (1.3) defines a whole ensemble of hypothesis maps, each individual map corresponding to a particular choice for $w_1 \geq 0$ and w_0. We refer to such an ensemble of potential predictor maps as the **model** or **hypothesis space** of a ML method.

We say that the map (1.3) is parameterized by the weight vector $\mathbf{w} = \big(w_1, w_0\big)$ and indicate this by writing $h^{(\mathbf{w})}$. For a given weight vector $\mathbf{w} = \big(w_1, w_0\big)$, we obtain the map $h^{(\mathbf{w})}(x) = w_1 x + w_0$. Figure 1.3 depicts three maps $h^{(\mathbf{w})}$ obtained for three different choices for the weights $\mathbf{w}$.

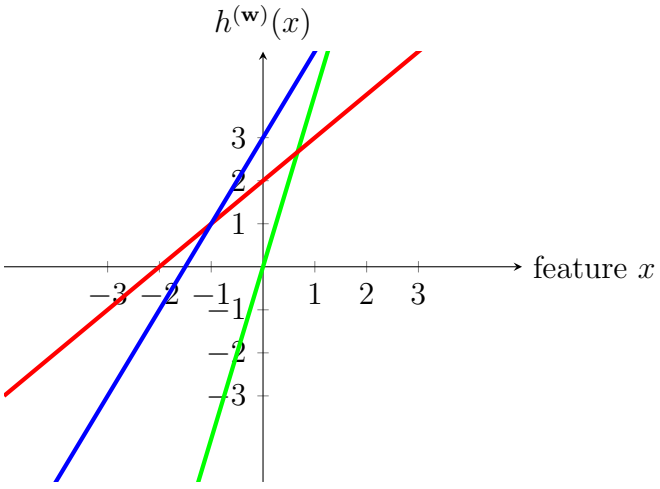


Figure 1.3: Three hypothesis maps of the form (1.3).

ML would be trivial if there is only one single hypothesis. Having only a single hypothesis

means that there is no need to try out different hypotheses to find the best one. To enable ML, we need to choose between a whole space of different hypotheses. ML methods are computationally efficient methods to choose (learn) a good hypothesis out of (typically very large) hypothesis spaces. The hypothesis space constituted by the maps (1.3) for different weights is uncountably infinite.

To find, or **learn**, a good hypothesis out of the infinite set (1.3), we need to somehow assess the quality of a particular hypothesis map. ML methods use data and a loss function for this purpose.

A loss function is a measure for the difference between the actual data and the predictions obtained from a hypothesis map (see Figure 1.4). One widely-used example of a loss function is the squared error loss $(y-h(x))^2$. Using this loss function, ML methods learn a hypothesis map out of the model (1.3) by tuning w_1, w_0 to minimize the average loss

$$(1/m)\sum_{i=1}^{m}\big(y^{(i)}-h\big(x^{(i)}\big)\big)^2.$$

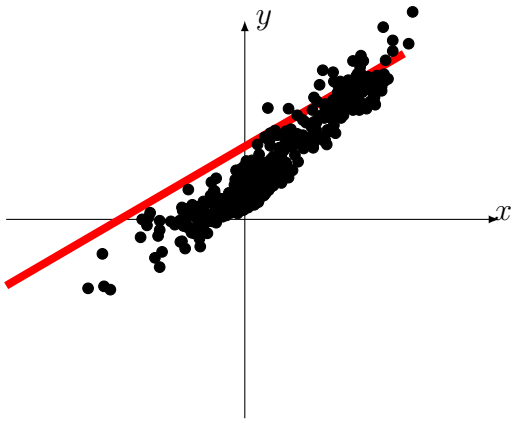


Figure 1.4: Dots represent days characterized by its minimum daytime temperature x and its maximum daytime temperature y. We also depict a straight line representing a linear predictor map. ML methods learn a predictor map with minimum discrepancy between predictor map and datapoints.

The above weather prediction is prototypical for many other ML applications. Figure 1 illustrates the typical workflow of a ML method. Starting from some initial guess, ML methods repeatedly improve their current hypothesis based on (new) observed data.

Using the current hypothesis, ML methods make predictions or forecasts about future observations. The discrepancy between the predictions and the actual observations, as measured using some loss function, is used to improve the hypothesis. Learning happens during improving the current hypothesis based on the discrepancy between its predictions and the actual observations.

ML methods must start with some initial guess or choice for a good hypothesis. This initial guess can be based on some prior knowledge or domain expertise [51]. While the initial guess for a hypothesis might not be made explicit in some ML methods, each method must use such an initial guess. In our weather prediction application discussed above, we used the approximate linear model (1.2) as the initial hypothesis.

1.1 Relation to Other Fields

ML builds on concepts from several other scientific fields.

1.1.1 Linear Algebra

Modern ML methods are computationally efficient methods to fit high-dimensional models to large amounts of data. The models underlying state-of-the-art ML methods can contain billions of tunable or learnable parameters. To make ML methods computationally efficient we need to use suitable representations for data and models.

Maybe the most widely used mathematical structure to represent data is the Euclidean space $\mathbb{R}^n$ with some dimension n. The rich algebraic and geometric structure of $\mathbb{R}^n$ allows to design of ML algorithms that can process vast amounts of data to quickly update a model (parameters).

The scatter plot in Figure 1.2 depicts datapoints (individual days) using vectors $\mathbf{z} \in \mathbb{R}^2$. We obtain a vector representation $\mathbf{z} = (x, y)^T$ of a particular day by stacking the minimum daytime temperature x and the maximum daytime temperature y into a vector $\mathbf{z}$ of length two.

We can use the Euclidean space $\mathbb{R}^n$ not only to represent datapoints but also to represent models for the data. One such class of models is obtained by linear subsets of $\mathbb{R}^n$, such as those depicted in Figure 1.3. We can then use the geometric structure of $\mathbb{R}^n$, defined by the Euclidean norm, to search for the best model. As an example, we could search for the linear model, represented by a straight line, such that the average distance to the data points in Figure 1.2 is as small as possible (see Figure 1.4).

The properties of linear structures, such as straight lines, are studied within linear algebra [67]. The basic principles behind important ML methods, such as linear regression or principal component analysis, are deeply rooted in the theory of linear algebra (see Sections 3.1 and 9.2).

1.1.2 Optimization

A main design principle for ML methods is to formulate learning tasks as optimization problems [65]. The weather prediction problem above can be formulated as the problem of optimizing (minimizing) the prediction error for the maximum daytime temperature. ML methods are then obtained by applying optimization methods to these learning problems.

The statistical and computational properties of such ML methods can be studied using tools from the theory of optimization. What sets the optimization problems arising in ML apart from "standard" optimization problems is that we do not have full access to the objective function to be minimized. Section 4 discusses methods that are based on estimating the correct objective function by empirical averages that are computed over subsets of datapoints (the training set).

1.1.3 Theoretical Computer Science

On a high level, ML methods take data as input and compute predictions as their output. The predictions are computed using algorithms such as linear solvers or optimization methods. These algorithms are implemented using some finite computational infrastructure.

One example for such a computational infrastructure is a single desktop computer. Another example for a computational infrastructure is an interconnected collection of computing nodes. ML methods must implement their computations within the available finite computational resources such as time, memory or communication bandwidth.

Therefore, engineering efficient ML methods requires a good understanding of algorithm design and their implementation on physical hardware. A huge algorithmic toolbox is provided by numerical linear algebra [67, 66].

The recent success of ML methods in several application domains might be attributed to their use of vectors and matrices to represent data and models. Using this representation allows to implement the resulting ML methods using highly efficient hard- and software implementations for numerical linear algebra.

1.1.4 Communication

We can interpret ML as a particular form of data processing. A ML algorithm is fed with observed data in order to adjust some model and, in turn, compute a prediction of some future event. Thus, ML involves transferring or communicating data to some computer which executes a ML algorithm.

The design of efficient ML systems also involves the design of efficient communication between data source and ML algorithm. The learning progress of an ML method will be slowed down if it cannot be fed with data at sufficiently large rate. Given limited memory or storage capacity, being too slow to process data at their rate of arrival (in real-time) means that we need to "throw away" data. The lost data might have carried relevant information for the ML task at hand.

1.1.5 Statistics

Consider the datapoints depicted in Figure 1.2. Each datapoint represents some previous day. Each datapoint (day) is characterized by the minimum and maximum daytime temperature as measured by some weather observation station. It might be useful to interpret these datapoints as independent and identically distributed (i.i.d.) realizations of a random vector $\mathbf{z} = \big(x, y\big)^T$. The random vector $\mathbf{z}$ is distributed according to some fixed but typically unknown probability distribution $p(\mathbf{z})$. Figure 1.5 extends the scatter plot of Figure 1.2 with some contour line that indicates the probability distribution $p(\mathbf{z})$.

Probability theory offers a great selection on methods for estimating the probability distribution from observed data (see Section 3.12). Given (an estimate of) the probability distribution $p(\mathbf{z})$, we can compute estimates for the label of a datapoint based on its features.

Having a probability distribution $p(\mathbf{z})$ for a randomly drawn datapoint $\mathbf{z} = (x, y)$, allows us to not only compute a single prediction (point estimate) $\hat{y}$ of the label y but rather an entire probability distribution $q(\hat{y})$ over all possible prediction values $\hat{y}$.

The distribution $q(\hat{y})$ represents, for each value $\hat{y}$, the probability or how likely it is that this is the true label value of the datapoint. By its very definition, this distribution $q(\hat{y})$ is precisely the conditional probability distribution $p(y|x)$ of the label value y, given the feature value x of a randomly drawn datapoint $\mathbf{z} = (x, y) \sim p(\mathbf{z})$.

Having an (estimate of) probability distribution $p(\mathbf{z})$ for the observed datapoints not only allows us to compute predictions but also to generate new datapoints. Indeed, we can artificially augment the available data by randomly drawing new datapoints according the

probability distribution $p(\mathbf{z})$ (see Section 7.3). A recently popularized class of ML methods that use probabilistic models to generate synthetic data is known as **generative adversarial networks** [28].

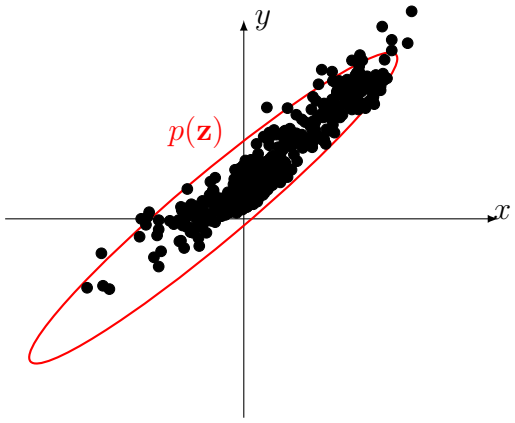


Figure 1.5: A scatterplot where each dot represents some day that is characterized by its minimum daytime temperature x and its maximum daytime temperature y.

1.1.6 Artificial Intelligence

ML is instrumental for the design and analysis of artificial intelligence (AI). AI systems (hard and software) interacts with their environment by taking certain actions. These actions influence the environment as well as the state of the AI system. The behaviour of an AI system is determined by how the perceptions made about the environment are used to form the next action.

From an engineering point of view, AI aims at optimizing behaviour to maximize a long-term **return**. The optimization of behaviour is based solely on the perceptions made by the agent. Let us consider some application domains where AI systems can be used:

- a **forest fire management system**: perceptions given by satellite images and local observations using sensors or "crowd sensing" via some mobile application which allows humans to notify about relevant events; actions amount to issuing warnings and bans of open fire; return is the reduction of number of forest fires.

- a **control unit** for combustion engines: perceptions given by various measurements such as temperature, fuel consistency; actions amount to varying fuel feed and timing and the amount of recycled exhaust gas; return is measured in reduction of emissions.
- a **severe weather warning service**: perceptions given by weather radar; actions are preventive measures taken by farmers or power grid operators; return is measured by savings in damage costs (see `https://www.munichre.com/`)
- an automated **benefit application system** for a social insurance institute (like "Kela" in Finland): perceptions given by information about application and applicant; actions are either to accept or to reject the application along with a justification for the decision; return is measured in reduction of processing time (applicants tend to prefer getting decisions quickly)
- a **personal diet assistant**: perceived environment is the food preferences of the app user and their health condition; actions amount to personalized suggestions for healthy and tasty food; return is the increase in well-being or the reduction in public spending for health-care.
- the **cleaning robot** Rumba (see Figure 1.6) perceives its environment using different sensors (distance sensors, on-board camera); actions amount to choosing different moving directions ("north", "south", "east", "west"); return might be the amount of cleaned floor area within a particular time period.
- **personal health assistant:** perceptions given by current health condition (blood values, weight,...), lifestyle (preferred food, exercise plan); actions amount to personalized suggestions for changing lifestyle habits (less meat, more jogging,...); return is measured via the level of well-being (or the reduction in public spending for health-care).
- **government-system** for a community: perceived environment is constituted by current economic and demographic indicators such as unemployment rate, budget deficit, age distribution,...; actions involve the design of tax and employment laws, public investment in infrastructure, organization of health-care system; return might be determined by the gross domestic product, the budget deficit or the gross national happiness (cf. `https://en.wikipedia.org/wiki/Gross_National_Happiness`).

ML methods are used on different levels within AI systems. On a low-level, ML methods help to extract the relevant information from raw data. AI systems use ML methods to

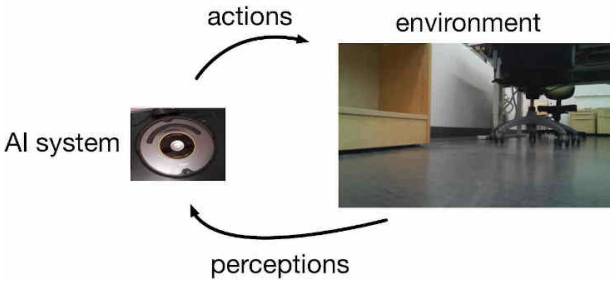


Figure 1.6: A cleaning robot chooses actions (moving directions) to maximize a long-term reward measured by the amount of cleaned floor area per day.

classify images into different categories. The AI system subsequently only needs to process the category of the image instead of its raw digital form.

ML methods are also used for higher-level tasks of an AI system. To behave optimally, an AI system or agent is required to learn a good hypothesis about how its behaviour affects its environment. We can think of optimal behaviour as the consequent choice of actions that are predicted as optimal according to some hypothesis which could be obtained by ML methods.

What sets AI methods apart from other ML methods is that they must compute predictions in real-time while collecting data and choosing the next action. Consider an AI system that steers a toy car. In any given state (point of time) the resulting prediction influences immediately the features of the following datapoints.

Consider datapoints that represent different states of a toy car. For such datapoints we could define their labels as the optimal steering angle for these states. However, it might be very challenging to obtain accurate label values for any of these datapoints. Instead, we could evaluate the usefulness of a particular steering angle only in an indirect fashion by using a reward signal. For the toy car example, we might obtain a reward from a distance sensor that indicates if the car reduces the distance to some goal or target location.

1.2 Flavours of Machine Learning

The main focus of this tutorial is on **supervised ML methods**. Supervised ML assigns labels to each datapoint. The label of a data points is some quantity of interest or higher-level fact. Roughly speaking, labels are properties of a datapoint that cannot be measured or computed easily. This is in contrast to features which are properties of datapoints that

can be measured or computed easily (see Chapter 2.1).

Supervised Learning. Supervised learning methods learn a (predictor or classifier) map that reads in features of a datapoint and outputs a prediction for its label (quantity of interest). The prediction should be an accurate approximation to the true label (see Chapter 2). To find such a map, supervised ML methods use labeled (training) data to try out different choices for the map.

The basic idea of supervised ML methods, as illustrated in Figure 1.7, is to fit a curve (representing the predictor map) to datapoints obtained from historic data (see Chapter 4). While this sounds like a simple task, the challenge of modern ML applications is the sheer amount of datapoints.

ML methods must process billions of datapoints with each single datapoint characterized by a potentially vast number of features. Consider datapoints representing social network users, whose features include all media that has been posted (videos, images, text).

Besides the sheer size of datasets, another challenge in modern ML methods is that they must be able to fit highly non-linear predictor maps. Deep learning methods address this challenge by using a computationally convenient representation of non-linear maps via artificial neural networks [27].

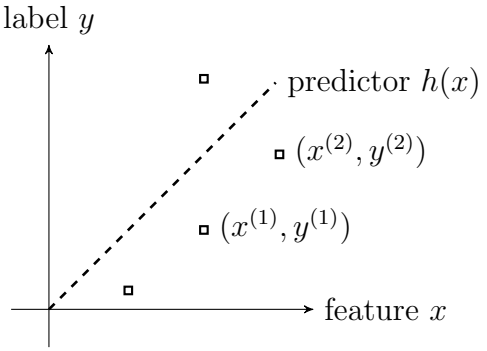


Figure 1.7: Supervised ML methods fit a curve to (a huge number of) datapoints.

Unsupervised Learning. Some ML applications do not need the concept of labels but require only to understand the intrinsic structure of data points. We refer to such applications as **unsupervised ML.** One important example for an intrinsic structure of a dataset is when its datapoints can be grouped into a few coherent subsets of cluster (see Chapter 8). Another example for such an intrinsic structure is when the datapoints are localized around a low-dimensional subspace (see Chapter 9). Unsupervised ML methods allow to determine such an intrinsic structure.

Reinforcement Learning. Another main flavour of ML considers datapoints that are characterized by labels but which cannot be determined easily beforehand. Reinforcement learning studies applications where the label values can only be determined in an indirect fashion. Consider the problem of predicting the next optimal moving direction for a toy car given its current state. datapoints represent a particular state of the car, its label is the optimum steering direction.

It is typically impossible to get labeled datapoints here since there are so many different driving scenarios that each have a different optimal steering direction. Instead, RL methods must try out some predictor of the optimal steering direction and then evaluate the quality of this prediction by some feedback signal. Such a feedback signal might be obtained from GPS sensors that allow to determine if the car stays in the lane.

1.3 Organization of this Book

Chapter 2 introduces the concepts of data, model and loss function as main components of ML. We also highlight that each component involves design choices that must take into account computational and statistical aspects.

Chapter 3 shows how well-known ML methods are obtained by specific design choices for the data, model and loss function. The aim of this chapter is to organize ML methods according to three dimensions representing data, model and loss.

Chapter 4 explains how a simple probabilistic model for data lends to the principle of **empirical risk minimization (ERM)**. This principle translates the problem of learning into an optimization problem. ML methods based on the ERM are therefore a special class of optimization methods. The ERM principle can be interpreted as a precise mathematical formulation of the "learning by trial and error" paradigm.

Chapter 5 discusses a powerful principle for learning predictors with a good performance. This principle uses the concept of gradients to locally approximate an objective function used to score predictors. A basic implementation of gradient-based optimization is the gradient descent (GD) algorithm. Variations of GD are currently the de-facto standard method for training deep neural networks [27].

Chapter 6 discusses one of the most important ideas in applied ML. This idea is to validate a predictor by trying it out on validation or test data which is different from the training data that has been used to fit a model to data. As detailed in Chapter 7, a main reason for doing validation is to detect and avoid **overfitting** which is a main reason for ML

methods to fail.

Chapter 8 presents some basic methods for **clustering** data. These methods group or partition datapoints into coherent groups which are referred to as clusters.

The efficiency of ML methods often depends crucially on the choice of data representation. Ideally we would like to have a small number of highly relevant features to characterize datapoints. If we use too many features we risk to waste computations on exploring irrelevant features. If we use too few features we might not have enough information to predict the label of a datapoint. Chapter 9 discusses **feature learning** methods that automatically determine the most relevant features from the "raw features" of a datapoint.

Two main challenges for the widespread use of ML techniques in critical application domains is privacy-preservation and explainability. Chapters 10 and 11 will discuss recent approaches to solve these challenges. We will see that the concepts developed in Chapter 9 for feature learning will be perfect tools for **privacy-preserving and explainable ML**.

Prerequisites. We assume some familiarity with basic concepts of linear algebra, real analysis, and probability theory. For a review of those concepts, we recommend [27, Chapter 2-4] and the references therein.

Notation. We mainly follow the notational conventions used in [27]. Boldface upper case letters such as $\mathbf{A}, \mathbf{X}, \ldots$ denote matrices. Boldface lower case letters such as $\mathbf{y}, \mathbf{x}, \ldots$) denote vectors. The generalized identity matrix $\mathbf{I}_{n\times r} \in \{0,1\}^{n\times r}$ is a diagonal matrix with ones on the main diagonal. The Euclidean norm of a vector $\mathbf{x} = (x_1, \ldots, x_n)^T$ is denoted $\|\mathbf{x}\| = \sqrt{\sum_{r=1}^{n} x_r^2}$.

Chapter 2

Three Components of ML: Data, Model and Loss

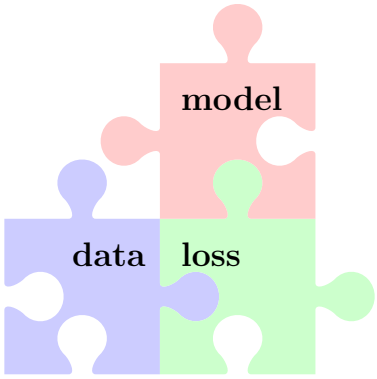


Figure 2.1: ML methods fit a model to data via minimizing a loss function.

This book portrays ML as combinations of three components:

- **data** as collections of datapoints characterized by **features** (see Section 2.1.1) and **labels** (see Section 2.1.2)
- a **model** or **hypothesis space** (see Section 2.2) of computationally feasible maps (called "predictors" or "classifiers") from feature to label space
- a **loss function** (see Section 2.3) to measure the quality of a predictor (or classifier).

We formalize a ML problem or application by identifying these three components for a given application. A formal ML problem is obtained by specific design choices for how to represent

data, which hypothesis space or model to use and with which loss function to measure the quality of a hypothesis. Once the ML problem is formally defined, we can readily apply off-the-shelf ML methods to solve them.

Similar to ML problems (or applications) we also think of ML methods as specific combinations of the three above components. We detail in Chapter 3 how some of the most popular ML methods, such as linear regression and deep learning methods, are obtained by specific design choices for the three components.

Linear regression is a ML method which uses linear maps for the hypothesis space and the squared error loss function. Deep learning methods are characterized by using artificial neural networks to represent hypothesis spaces constituted by highly non-linear predictor maps. The remainder of this chapter discusses in some depth each of the three main components of ML.

2.1 The Data

Data as Collections of datapoints. Maybe the most important component of any ML problem (and method) is data. We consider data as collections of individual datapoints which are atomic units of "information containers". datapoints can represent text documents, signal samples of time series generated by sensors, entire time series generated by collections of sensors, frames within a single video, videos within a movie database, cows within a herd, trees within a forest, forests within a collection of forests. Consider the problem of predicting the duration of a mountain hike (see Figure 2.2). Here, datapoints could represent different hiking tours.

Figure 2.2: Snapshot taken at the beginning of a mountain hike.

We use the concept of datapoints in a highly abstract and therefore very flexible manner.

datapoints can represent very different types of objects. For an image processing application it might be useful to define datapoints as images.

A recommendation system might use datapoints to represent customers. datapoints might represent time periods, animals, mountain hikes, proteins or humans. The meaning or definition of what datapoints represent is nothing but a design choice.

One practical requirement for a useful definition of datapoints is that we should have access to many of them. ML methods typically rely on constructing estimates for quantities of interest by averaging over datapoints. These estimates are often more accurate the more datapoints are used for the averaging.

A key parameter of a dataset is the number m of individual datapoints it contains. The number of datapoints within a dataset is also referred to as the sample size. Statistically, the larger the sample size m the better. However, there might be restrictions on computational resources that limit the maximum sample size m that can be processed.

Figure 2.3 illustrates two key parameters of a dataset. Beside the sample size m, a second key parameter of a dataset is the number of features used to characterize individual datapoints.

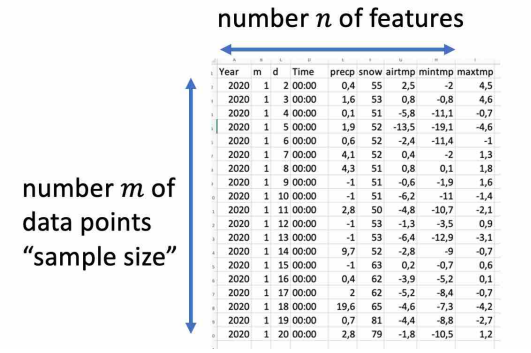


Figure 2.3: Two main parameters of a dataset are the number m of datapoints it contains and the number n of features used to characterize individual datapoints. A very important characteristic of a dataset is the ratio m/n.

For most applications, it is impossible to have full access to every single microscopic property of a datapoint. Consider a datapoint that represents a vaccine. A full characterization of such a datapoint would require to specify its chemical composition down to level of molecules and atoms. Moreover, there are properties of a vaccine that depend on the patient who received the vaccine.

It is useful to distinguish between two different groups of properties of a datapoint. The first group of properties is referred to as **features** and the second group of properties is

referred to as **label** (some authors refer to labels also as **target** or **output**).

The distinction between features and labels is somewhat blurry. The same property of a datapoint might be used as a feature in one application, while it might be used as a label in another application.

As an example, consider feature learning for datapoints representing images. One approach to learn representative features of an image is to use some of the image pixels as the label or target pixels. We can then learn new features by learning a feature map that allows to predict the target pixels.

2.1.1 Features

Similar to the definition of datapoints, also the choice of which properties to be used as features of a datapoint is a design choice. As a rule of thumb, features are properties or quantities that can be computed or measured easily. This is only an informal characterization since there is no widely-applicable measure for the difficulty in measuring a specific property.

Modern information-technology, including smartphones or wearables, allows to measure a huge number of properties about datapoints in many application domains. Consider a datapoint representing the book author "Alex Jung". Alex uses a smartphone to take snapshots. Let us assume that Alex takes five snapshots per day on average (sometimes more, e.g., during a mountain hike).

We conclude that Alex takes more than 1000 snapshots per year. Each snapshot contains around 10^6 pixels. Let us use each greyscale level of an individual pixel in those snapshots as features. This results in more than 10^9 features (per year!). If we stack all those features into a single feature vector $\mathbf{x}$, its length n would be of the order of 10^9.

Many important ML applications involve datapoints represented by very long feature vectors. To process such high-dimensional data, modern ML methods rely on concepts from high-dimensional statistics [14, 72]. One such concept from high-dimensional statistics is the notion of sparsity. Sparsity based methods, which we discuss in Section 3.4, exploits the tendency of high-dimensional datapoints to be aligned along some low-dimensional subspace.

At first sight it might seem that "the more features the better" since using more features might convey more relevant information to achieve the overall goal. However, as we discuss in Chapter 7, it can actually be detrimental to the performance of ML methods to use an excessive amount of (irrelevant) features.

Using too many irrelevant features might overwhelm or jam ML algorithms, which should invest their computational resources mainly in the processing of the most relevant features.

It is difficult to give a precise characterization on the maximum number of features that should be used to characterize the datapoints arising in a given application. As a rule of thumb, the number m of (labeled) datapoints provided to a ML method should be much larger than the number n of numeric features.

The informal condition $m/n \gg 1$ can be ensured by either collecting a sufficiently large number m of datapoints or by using a sufficiently small number n of features. In what follows, we discuss how each of these two complementary approaches can be implemented.

The acquisition of (labeled) datapoints might be costly, requiring human experts labour. Instead of collecting more raw data, it might be possible to generate synthetic data. Section 7.3 discusses how intrinsic symmetries of datasets can be used to augment the raw data with synthetic data.

As an example for an intrinsic symmetry of data, consider datapoints representing an image. We assign each image the label $y = 1$ if it shows a cat and $y = -1$ otherwise. For each image with known label we can generate several other images with the same label. This other images are simply obtained by image transformation such as rotations or re-scaling (zoom-in or zoom-out) that do not change the depicted objects. We will see in Chapter 7 that regularization techniques can be interpreted as an implicit form of data augmentation.

Instead of collecting more datapoints to ensure the condition m/n, we can use try to reduce the number n of features used to characterize datapoints. In some applications, we might use some domain knowledge to choose the most relevant features. For other applications, it might be difficult to tell which quantities are the best choice for features. Chapter 9 will discuss some data-driven methods for extracting a small number of relevant features of datapoints.

A datapoint is typically characterized by several individual features $x_1, \ldots, x_n$. It is convenient to stack the individual features into a single feature vector

$$\mathbf{x} = \big(x_1, \ldots, x_n\big)^T \in \mathbb{R}^n.$$

Each datapoint is then characterized by such a feature vector $\mathbf{x}$. The feature vectors of datapoints take on values out of some **feature space**, which we denote as $\mathcal{X}$.

The feature space is a design choice as it depends on what properties of a datapoint we use as its features. This design choice should take into account the statistical properties of the data as well as the available computational infrastructure. If the computational infrastructure allows for efficient numerical linear algebra, then using $\mathcal{X} = \mathbb{R}^n$ might be a good choice. For a computational infrastructure that allows to efficiently process networks

or graphs, we could use the nodes of a graph as the feature space (see Section 3.18).

The Euclidean space $\mathbb{R}^n$ is an example of a feature space with a rich geometric and algebraic structure [61]. The algebraic structure of $\mathbb{R}^n$ is defined by vector addition and multiplication of vectors with scalars.

The geometric structure of $\mathbb{R}^n$ is obtained by the Euclidean norm as a measure for the distance between two elements of $\mathbb{R}^n$.

The algebraic and geometric structure of $\mathbb{R}^n$ often enables an efficient search over $\mathbb{R}^n$ to find elements with desired properties. Chapter 4.3 discusses examples of such search problems in the context of learning an optimal hypothesis.

Beside the computational infrastructure, also the statistical properties of the data should be taken into account for the choice of feature space. The linear algebraic structure of $\mathbb{R}^n$ allows to efficiently represent and approximate datasets that are well aligned along linear subspaces. Section 9.2 discusses methods that learn such approximations. The geometric structure of $\mathbb{R}^n$ is used in Chapter 8 to organize data sets into few coherent groups or cluster.

Throughout this book we will mainly use the feature space $\mathbb{R}^n$ with dimension n being the number of features of a datapoint. This feature space has proven useful in many ML applications due to availability of efficient soft- and hardware for numerical linear algebra. Moreover, the algebraic and geometric structure of $\mathbb{R}^n$ seems to reflect the intrinsic structure of the data generated in many important application domains.[1]

Consider datapoints representing images with 512 by 512 pixels. A natural construction for the feature vector of such datapoints is to stack the red, green and blue intensities for all image pixels (see Figure 2.4). We end up with a feature vectors belonging to the feature space $\mathcal{X} = \mathbb{R}^n$ of (rather large) dimension $n = 3 \cdot 512^2$. Indeed, for each of the 512×512 pixels we obtain 3 numbers which encode the red, green and blue colour intensity of the respective pixel (see Figure 2.4).

In some applications, it is less obvious how to represent datapoints by a numeric feature vector in $\mathbb{R}^n$. The subfield of feature learning studies methods that map non-numeric data to numeric feature vectors in $\mathbb{R}^n$. These methods aim to structures in datasets that resemble the algebraic and geometric structure of $\mathbb{R}^n$. A quite impressive example for such methods have been developed for textual data [50].

[1] This should not be too surprising as the Euclidean space has evolved as a mathematical abstraction of physical phenomena.

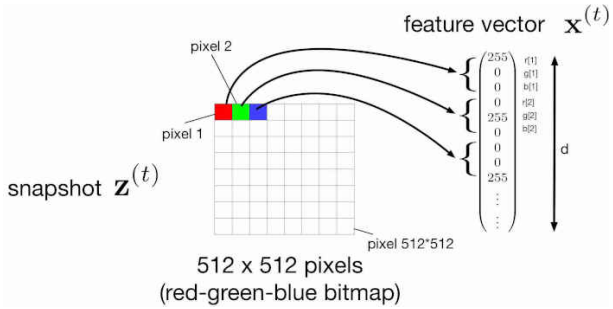


Figure 2.4: If the snapshot $\mathbf{z}^{(i)}$ is stored as a 512×512 RGB bitmap, we could use as features $\mathbf{x}^{(i)} \in \mathbb{R}^n$ the red-, green- and blue component of each pixel in the snapshot. The length of the feature vector would then be $n = 3 \cdot 512 \cdot 512 \approx 786000$.

2.1.2 Labels

Besides the features of a datapoint, there are other properties of a datapoint that represent some higher-level information or "quantity of interest" associated with the datapoint. We refer to the higher level information, or quantity of interest, associated with a datapoint as its *label* (or "output" or "target"). In contrast to features, determining the value of labels is more difficult to automate. Many ML methods revolve around finding efficient ways to determine the label of a datapoint given its features.

As already mentioned above, the distinction of datapoint properties into labels and those that are features is blurry. Roughly speaking, labels are properties of datapoints that might only be determined with the help of human experts. For datapoints representing humans we could define its label y as an indicator if the person has flu ($y = 1$) or not ($y = 0$). This label value can typically only be determined by a physician. However, in another application we might have enough resources to determine the flu status of any person of interest and could use it as a feature that characterizes a person.

Consider a datapoint that represents some hike, at the start of which the snapshot in Figure 2.2 has been taken. The features of this datapoint could be the red, green and blue (RGB) intensities of each pixel in the snapshot in Figure 2.2. We stack these RGB values into a vector $\mathbf{x} \in \mathbb{R}^n$ whose length n is three times the number of pixels in the image.

The label y associated with a datapoint (which represents a hike) could be the expected hiking time to reach the mountain in the snapshot. Alternatively, we could define the label y as the water temperature of the lake visible in the snapshot.

The label space $\mathcal{Y}$ of an ML problem contains all possible label values of datapoints. We refer to ML problems involving the $\mathcal{Y} = \mathbb{R}$ as a **regression problem**. It is also common

to refer to ML problems involving a discrete (finite or countably infinite) label space as **classification problems**.

ML problems with only two different label values are referred to as **binary classification problems**. Examples of classification problems are: detecting the presence of a tumour in a tissue, classifying persons according to their age group or detecting the current floor conditions ("grass", "tiles" or "soil") for a mower robot.

The distinction between regression and classification problems is somewhat blurry. Consider a binary classification problem based on datapoints whose label y takes on values -1 or 1. We could turn this into a regression problem by using a new label y' which is defined as the confidence in the label y being equal to 1. Given y' we can obtain y by thresholding, $y = 1$ if $y' \geq 0$ whereas $y = -1$ if $y' < 0$.

A datapoint is called *labeled* if, besides its features $\mathbf{x}$, the value of its label y is known.The acquisition of labeled data points typically involves human labour, such as handling a water thermometer at certain locations in a lake. In other applications, acquiring labels might require sending out a team of marine biologists to the Baltic sea [64], running a particle physics experiment at the European organization for nuclear research (CERN) [15], running animal testing in pharmacology [24].

There are also online market places for human labelling workforce [52]. In these market places, one can upload datapoints, such as images, and then pay some money to humans that label the datapoints, such as marking images that show a cat.

Many applications involve datapoints whose features can be determined easily, but whose labels are known for few datapoints only. Labeled data is a scarce resource. Some of the most successful ML methods have been devised in application domains where label information can be acquired easily [30]. ML methods for speech recognition and machine translation can make use of massive labeled datasets that are freely available [41].

In the extreme case, we do not know the label of any single datapoint. Even in the absence of any labeled data, ML methods can be useful for extracting relevant information from features only. We refer to ML methods which do not require any labeled datapoints as **unsupervised ML methods**. We discuss some of the most important unsupervised ML methods in Chapter 8 and Chapter 9).

As discussed next, many ML methods aim at constructing (or finding) a "good" predictor $h : \mathcal{X} \to \mathcal{Y}$ which takes the features $\mathbf{x} \in \mathcal{X}$ of a datapoint as its input and outputs a predicted label (or output, or target) $\hat{y} = h(\mathbf{x}) \in \mathcal{Y}$. A good predictor should be such that $\hat{y} \approx y$, i.e., the predicted label $\hat{y}$ is close (with small error $\hat{y} - y$) to the true underlying label y.

2.1.3 Scatterplot

Consider datapoints characterized by a single numeric feature x and label y. To gain more insight into the relation between the features and label of a datapoint, it can be instructive to generate a scatter plot as shown in Figure 1.2. A scatter plot depicts the datapoints $\mathbf{z}^{(i)} = (x^{(i)}, y^{(i)})$ in a two-dimensional plane with the axes representing the values of feature x and label y.

A visual inspection of a scatterplot might suggest potential relationships between feature x and label y. From Figure 1.2, it seems that there might be a relation between feature x and label y since datapoints with larger x tend to have larger y. This makes sense since having a larger minimum daytime temperature typically implies also a larger maximum daytime temperature.

We can obtain scatter plots for datapoints with more than two features using feature learning methods (see Chapter 9). These methods transform high-dimensional datapoints, having billions of raw features, to three or two new features. These new features can then be used as the coordinates of the datapoints in a scatter plot.

2.1.4 Probabilistic Models for Data

In what follows, we consider datapoints that are characterized by a single feature x. Many successful ML methods are based on a simple but crucial idea: Interpret datapoints as realizations of random variables. One of the most basic examples of a probabilistic model for the datapoints in ML is the **"independent and identically distributed" (i.i.d.)** assumption. This assumption interprets datapoints $x^{(1)}, \ldots, x^{(m)}$ as statistically independent realizations of one single random variable x.

It might seem somewhat awkward to interpret datapoints as realizations of random variables with some probability distribution $p(x)$. However, this interpretation allows us to use the properties of the probability distribution to characterize the statistical properties of collections of datapoints.

The probability distribution $p(x)$ is either assumed to be known or estimated from data. It is often enough to estimate only some parameters of the distribution $p(x)$ such as the mean and the variance. Section 3.12) discusses one particular approach to estimating the parameters of a probability distribution from datapoints.

Some of the most basic and widely used parameters of a probability distribution $p(x)$ are

the expected value or mean

$$\mu_x = \mathbb{E}\{x\} := \int_x xp(x)dx$$

and the variance

$$\sigma_x^2 := \mathbb{E}\big\{\big(x - \mathbb{E}\{x\}\big)^2\big\}.$$

These parameters can be estimated using the sample mean (average) and sample variance,

$$\hat{\mu}_x := (1/m)\sum_{i=1}^{m} x^{(i)} \text{ , and } \widehat{\sigma_x^2} := (1/m)\sum_{i=1}^{m}\big(x^{(i)} - \hat{\mu}_x\big)^2. \tag{2.1}$$

The sample mean and variance (2.1) can be shown to be maximum likelihood estimators of the mean and variance of a normal (Gaussian) distribution $p(x)$ (see [10, Chapter 2.3.4].

2.2 The Model

Consider a ML application generating datapoints, each characterized by features $\mathbf{x} \in \mathcal{X}$ and label $y \in \mathcal{Y}$. The goal of ML is to learn a map $h(\mathbf{x})$ such that

$$y \approx \underbrace{h(\mathbf{x})}_{\hat{y}} \text{ for any datapoint.} \tag{2.2}$$

The informal goal (2.2) needs to be made precise in two aspects. First, we need to quantify the approximation error (2.2) incurred by a given hypothesis map h. Second, we need to make precise what we actually mean by requiring (2.2) to hold for "any datapoint". We solve the first issue by the concept of a loss function in Section 2.3. The second issue is then solved in Chapter 4 by using a simple probabilistic model for data.

The main goal of ML is to learn a good hypothesis h from some training data. Given a good hypothesis map h, such that (2.2) is satisfied, ML methods use it to predict the label of any datapoint. The prediction $\hat{y} = h(\mathbf{x})$ is obtained by evaluating the hypothesis for the features $\mathbf{x}$ of a datapoint. We will use the term predictor map for the hypothesis map to highlight its use for computing predictions.

If the label space $\mathcal{Y}$ is finite, such as $\mathcal{Y} = \{-1, 1\}$, we refer to a hypothesis also as a **classifier**. For a finite label space $\mathcal{Y}$ and feature space $\mathcal{X} = \mathbb{R}^n$, we can characterize a particular classifier map h using its **decision boundary**. The decision boundary of a

classifier h is the set of boundary points between the different **decision regions**

$$\mathcal{R}_a := \{\mathbf{x} : \hat{y} = a\} \subseteq \mathcal{X}. \tag{2.3}$$

Each label value $a \in \mathcal{Y}$ is associated with a decision region $\mathcal{R}_a$. For a given label value $a \in \mathcal{Y}$, the decision region $\mathcal{R}_a$ contains all feature vectors $\mathbf{x} \in \mathcal{X}$ which are mapped to this label value, $\hat{y} = a \in \mathcal{Y}$.

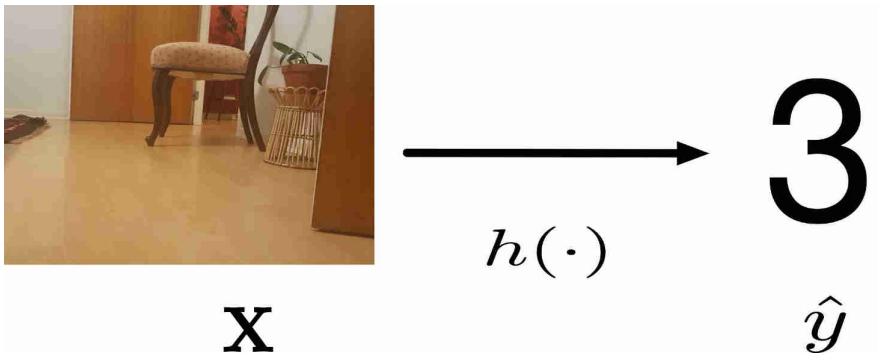


Figure 2.5: A predictor (hypothesis) h maps features $\mathbf{x} \in \mathcal{X}$, of an on-board camera snapshot, to the prediction $\hat{y} = h(\mathbf{x}) \in \mathcal{Y}$ for the coordinate of the current location of a cleaning robot. ML methods use data to learn predictors h such that $\hat{y} \approx y$ (with true label y).

In principle, ML methods could use any possible map $h : \mathcal{X} \to \mathcal{Y}$ to predict the label $y \in \mathcal{Y}$ via computing $\hat{y} = h(\mathbf{x})$. However, any ML method has only **limited computational resources** and therefore can only make use of a subset of all possible predictor maps. This subset of computationally feasible ("affordable") predictor maps is referred to as the **hypothesis space** or **model** underlying a ML method.

The largest possible hypothesis space $\mathcal{H}$ is the set $\mathcal{Y}^{\mathcal{X}}$ constituted by all maps from the feature space $\mathcal{X}$ to the label space $\mathcal{Y}$. The notation $\mathcal{Y}^{\mathcal{X}}$ has to be understood a symbolic shorthand denoting the set of all maps from $\mathcal{X}$ to $\mathcal{Y}$. The set $\mathcal{Y}^{\mathcal{X}}$ does in general not behave like powers of numbers such as 4^5.

The hypothesis space $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ is rarely used in practice since it is simply too large to work within a reasonable amount of computational resources. As depicted in Figure 2.9, ML methods typically use a hypothesis space $\mathcal{H}$ that is a tiny subset of $\mathcal{Y}^{\mathcal{X}}$.

The preference for a particular hypothesis space often depends on the available computational infrastructure available to a ML method. Different computational infrastructures favour different hypothesis spaces. ML methods implemented in a small embedded system, might prefer a linear hypothesis space which results in algorithms that require a small number of

arithmetic operations. Deep learning methods implemented in a cloud computing environment typically use much larger hypothesis spaces obtained from deep neural networks.

For the computational infrastructure provided by a **spreadsheet software**, we might use a hypothesis space constituted by maps $h : \mathcal{X} \to \mathcal{Y}$ which can be implemented easily by a spreadsheet (see Table 2.1). If we instead use the programming language Python to implement a ML method, we can obtain a hypothesis class by collecting all possible Python subroutines with one input (scalar feature x), one output argument (predicted label $\hat{y}$) and consisting of less than 100 lines of code.

If the computational infrastructure allows for efficient numerical linear algebra and the feature space is the Euclidean space $\mathbb{R}^n$, a popular choice of the hypothesis space is

$$\mathcal{H}^{(n)} := \{h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R} : h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{x}^T\mathbf{w} \text{ with some weight vector } \mathbf{w} \in \mathbb{R}^n\}. \tag{2.4}$$

The hypothesis space (2.4) is constituted by the linear maps (functions) $h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R}$. The function $h^{(\mathbf{w})}$ maps the feature vector $\mathbf{x} \in \mathbb{R}^n$ to the predicted label (or output) $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{x}^T\mathbf{w} \in \mathbb{R}$. For $n = 1$ the feature vector reduces a single feature x and the hypothesis space (2.4) consists of all maps $h^{(w)}(x) = wx$ with some weight $w \in \mathbb{R}$ (see Figure 2.7).

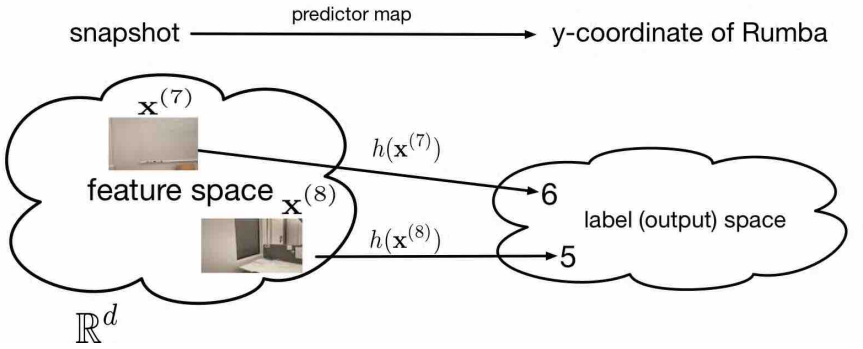


Figure 2.6: A predictor (hypothesis) $h : \mathcal{X} \to \mathcal{Y}$ takes the feature vector $\mathbf{x}^{(t)} \in \mathcal{X}$ (e.g., representing the snapshot taken by Rumba at time t) as input and outputs a predicted label $\hat{y}_t = h(\mathbf{x}^{(t)})$ (e.g., the predicted y-coordinate of Rumba at time t). A key problem studied within ML is how to automatically learn a good (accurate) predictor h such that $y_t \approx h(\mathbf{x}^{(t)})$.

The elements of the hypothesis space $\mathcal{H}$ in (2.4) are parameterized by the weight vector $\mathbf{w} \in \mathbb{R}^n$. Each map $h^{(\mathbf{w})} \in \mathcal{H}$ is fully specified by the weight vector $\mathbf{w}$. Instead of searching over the function space $\mathcal{H}$ (its elements are functions!), we can equivalently search over all possible weight vectors $\mathbf{w} \in \mathbb{R}^n$.

The search space $\mathbb{R}^n$ is still (unaccountably) infinite but it has a rich geometric and

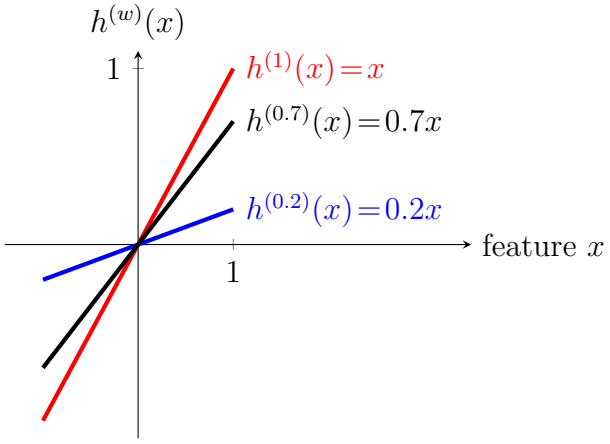


Figure 2.7: Three particular members of the hypothesis space $\mathcal{H} = \{h^{(w)} : \mathbb{R} \to \mathbb{R}, h^{(w)}(x) = w \cdot x\}$ which consists of all linear functions of the scalar feature x. We can parametrize this hypothesis space conveniently using the weight $w \in \mathbb{R}$ as $h^{(w)}(x) = w \cdot x$.

algebraic structure that allows to efficiently search over this space. Chapter 5 discusses methods that use the concept of gradients to implement an efficient search for good weights $\mathbf{w} \in \mathbb{R}^n$.

The hypothesis space (2.4) is also appealing because of the broad availability of computing hardware such as graphic processing units. Another factor boosting the widespread use of (2.4) might be the offer for optimized software libraries for numerical linear algebra.

The hypothesis space (2.4) can also be used for classification problems, e.g., with label space $\mathcal{Y} = \{-1, 1\}$. Indeed, given a linear predictor map $h^{(\mathbf{w})}$ we can classify data points according to $\hat{y} = 1$ for $h^{(\mathbf{w})}(\mathbf{x}) \geq 0$ and $\hat{y} = -1$ otherwise. The resulting classifier is an example of a **linear classifier**.

The defining property of a linear classifier, which maps data points with features $\mathbf{x}$ to a predicted label $\hat{y}$, is that its decision regions (2.3) are half-spaces. As illustrated in Figure 2.8, the decision boundary of a linear classifier is a hyperplane $\{\mathbf{x} : \mathbf{w}^T\mathbf{x} = b\}$.

ML methods that use linear classifiers include logistic regression (see Section 3.6), the SVM (see Section 3.7) and naive Bayes' classifiers (see Section 3.8).

Despite all the above mentioned benefits of the hypothesis space (2.4) it might seem too restrictive to consider only hypotheses that are linear functions of the features. Indeed, in most applications the relation between features $\mathbf{x}$ and label y of a datapoint is highly non-linear. We can then upgrade the linear hypothesis space by replacing the original features $\mathbf{x}$ of a data point with some new features $\mathbf{z} = \mathbf{\Phi}(\mathbf{x})$. The new features $\mathbf{z}$ are obtained by applying some feature map ϕ. If we apply a linear hypothesis to the new features $\mathbf{z}$, we

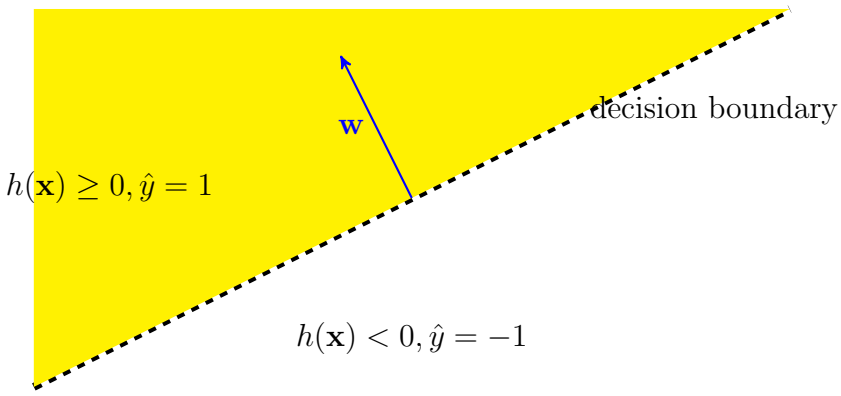


Figure 2.8: A hypothesis $h : \mathcal{X} \to \mathcal{Y}$ for a binary classification problem, with label space $\mathcal{Y} = \{-1, 1\}$ and feature space $\mathcal{X} = \mathbb{R}^2$, can be represented conveniently via the decision boundary (dashed line) which separates all feature vectors $\mathbf{x}$ with $h(\mathbf{x}) \geq 0$ from the region of feature vectors with $h(\mathbf{x}) < 0$. If the decision boundary is a hyperplane $\{\mathbf{x} : \mathbf{w}^T\mathbf{x} = b\}$ (with normal vector $\mathbf{w} \in \mathbb{R}^n$), we refer to the map h as a **linear classifier**.

obtain a non-linear map from the original features $\mathbf{x}$ to the predicted label $\hat{y}$,

$$\hat{y} = \mathbf{w}^T\mathbf{z} = \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}). \tag{2.5}$$

Section 3.9 discusses the family of kernel methods which are based on the concatenation (2.5) of (high-dimensional) feature maps $\mathbf{\Phi}(\cdot)$ with a linear hypothesis map.

The hypothesis space (2.4) can only be used for datapoints whose features are numeric vectors $\mathbf{x} = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$. In some application domains, such as natural language processing, there is no obvious natural choice for numeric features. However, since ML methods based on the hypothesis space (2.4) are well developed (using numerical linear algebra), it might be useful to construct numerical features even for non-numeric data (such as text). For text data, there has been significant progress recently on methods that map a human-generated text into sequences of vectors (see [27, Chap. 12] for more details).

The hypothesis space $\mathcal{H}$ used by a ML method is a **design choice**. Some choices have proven useful for a wide range of applications (see Chapter 3). In general, choosing a suitable hypothesis space requires a good understanding ("domain expertise") of statistical properties of the data and the limitations of the available computational infrastructure.

The design choice of the hypothesis space $\mathcal{H}$ has to balance between two conflicting requirements.

- It has to be **sufficiently large** such that it contains at least one accurate predictor

map $\hat{h} \in \mathcal{H}$. A hypothesis space $\mathcal{H}$ that is too small might fail to include a predictor map required to reproduce the (potentially highly non-linear) relation between features and label.

Consider the task of grouping or classifying images into "cat" images and "no cat image". The classification of each image is based solely on the feature vector obtained from the pixel colour intensities.

The relation between features and label ($y \in \{\text{cat}, \text{no cat}\}$) is highly non-linear. Any ML method that uses a hypothesis space consisting only of linear maps will most likely fail to learn a good predictor (classifier). We say that a ML method **underfits** the data if it uses a too small hypothesis space.

- It has to be **sufficiently small** such that its processing fits the available computational resources (memory, bandwidth, processing time). We must be able to efficiently search over the hypothesis space to find good predictors (see Section 2.3 and Chapter 4). This requirement implies also that the maps $h(\mathbf{x})$ contained in $\mathcal{H}$ can be evaluated (computed) efficiently [5]. Another important reason for using a hypothesis space $\mathcal{H}$ not too large is to avoid **overfitting** (see Chapter 7). If the hypothesis space $\mathcal{H}$ is too large, then just by luck we might find a predictor which fits the training dataset well. Such a predictor might perform poorly on data which is different from the training data (it will not generalize well).

The notion of a hypothesis space being too small or being too large can be made precise in different ways. The size of a finite hypothesis space $\mathcal{H}$ can be defined as its cardinality $|\mathcal{H}|$ which is simply the number of its elements.

Example. Consider datapoints represented by $100 \times 10 = 1000$ black-and-white pixels (see Figure 2.4) and characterized by a binary label $y \in \{0, 1\}$. We can model such datapoints using the feature space $\mathcal{X} = \{0, 1\}^{1000}$ and label space $\mathcal{Y} = \{0, 1\}$. The largest possible hypothesis space $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ consists of all maps from $\mathcal{X}$ to $\mathcal{Y}$. The size or cardinality of this space is $|\mathcal{H}| = 2^{2^{1000}}$.

Many ML methods use a hypothesis space which contains infinitely many different predictor maps (see, e.g., (2.4)). For an infinite hypothesis space, we cannot simply use the number of its elements as a measure for its size (since this number is not well-defined). Different concepts have been studied for measuring the size of infinite hypothesis spaces with the **Vapnik–Chervonenkis (VC) dimension** being maybe the most famous one [70].

We will use a simplified variant of the VC dimension and define the size of a hypothesis

feature x	prediction $h(x)$
0	0
1/10	10
2/10	3
$\vdots$	$\vdots$
1	22.3

Table 2.1: A spreadsheet representing a hypothesis map h in the form of a look-up table. The value $h(x)$ is given by the entry in the second column of the row whose first column entry is x.

space $\mathcal{H}$ as the maximum number D of arbitrary datapoints that can be perfectly fit (with probability one). For any set of D datapoints with different features, we can always find a hypothesis $h \in \mathcal{H}$ such that $y = h(\mathbf{x})$ for all datapoints $(\mathbf{x}, y) \in \mathcal{D}$.

Let us illustrate our concept for the size of a hypothesis space with two examples: linear regression and polynomial regression. Linear regression uses the hypothesis space

$$\mathcal{H}^{(n)} = \{h : \mathbb{R}^n \to \mathbb{R} : h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with some vector } \mathbf{w} \in \mathbb{R}^n\}.$$

Consider a dataset $\mathcal{D} = \big(\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big)$ consisting of m datapoints. Each datapoint is characterized by a feature vector $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and a numeric label $y^{(i)} \in \mathbb{R}$.

Let us assume that datapoints are realizations of i.i.d. continuous random variables with the same probability density function. Under this assumption, the matrix

$$\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\big) \in \mathbb{R}^{n \times m},$$

which is obtained by stacking (column-wise) the feature vectors $\mathbf{x}^{(i)}$ (for $i = 1, \ldots, m$), is full rank with probability one. Basic linear algebra allows to show that the datapoints in $\mathcal{D}$ can be perfectly fit by a linear map $h \in \mathcal{H}^{(n)}$ as long as $m \leq n$. In other words, for $m \leq n$, we can find (with probability one) a weight vector $\hat{\mathbf{w}}$ such that $y^{(i)} = \hat{\mathbf{w}}^T\mathbf{x}^{(i)}$ for all $i = 1, \ldots, m$. The size of the linear hypothesis space $\mathcal{H}^{(n)}$ is therefore $D = n$.

As a second example, consider the hypothesis space $\mathcal{H}^{(n)}_{\text{poly}}$ which is constituted by the set of polynomials with maximum degree n. The fundamental theorem of algebra tells us that any set of m datapoints with different features can be perfectly fit by a polynomial of degree n as long as $n \geq m$. Therefore, the size of the hypothesis space $\mathcal{H}^{(n)}_{\text{poly}}$ is $D = n$. Section 3.2 discusses polynomial regression in more detail.

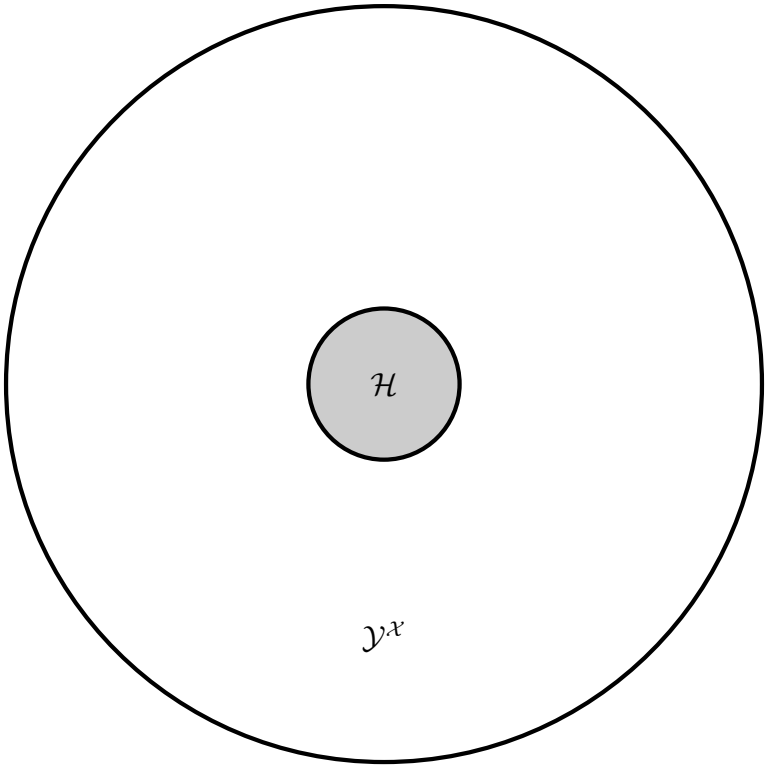


Figure 2.9: The hypothesis space $\mathcal{H}$ is a (typically very small) subset of the (typically very large) set $\mathcal{Y}^{\mathcal{X}}$ of all possible maps from feature space $\mathcal{X}$ into the label space $\mathcal{Y}$.

2.3 The Loss

Every practical ML method uses some hypothesis space $\mathcal{H}$ which consists of all **computationally feasible** predictor maps h. Which predictor map h out of all the maps in the hypothesis space $\mathcal{H}$ is the best for the ML problem at hand? We answer this question by using the concept of a **loss** as a measure the error incurred by the prediction $h(\mathbf{x})$ when the true label is y.

We formally define a loss function $\mathcal{L} : \mathcal{X} \times \mathcal{Y} \times \mathcal{H} \to \mathbb{R}$ which measures the loss $\mathcal{L}((\mathbf{x}, y), h)$ incurred by predicting the label y of a datapoint using the prediction $h(\mathbf{x})(=: \hat{y})$. The concept of loss functions is best understood by considering some examples.

Regression Loss. For ML problems involving numeric labels $y \in \mathbb{R}$, a good first choice for the loss function can be the **squared error loss** (see Figure 2.10)

$$\mathcal{L}((\mathbf{x}, y), h) := \big(y - \underbrace{h(\mathbf{x})}_{=\hat{y}}\big)^2. \tag{2.6}$$

The squared error loss (2.6) depends on the features $\mathbf{x}$ only via the predicted label value $\hat{y} = h(\mathbf{x})$. We can evaluate the squared error loss solely using the prediction $h(\mathbf{x})$ and the true label value y. Besides the prediction $h(\mathbf{x})$, no other properties of the datapoint's features $\mathbf{x}$ are required to determine the squared error loss. We will use the shorthand $\mathcal{L}(y, \hat{y})$ for any loss function that depends on the features only via the prediction $\hat{y} = h(\mathbf{x})$.

The squared error loss (2.6) has appealing computational and statistical properties. For linear predictor maps $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, the squared error loss is a convex and differentiable function of the weight vector $\mathbf{w}$. This allows, in turn, to efficiently search for the optimal linear predictor using efficient iterative optimization methods (see Chapter 5).

The squared error loss also has a useful interpretation in terms of a probabilistic model for the features and labels. Minimizing the squared error loss is equivalent to maximum likelihood estimation within a linear Gaussian model [31, Sec. 2.6.3].

Another loss function used in regression problems is the absolute error loss $|\hat{y} - y|$. Using this loss function to learn a good predictor results in methods that are robust against few outliers in the training set (see Section 3.3).

Classification Loss. In classification problems with a discrete label space $\mathcal{Y}$, such as in binary classification where $\mathcal{Y} = \{-1, 1\}$, the squared error $(y - h(\mathbf{x}))^2$ is not a useful

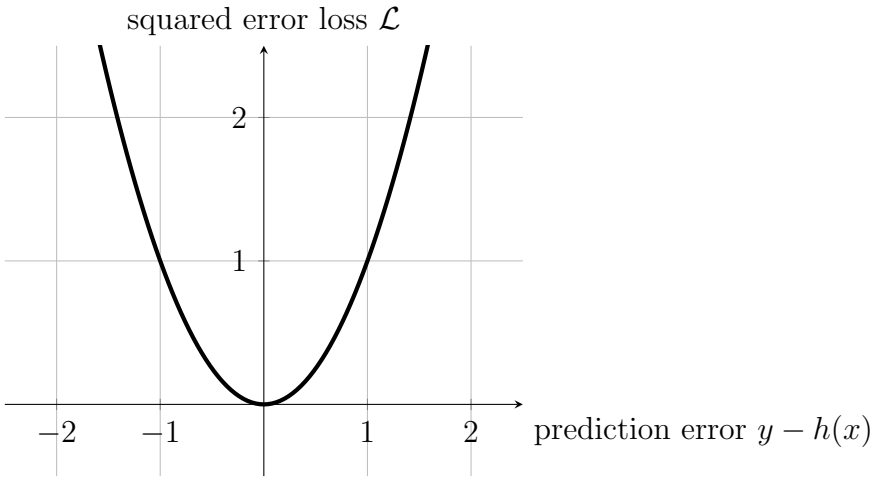


Figure 2.10: A widely used choice for the loss function in regression problems (with label space $\mathcal{Y} = \mathbb{R}$) is the squared error loss $\mathcal{L}((\mathbf{x}, y), h) := (y - h(\mathbf{x}))^2$. To evaluate the loss function for a given hypothesis h we need to know the feature $\mathbf{x}$ and the label y of the datapoint.

measure for the quality of a classifier $h(\mathbf{x})$. We would like the loss function to punish wrong classifications, e.g., when the true label is $y = -1$ but the classifier produces a large positive number, e.g., $h(\mathbf{x}) = 1000$. On the other hand, for a true label $y = -1$, we do not want to punish a classifier h which yields a large negative number, e.g., $h(\mathbf{x}) = -1000$. But exactly this unwanted result would happen for the squared error loss.

Figure 2.11 depicts a dataset consisting of 4 datapoints with binary labels. datapoints with label $y = 1$ are depicted by squares, while those with label $y = -1$ are depicted by circles. We could use these four datapoints to learn a linear hypothesis $h(x) = w_1 x + w_0$ to classify datapoints according to $\hat{y} = 1$ for $h(x) \geq 0$ and $\hat{y} = -1$ for $h(x) < 0$.

Figure 2.11 depicts two examples, denoted $h^{(1)}(x)$ and $h^{(2)}(x)$, of such a linear hypothesis. The classifications $\hat{y}$ obtained by thresholding $h^{(2)}(x)$ would perfectly match the labels of the four training datapoints. In contrast, the classifications $\hat{y}$ obtained by thresholding $h^{(1)}(x)$ disagree with the true labels for the datapoints with $y = -1$.

Based on the training data, we should prefer using $h^{(2)}(x)$ over $h^{(1)}$ to classify datapoints. However, the squared error loss incurred by the (reasonable) classifier $h^{(1)}$ is much larger than the squared error loss incurred by the (poor) classifier $h^{(2)}$. The squared error loss is typically a bad choice for assessing the quality of a hypothesis map that is used for classifying datapoints into different categories.

We now discuss some loss functions which are suitable for assessing the quality of a

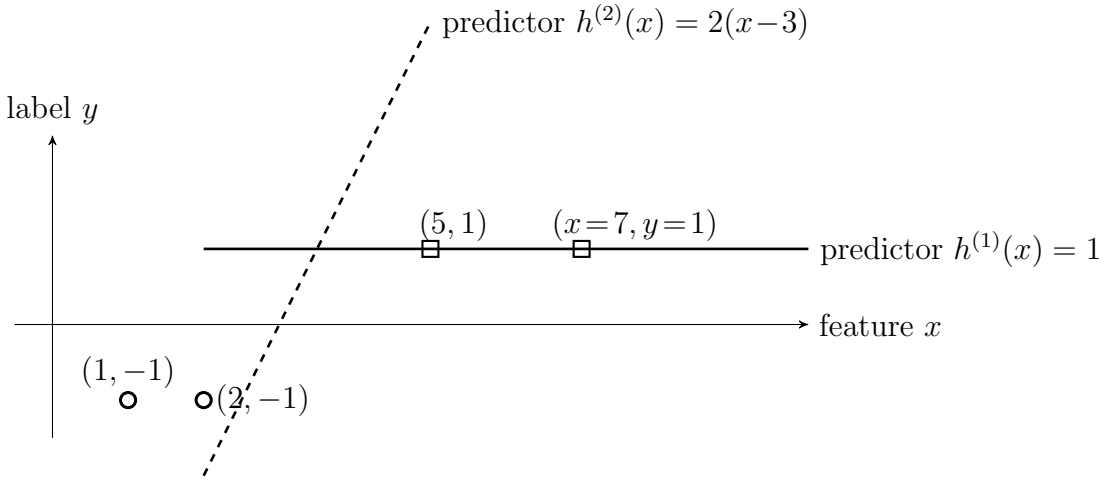


Figure 2.11: Minimizing the squared error loss would prefer the (poor) classifier $h^{(1)}$ over the (reasonable) classifier $h^{(2)}$.

hypothesis map that is used to classify datapoints according to their binary labels. It will be convenient to encode the two label values by the real numbers -1 and 1. The formulas for the loss functions we present only apply to this encoding. The modification of these formulas to a different encoding, such as label values 0 and 1, is not difficult.

Consider the problem of detecting forest fires as early as possible using webcam snapshots such as the one depicted in Figure 2.12. A particular snapshot is characterized by the features

Figure 2.12: A webcam snapshot taken near a ski resort in Lapland.

$\mathbf{x}$ and the label $y \in \mathcal{Y} = \{-1, 1\}$ with $y = 1$ if the snapshot shows a forest fire and $y = -1$ if there is no forest fire. We would like to find or learn a classifier $h(\mathbf{x})$ which takes the features

$\mathbf{x}$ as input and provides a classification according to

$$\hat{y} = \begin{cases} 1 & \text{if } h(\mathbf{x}) \geq 0 \\ -1 & \text{if } h(\mathbf{x}) \leq 0. \end{cases} \tag{2.7}$$

Ideally we would like to have $\hat{y} = y$ for any datapoint. This suggests to learn a hypothesis $h(\mathbf{x})$ by minimizing the 0/1 loss

$$\mathcal{L}((\mathbf{x}, y), h) := \begin{cases} 1 & \text{if } yh(\mathbf{x}) < 0 \\ 0 & \text{else.} \end{cases} \tag{2.8}$$

Figure 2.13 illustrates the 0/1 loss (2.8) for a datapoint with features $\mathbf{x}$ and label $y = 1$ as a function of the hypothesis h. For a given datapoint $(\mathbf{x}, y)$ and hypothesis h, the 0/1 loss is equal to zero if $\hat{y} = y$ (see (2.7)) and equal to one otherwise.

The 0/1 loss (2.8) is conceptually appealing when datapoints are interpreted as realizations of i.i.d. random variables with underlying probability distribution $p(\mathbf{x}, y)$. Given m realizations $(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ of such i.i.d. random variables,

$$(1/m) \sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) \approx \mathrm{P}(y \neq \hat{y}) \tag{2.9}$$

with high probability for sufficiently large sample size m. The precise formulation of the approximation (2.9) is known as the "law of large numbers" [8, Section 1]. We can apply the law of large numbers since the loss values $\mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h)$ are realizations of i.i.d. random variables.

In view of (2.9), the 0/1 loss seems a very natural choice for assessing the quality of a classifier if our goal is to enforce correct classification ($\hat{y} = y$). This appealing statistical property of the 0/1 loss comes at the cost of high computational complexity. Indeed, for a given datapoint $(\mathbf{x}, y)$, the 0/1 loss (2.8) is neither convex nor differentiable when viewed as a function of the classifier h. Thus, using the 0/1 loss for binary classification problems typically involves advanced optimization methods for solving the resulting learning problem (see Section 3.8).

In order to "cure" the non-convexity of the 0/1 loss we approximate it by a convex loss

function. This convex approximation results in the **hinge loss**

$$\mathcal{L}((\mathbf{x}, y), h) := \max\{0, 1 - y \cdot h(\mathbf{x})\}. \tag{2.10}$$

Figure 2.13 depicts the hinge loss (2.10) as a function of the hypothesis $h(\mathbf{x})$. While the hinge loss avoids the non-convexity of the 0/1 loss it still is a non-differentiable function of the classifier h.

Section 3.6 discusses the **logistic loss** which is a differentiable loss function that is useful for classification problems. The logistic loss

$$\mathcal{L}((\mathbf{x}, y), h) := \log(1 + \exp(-yh(\mathbf{x}))), \tag{2.11}$$

is used within logistic regression to measure the usefulness of a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$.

For a fixed feature vector $\mathbf{x}$ and label y, both, the hinge and the logistic loss function are **convex functions** of the hypothesis h. The logistic loss (2.11) depends **smoothly** on h such that we could define a derivative of the loss with respect to h. In contrast, the hinge loss (2.10) is **non-smooth** which makes it more difficult to minimize.

ML methods that use the logistic loss function, such as logistic regression in Section 3.6, can apply simple **gradient descent methods** (see Chapter 5) to minimize the average loss.

ML methods based on the hinge loss, such as those presented in Section 3.7, must use more sophisticated optimization methods to learn a predictor by minimizing the loss (see Chapter 4). We cannot directly apply GD to minimize the hinge loss since it is not differentiable. However, we can apply a generalization of GD which is known as **subgradient descent** [13]. Loosely speaking, subgradient descent is obtained from GD by replacing the concept of a gradient with the concept of a subgradient.

Let us emphasize that, very much like the choice of features and hypothesis space, the question of which particular loss function to use within an ML method is a **design choice**. The choice for the loss function should take into account the available computational resources and the statistical properties of the data (see Section 4.2). If we do not have access to any labeled datapoint, we might not use the squared error loss for measuring the quality of a hypothesis.

An important aspect guiding the choice of the loss function is the computational complexity of the resulting ML method. The basic idea behind ML methods is quite simple: learn (find)

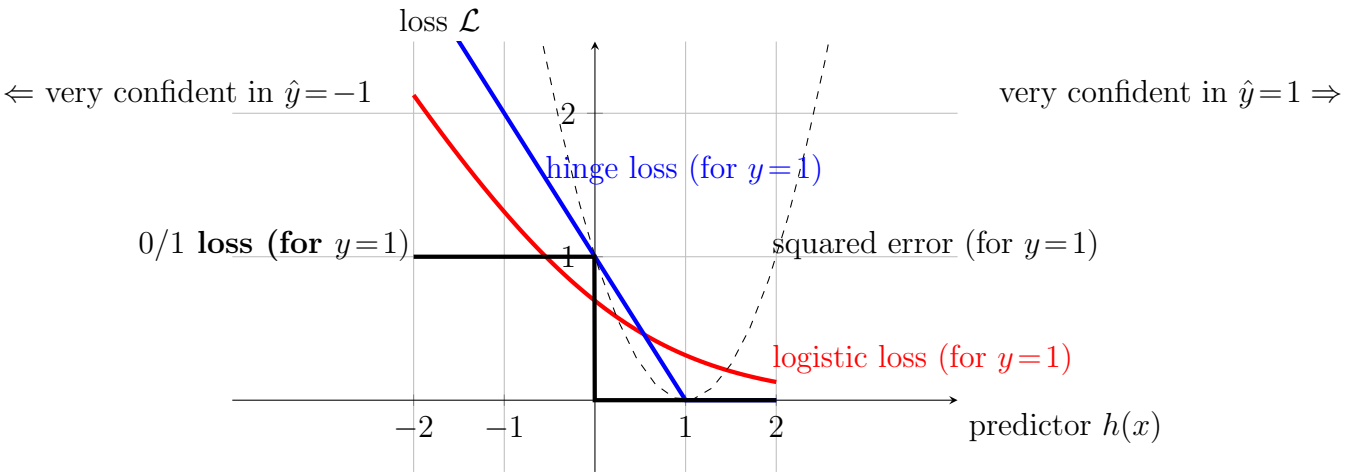


Figure 2.13: Three loss functions for assessing the quality of a hypothesis h which is used to classify a datapoint with true label $y = 1$ according to (2.7). The more confident we are in a correct classification ($\hat{y} = 1$), i.e, the more positive $h(x)$, the smaller the loss. Each of the three loss functions tends monotonically to 0 for increasing $h(x)$.

the particular hypothesis out of a given hypothesis space which yields the smallest (average) loss. Section 4.2 will discuss how the choice for the loss function influences the computational complexity of the resulting ML method. Some loss functions can be minimized using efficient iterative methods that will be discussed in Chapter 5.

Empirical and Generalization Risk. Many ML methods are based on a simple probabilistic model for the observed datapoints (i.i.d.). Using this assumption, we can define the average or generalization risk as the expectation of the loss. A large class of ML methods is based on approximating the expected value of the loss by an empirical (sample) average over a finite set of labeled datapoints (referred to as training set).

We define the empirical risk of some predictor when applied to labeled datapoints $\mathcal{D} = \big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)$ as

$$\mathcal{E}(h|\mathcal{D}) = (1/m) \sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h). \tag{2.12}$$

To ease notational burden, if the dataset $\mathcal{D}$ is clear from the context, we use $\mathcal{E}(h)$.

Regret. In some application, we might have access to the predictions obtained from some reference methods or **experts**. The quality of a hypothesis h can then be measured via the difference between the loss incurred by its predictions $h(\mathbf{x})$ and the loss incurred by the predictions of the experts [32]. This difference is referred to as the **regret**. This different measures how much we regret to have used the prediction $h(\mathbf{x})$ instead of having followed

the prediction of the expert. The goal of regret minimization is to learn a hypothesis with small regret compared to all considered experts.

The concept of regret minimization is useful when we do not make any probabilistic assumptions (such as i.i.d.) about the datapoints. Without a probabilistic model we cannot use the Bayes risk (of Bayes optimal estimator) as benchmark. Regret minimization techniques can be designed and analyzed without any such probabilistic model for the data [16]. This approach replaces the Bayes risk with the regret relative to given reference predictors (experts) as the benchmark.

Partial Feedback, "Reward". Some applications involve datapoints whose labels are so difficult or costly to determine that we cannot assume to have any labeled data available. Without any labeled data, we cannot use the concept of a loss function to measure the quality of a prediction.[2] Instead we must use some other form of indirect feedback or "reward" that indicates the usefulness of a particular prediction [16, 68].

Consider the ML problem of predicting the optimal steering directions for a toy car. The prediction has to be recalculated for each new state of the toy car. ML methods can sense the state via a feature vector $\mathbf{x}$ whose entries are pixel intensities of a snapshot. The goal is to learn a hypothesis map from the feature vector $\mathbf{x}$ to a guess $\hat{y} = h(\mathbf{x})$ for the optimal steering direction y (true label).

In some applications, we might have no access to the true label of any datapoint. This means that we cannot evaluate the quality of a particular map based on the average loss on training data. Instead, we might have only some indirect signal about the loss incurred by the prediction $\hat{y} = h(\mathbf{x})$. Such a feedback signal, or reward, could be obtained by a distance sensor which measures the change in the distance between the car and its goal such as the charging station.

2.4 Putting Together the Pieces

To illustrate how ML methods combine particular design choices for data, model and loss, we consider datapoints characterized by a single numeric feature $x \in \mathbb{R}$ and a numeric label $y \in \mathbb{R}$. We assume to have access to m labeled datapoints

$$\big(x^{(1)}, y^{(1)}\big), \ldots, \big(x^{(m)}, y^{(m)}\big) \tag{2.13}$$

[2]The evaluation of the loss function requires that the label value is known!

for which we know the true label values $y^{(i)}$.

The assumption of knowing the exact true label values $y^{(i)}$ for any datapoint is an idealization. We might often face labelling or measurement errors such that the observed labels are noisy versions of the true label. Later on, we will discuss techniques that allow ML methods to cope with noisy labels in Chapter 7.

Our goal is to learn a predictor map $h(x)$ such that $h(x) \approx y$ for any datapoint. We require the predictor map to belong to the hypothesis space $\mathcal{H}$ of linear predictors

$$h^{(w_0,w_1)}(x) = w_1 x + w_0. \tag{2.14}$$

The predictor (2.14) is parameterized by the slope w_1 and the intercept (bias or offset) w_0. We indicate this by the notation $h^{(w_0,w_1)}$. A particular choice for w_1, w_0 defines some linear predictor $h^{(w_0,w_1)}(x) = w_1 x + w_0$.

Let us use some linear predictor $h^{(w_0,w_1)}(x)$ to predict the labels of training datapoints. In general, the predictions $\hat{y}^{(i)} = h^{(w_0,w_1)}\big(x^{(i)}\big)$ will not be perfect and incur a non-zero prediction error $\hat{y}^{(i)} - y^{(i)}$ (see Figure 2.14).

We measure the goodness of the predictor map $h^{(w_0,w_1)}$ using the average squared error loss (see (2.6))

$$\begin{aligned} f(w_0, w_1) &:= (1/m) \sum_{i=1}^{m} \big(y^{(i)} - h^{(w_0,w_1)}(x^{(i)})\big)^2 \\ &\overset{(2.14)}{=} (1/m) \sum_{i=1}^{m} \big(y^{(i)} - (w_1 x^{(i)} + w_0)\big)^2. \end{aligned} \tag{2.15}$$

The training error $f(w_0, w_1)$ is the average of the squared prediction errors incurred by the predictor $h^{(w_0,w_1)}(x)$ to the labeled datapoints (2.13).

It seems natural to learn a good predictor (2.14) by choosing the weights w_0, w_1 to minimize the training error

$$\min_{w_1,w_0 \in \mathbb{R}} f(w_0, w_1) \overset{(2.15)}{=} \min_{w_1,w_0 \in \mathbb{R}} (1/m) \sum_{i=1}^{m} \big(y^{(i)} - (w_1 x^{(i)} + w_0)\big)^2. \tag{2.16}$$

The optimal weights w_0', w_1' are characterized by the **zero-gradient condition**,[3]

$$\frac{\partial f(w_0', w_1')}{\partial w_0} = 0, \text{ and } \frac{\partial f(w_0', w_1')}{\partial w_1} = 0. \tag{2.17}$$

Inserting (2.15) into (2.17) and by using basic rules for calculating derivatives, we obtain the following optimality conditions

$$(1/m)\sum_{i=1}^{m}\big(y^{(i)} - (w_1'x^{(i)} + w_0')\big) = 0, \text{ and } (1/m)\sum_{i=1}^{m} x^{(i)}\big(y^{(i)} - (w_1'x^{(i)} + w_0')\big) = 0. \tag{2.18}$$

Any weights w_0', w_1' that satisfy (2.18) define a predictor $h^{(w_0', w_1')} = w_1'x + w_0'$ that is optimal in the sense of incurring minimum training error,

$$f(w_0', w_1') = \min_{w_0, w_1 \in \mathbb{R}} f(w_0, w_1).$$

We find it convenient to rewrite the optimality condition (2.18) using matrices and vectors. To this end, we first rewrite the predictor (2.14) as

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with } \mathbf{w} = \big(w_0, w_1\big)^T, \mathbf{x} = \big(1, x\big)^T.$$

Let us stack the feature vectors $\mathbf{x}^{(i)} = \big(1, x^{(i)}\big)^T$ and labels $y^{(i)}$ of training datapoints (2.13) into the feature matrix and label vector,

$$\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\big)^T \in \mathbb{R}^{m\times 2}, \mathbf{y} = \big(y^{(1)}, \ldots, y^{(m)}\big)^T \in \mathbb{R}^m. \tag{2.19}$$

We can then reformulate (2.18) as

$$\mathbf{X}^T\big(\mathbf{y} - \mathbf{X}\mathbf{w}'\big) = \mathbf{0}. \tag{2.20}$$

The entries of any weight vector $\mathbf{w}' = \big(w_0', w_1'\big)$ that satisfies (2.20) are solutions to (2.18).

[3] A necessary and sufficient condition for $\mathbf{w}'$ to minimize a convex differentiable function $f(\mathbf{w})$ is $\nabla f(\mathbf{w}') = \mathbf{0}$ [12, Sec. 4.2.3].

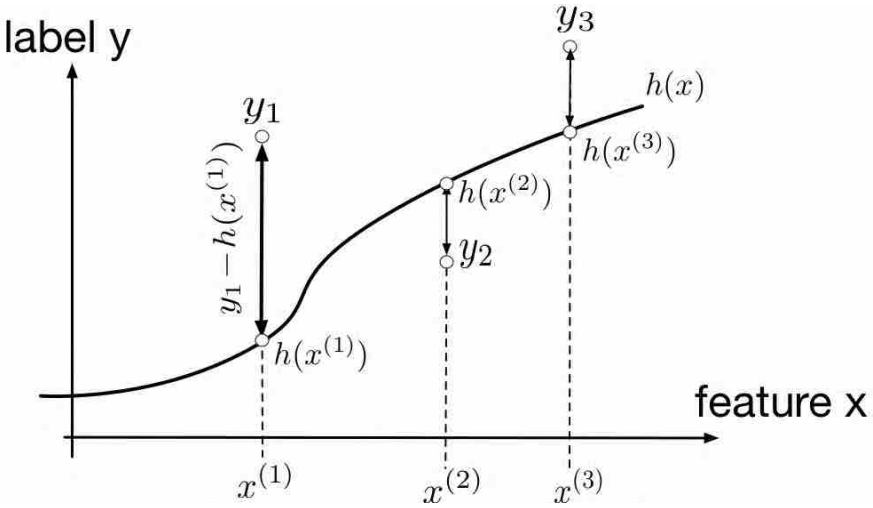


Figure 2.14: We can evaluate the quality of a particular predictor $h \in \mathcal{H}$ by measuring the prediction error $y - h(x)$ obtained for a labeled datapoint (x, y).

2.5 Exercises

2.5.1 How Many Features?

Consider the ML problem underlying a music information retrieval smartphone app [73]. Such an app aims at identifying the song title based on a short audio recording of (an interpretation of) the song obtained via the microphone of a smartphone. Here, the feature vector $\mathbf{x}$ represents the sampled audio signal and the label y is a particular song title out of a huge music database. What is the length n of the feature vector $\mathbf{x} \in \mathbb{R}^n$ if its entries are the signal amplitudes of a 20-second long recording which is sampled at a rate of 44 kHz?

2.5.2 Multilabel Prediction

Consider datapoints, each characterized by a feature vector $\mathbf{x} \in \mathbb{R}^{10}$ and vector-valued labels $\mathbf{y} \in \mathbb{R}^{30}$. Such vector-valued labels might be useful in multi-label classification problems. We might try to predict the label vector based on the features of a datapoint using a linear predictor map

$$\mathbf{h}(\mathbf{x}) = \mathbf{W}\mathbf{x} \text{ with some matrix } \mathbf{W} \in \mathbb{R}^{30 \times 10}. \tag{2.21}$$

How many different linear predictors (2.21) are there ? 10, 30,40, infinite.

2.5.3 Average Squared Error Loss as Quadratic Form

Consider linear hypothesis space consisting of linear maps parameterized by weights $\mathbf{w}$. We try to find the best linear map by minimizing the average squared error loss (empirical risk) incurred on some labeled training datapoints $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$. Is it possible to write the resulting empirical risk, viewed as a function $f(\mathbf{w})$ as a convex quadratic form $f(\mathbf{w}) = \mathbf{w}^T\mathbf{C}\mathbf{w} + \mathbf{b}\mathbf{w} + c$? If this is possible, how are the matrix $\mathbf{C}$, vector $\mathbf{b}$ and constant c related to the feature vectors and labels of the training data ?

2.5.4 Find Labeled Data for Given Empirical Risk

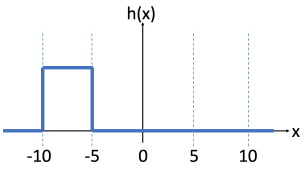
Consider linear hypothesis space consisting of linear maps parameterized by weights $\mathbf{w}$. We try to find the best linear map by minimizing the average squared error loss (empirical risk) incurred on some labeled training datapoints. Assume we know the shape of the empirical risk as a function of the weight. Can you reconstruct the labeled training data that resulted in this empirical risk function? Is the resulting labeled training data unique or are there different training sets that could have resulted in the same empirical risk function?

2.5.5 Dummy Feature Instead of Intercept

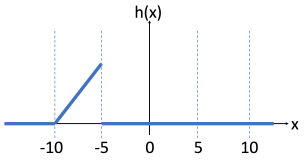
Show that any predictor of the form $h(x) = w_1 x + w_0$ can be emulated by combining a feature map $x \mapsto \mathbf{z}$ with a predictor of the form $\mathbf{w}^T\mathbf{z}$.

2.5.6 Approximate Non-Linear Maps Using Indicator Functions for Feature Maps

Consider an ML application generating datapoints characterized by a scalar feature $x \in \mathbb{R}$ and numeric label $y \in \mathbb{R}$. We construct non-linear predictor maps by first mapping the feature x to a new feature vector $\mathbf{z} = (\phi_1(x), \phi_2(x), \phi_3(x), \phi_4(x))$. The components $\phi_1(x), \ldots, \phi_4(x)$ are indicator functions of intervals $[-10, -5), [-5, 0), [0, 5), [5, 10]$. In particular, $\phi_1(x) = 1$ for $x \in [-10, -5)$ and $\phi_1(x) = 0$ otherwise. We construct a hypothesis space $\mathcal{H}_1$ by all maps of the form $\mathbf{w}^T\mathbf{z}$. Note that the map is a function of the feature x since the feature vector $\mathbf{z}$ is a function of x. Which of the following predictor maps belong to $\mathcal{H}_1$?



(a)



(b)

2.5.7 Python Hypothesis Space

Consider the source codes below for five different Python functions that read in the feature x and return some prediction $\hat{y}$. How many elements does the hypothesis space contain that is constituted by all maps $h(x)$ that can be represented by one of those Python functions.

2.5.8 A Lot of Features

In many application domains, we have access to a large number of features for each individual datapoint. Consider healthcare, where datapoints represent human patients. We could use all the measurements and diagnosis stored in the patient health record as features. When we use ML algorithms to analyse these datapoints, is it in general a good idea to use as much features as possible for datapoints ?

2.5.9 Over-Parameterization

Consider datapoints characterized by feature vectors $\mathbf{x} \in \mathbb{R}^2$ and a numeric label $y \in \mathbb{R}$. We want to learn the best predictor out of the hypothesis space

$$\mathcal{H} = \big\{h(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{w} : \mathbf{w} \in \mathcal{S}\}.$$

Here, we used the matrix $\mathbf{A} = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$ and the set

$$\mathcal{S} = \big\{(1,1)^T, (2,2)^T, (-1,3)^T, (0,4)^T\big\} \subseteq \mathbb{R}^2.$$

What is the cardinality of $\mathcal{H}$, i.e., how many different predictor maps does $\mathcal{H}$ contain?

2.5.10 Squared Error Loss

Consider a hypothesis space $\mathcal{H}$ constituted by three predictors $h_1(\cdot), h_2(\cdot), h_3(\cdot)$. Each predictor $h_j(x)$ is a real-valued function of a real-valued argument x. Moreover, for each $j \in \{1,2,3\}$, $h_j(x) = 0$ for all $x^2 \leq 1$. Can you tell which of these predictors is optimal in the sense of incurring the smallest average squared error loss on the three (training) datapoints $(x = 1/10, y = 3)$, $(0,0)$ and $(1,-1)$.

2.5.11 Classification Loss

Exercise. How would Figure 2.13 change if we consider the loss functions for a datapoint $z = (x, y)$ with known label $y = -1$?

2.5.12 Intercept Term

Linear regression models the relation between the label y and feature x of a datapoint by $y = h(x) + e$ with some small additive term e. The predictor map $h(x)$ is assumed to be linear $h(x) = w_1 x + w_0$. The weight w_0 is sometimes referred to as intercept or bias term. Assume we know for a given linear predictor map its values $h(x)$ for $x = 1$ and $x = 3$. Can you determine the weights w_1 and w_0 based on $h(1)$ and $h(3)$?

2.5.13 Picture Classification

Consider a huge collection of outdoor pictures you have taken during your last adventure trip. You want to organize these pictures as three categories (or classes) *dog*, *bird* and *fish*. How could you formalize this task as a ML problem?

2.5.14 Maximum Hypothesis Space

Consider datapoints characterized by a single real-valued feature x and a single real-valued label y. How large is the largest possible hypothesis space of predictor maps $h(x)$ that read in the feature value of a datapoint and deliver a real-valued prediction $\hat{y} = h(x)$?

2.5.15 A Large but Finite Hypothesis Space

Consider datapoints whose features are 10×10 black-and-white (bw) pixel images. Each datapoint is also characterized by a binary label $y \in \{0,1\}$. Consider the hypothesis space

which is constituted by all maps that take a bw image as input and deliver a prediction for the label. How large is this hypothesis space?

2.5.16 Size of Linear Hypothesis Space

Consider a training set of m datapoints with feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and numeric labels $y^{(1)}, \ldots, y^{(m)}$. The feature vectors and label values of the training set are arbitrary except that we assume the feature matrix $\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots\big)$ is full rank. What condition on m and n guarantee that we can find a linear predictor $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ that perfectly fits the training set, i.e., $y^{(1)} = h\big(\mathbf{x}^{(1)}\big), \ldots, y^{(m)} = h\big(\mathbf{x}^{(m)}\big)$.

Chapter 3

Some Examples

As discussed in Chapter 2, ML methods combine three main components:

- the data which is characterized by **features** which can be computed or measured easily and **labels** which represent high-level facts.
- a **model** or **hypothesis space** $\mathcal{H}$ which consists of computationally feasible predictor maps $h \in \mathcal{H}$.
- a **loss function** to measure the quality of a particular predictor map h.

Each of these three components involves design choices for the data features and labels, the model and loss function. This chapter details the specific design choices used by some of the most popular ML methods.

3.1 Linear Regression

Linear regression uses the feature space $\mathcal{X}=\mathbb{R}^n$, label space $\mathcal{Y}=\mathbb{R}$ and the linear hypothesis space

$$\mathcal{H}^{(n)} = \{h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R} : h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with some weight vector } \mathbf{w} \in \mathbb{R}^n\}. \tag{3.1}$$

The quality of a particular predictor $h^{(\mathbf{w})}$ is measured by the squared error loss (2.6). Using labeled training data $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, linear regression learns a predictor $\hat{h}$ which

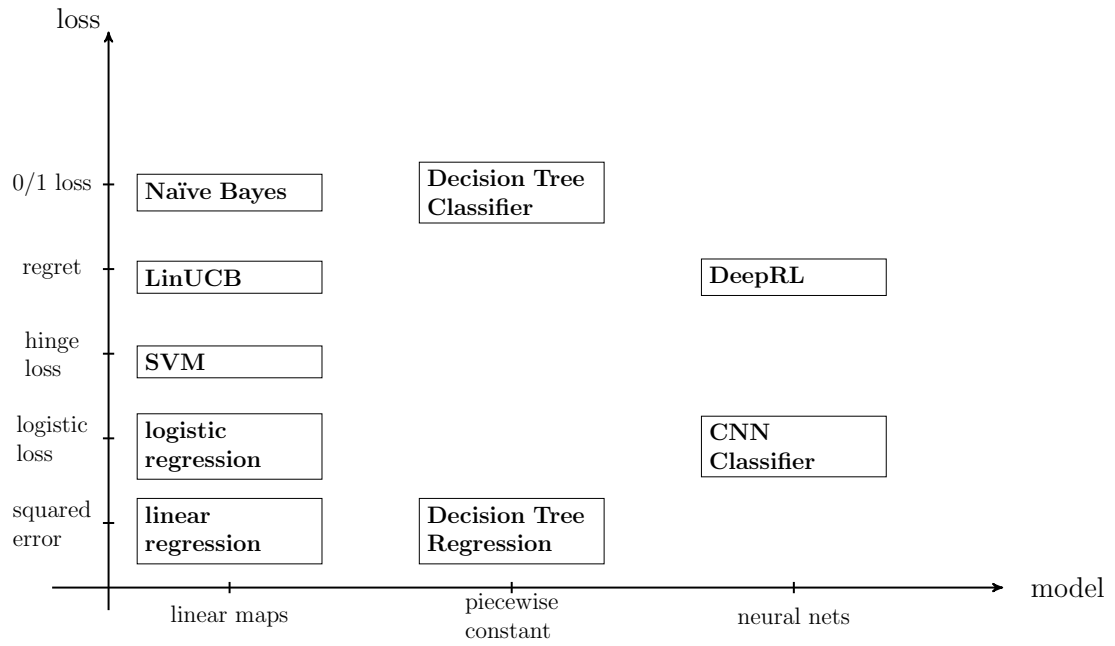


Figure 3.1: ML methods fit a model to data by minimizing a loss function. Different ML methods use different design choices for model, data and loss.

minimizes the average squared error loss, or **mean squared error**, (see (2.6))

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}^{(n)}} \mathcal{E}(h|\mathcal{D}) \tag{3.2}$$
$$\stackrel{(2.12)}{=} \operatorname*{argmin}_{h \in \mathcal{H}^{(n)}} (1/m) \sum_{i=1}^{m} (y^{(i)} - h(\mathbf{x}^{(i)}))^2.$$

Since the hypothesis space $\mathcal{H}^{(n)}$ is parameterized by the weight vector $\mathbf{w}$ (see (3.1)), we can rewrite (3.2) as an optimization problem directly over the weight vector $\mathbf{w}$:

$$\mathbf{w}_{\text{opt}} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} (1/m) \sum_{i=1}^{m} (y^{(i)} - h^{(\mathbf{w})}(\mathbf{x}^{(i)}))^2$$
$$\stackrel{h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}}{=} \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} (1/m) \sum_{i=1}^{m} (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)})^2. \tag{3.3}$$

The optimization problems (3.2) and (3.3) are equivalent in the following sense: Any optimal weight vector $\mathbf{w}_{\text{opt}}$ which solves (3.3), can be used to construct an optimal predictor $\hat{h}$, which solves (3.2), via $\hat{h}(\mathbf{x}) = h^{(\mathbf{w}_{\text{opt}})}(\mathbf{x}) = \mathbf{w}_{\text{opt}}^T \mathbf{x}$.

3.2 Polynomial Regression

Consider an ML problem involving datapoints which are characterized by a single numeric feature $x \in \mathbb{R}$ (the feature space is $\mathcal{X} = \mathbb{R}$) and a numeric label $y \in \mathbb{R}$ (the label space is $\mathcal{Y} = \mathbb{R}$). We observe a bunch of labeled datapoints which are depicted in Figure 3.2.

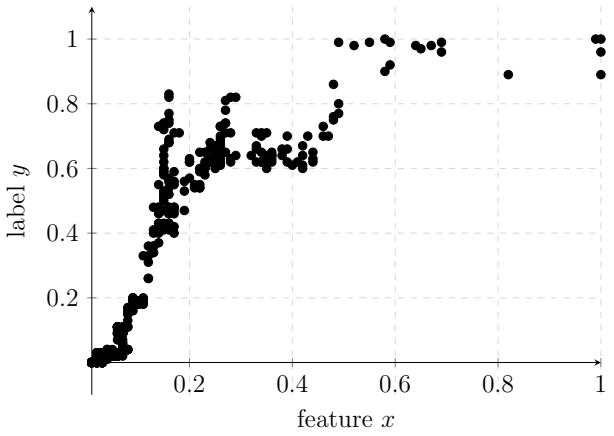


Figure 3.2: A scatterplot of some datapoints $(x^{(i)}, y^{(i)})$.

Figure 3.2 suggests that the relation $x \mapsto y$ between feature x and label y is highly non-linear. For such non-linear relations between features and labels it is useful to consider a hypothesis space which is constituted by polynomial functions

$$\mathcal{H}^{(n)}_{\rm poly} = \{h^{(\mathbf{w})} : \mathbb{R} \to \mathbb{R} : h^{(\mathbf{w})}(x) = \sum_{r=1}^{n+1} w_r x^{r-1}, \text{ with some } \mathbf{w} = (w_1, \ldots, w_{n+1})^T \in \mathbb{R}^{n+1}\}. \tag{3.4}$$

We can approximate any non-linear relation $y = h(x)$ with any desired level of accuracy using a polynomial $\sum_{r=1}^{n+1} w_r x^{r-1}$ of sufficiently large degree n.[1]

As for linear regression (see Section 3.1), we measure the quality of a predictor by the squared error loss (2.6). Based on labeled training data $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$, with scalar features $x^{(i)}$ and labels $y^{(i)}$, polynomial regression amounts to minimizing the average squared error loss (mean squared error) (see (2.6)):

$$\min_{h \in \mathcal{H}^{(n)}_{\rm poly}} (1/m) \sum_{i=1}^{m} (y^{(i)} - h^{(\mathbf{w})}(x^{(i)}))^2. \tag{3.5}$$

[1]The precise formulation of this statement is known as the "Stone-Weierstrass Theorem" [61, Thm. 7.26].

It is useful to interpret polynomial regression as a combination of a feature map (transformation) (see Section 2.1.1) and linear regression (see Section 3.1). Indeed, any polynomial predictor $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}_{\text{poly}}$ is obtained as a concatenation of the feature map

$$\phi(x) \mapsto (1, x, \ldots, x^n)^T \in \mathbb{R}^{n+1} \tag{3.6}$$

with some linear map $g^{(\mathbf{w})} : \mathbb{R}^{n+1} \to \mathbb{R} : \mathbf{x} \mapsto \mathbf{w}^T\mathbf{x}$, i.e.,

$$h^{(\mathbf{w})}(x) = g^{(\mathbf{w})}(\phi(x)). \tag{3.7}$$

Thus, we can implement polynomial regression by first applying the feature map $\mathbf{\Phi}$ (see (3.6)) to the scalar features $x^{(i)}$, resulting in the transformed feature vectors

$$\mathbf{x}^{(i)} = \mathbf{\Phi}\big(x^{(i)}\big) = \big(1, x^{(i)}, \ldots, \big(x^{(i)}\big)^n\big)^T \in \mathbb{R}^{n+1}, \tag{3.8}$$

and then applying linear regression (see Section 3.1) to these new feature vectors. By inserting (3.7) into (3.5), we end up with a linear regression problem (3.3) with feature vectors (3.8). Thus, while a predictor $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}_{\text{poly}}$ is a non-linear function $h^{(\mathbf{w})}(x)$ of the original feature x, it is a linear function, given explicitly by $g^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ (see (3.7)), of the transformed features $\mathbf{x}$ (3.8).

3.3 Least Absolute Deviation Regression

Learning a linear predictor by minimizing the average squared error loss incurred on training data is not robust against outliers. This sensitivity to outliers is rooted in the properties of the squared error loss $(\hat{y} - y)^2$. Minimizing the average squared error forces the resulting predictor $\hat{y}$ to not be too far away from any datapoint. However, it might be useful to tolerate a large prediction error $\hat{y} - y$ for few datapoints if they can be considered as outliers.

Replacing the squared loss with a different loss function can make the learning robust against few outliers. One such robust loss function is the **Huber loss** [33]

$$\mathcal{L}(y, \hat{y}) = \begin{cases} (1/2)(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \varepsilon \\ \varepsilon(|y - \hat{y}| - \varepsilon/2) & \text{else.} \end{cases} \tag{3.9}$$

The Huber loss contains a parameter ϵ, which has to be adapted to the application at

hand. The Huber loss is robust to outliers since the corresponding (large) prediction errors $y - \hat{y}$ are not squared. Outliers have a smaller effect on the average Huber loss over the entire dataset.

The Huber loss contains two important special cases. The first special case occurs when ε is chosen to be very large, such that the condition $|y - \hat{y}| \leq \varepsilon$ is satisfied for most datapoints. In this case, the Huber loss resembles the squared error loss $(y - \hat{y})^2$ (up to a scaling factor $1/2$).

The second special case occurs when ε is chosen to be very small (close to 0) such that the condition $|y - \hat{y}| \leq \varepsilon$ is almost never satisfied. In this case, the Huber loss is equivalent to the absolute loss $|y - \hat{y}|$ scaled by a factor ε.

3.4 The Lasso

We will see in Chapter 6 that linear regression (see Section 3.1) does not work well for datapoints having more features than the number of training datapoints (this is the high-dimensional regime). One approach to avoid overfitting is to modify the squared error loss (2.6) by taking into account the weight vector of the linear predictor $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$.

The Least Absolute Shrinkage and Selection Operator (Lasso) is obtained from linear regression by replacing the squared error loss with the regularized loss

$$\mathcal{L}((\mathbf{x}, y), h^{(\mathbf{w})}) = (y - \mathbf{w}^T\mathbf{x})^2 + \alpha\|\mathbf{w}\|_1. \tag{3.10}$$

The choice for the tuning parameter α can be guided by using a probabilistic model,

$$y = \overline{\mathbf{w}}^T\mathbf{x} + \varepsilon.$$

Here, $\overline{\mathbf{w}}$ denotes some true underlying weight vector and ε is as a random variable.

Appropriate values for α can then be determined based on the variance of the noise, the number of non-zero entries in $\overline{\mathbf{w}}$ and a lower bound on the non-zero values. Another option for choosing the value α is to try out different candidate values and pick the one resulting in smallest validation loss (see Section 6.2).

3.5 Gaussian Basis Regression

As discussed in Section 3.2, we can extend the basic linear regression problem by first transforming the features x using a vector-valued feature map $\phi : \mathbb{R} \to \mathbb{R}^n$ and then applying a weight vector $\mathbf{w}$ to the transformed features $\phi(x)$. For polynomial regression, the feature map is constructed using powers x^l of the scalar feature x.

It is possible to use other functions, different from polynomials, to construct the feature map ϕ. We can extend linear regression using an arbitrary feature map

$$\mathbf{\Phi}(x) = (\phi_1(x), \ldots, \phi_n(x))^T \tag{3.11}$$

with the scalar maps $\phi_j : \mathbb{R} \to \mathbb{R}$ which are referred to as **basis functions**. The choice of basis functions depends heavily on the particular application and the underlying relation between features and labels of the observed datapoints. The basis functions underlying polynomial regression are $\phi_j(x) = x^j$.

Another popular choice for the basis functions are "Gaussians"

$$\phi_{\sigma,\mu}(x) = \exp(-(1/(2\sigma^2))(x-\mu)^2). \tag{3.12}$$

The family (3.12) of maps is parameterized by the variance σ^2 and the mean (shift) μ. We obtain **Gaussian basis linear regression** by combining the feature map

$$\phi(x) = \big(\phi_{\sigma_1,\mu_1}(x), \ldots, \phi_{\sigma_n,\mu_n}(x)\big)^T \tag{3.13}$$

with linear regression (see Figure 3.3). The resulting hypothesis space is then

$$\begin{aligned}\mathcal{H}^{(n)}_{\text{Gauss}} = \{h^{(\mathbf{w})} : \mathbb{R} \to \mathbb{R} : h^{(\mathbf{w})}(x) &= \sum_{j=1}^{n} w_j \phi_{\sigma_j,\mu_j}(x) \\ &\text{with weights } \mathbf{w} = (w_1, \ldots, w_n)^T \in \mathbb{R}^n\}.\end{aligned} \tag{3.14}$$

Different choices for the variance σ^2 and shifts μ_j of the Gaussian function in (3.12) results in different hypothesis spaces $\mathcal{H}_{\text{Gauss}}$. Chapter 6.3 will discuss model selection techniques that allow to find useful values for these parameters.

The hypotheses of (3.14) are parameterized by a weight vector $\mathbf{w} \in \mathbb{R}^n$. Each hypothesis in $\mathcal{H}_{\text{Gauss}}$ corresponds to a particular choice for the weight vector $\mathbf{w}$. Thus, instead of searching over $\mathcal{H}_{\text{Gauss}}$ to find a good hypothesis, we can search over $\mathbb{R}^n$.

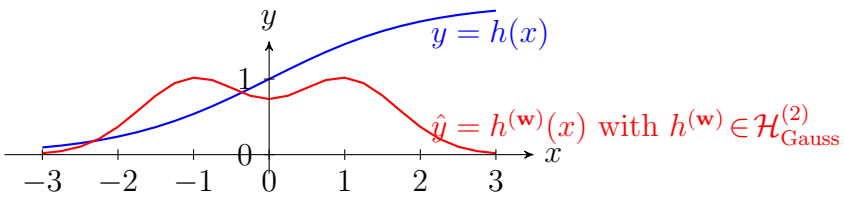


Figure 3.3: The true relation $x \mapsto y = h(x)$ (blue) between feature x and label y is highly non-linear. We might predict the label using a non-linear predictor $\hat{y} = h^{(\mathbf{w})}(x)$ with some weight vector $\mathbf{w} \in \mathbb{R}^2$ and $h^{(\mathbf{w})} \in \mathcal{H}^{(2)}_{\text{Gauss}}$.

Exercise. Try to approximate the hypothesis map depicted in Figure 3.12 by an element of $\mathcal{H}_{\text{Gauss}}$ (see (3.14)) using $\sigma = 1/10$, $n = 10$ and $\mu_j = -1 + (2j/10)$.

3.6 Logistic Regression

Logistic regression is a method for classifying datapoints which are characterized by feature vectors $\mathbf{x} \in \mathbb{R}^n$ (feature space $\mathcal{X} = \mathbb{R}^n$) according to two categories which are encoded by a label y.

It will be convenient to use the label space $\mathcal{Y} = \mathbb{R}$ and encode the two label values as $y = 1$ and $y = -1$. Logistic regression learns a predictor out of the hypothesis space $\mathcal{H}^{(n)}$ (see (3.1)).[2] Note that the hypothesis space is the same as used in linear regression (see Section 3.1).

At first sight, it seems wasteful to use a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, with some weight vector $\mathbf{w} \in \mathbb{R}^n$, to predict a binary label y. Indeed, while the prediction $h(\mathbf{x})$ can take any real number, the label $y \in \{-1, 1\}$ takes on only one of the two real numbers 1 and -1.

It turns out that even for binary labels it is quite useful to use a hypothesis map h which can take on arbitrary real numbers. We can always obtain a predicted label $\hat{y} \in \{-1, 1\}$ by comparing hypothesis value $h(\mathbf{x})$ with a threshold. A datapoint with features $\mathbf{x}$, is classified as $\hat{y} = 1$ if $h(\mathbf{x}) \geq 0$ and $\hat{y} = -1$ for $h(\mathbf{x}) < 0$. Thus, we use the sign of the predictor map $h(\mathbf{x})$ to determine the final prediction for the label. The absolute value $|h(\mathbf{x})|$ is then used to quantify the reliability of (or confidence in) the classification $\hat{y}$.

Consider two datapoints with features $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}$ and a linear classifier map h yielding the function values $h(\mathbf{x}^{(1)}) = 1/10$ and $h(\mathbf{x}^{(2)}) = 100$. Whereas the predictions for both datapoints result in the same label predictions, i.e., $\hat{y}^{(1)} = \hat{y}^{(2)} = 1$, the classification of the datapoint with feature vector $\mathbf{x}^{(2)}$ seems to be much more reliable.

[2] It is important to note that logistic regression can be used with an arbitrary label space which contains two different elements. Another popular choice for the label space is $\mathcal{Y} = \{0, 1\}$.

In logistic regression, we assess the quality of a particular classifier $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}$ using the logistic loss (2.11) Given some labeled training data $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m}$, logistic regression amounts to minimizing the empirical risk (average logistic loss)

$$\begin{aligned}\mathcal{E}(\mathbf{w}|\mathcal{D}) &= (1/m)\sum_{i=1}^{m} \log(1+\exp(-y^{(i)}h^{(\mathbf{w})}(\mathbf{x}^{(i)}))) \\ &\overset{h^{(\mathbf{w})}(\mathbf{x})=\mathbf{w}^T\mathbf{x}}{=} (1/m)\sum_{i=1}^{m} \log(1+\exp(-y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})). \end{aligned} \tag{3.15}$$

Once we have found the optimal weight vector $\widehat{\mathbf{w}}$ which minimizes (3.15), we classify a datapoint based on its features $\mathbf{x}$ according to

$$\hat{y} = \begin{cases} 1 & \text{if } h^{(\widehat{\mathbf{w}})}(\mathbf{x}) \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{3.16}$$

Since $h^{(\widehat{\mathbf{w}})}(\mathbf{x}) = \big(\widehat{\mathbf{w}}\big)^T\mathbf{x}$ (see (3.1)), the classifier (3.16) amounts to testing whether $\big(\widehat{\mathbf{w}}\big)^T\mathbf{x} \geq 0$ or not.

The classifier (3.16) partitions the feature space $\mathcal{X} = \mathbb{R}^n$ into two half-spaces $\mathcal{R}_1 = \big\{\mathbf{x} : \big(\widehat{\mathbf{w}}\big)^T\mathbf{x} \geq 0\big\}$ and $\mathcal{R}_{-1} = \big\{\mathbf{x} : \big(\widehat{\mathbf{w}}\big)^T\mathbf{x} < 0\big\}$ which are separated by the hyperplane $\big(\widehat{\mathbf{w}}\big)^T\mathbf{x} = 0$ (see Figure 2.8). Any datapoint with features $\mathbf{x} \in \mathcal{R}_1$ ($\mathbf{x} \in \mathcal{R}_{-1}$) is classified as $\hat{y} = 1$ ($\hat{y} = -1$).

Logistic regression can be interpreted as a maximum likelihood estimator within a particular probabilistic model for the datapoints. This interpretation is based on modelling the label $y \in \{-1, 1\}$ of a datapoint as random variables with the probability

$$\begin{aligned}\mathrm{P}(y=1;\mathbf{w}) &= 1/(1+\exp(-\mathbf{w}^T\mathbf{x})) \\ &\overset{h^{(\mathbf{w})}(\mathbf{x})=\mathbf{w}^T\mathbf{x}}{=} 1/(1+\exp(-h^{(\mathbf{w})}(\mathbf{x}))). \end{aligned} \tag{3.17}$$

As the notation indicates, the probability (3.17) is parameterized by the weight vector $\mathbf{w}$ of the linear hypothesis $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$. Given the probabilistic model (3.17), we can interpret the classification (3.16) as choosing $\hat{y}$ to maximize the probability $\mathrm{P}(y = \hat{y}; \mathbf{w})$.

Since $\mathrm{P}(y=1)+\mathrm{P}(y=-1)=1$,

$$\begin{aligned}\mathrm{P}(y=-1) &= 1-\mathrm{P}(y=1)\\ &\overset{(3.17)}{=} 1-1/(1+\exp(-\mathbf{w}^T\mathbf{x}))\\ &= 1/(1+\exp(\mathbf{w}^T\mathbf{x})).\end{aligned} \tag{3.18}$$

In practice we do not know the weight vector in (3.17). Rather, we have to estimate the weight vector $\mathbf{w}$ in (3.17) from observed datapoints. A principled approach to estimate the weight vector is to maximize the probability (or likelihood) of actually obtaining the dataset $\mathcal{D}=\{(\mathbf{x}^{(i)},y^{(i)})\}_{i=1}^m$ as realizations of i.i.d. datapoints whose labels are distributed according to (3.17). This yields the maximum likelihood estimator

$$\begin{aligned}\hat{\mathbf{w}} &= \operatorname*{argmax}_{\mathbf{w}\in\mathbb{R}^n} \mathrm{P}(\{y^{(i)}\}_{i=1}^m)\\ &\overset{y^{(i)}\ \text{i.i.d.}}{=} \operatorname*{argmax}_{\mathbf{w}\in\mathbb{R}^n} \prod_{i=1}^m \mathrm{P}(y^{(i)})\\ &\overset{(3.17),(3.18)}{=} \operatorname*{argmax}_{\mathbf{w}\in\mathbb{R}^n} \prod_{i=1}^m 1/(1+\exp(-y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})).\end{aligned} \tag{3.19}$$

Note that the last expression (3.19) is only valid if we encode the binary labels using the values 1 and -1. Using different label values results in a different expression.

Maximizing a positive function $f(\mathbf{w})>0$ is equivalent to maximizing $\log f(x)$,

$$\operatorname*{argmax}_{\mathbf{w}\in\mathbb{R}^n} f(\mathbf{w}) = \operatorname*{argmax}_{\mathbf{w}\in\mathbb{R}^n} \log f(\mathbf{w}).$$

Therefore, (3.19) can be further developed as

$$\begin{aligned}\hat{\mathbf{w}} &\overset{(3.19)}{=} \operatorname*{argmax}_{\mathbf{w}\in\mathbb{R}^n} \sum_{i=1}^m -\log\left(1+\exp(-y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})\right)\\ &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} (1/m)\sum_{i=1}^m \log\left(1+\exp(-y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})\right).\end{aligned} \tag{3.20}$$

Comparing (3.20) with (3.15) reveals that logistic regression is nothing but maximum likelihood estimation of the weight vector $\mathbf{w}$ in the probabilistic model (3.17).

3.7 Support Vector Machines

Support vector machines (SVM) use the hinge loss (2.10) to assess the usefulness of a hypothesis map $h \in \mathcal{H}$ for classifying datapoints. The most basic variant of SVM applies to ML problems with feature space $\mathcal{X} = \mathbb{R}^n$, label space $\mathcal{Y} = \{-1, 1\}$ and the hypothesis space $\mathcal{H}^{(n)}$ (3.1). This is the same hypothesis space as used by linear and logistic regression which we have discussed in Section 3.1 and Section 3.6, respectively.

The **soft-margin** SVM [44, Chapter 2] uses the loss

$$\begin{aligned} \mathcal{L}((\mathbf{x}, y), h^{(\mathbf{w})}) &:= \max\{0, 1 - y \cdot h^{(\mathbf{w})}(\mathbf{x})\} + \lambda \|\mathbf{w}\|^2 \\ &\overset{h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}}{=} \max\{0, 1 - y \cdot \mathbf{w}^T \mathbf{x}\} + \lambda \|\mathbf{w}\|^2 \end{aligned} \tag{3.21}$$

with a tuning parameter $\lambda > 0$. According to [44, Chapter 2], a classifier $h^{(\mathbf{w}_{\rm SVM})}$ minimizing the loss (3.21), averaged over some labeled datapoints $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, is equivalent to maximizing the distance (margin) ξ between the decision boundary, given by the set of points $\mathbf{x}$ satisfying $\mathbf{w}_{\rm SVM}^T \mathbf{x} = 0$, and each of the two classes $\mathcal{C}_1 = \{\mathbf{x}^{(i)} : y^{(i)} = 1\}$ and $\mathcal{C}_2 = \{\mathbf{x}^{(i)} : y^{(i)} = -1\}$.

Making the margin as large as possible is reasonable as it ensures that the resulting classifications are robust against small (relative to the margin) perturbations of the features (see Section 7.2).

As depicted in Figure 3.4, the margin between the decision boundary and the classes $\mathcal{C}_1$ and $\mathcal{C}_2$ is typically determined by few datapoints (such as $\mathbf{x}^{(6)}$ in Figure 3.4) which are closest to the decision boundary. These datapoints have minimum distance to the decision boundary and are referred to as **support vectors**.

We highlight that both, the SVM and logistic regression amount to linear classifiers $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}$ (see (3.1)) whose decision boundary is a hyperplane in the feature space $\mathcal{X} = \mathbb{R}^n$ (see Figure 2.8). The difference between SVM and logistic regression is the loss function used for evaluating the quality of a particular classifier $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}$. The SVM uses the hinge loss (2.10) which is the best convex approximation to the 0/1 loss (2.8). Thus, we expect the classifier obtained by the SVM to yield a smaller classification error probability $\mathrm{P}(\hat{y} \neq y)$ (with $\hat{y} = 1$ if $h(\mathbf{x}) \geq 0$ and $\hat{y} = -1$ otherwise) compared to logistic regression which uses the logistic loss (2.11).

The statistical superiority of the SVM comes at the cost of increased computational complexity. In particular, the hinge loss (2.10) is non-differentiable which prevents the use of simple gradient-based methods (see Chapter 5) and requires more advanced optimization

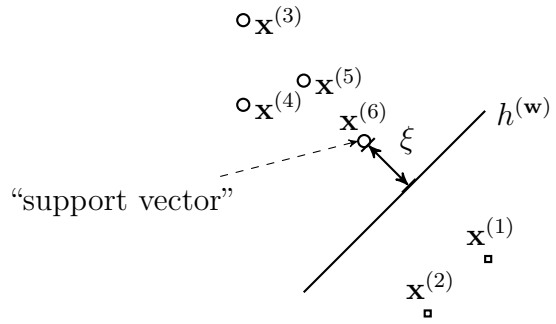


Figure 3.4: The SVM aims at a classifier $h^{(\mathbf{w})}$ with small hinge loss. Minimizing hinge loss of a classifier is the same as maximizing the margin ξ between the decision boundary (of the classifier) and each class of the training set.

methods. In contrast, the logistic loss (2.11) is convex and differentiable which allows to apply simple iterative methods for minimization of the loss (see Chapter 5).

3.8 Bayes' Classifier

Consider datapoints characterized by features $\mathbf{x} \in \mathcal{X}$ and some binary label $y \in \mathcal{Y}$. We can use any two different label values but let us assume that the two possible label values are $y = -1$ and $y = 1$.

The goal of ML is to find (or learn) a classifier $h : \mathcal{X} \to \mathcal{Y}$ such that the predicted (or estimated) label $\hat{y} = h(\mathbf{x})$ agrees with the true label $y \in \mathcal{Y}$ as much as possible. Thus, it is reasonable to assess the quality of a classifier h using the 0/1 loss (2.8). We could then learn a classifier using the ERM with the loss function (2.8). However, the resulting optimization problem is typically intractable since the loss (2.8) is non-convex and non-differentiable.

We take a different route to construct a classifier, which we refer to as Bayes' classifier. This construction is based on a simple probabilistic model for the datapoints. Using this model, we can interpret the average 0/1 loss on training data as an approximation for the probability $P_{\rm err} = \mathrm{P}(y \neq h(\mathbf{x}))$.

An important subclass of Bayes' classifiers uses the hypothesis space (3.1) which is also underlying logistic regression (see Section 3.6) and the SVM (see Section 3.7). Logistic regression, the SVM and Bayes' classifiers are different instances of linear classifiers (see Figure 2.8).

Linear classifiers partition the feature space $\mathcal{X}$ into two half-spaces. One half-space

consists of all feature vectors $\mathbf{x}$ which result in the predicted label $\hat{y} = 1$ and the other half-space constituted by all feature vectors $\mathbf{x}$ which result in the predicted label $\hat{y} = -1$. The difference between these three linear classifiers is how they choose these half-spaces by using different loss functions. We will discuss Bayes' classifier methods in more detail in Section 4.5.

3.9 Kernel Methods

Consider a ML (classification or regression) problem with an underlying feature space $\mathcal{X}$. In order to predict the label $y \in \mathcal{Y}$ of a datapoint based on its features $\mathbf{x} \in \mathcal{X}$, we apply a predictor h selected out of some hypothesis space $\mathcal{H}$. Let us assume that the available computational infrastructure only allows to use a linear hypothesis space $\mathcal{H}^{(n)}$ (see (3.1)).

For some applications, using a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ is not suitable since the relation between features $\mathbf{x}$ and label y might be highly non-linear. One approach to extend the capabilities of linear hypotheses is to transform the raw features of a data point before applying a linear hypothesis h.

The family of kernel methods is based on transforming the features $\mathbf{x}$ to new features $\hat{\mathbf{x}} \in \mathcal{X}'$ which belong to a (typically very) high-dimensional space $\mathcal{X}'$ [44]. It is not uncommon that, while the original feature space is a low-dimensional Euclidean space (e.g., $\mathcal{X} = \mathbb{R}^2$), the transformed feature space $\mathcal{X}'$ is an infinite-dimensional function space.

The rationale behind transforming the original features into a new (higher-dimensional) feature space $\mathcal{X}'$ is to reshape the intrinsic geometry of the feature vectors $\mathbf{x}^{(i)} \in \mathcal{X}$ such that the transformed feature vectors $\hat{\mathbf{x}}^{(i)}$ have a "simpler" geometry (see Figure 3.5).

Kernel methods are obtained by formulating ML problems (such as linear regression or logistic regression) using the transformed features $\hat{\mathbf{x}} = \phi(\mathbf{x})$. A key challenge within kernel methods is the choice of the feature map $\phi : \mathcal{X} \to \mathcal{X}'$ which maps the original feature vector $\mathbf{x}$ to a new feature vector $\hat{\mathbf{x}} = \phi(\mathbf{x})$.

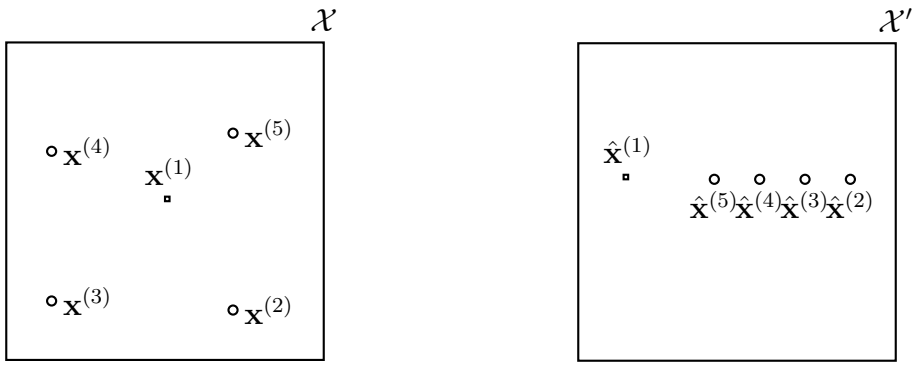


Figure 3.5: Consider a data set $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{5}$ constituted by datapoints with features $\mathbf{x}^{(i)}$ and binary labels $y^{(i)}$. Left: In the original feature space $\mathcal{X}$, the datapoints cannot be separated perfectly by any linear classifier. Right: The feature map $\phi : \mathcal{X} \to \mathcal{X}'$ transforms the features $\mathbf{x}^{(i)}$ to the new features $\hat{\mathbf{x}}^{(i)} = \phi\big(\mathbf{x}^{(i)}\big)$ in the new feature space $\mathcal{X}'$. In the new feature space $\mathcal{X}'$ the datapoints can be separated perfectly by a linear classifier.

3.10 Decision Trees

A decision tree is a flowchart-like description of a map $h : \mathcal{X} \to \mathcal{Y}$ which maps the features $\mathbf{x} \in \mathcal{X}$ of a datapoint to a predicted label $h(\mathbf{x}) \in \mathcal{Y}$ [31].

While decision trees can be used for arbitrary feature space $\mathcal{X}$ and label space $\mathcal{Y}$, we will discuss them for the particular feature space $\mathcal{X} = \mathbb{R}^2$ and label space $\mathcal{Y} = \mathbb{R}$.

We have depicted an example of a decision tree in Figure 3.6. The decision tree consists of nodes which are connected by directed edges. We might think of a decision tree as a step-by-step instruction, or a "recipe", for how to compute the function value $h(\mathbf{x})$ given the features $\mathbf{x} \in \mathcal{X}$ of a datapoint. This computation starts at the **root node** and ends at one of the **leaf nodes** of the decision tree.

A leaf node m, which does not have any outgoing edges, represents a decision region $\mathcal{R}_m \subseteq \mathcal{X}$ in the feature space. The hypothesis h associated with a decision tree is constant over the regions $\mathcal{R}_m$, such that $h(\mathbf{x}) = h_m$ for all $\mathbf{x} \in \mathcal{R}_m$ and some fixed number $h_m \in \mathbb{R}$.

In general, there are two types of nodes in a decision tree:

- decision (or test) nodes, which represent particular "tests" about the feature vector $\mathbf{x}$ (e.g., "is the norm of $\mathbf{x}$ larger than 10?").
- leaf nodes, which correspond to subsets of the feature space.

The particular decision tree depicted in Figure 3.6 consists of two decision nodes (including the root node) and three leaf nodes.

Given limited computational resources, we can only use decision trees which are not too deep. Consider the hypothesis space consisting of all decision trees which uses the tests "$\|\mathbf{x}-\mathbf{u}\| \leq r$" and "$\|\mathbf{x}-\mathbf{v}\| \leq r$" , with some vectors $\mathbf{u}$ and $\mathbf{v}$, some positive radius $r > 0$ and depth no larger than 2.[3]

To assess the quality of different decision trees we need to use some loss function. Examples of loss functions used to measure the quality of a decision tree are the squared error loss (for numeric labels) or the impurity of individual decision regressions (for discrete labels).

In general, we are not interested in one particular decision tree only but in a large set of different decision trees from which we choose the most suitable given some data (see Section 4.4). We can define a hypothesis space by collecting predictor maps h represented by a set of decision trees (such as depicted in Figure 3.7).

A collection of decision trees can be constructed based on a fixed set of "elementary tests" on the input feature vector, e.g., $\|\mathbf{x}\| > 3$, $x_3 < 1$ or a continuous ensemble of parametrized tests such as $\{x_2 > \eta\}_{\eta \in [0,10]}$. We then build a hypothesis space by considering all decision trees not exceeding a maximum depth and whose decision nodes carry out elementary tests.

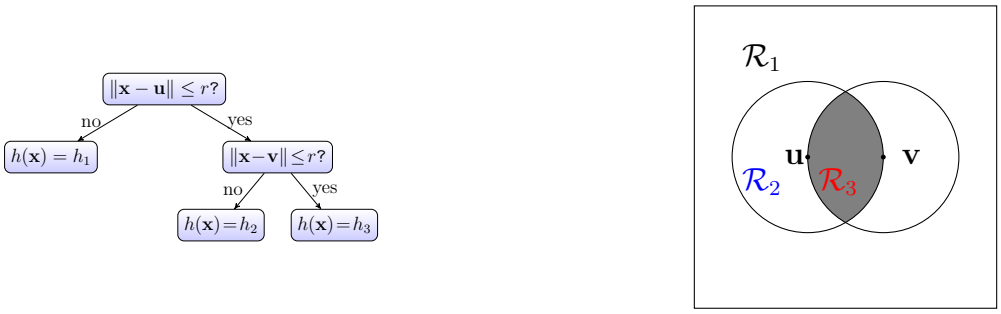


Figure 3.6: A decision tree represents a hypothesis h which is constant on subsets $\mathcal{R}_m$, i.e., $h(\mathbf{x}) = h_m$ for all $\mathbf{x} \in \mathcal{R}_m$. Each subset $\mathcal{R}_m \subseteq \mathcal{X}$ corresponds to a leaf node in the decision tree.

A decision tree represents a map $h : \mathcal{X} \to \mathcal{Y}$, which is piecewise-constant over regions of the feature space $\mathcal{X}$. These non-overlapping regions form a partitioning of the feature space. Each leaf node of a decision tree corresponds to one particular region. Using large decision trees, which involve many different test nodes, we can represent very complicated partitions that resemble any given labeled dataset (see Figure 3.8).

This is quite different from ML methods using the linear hypothesis space (3.1), such as

[3]The depth of a decision tree is the maximum number of hops it takes to reach a leaf node starting from the root and following the arrows. The decision tree depicted in Figure 3.6 has depth 2.

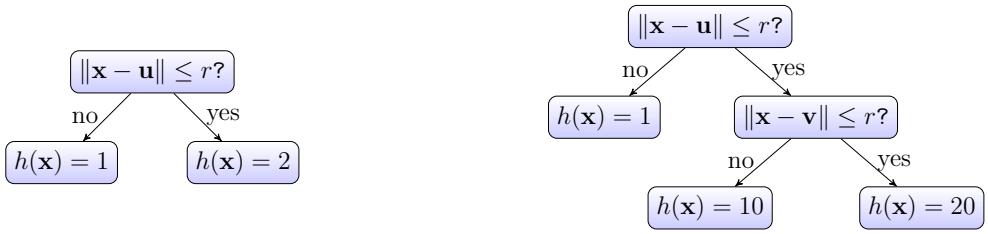


Figure 3.7: A hypothesis space $\mathcal{H}$ consisting of two decision trees with depth at most 2 and using the tests $\|\mathbf{x}-\mathbf{u}\| \leq r$ and $\|\mathbf{x}-\mathbf{v}\| \leq r$ with a fixed radius r and vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$.

linear regression, logistic regression or SVM. Such linear maps have a rather simple geometry. Indeed, a linear map is constant along hyperplanes. Moreover, the decision regions obtained from linear classifiers are always entire half-spaces (see Figure 2.8).

In contrast, the shape of a map represented by a decision tree can be highly complicated. Using a sufficiently large (deep) decision tree, we can obtain a map that closely resembles almost any given non-linear map. The decision regions obtained from a deep decision tree can be highly irregular.

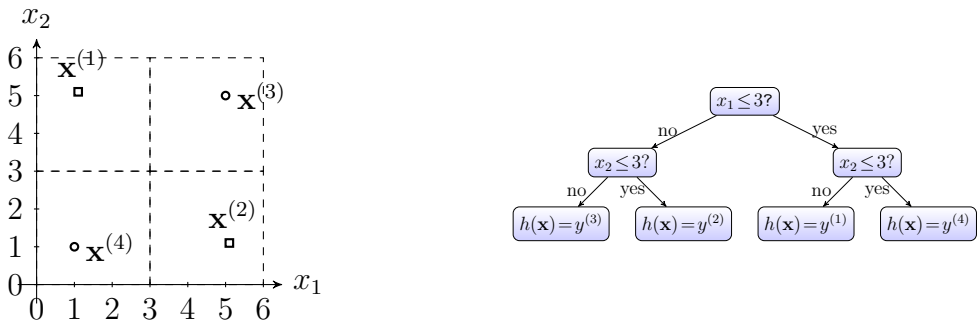


Figure 3.8: Using a sufficiently large (deep) decision tree, we can construct a map h that perfectly fits any given labeled dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ such that $h(\mathbf{x}^{(i)}) = y^{(i)}$ for $i = 1, \ldots, m$.

3.11 Artificial Neural Networks – Deep Learning

Another example of a hypothesis space, which has proven useful in a wide range of applications, e.g., image captioning or automated translation, is based on a **network representation** of a predictor $h : \mathbb{R}^n \to \mathbb{R}$. We can define a predictor $h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R}$ using an **artificial neural network** (ANN) structure as depicted in Figure 3.9. A feature vector $\mathbf{x} \in \mathbb{R}^n$ is

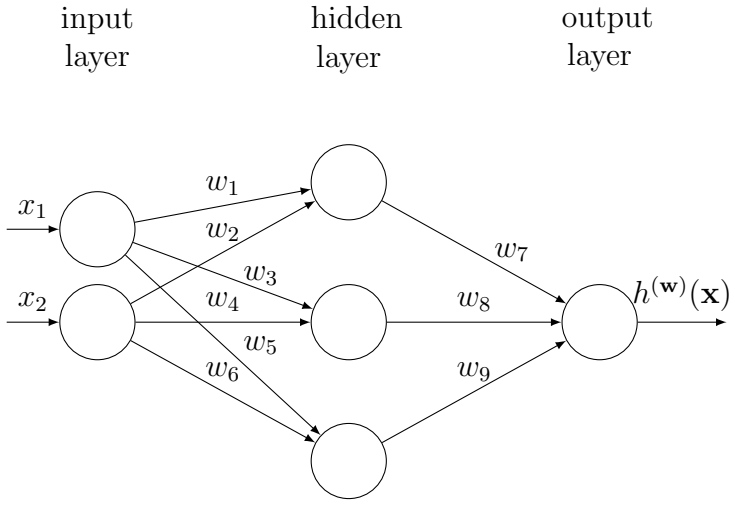


Figure 3.9: ANN representation of a predictor $h^{(\mathbf{w})}(\mathbf{x})$ which maps the input (feature) vector $\mathbf{x} = (x_1, x_2)^T$ to a predicted label (output) $h^{(\mathbf{w})}(\mathbf{x})$.

fed into the input units, each of which reads in one single feature $x_i \in \mathbb{R}$. The features x_i are then multiplied with the weights $w_{j,i}$ associated with the link between the i-th input node ("neuron") with the j-th node in the middle (hidden) layer. The output of the j-th node in the hidden layer is given by $s_j = g(\sum_{i=1}^{n} w_{j,i} x_i)$ with some (typically non-linear) **activation function** $g(z)$. The input (or activation) z for the activation (or output) $g(z)$ of a neuron is a weighted (linear) combination $\sum_{i=1}^{n} w_{j,i} s_i$ of the outputs s_i of the nodes in the previous layer. For the ANN depicted in Figure 3.9, the activation of the neuron s_1 is $z = w_{1,1}x_1 + w_{1,2}x_2$.

Two popular choices for the activation function used within ANNs are the **sigmoid function** $g(z) = \frac{1}{1+\exp(-z)}$ or the **rectified linear unit** $g(z) = \max\{0, z\}$. An ANN with many, say 10, hidden layers, is often referred to as a **deep neural network** and the obtained ML methods are known as **deep learning** methods (see [27] for an in-depth introduction to deep learning methods).

Remarkably, using some simple non-linear activation function $g(z)$ as the building block for ANNs allows to represent an extremely large class of predictor maps $h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R}$. The hypothesis space generated by a given ANN structure, i.e., the set of all predictor maps which can be implemented by a given ANN and suitable weights $\mathbf{w}$, tends to be much larger than the hypothesis space (2.4) of linear predictors using weight vectors $\mathbf{w}$ of the same length [27, Ch. 6.4.1.]. It can be shown that an ANN with only one single hidden layer can approximate any given map $h : \mathcal{X} \to \mathcal{Y} = \mathbb{R}$ to any desired accuracy [20]. However, a key insight which underlies many deep learning methods is that using several layers with few neurons, instead of one single layer containing many neurons, is computationally favourable [22].

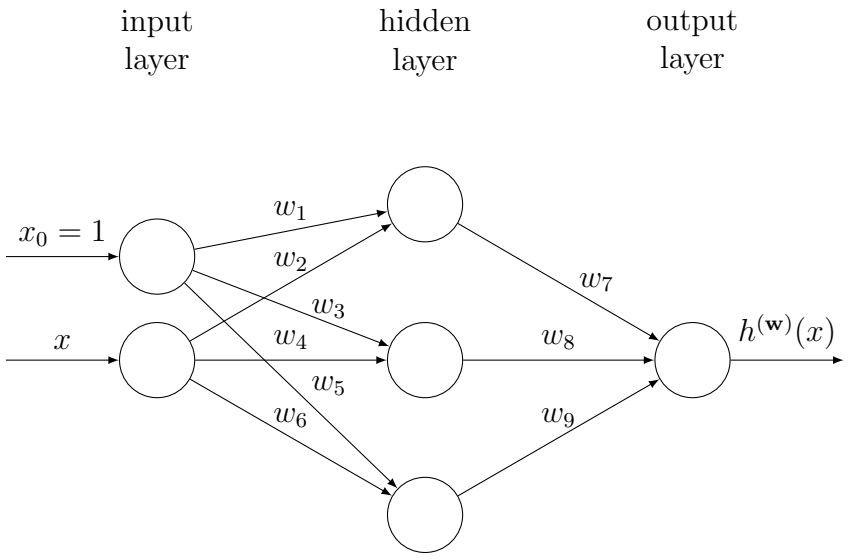


Figure 3.10: This ANN with one hidden layer defines a hypothesis space consisting of all maps $h^{(\mathbf{w})}(x)$ obtained by implementing the ANN with different weight vectors $\mathbf{w} = (w_1, \ldots, w_9)^T$.

Exercise. Consider the simple ANN structure in Figure 3.10 using the "ReLu" activation function $g(z) = \max\{z, 0\}$ (see Figure 3.11). Show that there is a particular choice for the weights $\mathbf{w} = (w_1, \ldots, w_9)^T$ such that the resulting hypothesis map $h^{(\mathbf{w})}(x)$ is a triangle as depicted in Figure 3.12. Can you also find a choice for the weights $\mathbf{w} = (w_1, \ldots, w_9)^T$ that produce the same triangle shape if we replace the ReLu activation function with the linear function $g(z) = 10 \cdot z$?

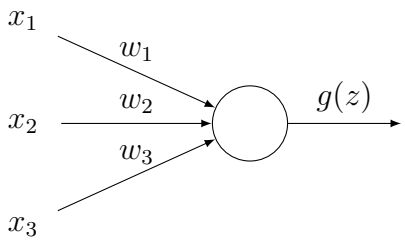


Figure 3.11: Each single neuron of the ANN depicted in Figure 3.10 implements a weighted summation $z = \sum_i w_i x_i$ of its inputs x_i followed by applying a non-linear activation function $g(z)$.

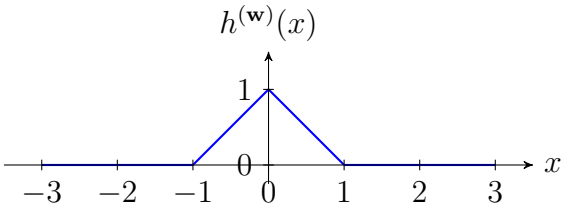


Figure 3.12: A hypothesis map with the shape of a triangle.

The recent success of ML methods based on ANN with many hidden layers (which makes them deep) might be attributed to the fact that the network representation of hypothesis maps is beneficial for the computational implementation of ML methods. First, we can evaluate a map $h^{(\mathbf{w})}$ represented by an ANN efficiently using modern parallel and distributed computing infrastructure via message passing over the network. Second, the graphical representation of a parametrized hypothesis in the form of a ANN allows to efficiently compute the gradient of the loss function via a message passing procedure known as **back-propagation** [27].

3.12 Maximum Likelihood Methods

For many applications it is useful to model the observed datapoints $\mathbf{z}^{(i)}$ as realizations of a random variable $\mathbf{z}$ with probability distribution $\mathrm{P}(\mathbf{z};\mathbf{w})$ which depends on some parameter vector $\mathbf{w} \in \mathbb{R}^n$. A principled approach to estimating the vector $\mathbf{w}$ based on several independent and identically distributed (i.i.d.) realizations $\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(m)} \sim \mathrm{P}(\mathbf{z};\mathbf{w})$ is **maximum likelihood estimation** [47].

Maximum likelihood estimation can be interpreted as an ML problem with a hypothesis space parameterized by the weight vector $\mathbf{w}$, i.e., each element $h^{(\mathbf{w})}$ of the hypothesis space $\mathcal{H}$ corresponds to one particular choice for the weight vector $\mathbf{w}$, and loss function

$$\mathcal{L}(\mathbf{z}, h^{(\mathbf{w})}) := -\log \mathrm{P}(\mathbf{z};\mathbf{w}). \tag{3.22}$$

A widely used choice for the probability distribution $\mathrm{P}(\mathbf{z};\mathbf{w})$ is a multivariate normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, both of which constitute the weight vector $\mathbf{w} = (\boldsymbol{\mu}, \boldsymbol{\Sigma})$ (we have to reshape the matrix $\boldsymbol{\Sigma}$ suitably into a vector form). Given the i.i.d. realizations $\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(m)} \sim \mathrm{P}(\mathbf{z};\mathbf{w})$, the maximum likelihood estimates $\hat{\boldsymbol{\mu}}$, $\widehat{\boldsymbol{\Sigma}}$ of the mean vector and the covariance matrix are obtained via

$$\hat{\boldsymbol{\mu}}, \widehat{\boldsymbol{\Sigma}} = \operatorname*{argmin}_{\boldsymbol{\mu}\in\mathbb{R}^n, \boldsymbol{\Sigma}\in\mathbb{S}^n_+} (1/m) \sum_{i=1}^{m} -\log \mathrm{P}(\mathbf{z}^{(i)}; (\boldsymbol{\mu}, \boldsymbol{\Sigma})). \tag{3.23}$$

The optimization in (3.23) is over choices for the mean vector $\boldsymbol{\mu} \in \mathbb{R}^n$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{S}^n_+$. Here, $\mathbb{S}^n_+$ denotes the set of all psd Hermitian $n \times n$ matrices.

The maximum likelihood problem (3.23) can be interpreted as an instance of ERM (4.2)

using the particular loss function (3.22). The resulting estimates are given explicitly as

$$\hat{\boldsymbol{\mu}} = (1/m)\sum_{i=1}^{m}\mathbf{z}^{(i)}, \text{ and } \widehat{\boldsymbol{\Sigma}} = (1/m)\sum_{i=1}^{m}(\mathbf{z}^{(i)}-\hat{\boldsymbol{\mu}})(\mathbf{z}^{(i)}-\hat{\boldsymbol{\mu}})^{T}. \tag{3.24}$$

Note that the expressions (3.24) are valid only when the probability distribution of the datapoints is modelled as a multivariate normal distribution.

3.13 Nearest Neighbour Methods

The class of k-nearest neighbour (k-NN) predictors (for continuous label space) or classifiers (for discrete label space) is defined for feature spaces $\mathcal{X}$ equipped with an intrinsic notion of distance between its elements. Mathematically, such spaces are referred to as metric spaces [61]. A prime example of a metric space is $\mathbb{R}^n$ with the Euclidean metric induced by the distance measure $\|\mathbf{x}-\mathbf{y}\|$ between two vectors $\mathbf{x},\mathbf{y}\in\mathbb{R}^n$.

The hypothesis space underlying k-NN problems consists of all maps $h:\mathcal{X}\rightarrow\mathcal{Y}$ such that the function value $h(\mathbf{x})$ for a particular feature vector $\mathbf{x}$ depends only on the (labels of the) k nearest datapoints of some labeled training data $\mathcal{D}=\{(\mathbf{x}^{(i)},y^{(i)})\}_{i=1}^{m}$.

In contrast to the ML problems discussed above in Section 3.1 - Section 3.11, the hypothesis space of k-NN depends on the training data $\mathcal{D}$.

3.14 Dimensionality Reduction

datapoints are whole datasets (bunch of datapoint); label is optimal hyperplane that allows for optimal dimensionality reduction by projecting onto it; the notion of optimality depends on the application at hand; one notion of optimality is obtained from approximation errors (PCA).

3.15 Clustering Methods

each datapoint is an entire dataset of lower-level datapoints; labels are correct partitioning/clustering; loss function is some notion of purity of clustering error; discuss partitional vs. hierarchical clustering (different choice for hypospace?)

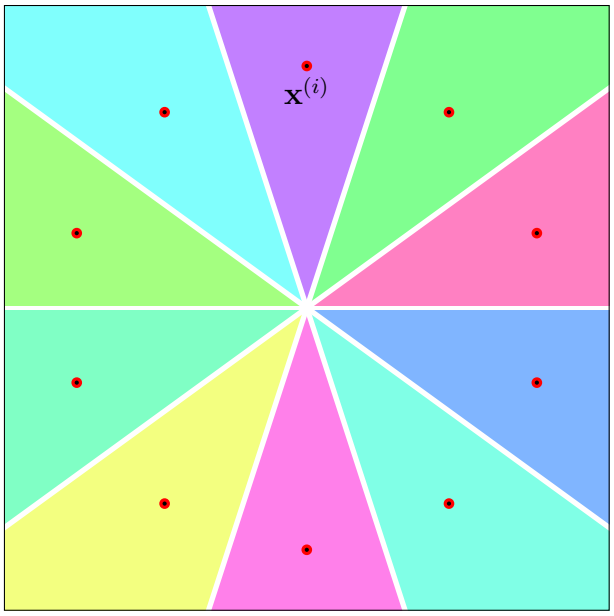


Figure 3.13: A hypothesis map h for k-NN with $k = 1$ and feature space $\mathcal{X} = \mathbb{R}^2$. The hypothesis map is constant over regions (indicated by the coloured areas) located around feature vectors $\mathbf{x}^{(i)}$ (indicated by a dot) of a dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}$.

3.16 Deep Reinforcement Learning

datapoints are the states of some (AI) agent characterized by features (sensor readings); labels are optimal actions; however we typically have no access to labeled data as we cannot try out each and any sequence of actions and such to find out the best action in each situation. instead we must construct the loss function via a (negative) reward collected over time (e.g. over an episode);

3.17 LinUCB

datapoints are customers characterized by feature vector; the label is discrete and indicates which product out of a finite set of products should be advertised to the customer;

3.18 Network Lasso

Maybe the most widely used choice for the feature space $\mathcal{X}$ in ML methods is the Euclidean space $\mathbb{R}^n$. If the features of the datapoints are available in numeric form it is quite natural

to stack them into feature vectors. But even for non-numerical data such as text it is often preferable to transform it to numeric features (word-embedding). The feature space $\mathbb{R}^n$ is attractive since it has a rich algebraic and geometric structure which allows to navigate (search) it efficiently.

A recent thread in ML is to use feature spaces whose structure better reflects the structure of non-Euclidean data. One example of non-Euclidean data is network-structured data where individual datapoints are related by some application-specific notion of similarity. For such data it might be useful to use as a feature space a graph whose nodes represent individual datapoints. Similar datapoints are connected by an edge.

Partially labeled network-structured data arises in many important application domains including signal processing [19, 18], image processing [48, 63], social networks, internet and bioinformatics [56, 17, 23]. Networked data can be described by an "empirical graph" $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ such as those depicted in Figure 3.14.

The nodes $\mathcal{V}$ of the empirical graph represent individual datapoints. Datapoints are connected by edges $\mathcal{E}$ if they are considered "similar" in an application-specific sense. We encode the extent of similarity between connected nodes $i, j \in \mathcal{V}$ using the edge weight $W_{i,j} > 0$. The edge weights are the entries of weight matrix $\mathbf{W} \in \mathbb{R}_{+}^{|\mathcal{V}| \times |\mathcal{V}|}$.

The notion of similarity between datapoints can be based on physical proximity (in time or space), communication networks or probabilistic graphical models [46, 10, 42].

Besides the graph structure, datasets carry additional information in the form of labels associated with individual datapoints. In a social network, we might define the personal preference for some product as the label associated with a datapoint (which represents a user profile).

Acquiring labels is often costly and requires manual labor or experiment design. Therefore, we assume to have access to the labels of only a few datapoints which belong to a small "training set".

The availability of accurate network models for datasets provides computational and statistical benefits. Computationally, network models lend naturally to highly scalable ML methods which can be implemented as message passing over the empirical graph [11].

Network models borrow statistical strength between connected data points, which allows semi-supervised learning (SSL) methods to capitalize on massive amounts of unlabeled data [17].

The key idea behind many SSL methods is the assumption that labels of close-by datapoints are similar, which allows to combine partially labeled data with its network structure in order

to obtain predictors which generalize well [17, 6]. While SSL methods on graphs have been applied to many application domains, the precise understanding of which type of data allow for accurate SSL is still in its infancy [76, 54, 1].

Besides the empirical graph structure $\mathcal{G}$, a dataset typically conveys additional information, e.g., features, labels or model parameters. We can represent this additional information by a graph signal defined over $\mathcal{G}$. A graph signal $h[\cdot]$ is a map $\mathcal{V} \rightarrow \mathbb{R}$, which associates every node $i \in \mathcal{V}$ with the signal value $h[i] \in \mathbb{R}$.

The graph signals arising in several important application domains can be well modelled using a cluster assumption [17]. The cluster assumption requires similar signal values $h[i] \approx h[j]$ at nodes $i, j \in \mathcal{V}$, which belong to the same well-connected subset of nodes ("cluster") of the empirical graph. The clusteredness of a graph signal $h[\cdot]$ can be measured by the total variation (TV):

$$\|h\|_{\rm TV} = \sum_{\{i,j\} \in \mathcal{E}} W_{i,j} |h(i) - h(j)|. \tag{3.25}$$

Clustered graph signals arise in digital signal processing which studies graph signals defined over the chain graph representing discrete time instants.

The signal values at adjacent time instants are correlated for sufficiently high sampling rate. Image processing methods use the tendency of close-by pixels to be coloured likely which amounts to a clustered graph signal over a grid graph representing pixels of a 2D image.

The recently introduced network Lasso (nLasso) amounts to a formal ML problem involving network-structured data which can be represented by an empirical graph $\mathcal{G}$. In particular, the hypothesis space of nLasso is constituted by graph signals on $\mathcal{G}$:

$$\mathcal{H} = \{h : \mathcal{V} \rightarrow \mathcal{Y}\}. \tag{3.26}$$

The loss function of nLasso is a combination of squared error and TV (see (3.25))

$$\mathcal{L}((\mathbf{x}, y), h) = (y - h(\mathbf{x}))^2 + \lambda \|h\|_{\rm TV}. \tag{3.27}$$

The regularization parameter λ allows to trade-off a small prediction error $y - h(\mathbf{x})$ against "clusteredness" of the predictor h.

Logistic Network Lasso. The logistic network Lasso [2, 3] is a modification of the network Lasso (see Section 3.18) for classification problems involving partially labeled networked data represented by an empirical graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$.

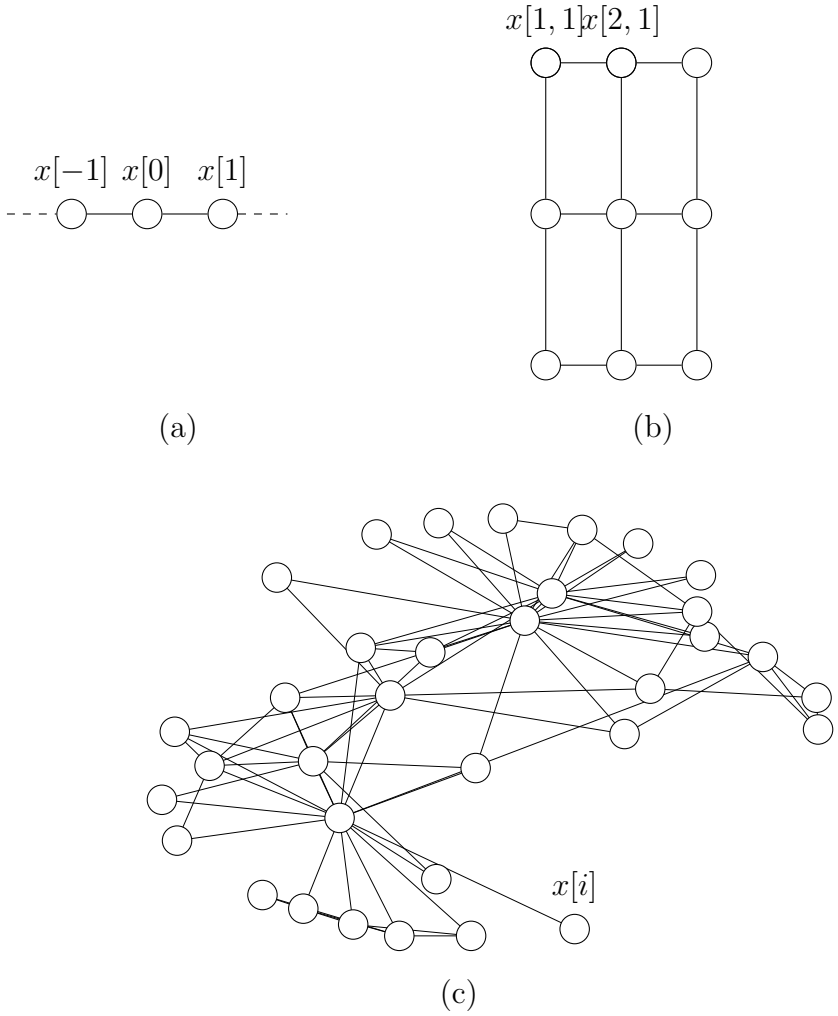


Figure 3.14: Examples for the empirical graph of networked data. (a) Chain graph representing signal amplitudes of discrete time signals. (b) Grid graph representing pixels of 2D-images. (c) Empirical graph $\mathcal{G}=(\mathcal{V},\mathcal{E},\mathbf{W})$ for a dataset obtained from the social relations between members of a Karate club [78]. The empirical graph with m nodes $i\in\mathcal{V}=\{1,\ldots,m\}$ represents m individual club members. Two nodes $i,j\in\mathcal{V}$ are connected by an edge $\{i,j\}\in\mathcal{E}$ if the corresponding club members have interacted outside the club.

Each datapoint $\mathbf{z}$ is characterized by the features $\mathbf{x}$ and is associated with a label $y \in \mathcal{Y}$, taking on values from a discrete label space $\mathcal{Y}$. The simplest setting is binary classification where each datapoint has a binary label $y \in \{-1, 1\}$. The hypothesis space underlying logistic network Lasso is given by the graph signals on the empirical graph:

$$\mathcal{H} = \{h : \mathcal{V} \to \mathcal{Y}\} \tag{3.28}$$

and the loss function is a combination of logistic loss and TV (see (3.25))

$$\mathcal{L}((\mathbf{x}, y), h) = -\log(1 + \exp(-yh(\mathbf{x}))) + \lambda \|h\|_{\mathrm{TV}}. \tag{3.29}$$

3.19 Exercises

3.19.1 How Many Neurons?

Consider a predictor map $h(x)$ which is piece-wise linear and consisting of 1000 pieces. Assume we want to represent this map by an ANN using neurons with ReLU activation functions. How many neurons must the ANN at least contain?

3.19.2 Linear Classifiers

Consider datapoints characterized by feature vectors $\mathbf{x} \in \mathbb{R}^n$ and binary labels $y \in \{-1, 1\}$. We are interested in finding a good linear classifier which is such that the feature vectors resulting in $h(\mathbf{x}) = 1$ is a half-space. Which of the methods discussed in this chapter aim at learning a linear classifier?

3.19.3 Data Dependent Hypothesis Space

Which of the following ML methods uses a hypothesis space that depends on the training data.

- logistic regression
- linear regression
- k-NN

Chapter 4

Empirical Risk Minimization

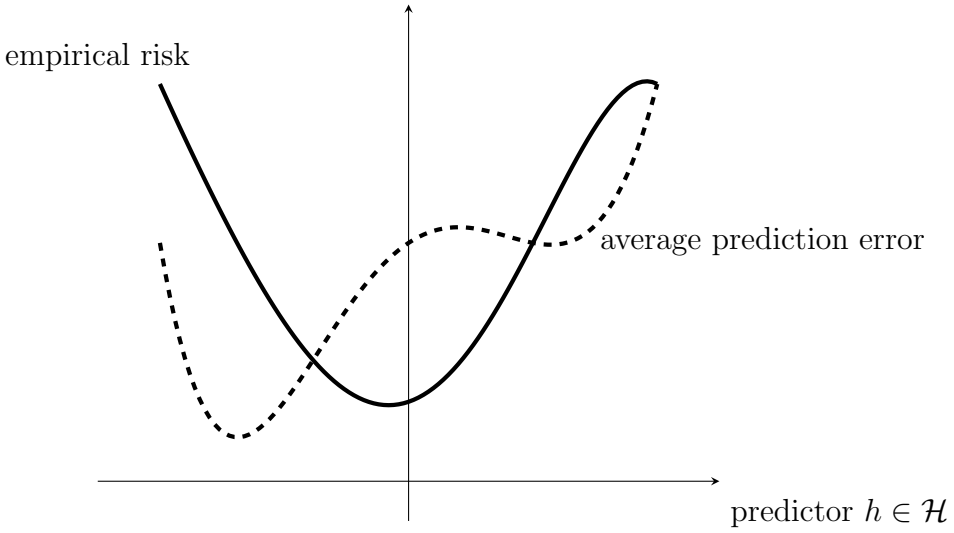


Figure 4.1: ML methods aim at learning a predictor $h \in \mathcal{H}$ that incurs small loss on any datapoint. Empirical risk minimization approximates the expected loss or risk with the empirical risk (solid curve) incurred on a finite set of labeled datapoints (the training set).

Chapter 2 explained three components of ML (see Figure 2.1):

- the feature space $\mathcal{X}$ and label space $\mathcal{Y}$,
- a hypothesis space $\mathcal{H}$ of computationally feasible predictor maps $\mathcal{X} \to \mathcal{Y}$,
- and a loss function $\mathcal{L}((\mathbf{x}, y), h)$ which measures the discrepancy between the predicted label $h(\mathbf{x})$ and the true label y of a datapoint. *error* incurred by predictor $h \in \mathcal{H}$.

Ideally we would like to learn a hypothesis h out of the model $\mathcal{H}$ such that $h(\mathbf{x}) \approx y$ or, in turn, $\mathcal{L}((\mathbf{x}, y), h)$ is very small, for any datapoint $(\mathbf{x}, y)$. However, in practice we can only use a given set of labeled datapoints (the training set) to measure the average loss of a hypothesis h.

How can we know the loss of a hypothesis h when predicting the label of datapoints outside the training set? One possible approach is to use a **probabilistic model** for the data. In this model, we interpret datapoints as realizations of i.i.d. random variables with the (same) probability distribution $p(\mathbf{x}, y)$. The training set is one particular set of such realizations drawn from $p(\mathbf{x}, y)$. Moreover, we can generate datapoints outside the training set by drawing realizations from the distribution $p(\mathbf{x}, y)$. Given this probability distribution over different realizations of datapoints allows to define the risk of a hypothesis h as the expectation of the loss incurred by h on a random datapoint.

If we would know the probability distribution $p(\mathbf{x}, y)$, from which the datapoints are drawn, we could minimize the risk using probability theory. Roughly speaking, the optimal hypothesis h can be read directly from the posterior probability distribution $p(y|\mathbf{x})$ of the label y given the features $\mathbf{x}$ of a datapoint. When using the squared error loss, the risk is minimized by the hypothesis $h(\mathbf{x}) = \mathbb{E}\{y|\mathbf{x}\}$.

In practice we do not know the true underlying probability distribution and have to estimate it from data. Therefore, we cannot compute the Bayes' optimal estimator exactly. However, we can approximately compute this estimator by replacing the exact probability distribution with an estimate. Moreover, the risk of the Bayes' optimal estimator provides a useful benchmark against which we can compare the average loss of practical ML methods.

Section (4.1) formally defines the risk of a hypothesis and motivates empirical risk minimization (ERM) as a natural approximation of the risk using labeled (training) datapoints. We then specialize the ERM for three particular ML problems. These three ML problems use different combinations of model (hypothesis space) and loss functions which result in ERM with different computational complexities.

In Section 4.3, we discuss the ERM obtained for linear regression (see Section 3.1). The resulting ERM amounts to minimizing a differentiable convex function, which can be done efficiently using gradient-based methods (see Chapter 5).

We then discuss in Section 4.4 the ERM obtained for decision tree models. The resulting ERM becomes a discrete optimization problem which are typically much harder than convex optimization problems. We cannot apply gradient-based methods to solve the ERM for decision trees. To solve the decision tree ERM we essentially must try out all possible

choices for the tree structure [].

Section 4.5 considers the ERM obtained when learning a linear hypothesis using the 0/1 loss for classification problems. The resulting ERM amounts to minimizing a non-differentiable and non-convex function. Instead of using computationally expensive methods for minimizing this function, we will use a different rout via probability theory to construct approximate solutions to this ERM instance.

As explained in Section 4.6, many ML methods use the ERM during a training period to learn a hypothesis which is then applied to new datapoints during the inference period. Section 4.7 demonstrates how an online learning method can be obtained by solving the ERM sequentially as new datapoints come in. Online learning methods continuously alternate between training and inference periods.

4.1 Why Empirical Risk Minimization?

We assume that datapoints are i.i.d. realizations drawn from some fixed probability distribution $p(\mathbf{x}, y)$. The probability distribution $p(\mathbf{x}, y)$ allows us to define the **expected loss** or **risk**

$$\mathbb{E}\big\{\mathcal{L}((\mathbf{x}, y), h)\big\}. \tag{4.1}$$

Many ML methods learn a predictor out of $\mathcal{H}$ such that (4.1) is minimal.

If we would know the probability distribution of the data, we could in principle readily determine the best predictor map by solving an optimization problem. This optimal predictor is known as the Bayes' predictor and depends on the probability distribution $p(\mathbf{x}, y)$ and the loss function. For the squared error loss, the Bayes' predictor is the posterior mean of y given the features $\mathbf{x}$.

We often do not know the probability distribution and therefore cannot evaluate the expectation in (4.1). **ERM** replaces the expectation in (4.1) with an average over a given set of labeled datapoints,

$$\begin{aligned} \hat{h} &= \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{E}(h|\mathcal{D}) \\ &\overset{(2.12)}{=} \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h). \end{aligned} \tag{4.2}$$

The ERM (4.2) amounts to learning (finding) a good predictor $\hat{h} \in \mathcal{H}$ by "training" it on

the dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, which is therefore referred to as the **training set**.

4.2 Computational and Statistical Aspects of ERM

Statistical Aspects: Objective in ERM is Noisy version of Actual Objective; sometimes we do not know most parts of objective function (reinforcement learning); even if we know the objective function (expected risk) perfectly, optimization might be hard for non-smooth, non-convex objective functions

Solving the optimization problem (4.2) provides two things. First, the minimizer $\hat{h}$ is a predictor which performs optimal on the training set $\mathcal{D}$. Second, the corresponding objective value $\mathcal{E}(\hat{h}|\mathcal{D})$ (the "training error") indicates how accurate the predictions of $\hat{h}$ will be.

As we will discuss in Chapter 7, for some datasets $\mathcal{D}$, the training error $\mathcal{E}(\hat{h}|\mathcal{D})$ obtained for $\mathcal{D}$ can be very different from the average prediction error of $\hat{h}$ when applied to new datapoints which are not contained in $\mathcal{D}$.

Many important ML methods use hypotheses that are parametrized by weight vector $\mathbf{w}$. For each possible weight vector, we obtain a hypothesis $h^{(\mathbf{w})}(\mathbf{x})$. Such a parametrization is used in linear regression which learns a linear hypotheses $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ with some weight vector $\mathbf{w}$. Another example for such a parametrization are ANNs with the weights assigned to inputs of individual neurons (see Figure 3.9).

For ML methods that use a parameterized hypothesis $h^{(\mathbf{w})}(\mathbf{x})$, we can reformulate the optimization problem (4.2) as an optimization of the weight vector,

$$\mathbf{w}_{\rm opt} = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} f(\mathbf{w}) \text{ with } f(\mathbf{w}) := \mathcal{E}\big(h^{(\mathbf{w})}|\mathcal{D}\big). \tag{4.3}$$

The objective function $f(\mathbf{w})$ in (4.3) is the empirical risk $\mathcal{E}\big(h^{(\mathbf{w})}|\mathcal{D}\big)$ achieved by $h^{(\mathbf{w})}$ when applied to the datapoints in the dataset $\mathcal{D}$.

The optimization problems (4.3) and (4.2) are fully equivalent. Given the optimal weight vector $\mathbf{w}_{\rm opt}$ solving (4.3), the predictor $h^{(\mathbf{w}_{\rm opt})}$ is an optimal predictor solving (4.2).

Learning a hypothesis via ERM (4.2) is a form of learning by "trial and error". An instructor (or supervisor) provides some snapshots $\mathbf{z}^{(i)}$ which are characterized by features $\mathbf{x}^{(i)}$ and associated with known labels $y^{(i)}$.

The learner then uses a hypothesis h to guess the labels $y^{(i)}$ only from the features $\mathbf{x}^{(i)}$ of all training data points. We then determine average loss or training error $\mathcal{E}(h|\mathcal{D})$ that is incurred by the predictions $\hat{y}^{(i)} = h\big(\mathbf{x}^{(i)}\big)$. If the error $\mathcal{E}(h|\mathcal{D})$ is too large, we should try

out another hypothesis map h' different from h with the hope of achieving a smaller training error $\mathcal{E}(h'|\mathcal{D})$.

We highlight that the precise shape of the objective function $f(\mathbf{w})$ in (4.3) depends heavily on the parametrization of the predictor functions, i.e., how does the predictor $h^{(\mathbf{w})}$ vary with the weight vector $\mathbf{w}$.

The shape of $f(\mathbf{w})$ depends also on the choice for the loss function $\mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h)$. As depicted in Figure 4.2, the different combinations of predictor parametrisation and loss functions can result in objective functions with fundamentally different properties such that their optimization is more or less difficult.

The objective function $f(\mathbf{w})$ for the ERM obtained for linear regression (see Section 3.1) is differentiable and convex and can therefore be minimized using simple iterative gradient descent methods (see Chapter 5). In contrast, the objective function $f(\mathbf{w})$ of ERM obtained for the SVM (see Section 3.7) is non-differentiable but still convex. The minimization of such functions is more challenging but still tractable as there exist efficient convex optimization methods which do not require differentiability of the objective function [58].

The objective function $f(\mathbf{w})$ obtained for ANN are typically **highly non-convex** having many local minima. The optimization of non-convex objective function is in general more difficult than optimizing convex objective functions. However, it turns out that despite the non-convexity, iterative gradient-based methods can still be successfully applied to solve the ERM [27]. Even more challenging is the ERM obtained for decision trees or Bayes' classifiers. These ML problems involve non-differentiable and non-convex objective functions.

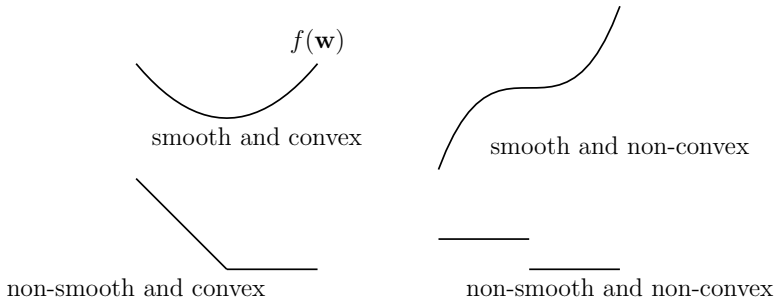


Figure 4.2: Different types of objective functions obtained for ERM in different settings.

4.3 ERM for Linear Regression

As discussed in Section 3.1, linear regression methods learn a linear hypothesis $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ with minimum squared error loss (2.6). For this choices, the ERM problem (4.3) specializes to

$$\mathbf{w}_{\rm opt} = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} f(\mathbf{w}) \\ \text{with } f(\mathbf{w}) := (1/m) \sum_{(\mathbf{x},y)\in\mathcal{D}} (y - \mathbf{x}^T\mathbf{w})^2. \tag{4.4}$$

Here, $m = |\mathcal{D}|$ denotes the (sample) size of the training set $\mathcal{D}$. The objective function $f(\mathbf{w})$ in (4.4) is computationally appealing since it is a convex and smooth function. Such a function can be minimized efficiently using the gradient-based methods discussed in Chapter 5.

The ERM problem (4.4) can rewritten in a more compact form by stacking the labels $y^{(i)}$ and feature vectors $\mathbf{x}^{(i)}$, for $i = 1, \ldots, m$, into a "label vector" $\mathbf{y}$ and "feature matrix" $\mathbf{X}$,

$$\mathbf{y} = (y^{(1)}, \ldots, y^{(m)})^T \in \mathbb{R}^m, \text{ and} \\ \mathbf{X} = \left(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\right)^T \in \mathbb{R}^{m\times n}. \tag{4.5}$$

This allows to rewrite the objective function in (4.4) as

$$f(\mathbf{w}) = (1/m)\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2. \tag{4.6}$$

Inserting (4.6) into (4.4), we obtain the quadratic problem

$$\min_{\mathbf{w}\in\mathbb{R}^n} \underbrace{(1/2)\mathbf{w}^T\mathbf{Q}\mathbf{w} - \mathbf{q}^T\mathbf{w}}_{=f(\mathbf{w})} \\ \text{with } \mathbf{Q} = (1/m)\mathbf{X}^T\mathbf{X}, \mathbf{q} = (1/m)\mathbf{X}^T\mathbf{y}. \tag{4.7}$$

Since $f(\mathbf{w})$ is a differentiable and convex function, a necessary and sufficient condition for $\mathbf{w}_{\rm opt}$ to be a minimizer $f(\mathbf{w}_{\rm opt}) = \min_{\mathbf{w}\in\mathbb{R}^n} f(\mathbf{w})$ is the **zero-gradient condition** [12, Sec. 4.2.3]

$$\nabla f(\mathbf{w}_{\rm opt}) = \mathbf{0}. \tag{4.8}$$

Combining (4.7) with (4.8), yields the following sufficient and necessary condition for a

weight vector $\mathbf{w}_{\rm opt}$ to solve the ERM (4.4),

$$(1/m)\mathbf{X}^T\mathbf{X}\mathbf{w}_{\rm opt} = (1/m)\mathbf{X}^T\mathbf{y}. \tag{4.9}$$

This condition can be rewritten as

$$(1/m)\mathbf{X}^T\big(\mathbf{y} - \mathbf{X}\mathbf{w}_{\rm opt}\big) = \mathbf{0}. \tag{4.10}$$

We might refer to this condition as "normal equations" as they require the vector

$$\big(\mathbf{y} - \mathbf{X}\mathbf{w}_{\rm opt}\big) = \big(\big(y^{(1)} - \hat{y}^{(1)}\big), \ldots, \big(y^{(m)} - \hat{y}^{(m)}\big)\big)^T,$$

whose entries are the prediction errors for the training datapoints, to be orthogonal (or normal) to the subspace spanned by the columns of the feature matrix $\mathbf{X}$.

It can be shown that, for any given feature matrix $\mathbf{X}$ and label vector $\mathbf{y}$, there always exists at least one optimal weight vector $\mathbf{w}_{\rm opt}$ which solves (4.9). The optimal weight vector might not be unique, such that there are several different vectors which achieve the minimum in (4.4). However, every vector $\mathbf{w}_{\rm opt}$ which solves (4.9) achieves the same minimum empirical risk

$$\mathcal{E}(h^{(\mathbf{w}_{\rm opt})} \mid \mathcal{D}) = \min_{\mathbf{w}\in\mathbb{R}^n} \mathcal{E}(h^{(\mathbf{w})} \mid \mathcal{D}) = \|(\mathbf{I} - \mathbf{P})\mathbf{y}\|^2. \tag{4.11}$$

Here, we used the orthogonal projection matrix $\mathbf{P} \in \mathbb{R}^{m\times m}$ on the linear span of the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})^T \in \mathbb{R}^{m\times n}$ (see (4.5)).[1]

If the feature matrix $\mathbf{X}$ (see (4.5)) has full column rank, which implies that the matrix $\mathbf{X}^T\mathbf{X}$ is invertible, the projection matrix $\mathbf{P}$ is given explicitly as

$$\mathbf{P} = \mathbf{X}\big(\mathbf{X}^T\mathbf{X}\big)^{-1}\mathbf{X}^T.$$

Moreover, the solution of (4.9) is then unique and given by

$$\mathbf{w}_{\rm opt} = \big(\mathbf{X}^T\mathbf{X}\big)^{-1}\mathbf{X}^T\mathbf{y}. \tag{4.12}$$

The closed-form solution (4.12) requires the inversion of the $n \times n$ matrix $\mathbf{X}^T\mathbf{X}$.

Computing the inverse of $\mathbf{X}^T\mathbf{X}$ can be computationally challenging for large number n of features. Figure 2.4 depicts a simple ML problem where the number of features is already

[1] The linear span of a matrix $\mathbf{A} = (\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(m)}) \in \mathbb{R}^{n\times m}$, denoted as $\text{span}\{\mathbf{A}\}$, is the subspace of $\mathbb{R}^n$ consisting of all linear combinations of the columns $\mathbf{a}^{(r)} \in \mathbb{R}^n$ of $\mathbf{A}$.

in the millions. The inversion of the matrix $\mathbf{X}^T\mathbf{X}$ is particularly challenging if this matrix is ill-conditioned. In general, we do not have any control on this condition number as we face datapoints with arbitrary feature vectors.

Section 5.4 discusses a method for computing the optimal weight vector $\mathbf{w}_{\rm opt}$ which does not require any matrix inversion. This method, referred to as **gradient descent**, constructs a sequence $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \ldots$ of increasingly accurate approximations of $\mathbf{w}_{\rm opt}$. This iterative method has two major benefits compared to evaluating the formula (4.12) using direct matrix inversion, such as Gauss-Jordan elimination [26]. First, gradient descent requires much fewer arithmetic operations compared to direct matrix inversion. This is crucial in modern ML applications involving large feature matrices. Second, gradient descent does not break when the matrix $\mathbf{X}$ is not full rank and the formula (4.12) cannot be used any more.

4.4 ERM for Decision Trees

Consider the ERM problem (4.2) for a regression problem with label space $\mathcal{Y} = \mathbb{R}$, feature space $\mathcal{X} = \mathbb{R}^n$ and using a hypothesis space defined by decision trees (see Section 3.10).

In stark contrast to the ERM problem obtained for linear or logistic regression, the ERM problem obtained for decision trees amounts to a **discrete optimization problem**. Consider the particular hypothesis space $\mathcal{H}$ depicted in Figure 3.7. This hypothesis space contains a finite number of predictor maps, each map corresponding to a particular decision tree.

For the small hypothesis space $\mathcal{H}$ in Figure 3.7, ERM is easy. Indeed, we just have to evaluate the empirical risk for each of the elements in $\mathcal{H}$ and pick the one yielding the smallest empirical risk. However, for increasing size of decision trees the computational complexity of exactly solving the ERM becomes intractable.

A popular approach to ERM for decision trees is to use greedy algorithms which try to expand (grow) a given decision tree by adding new branches to leaf nodes in order to reduce the empirical risk (see [34, Chapter 8] for more details).

> The idea behind many decision tree learning methods is quite simple: try out expanding a decision tree by replacing a leaf node with a decision node (implementing another "test" on the feature vector) in order to reduce the overall empirical risk as much as possible.

Consider the labeled dataset $\mathcal{D}$ depicted in Figure 4.3 and a given decision tree for predicting the label y based on the features $\mathbf{x}$. We start with a very simple tree shown in the

top of Figure 4.3. Then we try out growing the tree by replacing a leaf node with a decision node. According to Figure 4.3, replacing the right leaf node results in a decision tree which is able to perfectly represent the training dataset (it achieves zero empirical risk).

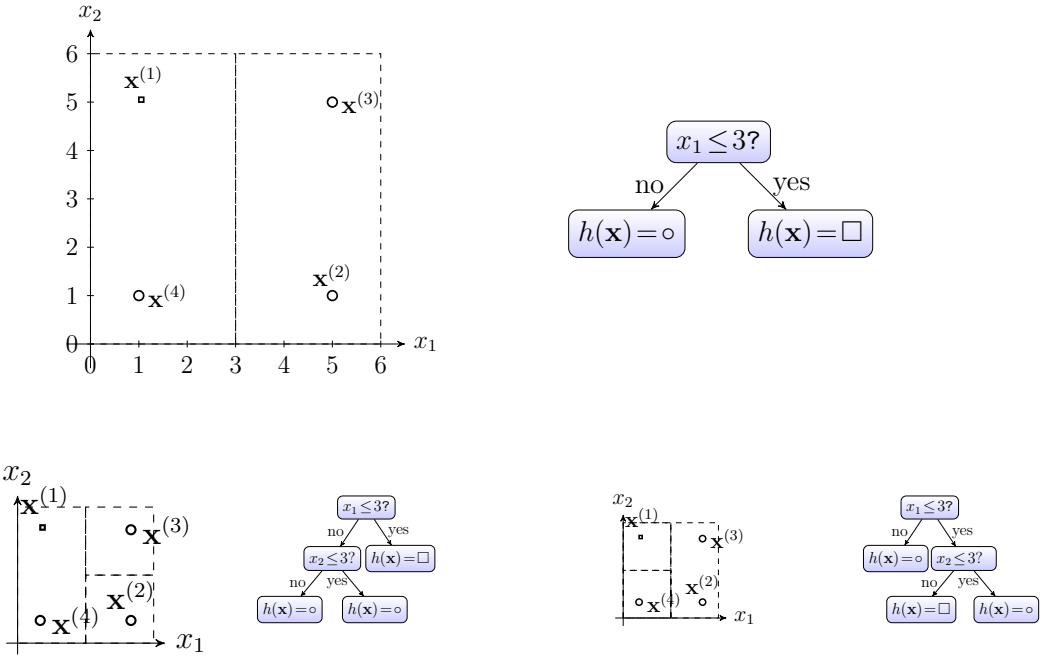


Figure 4.3: Given the labeled dataset and a decision tree in the top row, we grow the decision tree by expanding it at one of its two leaf nodes. The bottom row shows two different decision trees, along with their decision boundaries, obtained by expanding different leaf nodes of the tree in the top row.

One important aspect of learning decision trees from labeled data is the question of when to stop growing. A natural stopping criterion might be obtained from the limitations in computational resources, i.e., we can only afford to use decision trees up to certain maximum depth. Besides the computational limitations, we also face statistical limitations for the maximum size of decision trees. Very large decision trees, which represent highly complicated maps, we might end up overfitting the training data (see Figure 3.8 and Chapter 7) which is detrimental to the prediction performance of decision trees obtained for new data (which has not been used for training or growing the decision tree).

4.5 ERM for Bayes' Classifiers

The family of Bayes' classifiers is based on using the 0/1 loss (2.8) for measuring the quality of a classifier h. The resulting ERM is

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h)$$
$$\stackrel{(2.8)}{=} \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} \mathcal{I}(h(\mathbf{x}^{(i)}) \neq y^{(i)}). \tag{4.13}$$

The objective function in this optimization problem is non-differentiable and non-convex (see Figure 4.2). This prevents us from using gradient-based optimization methods (see Chapter 5) to solve (4.13).

We will now approach the ERM (4.13) via a different route by interpreting the datapoints $(\mathbf{x}^{(i)}, y^{(i)})$ as realizations of i.i.d. random variables which are distributed according to some probability distribution $p(\mathbf{x}, y)$.

As discussed in Section 2.3, the empirical risk obtained using 0/1 loss approximates the error probability $\mathrm{P}(\hat{y} \neq y)$ with the predicted label $\hat{y} = 1$ for $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ otherwise (see (2.9)). Thus, we can approximate the ERM (4.13) as

$$\hat{h} \stackrel{(2.9)}{\approx} \operatorname*{argmin}_{h \in \mathcal{H}} \mathrm{P}(\hat{y} \neq y). \tag{4.14}$$

Note that the hypothesis h, which is the optimization variable in (4.14), enters into the objective function of (4.14) via the definition of the predicted label $\hat{y}$, which is $\hat{y} = 1$ if $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ otherwise.

It turns out that if we would know the probability distribution $p(\mathbf{x}, y)$, which is required to compute $\mathrm{P}(\hat{y} \neq y)$, the solution of (4.14) can be found easily via elementary Bayesian decision theory [59]. In particular, the optimal classifier $h(\mathbf{x})$ is such that $\hat{y}$ achieves the maximum "a-posteriori" probability $p(\hat{y}|\mathbf{x})$ of the label being $\hat{y}$, given (or conditioned on) the features $\mathbf{x}$. However, since we do not know the probability distribution $p(\mathbf{x}, y)$, we have to estimate (or approximate) it from the observed datapoints $(\mathbf{x}^{(i)}, y^{(i)})$ which are modelled as i.i.d. random variables distributed according to $p(\mathbf{x}, y)$.

The estimation of $p(\mathbf{x}, y)$ can be based on a particular probabilistic model for the features and labels which depends on certain parameters and then determining the parameters using maximum likelihood (see Section 3.12). A widely used probabilistic model is based on

Gaussian random vectors. In particular, conditioned on the label y, we model the feature vector $\mathbf{x}$ as a Gaussian vector with mean $\boldsymbol{\mu}_y$ and covariance $\boldsymbol{\Sigma}$, i.e.,

$$p(\mathbf{x}|y) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_y, \boldsymbol{\Sigma}).^2 \tag{4.15}$$

Given (conditioned on) the label y of a data point, the conditional mean of the features $\mathbf{x}$ of this data point is $\boldsymbol{\mu}_1$ if $y = 1$, while for $y = -1$ the conditional mean of $\mathbf{x}$ is $\boldsymbol{\mu}_{-1}$. In contrast, the conditional covariance matrix $\boldsymbol{\Sigma} = \mathbb{E}\{(\mathbf{x} - \boldsymbol{\mu}_y)(\mathbf{x} - \boldsymbol{\mu}_y)^T|y\}$ of $\mathbf{x}$ is the same for both values of the label $y \in \{-1, 1\}$. The conditional probability distribution $p(\mathbf{x}|y)$ of the feature vector, given the label y, is multivariate normal. In contrast, the marginal distribution of the features $\mathbf{x}$ is a Gaussian mixture model (see Section 8.2).

For this probabilistic model of features and labels, the optimal classifier minimizing the error probability $\mathrm{P}(\hat{y} \neq y)$ is $\hat{y} = 1$ for $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ for $h(\mathbf{x}) \leq 0$ using the classifier map

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with } \mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1}). \tag{4.16}$$

Carefully note that this expression is only valid if the matrix $\boldsymbol{\Sigma}$ is invertible.

We cannot implement the classifier (4.16) directly, since we do not know the true values of the class-specific mean vectors $\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_{-1}$ and covariance matrix $\boldsymbol{\Sigma}$. Therefore, we have to replace those unknown parameters with some estimates $\hat{\boldsymbol{\mu}}_1$, $\hat{\boldsymbol{\mu}}_{-1}$ and $\widehat{\boldsymbol{\Sigma}}$. A principled approach is to use the maximum likelihood estimates (see (3.24))

$$\begin{aligned}
\hat{\boldsymbol{\mu}}_1 &= (1/m_1)\sum_{i=1}^{m} \mathcal{I}(y^{(i)} = 1)\mathbf{x}^{(i)}, \\
\hat{\boldsymbol{\mu}}_{-1} &= (1/m_{-1})\sum_{i=1}^{m} \mathcal{I}(y^{(i)} = -1)\mathbf{x}^{(i)}, \\
\hat{\boldsymbol{\mu}} &= (1/m)\sum_{i=1}^{m} \mathbf{x}^{(i)}, \\
\text{and } \widehat{\boldsymbol{\Sigma}} &= (1/m)\sum_{i=1}^{m} (\mathbf{z}^{(i)} - \hat{\boldsymbol{\mu}})(\mathbf{z}^{(i)} - \hat{\boldsymbol{\mu}})^T,
\end{aligned} \tag{4.17}$$

[2] We use the shorthand $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ to denote the probability density function

$$p(\mathbf{x}) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma})}} \exp\big(-(1/2)(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\big)$$

of a Gaussian random vector $\mathbf{x}$ with mean $\boldsymbol{\mu} = \mathbb{E}\{\mathbf{x}\}$ and covariance matrix $\boldsymbol{\Sigma} = \mathbb{E}\big\{(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^T\big\}$.

with $m_1 = \sum_{i=1}^{m} \mathcal{I}(y^{(i)} = 1)$ denoting the number of datapoints with label $y = 1$ (m_{-1} is defined similarly). Inserting the estimates (4.17) into (4.16) yields the implementable classifier

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with } \mathbf{w} = \widehat{\mathbf{\Sigma}}^{-1}(\hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_{-1}). \tag{4.18}$$

We highlight that the classifier (4.18) is only well-defined if the estimated covariance matrix $\widehat{\mathbf{\Sigma}}$ (4.17) is invertible. This requires to use a sufficiently large number of training datapoints such that $m \geq n$.

We derived the classifier (4.18) as an approximate solution to the ERM (4.13). The classifier (4.18) partitions the feature space $\mathbb{R}^n$ into two half-spaces. One half-space consists of feature vectors $\mathbf{x}$ for which the hypothesis (4.18) is non-negative and, in turn, $\hat{y} = 1$. The other half-space is constituted by feature vectors $\mathbf{x}$ for which the hypothesis (4.18) is negative and, in turn, $\hat{y} = -1$. Figure 2.8 illustrates these two half-spaces and the decision boundary between them.

The Bayes' classifier (4.18) is another instance of a linear classifier like logistic regression and the SVM. Each of these methods learns a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, whose decision boundary (vectors $\mathbf{x}$ with $h(\mathbf{x}) = 0$) is a hyperplane (see Figure 2.8). However, these methods use different loss functions for assessing the quality of a particular linear hypothesis $h(\mathbf{x}) = \mathbf{w}\mathbf{x}$ (which defined the decision boundary via $h(\mathbf{x}) = 0$). Therefore, these three methods typically learn classifiers with different decision boundaries.

For the estimator $\widehat{\mathbf{\Sigma}}$ (3.24) to be accurate (close to the unknown covariance matrix) we need a number of datapoints (sample size) which is at least of the order n^2. This sample size requirement might be infeasible for applications with only few datapoints available.

The maximum likelihood estimate $\widehat{\mathbf{\Sigma}}$ (4.17) is not invertible whenever $m < n$. In this case, the expression (4.18) becomes useless. To cope with small sample size $m < n$ we can simplify the model (4.15) by requiring the covariance to be diagonal $\mathbf{\Sigma} = \text{diag}(\sigma_1^2, \ldots, \sigma_n^2)$. This is equivalent to modelling the individual features $x_1, \ldots, x_n$ of a datapoint as conditionally independent, given its label y. The resulting special case of a Bayes' classifier is often referred to as a **naive Bayes** classifier.

We finally highlight that the classifier (4.18) is obtained using the generative model (4.15) for the data. Therefore, Bayes' classifiers belong to the family of generative ML methods which involve modelling the data generation. In contrast, logistic regression and SVM do not require a generative model for the datapoints but aim directly at finding the relation between features $\mathbf{x}$ and label y of a datapoint. These methods belong therefore to the family of discriminative ML methods.

Generative methods such as Bayes' classifier are preferable for applications with only very limited amounts of labeled data. Indeed, having a generative model such as (4.15) allows to synthetically generate more labeled data by generating random features and labels according to the probability distribution (4.15). We refer to [57] for a more detailed comparison between generative and discriminative methods.

4.6 Training and Inference Periods

Some ML methods repeat the cycle in Figure 1 in a highly irregular fashion. Consider a large image collection which we use to learn a hypothesis about how cat images look like. It might be reasonable to adjust the hypothesis by fitting a model to the image collection. This fitting or training amounts to repeating the cycle in Figure 1 during some specific time period (the "training time") for a large number.

After the training period, we only apply the hypothesis to predict the labels of new images. This second phase is also known as inference time and might be much longer compared to the training time. Ideally, we would like to only have a very short training period to learn a good hypothesis and then only use the hypothesis for inference.

4.7 Online Learning

So far we considered the training set to be an unordered set of datapoints whose labels are known. Many applications generate data in a sequential fashion, datapoints arrive incrementally over time. It is then desirable to update the current hypothesis as soon as new data arrives.

ML methods differ in the frequency of iterating the cycle in Figure 1. Consider a temperature sensor which delivers a new measurement every ten seconds. As soon as a new temperature measurement arrives, a ML method can use it to improve its hypothesis about how the temperature evolves over time. Such ML methods operate in an online fashion by continuously learning an improved model as new data arrives.

To illustrate online learning, we consider the ML problem discussed in Section 2.4. This problem amounts to learning a linear predictor for the label y of datapoints using a single numeric feature x. We learn the predictor based on some training data. The weight vector for the optimal linear predictor is characterized by (2.20).

Let us assume that the training data is built up sequentially, we start with $m = 1$

datapoints in the first time step, then in the next time step collect another datapoint to get $m = 2$ datapoints, ... We denote the feature matrix and label vector at time m by $\mathbf{X}^{(m)}$ and $\mathbf{y}^{(m)}$:

$$m = 1: \quad \mathbf{X}^{(1)} = \big(\mathbf{x}^{(1)}\big)^T, \quad \mathbf{y}^{(1)} = \big(y^{(1)}\big)^T, \tag{4.19}$$

$$m = 2: \quad \mathbf{X}^{(2)} = \big(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}\big)^T, \quad \mathbf{y}^{(2)} = \big(y^{(1)}, y^{(2)}\big)^T, \tag{4.20}$$

$$m = 3: \quad \mathbf{X}^{(3)} = \big(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}\big)^T, \quad \mathbf{y}^{(3} = \big(y^{(1)}, y^{(2)}, y^{(3)}\big)^T. \tag{4.21}$$

Note that in this online learning setting, the sample size m has the meaning of a time index.

Naively, we could try to solve the optimality condition (2.20) for each time step m. However, this approach does not reuse computations already invested in solving (2.20) at previous time steps $m' < m$.

4.8 Exercise

4.8.1 Uniqueness in Linear Regression

Consider linear regression with squared error loss. When is the optimal linear predictor unique. Does there always exist an optimal linear predictor?

4.8.2 A Simple Linear Regression Method

Consider datapoints characterized by single numeric feature x and label y. We learn a hypothesis map of the form $h(x) = x + b$ with some bias $b \in \mathbb{R}$. Can you write down a formula for the optimal b, that minimizes the average squared error on training data $\big(x^{(1)}, y^{(1)}\big), \ldots, \big(x^{(m)}, y^{(m)}\big)$.

4.8.3 A Simple Least Absolute Deviation Method

Consider datapoints characterized by single numeric feature x and label y. We learn a hypothesis map of the form $h(x) = x + b$ with some bias $b \in \mathbb{R}$. Can you write down a formula for the optimal b, that minimizes the average absolute error on training data $\big(x^{(1)}, y^{(1)}\big), \ldots, \big(x^{(m)}, y^{(m)}\big)$.

4.8.4 Polynomial Regression

Polynomial regression for datapoints with a single feature x and label y is equivalent to linear regression with the feature vectors $\mathbf{x} = \big(x^0, x^1, \ldots, x^{n-1}\big)^T$. Given $m = n$ datapoints $\big(x^{(1)}, y^{(1)}\big), \ldots, \big(x^{(m)}, y^{(m)}\big)$, we construct the feature matrix $\mathbf{X} \in \mathbb{R}^{m \times m}$. The columns of the feature matrix are the feature vectors $\mathbf{x}^{(i)}$. Is this feature matrix a Vandermonde matrix [25]? Can you say something about the determinant of the feature matrix?

4.8.5 Empirical Risk Approximates Expected Loss

Consider training datapoints $\big(x^{(i)}, y^{(i)}\big)$, for $i = 1, \ldots, 100$. The datapoints are i.i.d. realizations of a random datapoint (x, y). The feature x of a random datapoint is a Gaussian random variable with zero mean and unit variance. The label is modelled as via $y = x + e$ with noise $e \sim \mathcal{N}(0, 1)$ being a standard Gaussian RV. The feature x and noise e are statistically independent. For the hypothesis $h(x) = 0$, what is the probability that the empirical risk (average loss) on the training data is more than 20 % larger than the expected loss or risk? What is the expectation and variance of the training error and how are those related to the expected loss ?

Chapter 5

Gradient-Based Learning

ML methods are optimization methods, that learn an optimal hypothesis out of the model. The quality of each hypothesis is measured or scored by some average loss or empirical risk. This average loss, viewed as a function of the hypothesis, defines an objective function whose minimum is achieved by the optimal hypothesis.

Many ML methods use gradient-based methods to efficiently search for a (nearly) optimal hypothesis. These methods locally approximate the objective function by a linear function which is used to improve the current guess for the optimal hypothesis. The prototype of a gradient-based optimization method is **gradient descent** (GD).

Variants of GD are used to tune the weights of artificial neural networks within deep learning methods [27]. GD can also be applied to reinforcement learning applications. The difference between these applications is merely in the details for how to compute or estimate the gradient and how to incorporate the information provided by the gradients.

In the following, we will mainly focus on ML problems with hypothesis space $\mathcal{H}$ consisting of predictor maps $h^{(\mathbf{w})}$ which are parameterized by a weight vector $\mathbf{w} \in \mathbb{R}^n$. Moreover, we will restrict ourselves to loss functions $\mathcal{L}((\mathbf{x}, y), h^{(\mathbf{w})})$ which depend smoothly on the weight vector $\mathbf{w}$.

Many important ML problems, including linear regression (see Section 3.1) and logistic regression (see Section 3.6), involve in a smooth loss function. A smooth function $f : \mathbb{R}^n \to \mathbb{R}$ has continuous partial derivatives of all orders. In particular, we can define the gradient $\nabla f(\mathbf{w})$ for a smooth function $f(\mathbf{w})$ at every point $\mathbf{w}$.

For a smooth loss function, the resulting ERM (see (4.3))

$$\begin{aligned}\mathbf{w}_{\rm opt} &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} \mathcal{E}(h^{(\mathbf{w})} \mid \mathcal{D}) \\ &= \underbrace{(1/m)\sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h^{(\mathbf{w})})}_{:=f(\mathbf{w})} \end{aligned} \tag{5.1}$$

is a **smooth optimization problem**

$$\min_{\mathbf{w}\in\mathbb{R}^n} f(\mathbf{w}) \tag{5.2}$$

with a smooth function $f : \mathbb{R}^n \to \mathbb{R}$ of the vector argument $\mathbf{w} \in \mathbb{R}^n$.

We can approximate a smooth function $f(\mathbf{w})$ locally around some point $\mathbf{w}_0$ using a hyperplane. This hyperplane passes through the point $(\mathbf{w}_0, f(\mathbf{w}_0))$ and has the normal vector $\mathbf{n} = (\nabla f(\mathbf{w}_0), -1)$ (see Figure 5.1). Elementary calculus yields the following linear approximation (around a point $\mathbf{w}_0$) [61]

$$f(\mathbf{w}) \approx f(\mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T \nabla f(\mathbf{w}_0) \text{ for } \mathbf{w} \text{ sufficiently close to } \mathbf{w}_0. \tag{5.3}$$

The approximation (5.3) lends naturally to an iterative method for finding the minimum of the function $f(\mathbf{w})$. This method is known as gradient descent (GD) and (variants of it) underlies many state-of-the-art ML methods, including deep learning methods.

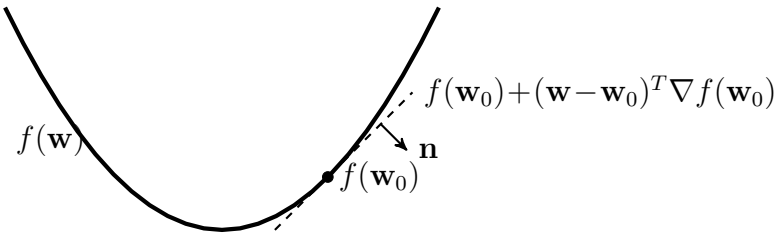


Figure 5.1: A smooth function $f(\mathbf{w})$ can be approximated locally around a point $\mathbf{w}_0$ using a hyperplane whose normal vector $\mathbf{n} = (\nabla f(\mathbf{w}_0), -1)$ is determined by the gradient $\nabla f(\mathbf{w}_0)$.

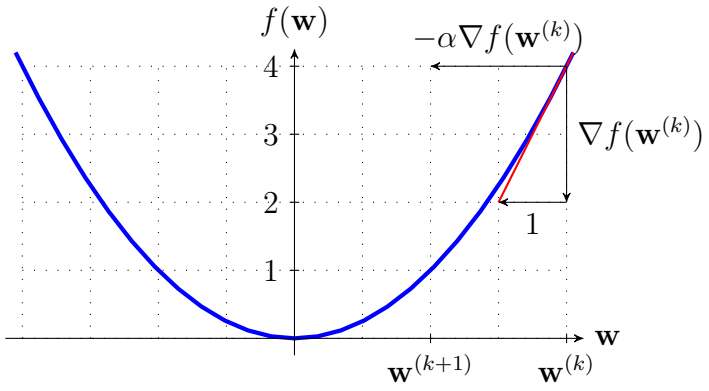


Figure 5.2: The GD step (5.4) amounts to a shift by $-\alpha\nabla f(\mathbf{w}^{(k)})$.

5.1 The Basic GD Step

We now discuss a very simple, yet quite powerful, algorithm for finding the weight vector $\mathbf{w}_{\rm opt}$ which solves continuous optimization problems like (5.1).

Let us assume we have already some guess (or approximation) $\mathbf{w}^{(k)}$ for the optimal weight vector $\mathbf{w}_{\rm opt}$ and would like to improve it to a new guess $\mathbf{w}^{(k+1)}$ which yields a smaller value of the objective function $f(\mathbf{w}^{(k+1)}) < f(\mathbf{w}^{(k)})$.

For a differentiable objective function $f(\mathbf{w})$, we can use the approximation $f(\mathbf{w}^{(k+1)}) \approx f(\mathbf{w}^{(k)}) + (\mathbf{w}^{(k+1)} - \mathbf{w}^{(k)})^T \nabla f(\mathbf{w}^{(k)})$ (cf. (5.3)) for $\mathbf{w}^{(k+1)}$ not too far away from $\mathbf{w}^{(k)}$. Thus, we should be able to enforce $f(\mathbf{w}^{(k+1)}) < f(\mathbf{w}^{(k)})$ by choosing

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha\nabla f(\mathbf{w}^{(k)}) \tag{5.4}$$

with a sufficiently small **step size** $\alpha > 0$ (a small α ensures that the linear approximation (5.3) is valid). Then, we repeat this procedure to obtain $\mathbf{w}^{(k+2)} = \mathbf{w}^{(k+1)} - \alpha\nabla f(\mathbf{w}^{(k+1)})$ and so on.

The update (5.4) amounts to a **gradient descent (GD) step**. For a convex differentiable objective function $f(\mathbf{w})$ and sufficiently small step size α, the iterates $f(\mathbf{w}^{(k)})$ obtained by repeating the GD steps (5.4) converge to a minimum, i.e., $\lim_{k\to\infty} f(\mathbf{w}^{(k)}) = f(\mathbf{w}_{\rm opt})$ (see Figure 5.2).

When the GD step is used within an ML method (see Section 5.4 and Section 3.6), the step size α is also referred to as the **learning rate**.

In order to implement the GD step (5.4) we need to choose the step size α and we need

to be able to compute the gradient $\nabla f(\mathbf{w}^{(k)})$. Both tasks can be very challenging for an ML problem.

The success of deep learning methods, which represent predictor maps using ANN (see Section 3.11), can be partially attributed to the ability of computing the gradient $\nabla f(\mathbf{w}^{(k)})$ efficiently via a message passing protocol known as **back-propagation** [27].

For the particular case of linear regression (see Section 3.1) and logistic regression (see Section 5.5), we will present precise conditions on the step size α which guarantee convergence of GD in Section 5.4 and Section 5.5. Moreover, the objective functions $f(\mathbf{w})$ arising within linear and logistic regression allow for closed-form expressions of the gradient $\nabla f(\mathbf{w})$.

5.2 Choosing Step Size

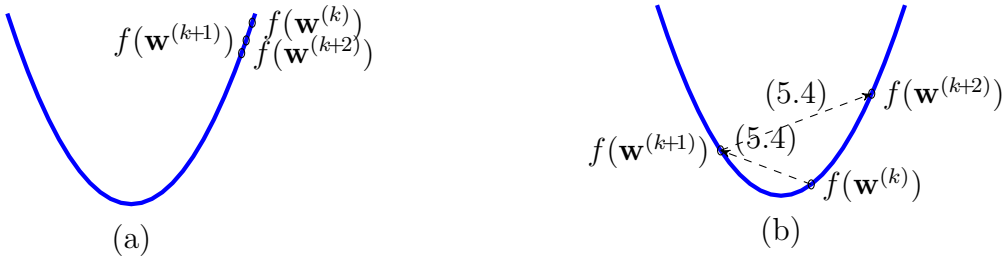


Figure 5.3: Effect of choosing learning rate α in GD step (5.4) too small (a) or too large (b). If the steps size α in the GD step (5.4) is chosen too small, the iterations make very little progress towards the optimum or even fail to reach the optimum at all. If the learning rate α is chosen too large, the iterates $\mathbf{w}^{(k)}$ might not converge at all (it might happen that $f(\mathbf{w}^{(k+1)}) > f(\mathbf{w}^{(k)})$!).

The choice of the step size α in the GD step (5.4) has a strong impact on the performance of Algorithm 1. If we choose the step size α too large, the GD steps (5.4) diverge (see Figure 5.3-(b)) and, in turn, Algorithm 1 fails to deliver a satisfactory approximation of the optimal weight vector $\mathbf{w}^{(\text{opt})}$ (see (5.7)).

If we choose the step size α too small (see Figure 5.3-(a)), the updates (5.4) make only very little progress towards approximating the optimal weight vector $\mathbf{w}_{\text{opt}}$. In applications that require real-time processing of data streams, it is possible to repeat the GD steps only for a moderate number. If the GD step size is chosen too small, Algorithm 1 will fail to deliver a good approximation of $\mathbf{w}_{\text{opt}}$ within an acceptable number of iterations (which translates to computation time).

The optimal choice of the step size α of GD can be a challenging task and many sophisticated approaches have been proposed for its solution (see [27, Chapter 8]). We will restrict ourselves to a simple sufficient condition on the step size which guarantees convergence of the GD iterations $\mathbf{w}^{(k)}$ for $k = 1, 2, \ldots$.

If the objective function $f(\mathbf{w})$ is convex and smooth, the GD steps (5.4) converge to an optimum $\mathbf{w}_{\rm opt}$ for any step size α satisfying [55]

$$\alpha \leq \frac{1}{\lambda_{\max}\big(\nabla^2 f(\mathbf{w})\big)} \text{ for all } \mathbf{w} \in \mathbb{R}^n. \tag{5.5}$$

Here, we use the Hessian matrix $\nabla^2 f(\mathbf{w}) \in \mathbb{R}^{n \times n}$ of a smooth function $f(\mathbf{w})$ whose entries are the second-order partial derivatives $\frac{\partial f(\mathbf{w})}{\partial w_i \partial w_j}$ of the function $f(\mathbf{w})$. It is important to note that (5.5) guarantees convergence for every possible initialization $\mathbf{w}^{(0)}$ of the GD iterations.

Note that while it might be computationally challenging to determine the maximum eigenvalue $\lambda_{\max}\big(\nabla^2 f(\mathbf{w})\big)$ for arbitrary $\mathbf{w}$, it might still be feasible to find an upper bound U for the maximum eigenvalue. If we know an upper bound $U \geq \lambda_{\max}\big(\nabla^2 f(\mathbf{w})\big)$ (valid for all $\mathbf{w} \in \mathbb{R}^n$), the step size $\alpha = 1/U$ still ensures convergence of the GD iteration.

5.3 When To Stop

Fixed number of iteration (for this we might use convergence analysis of GD methods); use gradient as indicator for distance to optimum; monitor decrease in objective function; monitor decrease in validation error

5.4 GD for Linear Regression

We will now formulate a complete ML algorithm. This algorithm is based on applying GD to the linear regression problem discussed in Section 3.1. This algorithm learns the weight vector for a linear hypothesis (see (3.1))

$$h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}. \tag{5.6}$$

The weight vector is chosen to minimize average squared error loss (2.6)

$$\mathcal{E}(h^{(\mathbf{w})}|\mathcal{D}) \overset{(4.3)}{=} (1/m) \sum_{i=1}^{m} (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)})^2, \tag{5.7}$$

incurred by the predictor $h^{(\mathbf{w})}(\mathbf{x})$ when applied to the labeled dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$. The optimal weight vector $\mathbf{w}_{\rm opt}$ for (5.6) is characterized as

$$\mathbf{w}_{\rm opt} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} f(\mathbf{w}) \text{ with } f(\mathbf{w}) = (1/m) \sum_{i=1}^{m} (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)})^2. \tag{5.8}$$

The optimization problem (5.8) is an instance of the smooth optimization problem (5.2). We can therefore use GD (5.4) to solve (5.8), to obtain the optimal weight vector $\mathbf{w}_{\rm opt}$. To implement GD, we need to compute the gradient $\nabla f(\mathbf{w})$.

The gradient of the objective function in (5.8) is given by

$$\nabla f(\mathbf{w}) = -(2/m) \sum_{i=1}^{m} (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)}) \mathbf{x}^{(i)}. \tag{5.9}$$

By inserting (5.9) into the basic GD iteration (5.4), we obtain Algorithm 1.

Algorithm 1 "Linear Regression via GD"

Input: labeled dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ containing feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and labels $y^{(i)} \in \mathbb{R}$; GD step size $\alpha > 0$.

Initialize: set $\mathbf{w}^{(0)} := \mathbf{0}$; set iteration counter $k := 0$

1: **repeat**

2: $\quad k := k + 1$ (increase iteration counter)

3: $\quad \mathbf{w}^{(k)} := \mathbf{w}^{(k-1)} + \alpha(2/m) \sum_{i=1}^{m} (y^{(i)} - \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)}) \mathbf{x}^{(i)}$ (do a GD step (5.4))

4: **until** convergence

Output: $\mathbf{w}^{(k)}$ (which approximates $\mathbf{w}_{\rm opt}$ in (5.8))

Let us have a closer look on the update in step 3 of Algorithm 1, which is

$$\mathbf{w}^{(k)} := \mathbf{w}^{(k-1)} + \alpha(2/m) \sum_{i=1}^{m} (y^{(i)} - \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)}) \mathbf{x}^{(i)}. \tag{5.10}$$

The update (5.10) has an appealing form as it amounts to correcting the previous guess (or approximation) $\mathbf{w}^{(k-1)}$ for the optimal weight vector $\mathbf{w}_{\rm opt}$ by the correction term

$$(2\alpha/m) \sum_{i=1}^{m} \underbrace{(y^{(i)} - \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)})}_{e^{(i)}} \mathbf{x}^{(i)}. \tag{5.11}$$

The correction term (5.11) is a weighted average of the feature vectors $\mathbf{x}^{(i)}$ using weights $(2\alpha/m) \cdot e^{(i)}$. These weights consist of the global factor $(2\alpha/m)$ (that applies equally to

all feature vectors $\mathbf{x}^{(i)}$) and a sample-specific factor $e^{(i)} = (y^{(i)} - \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)})$, which is the prediction (approximation) error obtained by the linear predictor $h^{(\mathbf{w}^{(k-1)})}(\mathbf{x}^{(i)}) = \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)}$ when predicting the label $y^{(i)}$ from the features $\mathbf{x}^{(i)}$.

> We can interpret the GD step (5.10) as an instance of "learning by trial and error". Indeed, the GD step amounts to "trying out" the predictor $h(\mathbf{x}^{(i)}) = \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)}$ and then correcting the weight vector $\mathbf{w}^{(k-1)}$ according to the error $e^{(i)} = y^{(i)} - \big(\mathbf{w}^{(k-1)}\big)^T \mathbf{x}^{(i)}$.

The choice of the step size α used for Algorithm 1 can be based on the sufficient condition (5.5) with the Hessian $\nabla^2 f(\mathbf{w})$ of the objective function $f(\mathbf{w})$ underlying linear regression (see (5.8)). This Hessian is given explicitly as

$$\nabla^2 f(\mathbf{w}) = (1/m)\mathbf{X}^T\mathbf{X}, \tag{5.12}$$

with the feature matrix $\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\big)^T \in \mathbb{R}^{m \times n}$ (see (4.5)). Note that the Hessian (5.12) does not depend on the weight vector $\mathbf{w}$.

Comparing (5.12) with (5.5), one particular strategy for choosing the step size in Algorithm 1 is to (i) compute the matrix product $\mathbf{X}^T\mathbf{X}$, (ii) compute the maximum eigenvalue $\lambda_{\max}\big((1/m)\mathbf{X}^T\mathbf{X}\big)$ of this product and (iii) set the step size to $\alpha = 1/\lambda_{\max}\big((1/m)\mathbf{X}^T\mathbf{X}\big)$.

While it might be challenging to compute the maximum eigenvalue $\lambda_{\max}\big((1/m)\mathbf{X}^T\mathbf{X}\big)$, it might be easier to find an upper bound U for it.[1] Given such an upper bound $U \geq \lambda_{\max}\big((1/m)\mathbf{X}^T\mathbf{X}\big)$, the step size $\alpha = 1/U$ still ensures convergence of the GD iteration. Consider a dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ with normalized features, i.e., $\|\mathbf{x}^{(i)}\| = 1$ for all $i = 1, \ldots, m$. Then, by elementary linear algebra, one can verify the upper bound $U = 1$, i.e., $1 \geq \lambda_{\max}\big((1/m)\mathbf{X}^T\mathbf{X}\big)$. We can then ensure convergence of the GD iterations $\mathbf{w}^{(k)}$ (see (5.10)) by choosing the step size $\alpha = 1$.

5.5 GD for Logistic Regression

As discussed in Section 3.6, logistic regression learns a linear hypothesis $h^{(\mathbf{w}_{\rm opt})}$ by minimizing the average logistic loss (3.15) obtained for a dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, with features $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and binary labels $y^{(i)} \in \{-1, 1\}$. This minimization problem is an instance of the

[1] The problem of computing a full eigenvalue decomposition of $\mathbf{X}^T\mathbf{X}$ has essentially the same complexity as solving the ERM problem directly via (4.9), which we want to avoid by using the "cheaper" GD algorithm.

smooth optimization problem (5.2),

$$\mathbf{w}_{\rm opt} = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} f(\mathbf{w})$$
$$\text{with } f(\mathbf{w}) = (1/m)\sum_{i=1}^{m} \log(1+\exp(-y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})). \tag{5.13}$$

To apply GD (5.4) to solve (5.13), we need to compute the gradient $\nabla f(\mathbf{w})$. The gradient of the objective function in (5.13) is given by

$$\nabla f(\mathbf{w}) = (1/m)\sum_{i=1}^{m} \frac{-y^{(i)}}{1+\exp(y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})}\mathbf{x}^{(i)}. \tag{5.14}$$

By inserting (5.14) into the basic GD iteration (5.4), we obtain Algorithm 2.

Algorithm 2 "Logistic Regression via GD"

Input: labeled dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ containing feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and labels $y^{(i)} \in \mathbb{R}$; GD step size $\alpha > 0$.

Initialize:set $\mathbf{w}^{(0)} := \mathbf{0}$; set iteration counter $k := 0$

1: **repeat**

2: $k := k+1$ (increase iteration counter)

3: $\mathbf{w}^{(k)} := \mathbf{w}^{(k-1)} + \alpha(1/m)\sum_{i=1}^{m} \frac{y^{(i)}}{1+\exp\left(y^{(i)}\left(\mathbf{w}^{(k-1)}\right)^T\mathbf{x}^{(i)}\right)}\mathbf{x}^{(i)}$ (do a GD step (5.4))

4: **until** convergence

Output: $\mathbf{w}^{(k)}$, which approximates a solution $\mathbf{w}_{\rm opt}$ of (5.13))

Let us have a closer look on the update in step 3 of Algorithm 2, which is

$$\mathbf{w}^{(k)} := \mathbf{w}^{(k-1)} + \alpha(1/m)\sum_{i=1}^{m} \frac{y^{(i)}}{1+\exp\left(y^{(i)}\left(\mathbf{w}^{(k-1)}\right)^T\mathbf{x}^{(i)}\right)}\mathbf{x}^{(i)}. \tag{5.15}$$

The update (5.15) has an appealing form as it amounts to correcting the previous guess (or approximation) $\mathbf{w}^{(k-1)}$ for the optimal weight vector $\mathbf{w}_{\rm opt}$ by the correction term

$$(\alpha/m)\sum_{i=1}^{m} \underbrace{\frac{y^{(i)}}{1+\exp(y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})}}_{e^{(i)}}\mathbf{x}^{(i)}. \tag{5.16}$$

The correction term (5.16) is a weighted average of the feature vectors $\mathbf{x}^{(i)}$, each of which is weighted by the factor $(\alpha/m)\cdot e^{(i)}$. These weighting factors are a product of the global

factor (α/m) that applies equally to all feature vectors $\mathbf{x}^{(i)}$. The global factor is multiplied by a datapoint-specific factor $e^{(i)} = \frac{y^{(i)}}{1+\exp(y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)})}$, which quantifies the error of the classifier $h^{(\mathbf{w}^{(k-1)})}(\mathbf{x}^{(i)}) = \big(\mathbf{w}^{(k-1)}\big)^T\mathbf{x}^{(i)}$ for a single datapoint with true label $y^{(i)} \in \{-1, 1\}$ and features $\mathbf{x}^{(i)} \in \mathbb{R}^n$.

We can use the sufficient condition (5.5) for the convergence of GD to guide the choice of the step size α in Algorithm 2. To apply condition (5.5), we need to determine the Hessian $\nabla^2 f(\mathbf{w})$ matrix of the objective function $f(\mathbf{w})$ underlying logistic regression (see (5.13)). Some basic calculus reveals (see [31, Ch. 4.4.])

$$\nabla^2 f(\mathbf{w}) = (1/m)\mathbf{X}^T\mathbf{D}\mathbf{X}. \tag{5.17}$$

Here, we used the feature matrix $\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\big)^T \in \mathbb{R}^{m\times n}$ (see (4.5)) and the diagonal matrix $\mathbf{D} = \mathrm{diag}\{d_1, \ldots, d_m\} \in \mathbb{R}^{m\times m}$ with diagonal elements

$$d_i = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x}^{(i)})}\left(1 - \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x}^{(i)})}\right). \tag{5.18}$$

We highlight that, in contrast to the Hessian (5.12) obtained for the objective function arising in linear regression, the Hessian (5.17) varies with the weight vector $\mathbf{w}$. This makes the analysis of Algorithm 2 and the optimal choice of step size somewhat more difficult compared to Algorithm 1. However, since the diagonal entries (5.18) take values in the interval $[0, 1]$, for normalized features (with $\|\mathbf{x}^{(i)}\| = 1$) the step size $\alpha = 1$ ensures convergence of the GD updates (5.15) to the optimal weight vector $\mathbf{w}_{\mathrm{opt}}$ solving (5.13).

5.6 Data Normalization

The convergence speed of the GD steps (5.4), i.e., the number of steps required to reach the minimum of the objective function (4.4) within a prescribed accuracy, depends crucially on the condition number $\kappa(\mathbf{X}^T\mathbf{X})$. This condition number is defined as the ratio

$$\kappa(\mathbf{X}^T\mathbf{X}) := \lambda_{\max}/\lambda_{\min} \tag{5.19}$$

between the largest and smallest eigenvalue of the matrix $\mathbf{X}^T\mathbf{X}$.

The condition number is only well defined if the columns of the feature matrix $\mathbf{X}$ (see (4.5)), which are precisely the feature vectors $\mathbf{x}^{(i)}$, are linearly independent. In this case the condition number is lower bounded as $\kappa(\mathbf{X}^T\mathbf{X}) \geq 1$.

It can be shown that the GD steps (5.4) converge faster for smaller condition number $\kappa(\mathbf{X}^T\mathbf{X})$ [35]. Thus, GD will be faster for datasets with a feature matrix $\mathbf{X}$ such that $\kappa(\mathbf{X}^T\mathbf{X}) \approx 1$. It is therefore often beneficial to pre-process the feature vectors using a **normalization** (or **standardization**) procedure as detailed in Algorithm 3.

Algorithm 3 "Data Normalization"

Input: labeled dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$

1: remove sample means $\bar{\mathbf{x}} = (1/m)\sum_{i=1}^m \mathbf{x}^{(t)}$ from features, i.e.,

$$\mathbf{x}^{(i)} := \mathbf{x}^{(i)} - \bar{\mathbf{x}} \text{ for } i = 1, \ldots, m$$

2: normalise features to have unit variance,

$$\hat{x}_j^{(i)} := x_j^{(i)}/\hat{\sigma} \text{ for } j = 1, \ldots, n \text{ and } i = 1, \ldots, m$$

with the empirical variance $\hat{\sigma}_j^2 = (1/m)\sum_{i=1}^m \left(x_j^{(i)}\right)^2$

Output: normalized feature vectors $\{\hat{\mathbf{x}}^{(i)}\}_{i=1}^m$

The preprocessing implemented in Algorithm 3 reshapes (transforms) the original feature vectors $\mathbf{x}^{(i)}$ into new feature vectors $\hat{\mathbf{x}}^{(i)}$ such that the new feature matrix $\widehat{\mathbf{X}} = (\hat{\mathbf{x}}^{(1)}, \ldots, \hat{\mathbf{x}}^{(m)})^T$ tends to be well-conditioned, i.e., $\kappa(\widehat{\mathbf{X}}^T\widehat{\mathbf{X}}) \approx 1$.

Exercise. Consider the dataset with feature vectors $\mathbf{x}^{(1)} = (100, 0)^T \in \mathbb{R}^2$ and $\mathbf{x}^{(2)} = (0, 1/10)^T$ which we stack into the matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)})^T$. What is the condition number of $\mathbf{X}^T\mathbf{X}$? What is the condition number of $\left(\widehat{\mathbf{X}}\right)^T\widehat{\mathbf{X}}$ with the matrix $\widehat{\mathbf{X}} = (\hat{\mathbf{x}}^{(1)}, \hat{\mathbf{x}}^{(2)})^T$ constructed from the normalized feature vectors $\hat{\mathbf{x}}^{(i)}$ delivered by Algorithm 3.

5.7 Stochastic GD

Consider an ML problem with a hypothesis space $\mathcal{H}$ which is parametreized by a weight vector $\mathbf{w} \in \mathbb{R}^n$ (such that each element $h^{(\mathbf{w})}$ of $\mathcal{H}$ corresponds to a particular choice of $\mathbf{w}$) and a loss function $\mathcal{L}((\mathbf{x}, y), h^{(\mathbf{w})})$ which depends smoothly on the weight vector $\mathbf{w}$. The resulting ERM (5.1) amounts to a smooth optimization problem which can be solved using GD (5.4).

The gradient $\nabla f(\mathbf{w})$ obtained for the optimization problem (5.1) has a particular structure.

Indeed, the gradient is a sum

$$\nabla f(\mathbf{w}) = (1/m) \sum_{i=1}^{m} \nabla f_i(\mathbf{w}) \text{ with } f_i(\mathbf{w}) := \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h^{(\mathbf{w})}), \tag{5.20}$$

with the components corresponding to the datapoints $(\mathbf{x}^{(i)}, y^{(i)})$, for $i = 1, \ldots, m$. Note that each GD step (5.4) requires to compute the gradient (5.20).

Computing the sum in (5.20) can be computationally challenging for at least two reasons. First, computing the sum exactly is challenging for extremely large datasets with m in the order of billions. Second, for datasets which are stored in different data centres located all over the world, the summation would require a huge amount of network resources. Moreover, the finite transmission rate of communication networks limits the rate by which the GD steps (5.4) can be executed.

ImageNet. The "ImageNet" database contains more than 10^6 images [43]. These images are labeled according to their content (e.g., does the image show a dog?). Let us assume that each image is represented by a (rather small) feature vector $\mathbf{x} \in \mathbb{R}^n$ of length $n = 1000$. Then, if we represent each feature by a floating point number, performing only one single GD update (5.4) per second would require at least 10^9 FLOPS.

The idea of **stochastic GD (SGD)** is to replace the exact gradient $\nabla f(\mathbf{w})$ by some approximation which can be computed easier than (5.20). The word "stochastic" in the name SGD hints already at the use of stochastic approximations.

A basic variant of SGD approximates the gradient $\nabla f(\mathbf{w})$ (see (5.20)) a randomly selected component $\nabla f_{\hat{i}}(\mathbf{w})$ in (5.20), with the index $\hat{i}$ being chosen randomly out of $\{1, \ldots, m\}$. SGD amounts to iterating the update

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla f_{\hat{i}}(\mathbf{w}^{(k)}). \tag{5.21}$$

It is important to use a fresh randomly chosen index $\hat{i}$ during each new iteration. The indices used in different iterations are statistically independent.

Note that SGD replaces the summation over all training datapoints in the GD step (5.4) just by the random selection of a single component of the sum. The resulting savings in computational complexity can be significant in applications where a large number of datapoints is stored in a distributed fashion. However, this saving in computational complexity

comes at the cost of introducing a non-zero gradient noise

$$\varepsilon = \nabla f(\mathbf{w}) - \nabla f_{\hat{i}}(\mathbf{w}), \tag{5.22}$$

into the SGD updates.

To avoid a detrimental accumulation of the gradient noise (5.22) during the SGD updates (5.24), the step size α needs to be gradually decreased. Thus, the step-size used in the SGD update (5.22) typically depends on the iteration number k, $\alpha = \alpha_k$. The sequence α_k of step-sizes is referred to as a **learning rate schedule** [27, Chapter 8]. One popular choice for the learning rate schedule is $\alpha = 1/k$ [53]. We consider conditions on the learning rate schedule that guarantee convergence of SGD in Exercise 5.8.2.

The SGD iteration (5.24) assumes that the training data is already collected but so large that the sum in (5.20) is computationally intractable. Another variant of SGD is obtained by assuming a different data generation mechanism. If datapoints are collected sequentially, one new datapoint $\mathbf{x}^{(t)}, y^{(t)}$ at each new time step t, we could use a SGD variant for online learning (see Section 4.7). This online SGD algorithm amounts to computing, for each time step t, the iteration

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha_t \nabla f_{t+1}(\mathbf{w}^{(t)}). \tag{5.23}$$

5.8 Exercises

5.8.1 Use Knowledge About Problem Class

Consider the space $\mathcal{P}$ of sequences $f = (f[0], f[1], \ldots)$ that have the following properties

- they are monotone increasing, $f[k'] \geq f[k]$ for any $k' \geq k$ and $f \in \mathcal{P}$
- a change point k, where $f[k] \neq f[k+1]$ can only be at integer multiples of 100, e.g., $k = 100$ or $k = 300$.

Given some unknown function $f \in \mathcal{P}$ and starting point k_0 the problem is to find the minimum value of f as quickly as possible. We consider iterative algorithms that can query the function at some point k to obtain the values $f[k], f[k-1]$ and $f[k+1]$.

5.8.2 SGD Learning Rate Schedule

Consider learning a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ from datapoints that arrive sequentially. In each time step $k = 1, \ldots,$, we collect a new datapoint $\big(\mathbf{x}^{(k)}, y^{(k)}\big)$. The datapoints are modelled as realizations of i.i.d. copies of a random data point $\big(\mathbf{x}, y\big)$. The probability distribution of the features $\mathbf{x}$ is a standard multivariate normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$. The label of a random datapoint is related to its features via $y = \bar{\mathbf{w}}^T\mathbf{x} + \varepsilon$ with standard Gaussian noise $\varepsilon \sim \mathcal{N}(0, 1)$. We use SGD to learn the weight vector $\mathbf{w}$ of a linear hypothesis,

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha_k \big(\big(\mathbf{w}^{(k)}\big)^T \mathbf{x}^{(k)} - y^{(k)} \big) \mathbf{x}^{(k)}. \tag{5.24}$$

with learning rate schedule $\alpha_k = \beta / k^\gamma$. Note that we implement one SGD iteration (5.24) during each time step k. Thus, the iteration counter is the time index in this case. What conditions on the hyper-parameters β, γ ensure that $\lim_{k \to \infty} \mathbf{w}^{(k)} = \bar{\mathbf{w}}$ in distribution?

5.8.3 Apple or No Apple?

Consider datapoints representing images. Each image is characterized by the RGB values (value range $0, \ldots, 255$) of 1024×1024 pixels, which we stack into a feature vector $\mathbf{x} \in \mathbb{R}^n$. We assign each image the label $y = 1$ if it shows an apple and $y = -1$ if it does not show an apple.

We use logistic regression to learn a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ for classifying an image according to $\hat{y} = 1$ if $h(\mathbf{x}) \geq 0$. We use a training set of $m = 10^{10}$ labeled images which are stored in the cloud. We implement the ML method on our own laptop which is connected to the internet with a rate of at most 100 Mbps. Unfortunately we only store at most five images on our computer. How long does one single GD step take at least?

Chapter 6

Model Validation and Selection

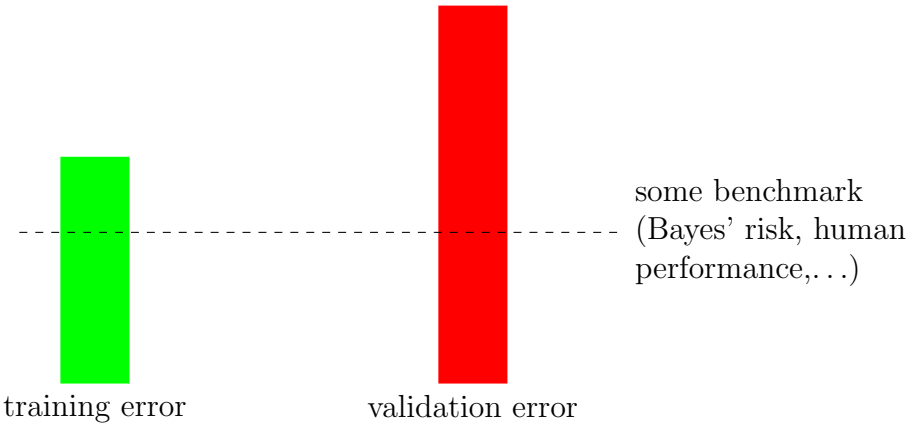


Figure 6.1: The diagnosis of ML methods is often based on comparing the training with validation error. Sometimes we also have some benchmark against which we can compare these errors.

Chapter 4 discussed ERM as a principled approach to learning a good hypothesis out of a hypothesis space or model. The idea of ERM is to learn a hypothesis $\hat{h} \in \mathcal{H}$ that incurs minimum average loss on a set of labeled datapoints (the training set). The minimum average loss achieved by a hypothesis that solves the ERM is referred to as the training error.

ERM makes sense if the training error of a hypothesis is also a good indicator for the typical loss incurred on datapoints outside the training set. Whether the training error of a hypothesis is a reliable indicator for its error on datapoints outside the training set depends on the statistical properties of the datapoints and on the hypothesis space used by the ML method.

Modern ML methods often use high-dimensional hypothesis spaces. One example is linear regression with datapoints having a vast number of features. This results in a high-dimensional space of linear maps. Another example are deep learning methods that use high-dimensional hypothesis spaces constituted by maps represented by deep neural networks with billions of tunable weights. For such vast hypothesis spaces, it is quite easy to find a hypothesis that fits the given training data well and, in turn, achieve a small training error. However, such a hypothesis might incur a large loss when predicting the labels of datapoints outside the training data.

A ML method overfits if it learns a predictor $h \in \mathcal{H}$ that accurately predicts the labels of the datapoints in the training set but does a poor job when predicting labels of datapoints outside the training set. Section 6.1 using a probabilistic model for datapoints to analyzes overfitting of linear regression methods.

This chapter discusses a simple validation procedure that allows to detect and avoid overfitting. Section 6.2 will discuss how **to validate** a learnt hypothesis by computing its average loss on datapoints which are different from the training set.

The datapoints used to validate the hypothesis are referred to as the **validation set**. When a ML method is overfitting the training set, it will learn a hypothesis whose training error is much smaller than the validation error.

We can use validation not only to detect if a ML method overfits. The validation error can also be used as a performance measure for the hypothesis space or model that is used by that method. This is similar in spirit to the the concept of a loss function that allows to evaluate the quality of a hypothesis $h \in \mathcal{H}$.

The validation error allows to evaluate the quality of the entire hypothesis space $\mathcal{H}$. Similar to ERM, which chooses between different hypotheses based on (average) loss on the training set, we can choose between different hypothesis spaces based on their validation errors. The resulting model selection strategy is detailed in Section 6.3.

Section 6.4 uses a simple probabilistic model for the data to study the relation between training error and the expected loss or risk of a hypothesis. This analysis reveals the interplay between the properties of the hypothesis space used by a ML method and the resulting training and validation error.

Computing validation errors not only allows to detect overfitting and select between different models (hypothesis spaces). Section 6.6 will show how we can debug and diagnose a ML method merely by comparing its training and validation error. The comparison of these two errors might help to identify possible improvements of the ML method. These

improvements might be obtained by collecting more datapoints, using more features of datapoints or to change the model (hypothesis space).

6.1 Overfitting

We illustrate the phenomenon of overfitting by considering a human child that learns the concept "tractor". This learning task amounts to finding an association between an image and the fact if the image shows a tractor or not. The association can be modelled as a hypothesis $h(\mathbf{x})$ that reads in the features of the image and outputs a value that indicates if the image shows a tractor or not.

Assume that we have taught the child using the image collection $\mathcal{D}^{(\text{train})}$ depicted in Figure 6.2. For some reason, one of the images is labeled erroneously as "tractor" but actually shows an ocean wave. As a consequence, if the child is good in memorizing images, it might predict the presence of tractors whenever looking at a wave (Figure 6.3).

Figure 6.2: A (misleading) training dataset $\mathcal{D}^{(\text{train})} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m_{\text{t}}}$ consisting of $m_{\text{t}} = 9$ images. The i-th image is characterized by the feature vector $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and labeled with $y^{(i)} = 1$ (tractor) or with $y^{(i)} = -1$ (no tractor).

For the sake of argument, we assume that the child uses a linear predictor $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{x}^T\mathbf{w}$, using the features $\mathbf{x}$ of the image, and encodes the fact of showing a tractor by $y = 1$ and if it is not showing a tractor by $y = -1$. We learn the weight vector by minimizing the squared error loss (4.4) on the training dataset $\mathcal{D}^{(\text{train})}$.

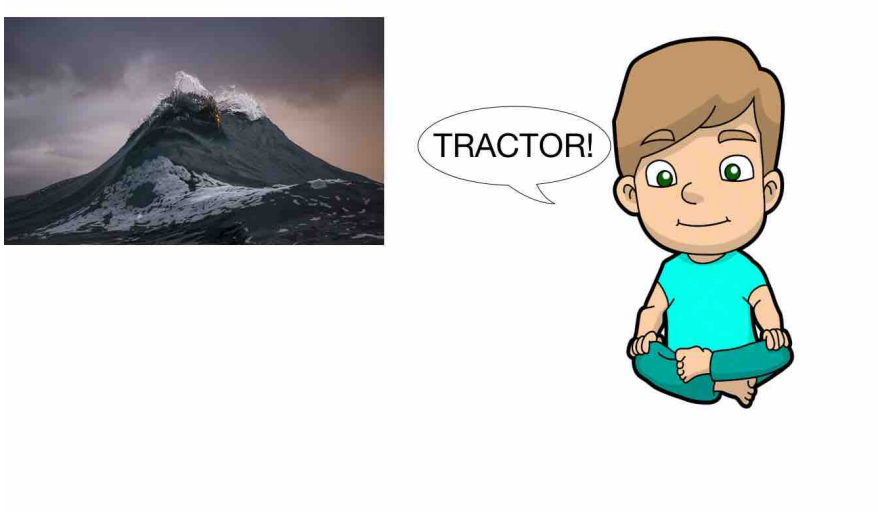


Figure 6.3: The child, who has been taught the concept "tractor" using the image collection $\mathcal{D}^{(\text{train})}$ in Figure 6.2, might "see" a lot of tractors during the next beach holiday.

If we stack the feature vectors $\mathbf{x}^{(i)}$ and labels $y^{(i)}$ into the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m_{\rm t})})^T$ and label vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(m_{\rm t})})^T$, the optimal linear predictor is obtained for the weight vector solving (4.9) and the associated training error is given by (4.11), which we repeat here for convenience:

$$\mathcal{E}(h^{(\mathbf{w}_{\rm opt})} \mid \mathcal{D}^{(\text{train})}) = \min_{\mathbf{w} \in \mathbb{R}^n} \mathcal{E}(h^{(\mathbf{w})} \mid \mathcal{D}^{(\text{train})}) = \|(\mathbf{I} - \mathbf{P})\mathbf{y}\|^2. \tag{6.1}$$

Here, we used the orthogonal projection matrix $\mathbf{P}$ on the linear span

$$\text{span}\{\mathbf{X}\} = \big\{\mathbf{X}\mathbf{a} : \mathbf{a} \in \mathbb{R}^n\big\} \subseteq \mathbb{R}^{m_{\rm t}},$$

of the feature matrix

$$\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m_{\rm t})})^T \in \mathbb{R}^{m_{\rm t} \times n}. \tag{6.2}$$

ML methods using linear predictors overfit as soon as the number n of features exceeds the sample size m,

$$n \geq m. \tag{6.3}$$

A set of m feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ is typically linearly independent whenever (6.3) is satisfied. If the feature vectors of the training datapoints are linearly independent, the span of the transposed feature matrix (6.2) coincides with $\mathbb{R}^m$ which implies, in turn, $\mathbf{P} = \mathbf{I}$.

Inserting $\mathbf{P} = \mathbf{I}$ into (4.11) yields

$$\mathcal{E}(h^{(\mathbf{w}_{\rm opt})} \mid \mathcal{D}^{(\rm train)}) = 0. \tag{6.4}$$

To sum up: as soon as the number of training examples $m_{\rm t} = |\mathcal{D}_{\rm train}|$ is smaller than the size n of the feature vector $\mathbf{x}$, there is a linear predictor $h^{(\mathbf{w}_{\rm opt})}$ achieving **zero empirical risk** (see (6.4)) on the training data. The result (6.4) only applies if the feature vectors of the training datapoints are linearly independent.

It can be shown that if the feature vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)} \in \mathbb{R}^n$ are realizations of i.i.d. RVs with a continuous probability distribution, then with probability one they are linearly independent whenever (6.3) holds.

While the learnt predictor $h^{(\mathbf{w}_{\rm opt})}$ is perfectly accurate on the training data (the training error is zero!), it will typically incur a non-zero average prediction error $y - h^{(\mathbf{w}_{\rm opt})}(\mathbf{x})$ on new datapoints $(\mathbf{x}, y)$ (which are different from the training data).

Using a simple toy model for the data generation, we obtained the expression (6.24) for the average prediction error. This average prediction error is lower bounded by the noise variance σ^2 which might be very large even when the training error is zero. Thus, when a method overfits, a small training error can be highly misleading regarding the average prediction error of a hypothesis.

A simple, yet quite useful, strategy to detect if a predictor $\hat{h}$ overfits the training dataset $\mathcal{D}^{(\rm train)}$, is to compare the resulting training error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\rm train)})$ (see (6.6)) with the validation error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\rm val)})$ (see (6.7)). The validation error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\rm val)})$ is the empirical risk of the predictor $\hat{h}$ on the validation dataset $\mathcal{D}^{(\rm val)}$. If overfitting occurs, the validation error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\rm val)})$ is significantly larger than the training error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\rm train)})$. The occurrence of overfitting for polynomial regression with degree n (see Section 3.2) chosen too large is depicted in Figure 7.2.

6.2 Validation

Consider an ML method using the hypothesis space $\mathcal{H}$ that learns a hypothesis $\hat{h} \in \mathcal{H}$ by ERM (4.2). The hypothesis $\hat{h}$ has minimum average loss (training error) on the labeled dataset $\mathcal{D}$. The basic idea of validating the predictor $\hat{h}$ is simple: compute the empirical risk of $\hat{h}$ on a new set of datapoints $(\mathbf{x}, y)$ which have not been already used for training.

It is very important to validate the predictor $\hat{h}$ using labeled data points which do not belong to the dataset which has been used to learn $\hat{h}$ (e.g., via ERM (4.2)). The hypothesis

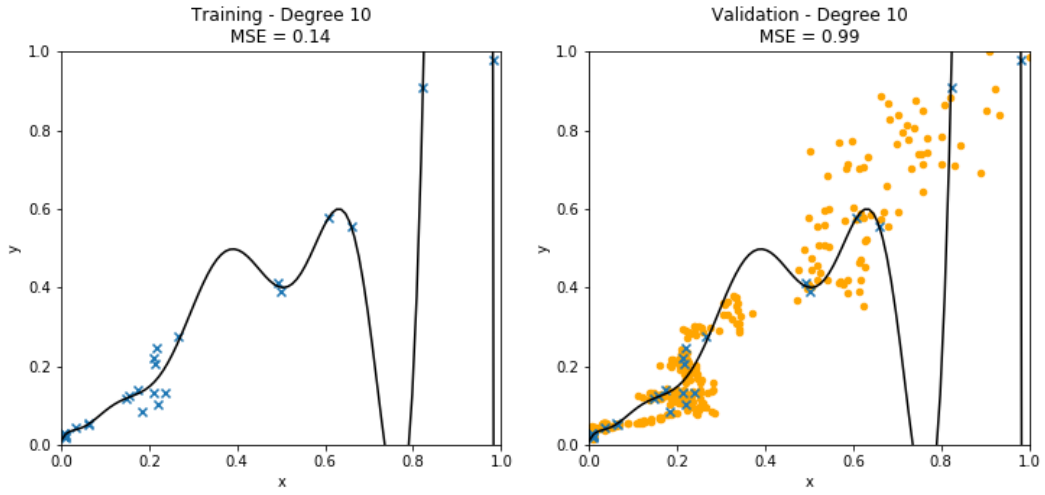


Figure 6.4: The training dataset consists of the blue crosses and can be almost perfectly fit by a high-degree polynomial. This high-degree polynomial gives only poor results for a different (validation) dataset indicated by the orange dots.

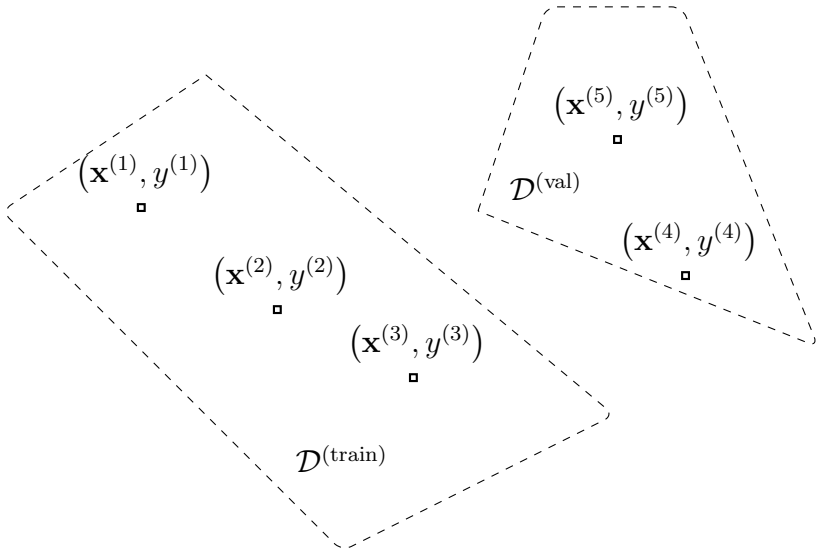


Figure 6.5: We split the entire dataset $\mathcal{D}$, constituted by datapoints with features $\mathbf{x}^{(i)}$ and label $y^{(i)}$, into a **training set** $\mathcal{D}^{(\text{train})}$ and a **validation set** $\mathcal{D}^{(\text{val})}$. We use the training set to learn (find) the hypothesis $\hat{h}$ with minimum empirical risk $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})})$ on the training set (4.2). We then validate $\hat{h}$ by computing its average loss $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$ on the validation set $\mathcal{D}^{(\text{val})}$. The average loss $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$ obtained on the validation set is the **validation error**. Note that $\hat{h}$ depends on the training set $\mathcal{D}^{(\text{train})}$ but is completely independent of the validation set $\mathcal{D}^{(\text{val})}$.

$\hat{h}$ tends to "look better" on the training set than for other datapoints, since it is optimized precisely for the datapoints in the training set.

> A golden rule of ML practice: use different datapoints for the learning (see (4.2)) and the validation of a hypothesis $\hat{h}$!

To validate the hypothesis learnt by a ML method we can use the following steps.

1. We randomly divide ("split") the entire set $\mathcal{D}$ of labeled datapoints into two disjoint subsets $\mathcal{D} = \mathcal{D}^{(\text{train})} \cup \mathcal{D}^{(\text{val})}$ (see Figure 6.5). The first subset $\mathcal{D}^{(\text{train})}$ is referred to as the **training set**. The second subset $\mathcal{D}^{(\text{val})}$ is referred to as the **validation set**.

2. We learn a hypothesis $\hat{h}$ via ERM using the training data $\mathcal{D}^{(\text{train})}$ (cf. (4.2)),

$$\begin{aligned} \hat{h} &= \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{E}(h|\mathcal{D}^{(\text{train})}) \\ &= \operatorname*{argmin}_{h \in \mathcal{H}} (1/m_{\text{t}}) \sum_{(\mathbf{x},y) \in \mathcal{D}^{(\text{train})}} \mathcal{L}((\mathbf{x},y),h) \end{aligned} \tag{6.5}$$

   with corresponding **training error**

$$\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})}) = (1/m_{\text{t}}) \sum_{(\mathbf{x},y) \in \mathcal{D}^{(\text{train})}} \mathcal{L}((\mathbf{x},y),\hat{h}). \tag{6.6}$$

3. We validate the hypothesis $\hat{h}$ obtained from (6.5) by computing the empirical risk

$$\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})}) = (1/m_{\text{v}}) \sum_{(\mathbf{x},y) \in \mathcal{D}^{(\text{val})}} \mathcal{L}((\mathbf{x},y),\hat{h}) \tag{6.7}$$

   on the **validation set** $\mathcal{D}^{(\text{val})}$. We refer to $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$ as the **validation error** of $\hat{h}$.

The choice of the split ratio (between the size of training and validation set) $|\mathcal{D}^{(\text{val})}|/|\mathcal{D}^{(\text{train})}|$ is often based on trial and error. It is difficult to make a precise statement on how to choose the split ratio which applies broadly [45].

The optimal choice for the size of training and validation set depends on the statistical properties of the datapoints. If we know the probability distribution from which datapoints are drawn, we can derive lower bounds on the size of the validation set such that the validation error (6.7) is close to the expected loss $\mathbb{E}\{\mathcal{L}((\mathbf{x},y),\hat{h})\}$ with high probability.

The basic idea of randomly splitting the available labeled data into training and validation sets underlies many validation techniques. A popular extension of the above approach, which is known as k-fold cross-validation, is based on repeating the splitting into training and validation sets k times. During each repetition, this method uses different subsets for training and validation. We refer to [31, Sec. 7.10] for a detailed discussion of k-fold cross-validation.

Imbalanced Data. The simple validation approach discussed above requires the validation set to be a good representative for the overall statistical properties of the data. This might not be the case in applications with discrete valued labels and some of the label values being very rare.

Consider datapoints characterized by a feature vector $\mathbf{x}$ and binary label $y \in \{-1, 1\}$. Assume we aim at learning a hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ to classify datapoints via $\hat{y} = 1$ if $h(\mathbf{x}) \geq 0$ while $\hat{y} = -1$ otherwise. The learning is based on a dataset $\mathcal{D}$ which contains only one single (!) datapoint with $y = -1$. If we then split the dataset into training and validation set, it is with high probability that the validation set does not include any datapoint with $y = -1$. This cannot happen when using k-fold CV since the single data point must be one of the validation folds. However, even when using k-fold CV for such an imbalanced dataset is problematic since we evaluate the performance of a hypothesis $h(\mathbf{x})$ using only one single datapoint with $y = -1$. The validation error will then be dominated by the performance of $h(\mathbf{x})$ only on datapoints with $y = 1$.

6.3 Model Selection

We will now discuss how to use the validation principle of Section 6.2 to perform model selection. As discussed in Chapter 2, the choice of the hypothesis space from which we select a predictor map (e.g., via solving the ERM (4.2)) is a design choice. However, it is often not obvious what a good first choice for the hypothesis space is. We might try out different choices $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_M$ for the hypothesis space.

Consider datapoints with a non-linear relation between its feature x and label y. We might then use polynomial regression (see Section 3.2) using the hypothesis space $\mathcal{H}^{(n)}_{\text{poly}}$ with some maximum degree n.

Different choices for the maximum degree n yield a different hypothesis space: $\mathcal{H}_1 = \mathcal{H}^{(0)}_{\text{poly}}, \mathcal{H}_2 = \mathcal{H}^{(1)}_{\text{poly}}, \ldots, \mathcal{H}_M = \mathcal{H}^{(M-1)}_{\text{poly}}$. We might also mix polynomial maps using maps obtained from Gaussian basis functions (see Section 3.5), with different choices for the

variance σ and shifts μ of the Gaussian basis function (3.12), e.g., $\mathcal{H}_1 = \mathcal{H}^{(2)}_{\text{Gauss}}$ with $\sigma = 1$ and $\mu_1 = 1$ and $\mu_2 = 2$, $\mathcal{H}_2 = \mathcal{H}^{(2)}_{\text{Gauss}}$ with $\sigma = 1/10$, $\mu_1 = 10$, $\mu_2 = 20$.

A principled approach for choosing a hypothesis space out of a list of candidate spaces $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_M$ is as follows:

- randomly divide (split) the entire dataset $\mathcal{D}$ of labeled snapshots into two disjoint subsets $\mathcal{D}^{(\text{train})}$ (the "training set") and $\mathcal{D}^{(\text{val})}$ (the "validation set"): $\mathcal{D} = \mathcal{D}^{(\text{train})} \cup \mathcal{D}^{(\text{val})}$ (see Figure 6.5).

- for each hypothesis space $\mathcal{H}_l$ learn predictor $\hat{h}_l \in \mathcal{H}_l$ via ERM (4.2) using training data $\mathcal{D}^{(\text{train})}$:

$$\begin{aligned} \hat{h}_l &= \operatorname*{argmin}_{h \in \mathcal{H}_l} \mathcal{E}(h|\mathcal{D}^{(\text{train})}) \\ &= \operatorname*{argmin}_{h \in \mathcal{H}_l} (1/m_{\text{t}}) \sum_{i=1}^{m_{\text{t}}} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) \end{aligned} \tag{6.8}$$

- compute the validation error of $\hat{h}_l$

$$\mathcal{E}(\hat{h}_l|\mathcal{D}^{(\text{val})}) = (1/m_{\text{v}}) \sum_{i=1}^{m_{\text{v}}} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}_l) \tag{6.9}$$

  obtained when applying the predictor $\hat{h}_l$ to the **validation dataset** $\mathcal{D}^{(\text{val})}$.

- pick the hypothesis space $\mathcal{H}_l$ resulting in the smallest validation error $\mathcal{E}(\hat{h}_l|\mathcal{D}^{(\text{val})})$

6.4 A Probabilistic Model of Generalization

More Data Beats Clever Algorithms ?; More Data Beats Clever Feature Selection?

A core problem or challenge within ML is the verification (or validation) whether a hypothesis incurring a small loss on a training dataset, will also incur a small loss on new datapoints (generalize well). Roughly speaking, we must validate a hypothesis by evaluating the loss of its predictions for datapoints that have not been used already for learning that hypothesis. However, if interpret datapoints as realizations of i.i.d. random variables, we can study the generalization ability (beyond the training set) of a hypothesis via probability theory.

To study generalization within a linear regression problem (see Section 3.1), we will invoke a **probabilistic toy model** for the data arising in an ML application. We assume that any observed datapoint $\mathbf{z} = (\mathbf{x}, y)$ with features $\mathbf{x} \in \mathbb{R}^n$ and label $y \in \mathbb{R}$ is an i.i.d. realization of a Gaussian random vector.

The feature vector $\mathbf{x}$ is assumed to have zero mean and covariance being the identity matrix, i.e., $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The label y of a datapoint is related to its features $\mathbf{x}$ via a **linear Gaussian model**

$$y = \bar{\mathbf{w}}^T \mathbf{x} + \varepsilon, \text{ with noise } \varepsilon \sim \mathcal{N}(0, \sigma^2). \tag{6.10}$$

The noise variance σ^2 is assumed fixed (non-random) and known. Note that the error component ε in (6.10) is intrinsic to the data (within our toy model) and cannot be overcome by any ML method. We highlight that this model for the observed data points might not be accurate for a particular ML application. However, this toy model will allow us to study some fundamental behaviour of ML methods.

We predict the label y from the features $\mathbf{x}$ using a linear hypothesis $h(\mathbf{x})$ that depends only on the first r features $x_1, \ldots, x_r$. Thus, we use the hypothesis space

$$\mathcal{H}^{(r)} = \{h^{(\mathbf{w})}(\mathbf{x}) = (\mathbf{w}^T, \mathbf{0})\mathbf{x} \text{ with } \mathbf{w} \in \mathbb{R}^r\}. \tag{6.11}$$

The design parameter r determines the size of the hypothesis space $\mathcal{H}^{(r)}$ and, in turn, the computational complexity of learning the optimal hypothesis in $\mathcal{H}^{(r)}$.

For $r < n$, the hypothesis space $\mathcal{H}^{(r)}$ is a proper subset of the space of linear predictors (2.4) used within linear regression (see Section 3.1). Note that each element $h^{(\mathbf{w})} \in \mathcal{H}^{(r)}$ corresponds to a particular choice of the weight vector $\mathbf{w} \in \mathbb{R}^r$.

The quality of a particular predictor $h^{(\mathbf{w})} \in \mathcal{H}^{(r)}$ is measured via the mean squared error $\mathcal{E}(h^{(\mathbf{w})} \mid \mathcal{D}^{(\text{train})})$ incurred over a labeled training set $\mathcal{D}^{(\text{train})} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m_{\text{t}}}$. Within our toy model (see (6.10), (6.12) and (6.13)), the training datapoints $(\mathbf{x}^{(i)}, y^{(i)})$ are i.i.d. copies of the datapoint $\mathbf{z} = (\mathbf{x}, y)$.

The datapoints in the training dataset and any other datapoints outside the training set are statistically independent. However, the training datapoints $(\mathbf{x}^{(i)}, y^{(i)})$ and any other datapoint $(\mathbf{x}, y)$ are drawn from the same probability distribution, which is a multivariate normal distribution,

$$\mathbf{x}, \mathbf{x}^{(i)} \text{ i.i.d. with } \mathbf{x}, \mathbf{x}^{(i)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{6.12}$$

and the labels $y^{(i)}, y$ are obtained as

$$y^{(i)} = \bar{\mathbf{w}}^T \mathbf{x}^{(i)} + \varepsilon^{(i)}, \text{ and } y = \bar{\mathbf{w}}^T \mathbf{x} + \varepsilon \tag{6.13}$$

with i.i.d. noise $\varepsilon, \varepsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$.

As discussed in Chapter 4, the training error $\mathcal{E}(h^{(\mathbf{w})} \mid \mathcal{D}^{(\text{train})})$ is minimized by the predictor $h^{(\hat{\mathbf{w}})}(\mathbf{x}) = \hat{\mathbf{w}}^T \mathbf{I}_{r\times n}\mathbf{x}$, with weight vector

$$\widehat{\mathbf{w}} = (\mathbf{X}_r^T \mathbf{X}_r)^{-1} \mathbf{X}_r^T \mathbf{y} \tag{6.14}$$

with feature matrix $\mathbf{X}_r$ and label vector $\mathbf{y}$ defined as

$$\begin{aligned} \mathbf{X}_r &= (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m_{\text{t}})})^T \mathbf{I}_{n\times r} \in \mathbb{R}^{m_{\text{t}}\times r}, \text{ and} \\ \mathbf{y} &= \big(y^{(1)}, \ldots, y^{(m_{\text{t}})}\big)^T \in \mathbb{R}^{m_{\text{t}}}. \end{aligned} \tag{6.15}$$

It will be convenient to tolerate a slight abuse of notation and denote both, the length-r vector (6.14) as well as the zero padded length-n vector $(\widehat{\mathbf{w}}^T, \mathbf{0})^T$, by $\widehat{\mathbf{w}}$. This allows us to write

$$h^{(\widehat{\mathbf{w}})}(\mathbf{x}) = \widehat{\mathbf{w}}^T \mathbf{x}. \tag{6.16}$$

We highlight that the formula (6.14) for the optimal weight vector $\widehat{\mathbf{w}}$ is only valid if the matrix $\mathbf{X}_r^T \mathbf{X}_r$ is invertible. However, it can be shown that within our toy model (see (6.12)), this is true with probability one whenever $m_{\text{t}} \geq r$. In what follows, we will consider the case of having more training samples than the dimension of the hypothesis space, i.e., $m_{\text{t}} > r$ such that the formula (6.14) is valid (with probability one). The case $m_{\text{t}} \leq r$ will be studied in Chapter 7.

The optimal weight vector $\widehat{\mathbf{w}}$ (see (6.14)) depends on the training data $\mathcal{D}^{(\text{train})}$ via the feature matrix $\mathbf{X}_r$ and label vector $\mathbf{y}$ (see (6.15)). Therefore, since we model the training data as random, the weight vector $\widehat{\mathbf{w}}$ (6.14) is a random quantity. For each different realization of the training dataset, we obtain a different realization of the optimal weight $\widehat{\mathbf{w}}$.

The probabilistic model (6.10) relates the features $\mathbf{x}$ of a datapoint to its label y via some (unknown) true weight vector $\bar{\mathbf{w}}$. Intuitively, the best linear hypothesis would be $h(\mathbf{x}) = \widehat{\mathbf{w}}^T \mathbf{x}$ with weight vector $\widehat{\mathbf{w}} = \bar{\mathbf{w}}$. However, in general this will not be achievable since we have to compute $\widehat{\mathbf{w}}$ based on the features $\mathbf{x}^{(i)}$ and noisy labels $y^{(i)}$ of the data points in the training dataset $\mathcal{D}$.

In general, learning the weights of a linear hypothesis by ERM (4.4) results in a non-zero

estimation error

$$\Delta \mathbf{w} := \widehat{\mathbf{w}} - \bar{\mathbf{w}}. \tag{6.17}$$

The estimation error (6.17) is a random quantity (realization of a random variable) since the learnt weight vector $\widehat{\mathbf{w}}$ (see (6.14)) is a random quantity itself.

Bias and Variance. As we will see below, the prediction quality achieved by $h^{(\widehat{\mathbf{w}})}$ depends crucially on the **mean squared estimation error (MSE)**

$$\mathcal{E}_{\rm est} := \mathbb{E}\{\|\Delta \mathbf{w}\|_2^2\} = \mathbb{E}\big\{\big\|\widehat{\mathbf{w}} - \bar{\mathbf{w}}\big\|_2^2\big\}. \tag{6.18}$$

We can decompose the MSE $\mathcal{E}_{\rm est}$ into two components. The first component is the **bias** which characterizes the properties of the learn hypothesis on average (over all different realizations of training sets). The second component is the **variance** which quantifies the amount of random fluctuations of the hypothesis obtained from ERM applied to different realizations of the training set. Both components depend on the model complexity parameter r.

It is not too difficult to show that

$$\mathcal{E}_{\rm est} = \underbrace{\|\bar{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2}_{\text{“bias”}\, B^2} + \underbrace{\mathbb{E}\|\widehat{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2}_{\text{“variance”}\, V} \tag{6.19}$$

The bias term in (6.19), which can be computed as

$$B^2 = \|\bar{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2 = \sum_{l=r+1}^{n} \bar{w}_l^2, \tag{6.20}$$

measures the distance between the “true hypothesis” $h^{(\bar{\mathbf{w}})}(\mathbf{x}) = \bar{\mathbf{w}}^T\mathbf{x}$ and the hypothesis space $\mathcal{H}^{(r)}$ (see (6.11)) of the linear regression problem.

The bias (6.20) is zero if $\bar{w}_l = 0$ for any index $l = r+1, \ldots, n$, or equivalently if $h^{(\bar{\mathbf{w}})} \in \mathcal{H}^{(r)}$. We can ensure that for every possible true weight vector $\bar{\mathbf{w}}$ in (6.10) only if we use the hypothesis space $\mathcal{H}^{(r)}$ with $r = n$.

When using the model $\mathcal{H}^{(r)}$ with $r < n$, we cannot guarantee a zero bias term since we have no access to the true underlying weight vector $\bar{\mathbf{w}}$ in (6.10). In general, the bias term decreases with an increasing model size r (see Figure 6.6). We highlight that the bias term does not depend on the variance σ^2 of the noise ε in our toy model (6.10).

Let us now consider the variance term in (6.19). Using the properties of our toy model

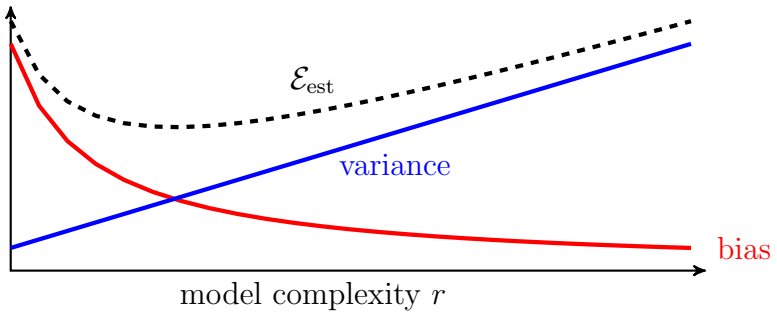


Figure 6.6: The estimation error $\mathcal{E}_{\text{est}}$ incurred by linear regression can be decomposed into a bias term B^2 and a variance term V (see (6.19)). These two components depend on the model complexity r in an opposite manner resulting in a bias-variance trade-off.

(see (6.10), (6.12) and (6.13))

$$V = \mathbb{E}\{\|\widehat{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2\} = \sigma^2 \text{tr}\big\{\mathbb{E}\{(\mathbf{X}_r^T\mathbf{X}_r)^{-1}\}\big\}. \tag{6.21}$$

By (6.12), the matrix $(\mathbf{X}_r^T\mathbf{X}_r)^{-1}$ is random and distributed according to an **inverse Wishart distribution** [49]. For $m_{\text{t}} > r+1$, its expectation is given as

$$\mathbb{E}\{(\mathbf{X}_r^T\mathbf{X}_r)^{-1}\} = 1/(m_{\text{t}} - r - 1)\mathbf{I}_{r\times r}. \tag{6.22}$$

By inserting (6.22) and $\text{tr}\{\mathbf{I}_{r\times r}\} = r$ into (6.21),

$$V = \mathbb{E}\{\|\widehat{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2\} = \sigma^2 r/(m_{\text{t}} - r - 1). \tag{6.23}$$

As indicated by (6.23), the variance term increases with increasing model complexity r (see Figure 6.6). This behaviour is in stark contrast to the bias term which decreases with increasing r. The opposite dependency of bias and variance on the model complexity is known as the **bias-variance trade-off**. Thus, the choice of model complexity r (see (6.11)) has to balance between a small variance and a small bias.

Generalization. Consider the linear hypothesis $h(\mathbf{x}) = \hat{\mathbf{w}}^T\mathbf{x}$ with the weight vector (6.14) which results in a minimum training error. We would like this predictor to generalize well to datapoints which are different from the training set. This generalization capability can

be quantified by the expected loss or risk

$$\begin{aligned}\mathcal{E}_{\rm pred} &= \mathbb{E}\big\{\big(y-\hat{y}\big)^2\big\} \\ &\overset{(6.10)}{=} \mathbb{E}\{\Delta\mathbf{w}^T\mathbf{x}\mathbf{x}^T\Delta\mathbf{w}\}+\sigma^2 \\ &\overset{(a)}{=} \mathbb{E}\{\mathbb{E}\{\Delta\mathbf{w}^T\mathbf{x}\mathbf{x}^T\Delta\mathbf{w}\mid\mathcal{D}\}\}+\sigma^2 \\ &\overset{(b)}{=} \mathbb{E}\{\Delta\mathbf{w}^T\Delta\mathbf{w}\}+\sigma^2 \\ &\overset{(6.17),(6.18)}{=} \mathcal{E}_{\rm est}+\sigma^2 \\ &\overset{(6.19)}{=} B^2+V+\sigma^2. \end{aligned} \tag{6.24}$$

Step (a) uses the law of total expectation [8] and step (b) uses that, conditioned on the dataset $\mathcal{D}$, the feature vector $\mathbf{x}$ of a new datapoint is a random vector with zero mean and a covariance matrix $\mathbb{E}\{\mathbf{x}\mathbf{x}^T\}=\mathbf{I}$ (see (6.12)).

According to (6.24), the average (expected) prediction error $\mathcal{E}_{\rm pred}$ is the sum of three components: (i) the bias B^2, (ii) the variance V and (iii) the noise variance σ^2. Figure 6.6 illustrates the typical dependency of the bias and variance on the model, which is parametrized by r.

The bias and variance, whose sum is the estimation error $\mathcal{E}_{\rm est}$, can be influenced by varying the model complexity r which is a design parameter. The noise variance σ^2 is the intrinsic accuracy limit of our toy model (6.10) and is not under the control of the ML engineer. It is impossible for any ML method - no matter how advanced it is - to achieve, on average, a prediction error smaller than the noise variance σ^2.

We finally highlight that our analysis of bias (6.20), variance (6.23) and the average prediction error (6.24) only applies if the observed datapoints are well modelled as realizations of random vectors according to (6.10), (6.12) and (6.13). The usefulness of this model for the data arising in a particular application has to be verified in practice by some validation techniques [77, 71].

An alternative approach for analyzing bias, variance and average prediction error of linear regression is to use simulations. Here, we generate a number of i.i.d. copies of the observed datapoints by some random number generator [4]. Using these i.i.d. copies, we can replace exact computations (expectations) by empirical approximations (sample averages).

6.5 The Bootstrap

Consider learning a hypothesis $\hat{h} \in \mathcal{H}$ by minimizing the average loss incurred on a dataset $\mathcal{D} = \{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\}$. The datapoints $\big(\mathbf{x}^{(i)}, y^{(i)}\big)$ are modelled as realizations of i.i.d. random variables. Let use denote the (common) probability distribution of these random variables by $p(\mathbf{x}, y)$.

If we interpret the datapoints $\big(\mathbf{x}^{(i)}, y^{(i)}\big)$ as realizations of random variables, also the learnt hypothesis $\hat{h}$ is a realization of a random variable. Indeed, the hypothesis $\hat{h}$ is obtained by solving an optimization problem (4.2) that involves realizations of random variables. The bootstrap is a method for estimating (parameters of) the probability distribution $p(\hat{h})$ [31].

Section 6.4 used a probabilistic model for datapoints to derive analytically (some parameters of) the probability distribution $p(\hat{h})$. While the analysis in Section 6.4 only applies to the specific probabilistic model (6.12), (6.13), the bootstrap can be used for datapoints drawn from an arbitrary probability distribution.

The core idea behind the bootstrap is to use the empirical distribution or histogram $\hat{p}(\mathbf{z})$ of the available datapoints $\mathcal{D}$ to generate B new datasets $\mathcal{D}^{(1)}, \ldots$. Each dataset is constructed such that is has the same size as the original dataset $\mathcal{D}$. The then solve ERM (4.2) for each dataset $\mathcal{D}^{(b)}$ to obtain the hypothesis $\hat{h}^{(b)}$. The hypothesis $\hat{h}^{(b)}$ is a realization of a random variable whose distribution is determined by the empirical distribution $\hat{p}(\mathbf{z})$ as well as the hypothesis space and loss function used in the ERM (4.2).

6.6 Diagnosing ML

compare training, validation and benchmark error. benchmark can be Bayes risk when using probabilistic model (such as i.i.d.), or human performance or risk of some other ML methods ("experts" in regret framework)

Consider a ML method which learns a hypothesis $\hat{h}$ using ERM (4.2) on a training set $\mathcal{D}^{(\text{train})}$ resulting in the training error

$$\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})}) = (1/|\mathcal{D}^{(\text{train})}|) \sum_{(\mathbf{x},y) \in \mathcal{D}^{(\text{train})}} \mathcal{L}((\mathbf{x}, y), \hat{h}).$$

We then validate the hypothesis by computing the validation error

$$\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})}) = (1/|\mathcal{D}^{(\text{val})}|) \sum_{(\mathbf{x},y) \in \mathcal{D}^{(\text{val})}} \mathcal{L}((\mathbf{x}, y), \hat{h}).$$

By comparing the two numbers $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})})$ and $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$ we diagnose the ML method. This diagnosis might provide insight into how the ML method could be improved.

In some applications we might know a benchmark $\mathcal{E}_0$ for the average loss that is considered as acceptable. Such a benchmark can be obtained using a probabilistic model for the datapoints. Given such a probabilistic model we can compute the minimum achievable expected loss or risk (4.1). This minimum risk can often be read from the posterior distribution $p(y|\mathbf{x})$ of the label y, given the features $\mathbf{x}$ of a datapoint.

Another option to obtain such a benchmark is by considering other ML methods ("experts") which are computationally more expensive. Finally, such a benchmark might simply be prescribed by the specification of the overall product which uses the ML method.

We can diagnose a ML method by comparing the training error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})})$ with the validation error $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$ and (if available) the benchmark $\mathcal{E}_0$.

- $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})}) \approx \mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})}) \approx E_0$: There is not much to improve here since the validation error is already on the desired error level. Moreover, the training error is not much smaller than the validation error which indicates that there is no overfitting and we cannot reduce the validation error by much.

- $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})}) \gg \mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})}) \approx E_0$: The ERM (4.2) results in a hypothesis $\hat{h}$ with sufficiently small training error but when applied to new datapoints, such as those in the validation set, the performance of $\hat{h}$ is significantly worse. This is an indicator for overfitting which can be addressed by using using a smaller hypothesis space. Reducing the size of the hypothesis space can be achieved by using only a subset of features in a linear model (3.1), by using a shorter decision tree (Section 3.10) or by using a smaller ANN (Section 3.11). Another option to avoid overfitting is to use regularization techniques, which will be discussed in Chapter 7.

- $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})}) \gg \mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$: This indicates that the method for solving the ERM (4.2) fails to find (an approximation of) the minimum in (4.2). Indeed, the training error obtained by solving the ERM (4.2) should typically be smaller than the validation error. When using GD for solving ERM, one reason for obtaining $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})}) \gg \mathcal{E}(\hat{h}|\mathcal{D}^{(\text{val})})$ could be that the step size α in the GD step (5.4) is chosen too large (see Figure 5.3-(b)).

6.7 Exercises

6.7.1 Validation Set Size

Consider a linear regression problem with datapoints characterized by a scalar feature and a numeric label. Assume datapoints are i.i.d. Gaussian with zero-mean and covariance matrix $\mathbf{C}$. How many datapoints do we need to include in the validation set such that with probability of at least 0.8 the validation error does not deviate by more than 20 percent from the expected loss or risk?

6.7.2 Validation Error Smaller Than Training Error?

Consider learning a linear hypothesis by minimizing the average squared error on some training set. The resulting linear predictor is then validated on some other validation set. Can you construct a training and validation set such that the validation error is strictly smaller than the training set?

Chapter 7

Regularization

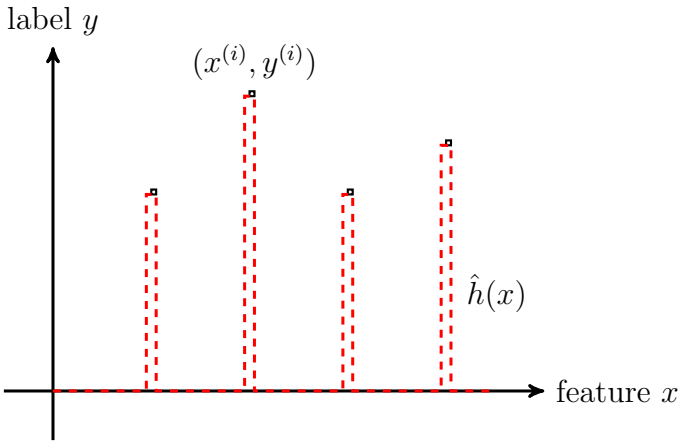


Figure 7.1: Many ML methods learn a hypothesis $\hat{h}$ by minimizing the average loss incurred by the predictions $\hat{h}(x^{(i)})$ on a training set $\big(x^{(i)}, y^{(i)}\big)$, for $i = 1, \ldots, m$. Regularization techniques try to estimate (or to anticipate) how the performance of a hypothesis $\hat{h}$ differs between the training set and other datapoints. The average loss incurred by $\hat{h}$ on arbitrary datapoints is typically larger than the average loss on the training set.

Many ML methods use the principle of ERM (see Chapter 4) to learn a hypothesis out of a hypothesis space by minimizing the average loss (training error) on a set of labeled datapoints (training set).

Using ERM as a guiding principle for ML methods makes sense only if the training error is a good indicator for the loss it incurs on other datapoints which are different from the training set.

Chapter 6 discussed how validation techniques can be used to verify if the hypothesis performs also well on datapoints outside the training set. This is achieved by measuring the

validation error as the average loss on a validation set which is different from the training set. The validation error serves as an estimate for the expected loss or risk of a hypothesis.

Regularization techniques replace the computation of a validation error of a hypothesis by constructing an estimate for the loss of the hypothesis incurred on arbitrary datapoints. In particular, regularization aims at estimating by how much the loss incurred on arbitrary datapoints exceeds the (optimistic) training error. This increase is estimated by adding different regularizations terms to the average loss in ERM. We discuss the resulting regularized ERM in Section 7.1.

Section 7.4 analyzes the effect of regularization for linear regression using a simple probabilistic model for data points. This analysis parallels the analysis of model validation in Section 6.4. In particular, we will obtain another instance of a bias-variance tradeoff. While this trade off was traced out by a discrete model complexity parameter in Section 6.4, here it is traced out by a continuous regularization parameter.

The regularization terms used in regularized ERM can be obtained in different ways. Section 7.2 discusses the construction of regularization terms by requiring the ML method to be robust against (small) random perturbations in the training data. Conceptually we replace each training data point by a random variable that fluctuates around it.

Section 7.3 discusses data augmentation methods as a simulation-based variant of the techniques discussed in Section 7.2. Data augmentation adds for each training data points a certain number of perturbed copies. One such copy of a training data point can be obtained by adding the realizations of random numbers to its features.

Semi-supervised learning problems refer to ML problems that involve a mix of labeled and unlabeled data points. Section 7.5 shows how to use the statistical properties of unlabeled data point to construct regularization terms which are used for regularized ERM on the (typically small) subset of labeled datapoints.

Multitask learning methods exploit similarities between different ML problems. As an example consider ML problem that use the same datapoints and their features but different labels. The similarity between ML problems could mean that the same subset of features is relevant for all ML problems. Section 7.6 designs regularization terms for individual ML problems by using their similarities.

7.1 Regularized ERM

It seems reasonable to avoid overfitting by pruning the hypothesis space $\mathcal{H}$, i.e., removing some of its elements. In particular, instead of solving (4.2) we solve the restricted ERM

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}'} \mathcal{E}(h|\mathcal{D}) \text{ with pruned hypothesis space } \mathcal{H}' \subset \mathcal{H}. \tag{7.1}$$

Another approach to avoid overfitting is to regularize the ERM (4.2) by adding a penalty term $\mathcal{R}(h)$ which somehow measures the complexity or non-regularity of a predictor map h using a non-negative number $\mathcal{R}(h) \in \mathbb{R}_+$. We then obtain the regularized ERM

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{E}(h|\mathcal{D}) + \mathcal{R}(h). \tag{7.2}$$

The additional term $\mathcal{R}(h)$ aims at approximating (or anticipating) the increase in the empirical risk of a predictor $\hat{h}$ when it is applied to new datapoints, which are different from the dataset $\mathcal{D}$ used to learn the predictor $\hat{h}$ by (7.2).

The two approaches (7.1) and (7.2), for making ERM (4.2) robust against overfitting are closely related. In particular, these two approaches are, in a certain sense, **dual** to each other: for a given restriction $\mathcal{H}' \subset \mathcal{H}$ we can find a penalty $\mathcal{R}(h)$ term such that the solutions of (7.1) and (7.2) coincide. Similarly for a many popular types of penalty terms $\mathcal{R}(h)$, we can find a restriction $\mathcal{H}' \subset \mathcal{H}$ such that the solutions of (7.1) and (7.2) coincide. This statements can be made precise using the theory of duality for optimization problems (see [7]).

In what follows we will analyze the occurrence of overfitting in Section **??** and then discuss in Section **??** how to avoid overfitting using regularization.

7.2 Robustness

Overfitting is a main challenges in applying modern ML methods. Modern ML methods use large hypothesis spaces that allow to represent highly non-linear predictor maps. Just by pure luck we can find one such predictor map that perfectly fits the training set resulting in zero training error and, in turn, solving ERM (4.2).

Overfitting is closely related to another property of ML methods: robustness. If a method overfits it will typically be not robust to small perturbations in the training data. The robustness to small perturbations in the data is almost a mandatory requirement for ML

methods to be useful in important application domains.

The ML methods discussed in Chapter 4 rest on the idealizing assumption that we have access to the true label values and feature values of a set of datapoints (the training set). However, the means by which the label and feature values are determing are prone to errors. These errors might stem from the measurement device itself (hardward failures) or might be due to modelling errors. We need ML methods that do not "break" if we feed it slightly perturbed label values for the training data.

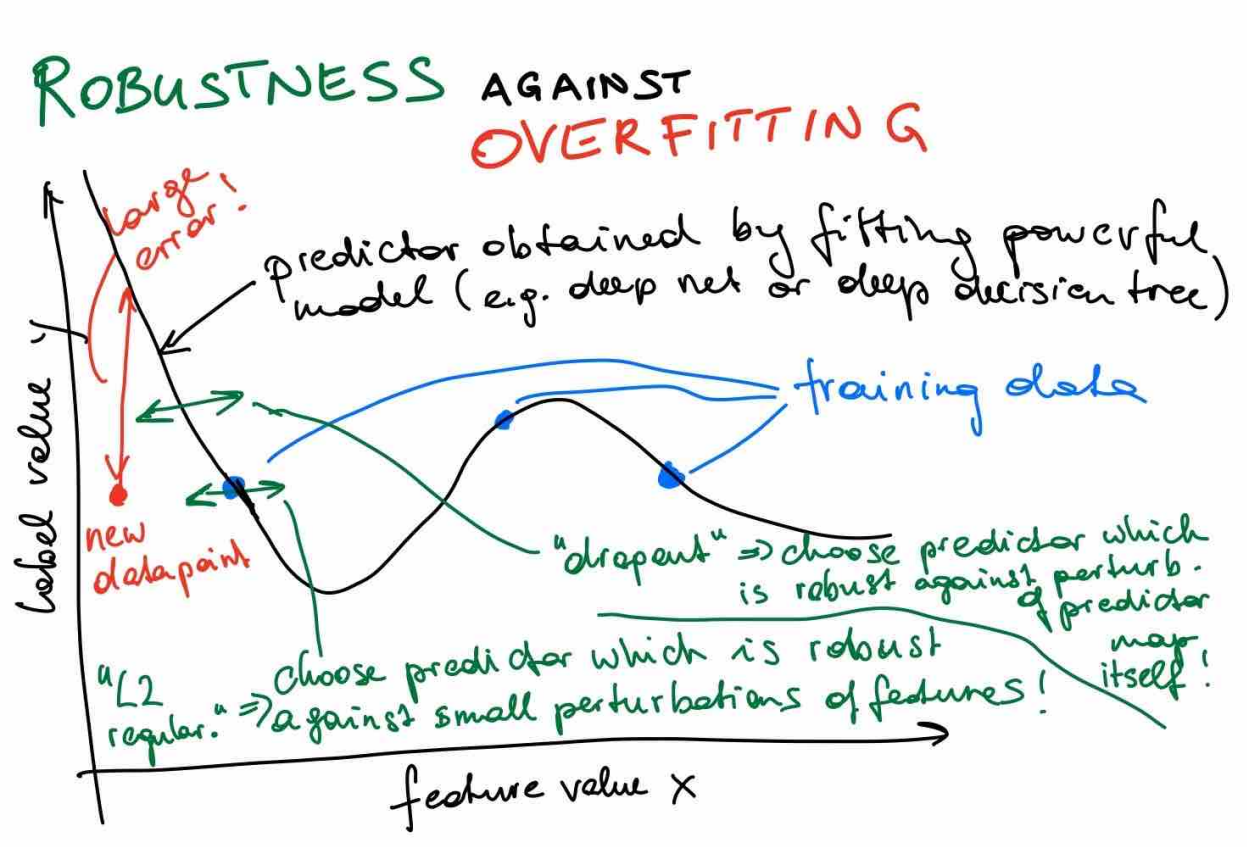


Figure 7.2: Modern ML methods allow to find a predictor map that perfectly fits training data. Such a predictor might perform poorly on a new datapoint outside the training set. To prevent learning such a predictor map we could require it to be robust against small perturbations in the features of the training datapoints or the predictor map itself.

7.3 Data Augmentation

implement robustness principle by augmenting dataset with random perturbations of original training data.

7.4 A Probabilistic Model for Regularization

As mentioned above, the overfitting of the training data $\mathcal{D}^{(\text{train})} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m_\text{t}}$ might be caused by choosing the hypothesis space too large. Therefore, we can avoid overfitting by making (pruning) the hypothesis space $\mathcal{H}$ smaller to obtain a new hypothesis space $\mathcal{H}_{\text{small}}$. This smaller hypothesis space $\mathcal{H}_{\text{small}}$ can be obtained by pruning, i.e., removing certain maps h, from $\mathcal{H}$.

A more general strategy is **regularization**, which amounts to modifying the loss function of an ML problem in order to favour a subset of predictor maps. Pruning the hypothesis space can be interpreted as an extreme case of regularization, where the loss functions become infinite for predictors which do not belong to the smaller hypothesis space $\mathcal{H}_{\text{small}}$.

In order to avoid overfitting, we have to augment our basic ERM approach (cf. (4.2)) by **regularization techniques**. According to [27], regularization aims at "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error." By generalization error, we mean the average prediction error (see (6.24)) incurred by a predictor when applied to new datapoints (different from the training set).

A simple but effective method to regularize the ERM learning principle, is to augment the empirical risk (5.7) of linear regression by the penalty term $\mathcal{R}(h^{(\mathbf{w})}) := \lambda \|\mathbf{w}\|_2^2$, which penalizes overly large weight vectors $\mathbf{w}$. Thus, we arrive at **regularized ERM**

$$\begin{aligned} \hat{\mathbf{w}}^{(\lambda)} &= \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}} \big[\mathcal{E}(h^{(\mathbf{w})}|\mathcal{D}^{(\text{train})}) + \lambda \|\mathbf{w}\|^2\big] \\ &= \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}} \big[(1/m_\text{t}) \sum_{i=1}^{m_\text{t}} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h^{(\mathbf{w})}) + \lambda \|\mathbf{w}\|^2\big], \end{aligned} \tag{7.3}$$

with the regularization parameter $\lambda > 0$. The parameter λ trades a small training error $\mathcal{E}(h^{(\mathbf{w})}|\mathcal{D})$ against a small norm $\|\mathbf{w}\|$ of the weight vector. In particular, if we choose a large value for λ, then weight vectors $\mathbf{w}$ with a large norm $\|\mathbf{w}\|$ are "penalized" by having a larger objective function and are therefore unlikely to be a solution (minimizer) of the optimization problem (7.3).

Specialising (7.3) to the squared error loss and linear predictors yields **regularized linear regression** (see (4.4)):

$$\hat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} \big[(1/m_{\rm t}) \sum_{i=1}^{m_{\rm t}} (y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})^2 + \lambda \|\mathbf{w}\|_2^2\big], \tag{7.4}$$

The optimization problem (7.4) is also known under the name **ridge regression** [31].

Using the feature matrix $\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m_{\rm t})}\big)^T$ and label vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(m_{\rm t})})^T$, we can rewrite (7.4) more compactly as

$$\hat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} \big[(1/m_{\rm t}) \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2\big]. \tag{7.5}$$

The solution of (7.5) is given by

$$\hat{\mathbf{w}}^{(\lambda)} = (1/m_{\rm t})((1/m_{\rm t})\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{7.6}$$

This reduces to the closed-form expression (6.14) when $\lambda = 0$ in which case regularized linear regression reduces to ordinary linear regression (see (7.4) and (4.4)). It is important to note that for $\lambda > 0$, the formula (7.6) is always valid, even when $\mathbf{X}^T\mathbf{X}$ is singular (not invertible). This implies, in turn, that for $\lambda > 0$ the optimization problem (7.5) (and (7.4)) have a unique solution (which is given by (7.6)).

We now study the effect of regularization on the bias, variance and average prediction error incurred by the predictor $h^{(\hat{\mathbf{w}}^{(\lambda)})}(\mathbf{x}) = \big(\hat{\mathbf{w}}^{(\lambda)}\big)^T\mathbf{x}$. To this end, we will again invoke the simple probabilistic toy model (see (6.10), (6.12) and (6.13)) used already in Section 6.4. In particular, we interpret the training data $\mathcal{D}^{(\rm train)} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m_{\rm t}}$ as realizations of i.i.d. random variables according to (6.10), (6.12) and (6.13).

As discussed in Section 6.4, the average prediction error is the sum of three components: the bias, the variance and the noise variance σ^2 (see (6.24)). The bias of regularized linear regression (7.4) is obtained as

$$B^2 = \big\|(\mathbf{I} - \mathbb{E}\{(\mathbf{X}^T\mathbf{X} + m\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{X}\})\bar{\mathbf{w}}\big\|_2^2. \tag{7.7}$$

For sufficiently large sample size $m_{\rm t}$ we can use the approximation

$$\mathbf{X}^T\mathbf{X} \approx m_{\rm t}\mathbf{I} \tag{7.8}$$

such that (7.7) can be approximated as

$$B^2 \approx \big\|(\mathbf{I}-(\mathbf{I}+\lambda\mathbf{I})^{-1})\bar{\mathbf{w}}\big\|_2^2 = \sum_{l=1}^{n} \frac{\lambda}{1+\lambda}\bar{w}_l^2. \tag{7.9}$$

Compare the (approximate) bias term (7.9) of regularized linear regression with the bias term (6.20) of ordinary linear regression. Using regularization typically increases the bias. The bias increases with increasing regularization parameter λ.

The variance of regularized linear regression (7.4) satisfies

$$V = (\sigma^2/m_{\rm t}^2)\times \mathrm{tr}\big\{\mathbb{E}\{((1/m_{\rm t})\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{X}((1/m_{\rm t})\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\}\big\}. \tag{7.10}$$

Inserting the approximation (7.8) into (7.10),

$$V \approx \sigma^2(n/m_{\rm t})(1/(1+\lambda)). \tag{7.11}$$

According to (7.11), the variance of regularized linear regression decreases with increasing regularization parameter λ. This is the opposite behaviour as for the bias (7.9), which increases with increasing λ.

As illustrated in Figure 7.3, the choice for λ has to balance between the bias B^2 (7.9) (which increases with increasing λ) and the variance V (7.11) (which decreases with increasing λ). This results in bias-variance trade off for the choice of λ. We have discussed a similar a bias-variance tradeoff already in Section 6.4. However, in contrast to the trade-off obtained from the discrete model complexity r in (6.11), here we obtain a continuous trade-off via the real-valued parameter λ.

So far, we only have discussed the statistical effect of regularization on the resulting ML method. Regularizations allows to trade off bias against variance in order to reduce the risk of the learnt hypothesis.

There is also a computational aspect of regularization. Adding a regularization term to the ERM changes the shape of the objective function that is minimized by a learning method. The shape of the objective function determines the difficulty in finding a (approximate) minimizer. Thus, regularization influences the computational complexity of the resulting ML method.

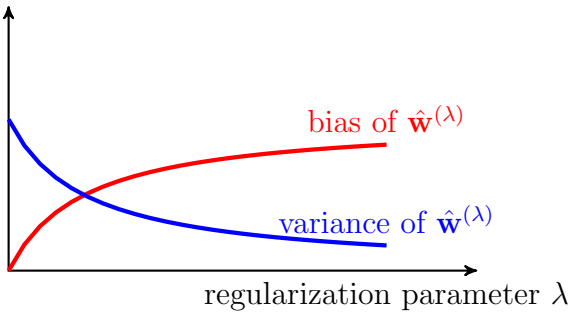


Figure 7.3: The bias and variance of regularized linear regression depend on the regularization parameter λ in an opposite manner resulting in a bias-variance tradeoff.

Note that the objective function in (7.5) is a smooth (infinitely often differentiable) convex function. Similar to linear regression, we can solve the regularization linear regression problem using GD (2.6) (see Algorithm 4).

Adding the regularization term $\lambda\|\mathbf{w}\|_2^2$ to the objective function of linear regression **speeds up GD**. To verify this claim, we first rewrite (7.5) as the quadratic problem

$$\min_{\mathbf{w}\in\mathbb{R}^n} \underbrace{(1/2)\mathbf{w}^T\mathbf{Q}\mathbf{w} - \mathbf{q}^T\mathbf{w}}_{=f(\mathbf{w})}$$
$$\text{with } \mathbf{Q} = (1/m)\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}, \mathbf{q} = (1/m)\mathbf{X}^T\mathbf{y}. \tag{7.12}$$

This is similar to the quadratic optimization problem (4.7) underlying linear regression but with different matrix $\mathbf{Q}$. It turns out that the convergence speed of GD (see (5.4)) applied to solving a quadratic problem of the form (7.12) depends crucially on the condition number $\kappa(\mathbf{Q}) \geq 1$ of the psd matrix $\mathbf{Q}$ [35]. In particular, GD methods are fast if the condition number $\kappa(\mathbf{Q})$ is small (close to 1).

This condition number is given by $\frac{\lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X})}{\lambda_{\min}((1/m)\mathbf{X}^T\mathbf{X})}$ for ordinary linear regression (see (4.7)) and given by $\frac{\lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X})+\lambda}{\lambda_{\min}((1/m)\mathbf{X}^T\mathbf{X})+\lambda}$ for regularized linear regression (7.12). For increasing regularization parameter λ, the condition number obtained for regularized linear regression (7.12) tends to 1:

$$\lim_{\lambda\to\infty} \frac{\lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X}) + \lambda}{\lambda_{\min}((1/m)\mathbf{X}^T\mathbf{X}) + \lambda} = 1. \tag{7.13}$$

Thus, according to (7.13), the GD implementation of regularized linear regression (see Algorithm 4) with a large value of the regularization parameter λ in (7.4) will converge faster compared to GD for linear regression (see Algorithm 1).

Algorithm 4 "Regularized Linear Regression via GD"

Input: labeled dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ containing feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and labels $y^{(i)} \in \mathbb{R}$; GD step size $\alpha > 0$.

Initialize:set $\mathbf{w}^{(0)} := \mathbf{0}$; set iteration counter $k := 0$

1: **repeat**

2: $k := k + 1$ (increase iteration counter)

3: $\mathbf{w}^{(k)} := (1 - \alpha\lambda)\mathbf{w}^{(k-1)} + \alpha(2/m)\sum_{i=1}^{m}(y^{(i)} - \big(\mathbf{w}^{(k-1)}\big)^T\mathbf{x}^{(i)})\mathbf{x}^{(i)}$ (do a GD step (5.4))

4: **until** convergence

Output: $\mathbf{w}^{(k)}$ (which approximates $\hat{\mathbf{w}}^{(\lambda)}$ in (7.5))

Let us finally point out a close relation between regularization (which amounts to adding the term $\lambda\|\mathbf{w}\|^2$ to the objective function in (7.3)) and model selection (see Section 6.3). The regularized ERM (7.3) can be shown (see [7, Ch. 5]) to be equivalent to

$$\hat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}^{(\lambda)}} (1/m_{\mathrm{t}}) \sum_{i=1}^{m_{\mathrm{t}}} (y^{(i)} - h^{(\mathbf{w})}(\mathbf{x}^{(i)}))^2 \tag{7.14}$$

with the restricted hypothesis space

$$\mathcal{H}^{(\lambda)} := \{h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R} : h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \\ \text{, with some } \mathbf{w} \text{ satisfying } \|\mathbf{w}\|^2 \leq C(\lambda)\} \subset \mathcal{H}^{(n)}. \tag{7.15}$$

For any given value λ, we can find a bound $C(\lambda)$ such that solutions of (7.3) coincide with the solutions of (7.14). Thus, by solving the regularized ERM (7.3) we are performing implicitly model selection using a continuous ensemble of hypothesis spaces $\mathcal{H}^{(\lambda)}$ given by (7.15). In contrast, the simple model selection strategy considered in Section 6.3 uses a discrete sequence of hypothesis spaces.

7.5 Semi-Supervised Learning

Can we use unlabelled datapoints to construct better regularizers? We could use unlabled data to learn some subspace of features that are most relevant ? (relation to feature learning ?)

7.6 Multitask Learning

Remember that a formal ML problem is specified by identifying datapoints, their features and labels, a model (hypothesis space) and loss function. Note that we can use the very same raw data, model and loss function and still define many different ML problems by using different choices for the label. Multitask learning aims at exploiting relations between similar ML problems or tasks.

Consider the ML problem (task) of predicting the confidence level of a hand-drawing showing an apple. To learn such a predictor we use a collection of hand-drawings. We then annotate each hand-drawing by the object that it depicts. These annotations can be used to define different labels of the hand-drawings. One choice for the label could be to define the label $y = 1$ if a hand-drawing shows an apple and $y = 0$ if it does not show a hand-drawing. Another definition for the label could be based on the fact that the hand-drawing shows an apple or not.

Different choices for the label result in different ML problems. These different ML problems are related since they are based on the same data points, the hand-drawings. Some ML problems (label choices) are more difficult to answer while others are easier to answer, e.g., if there no single hand-drawing of an orange but many drawings of an apple.

Consider the ML problem arising from guiding the operation of a mower robot. For a mowing robot, it is important to determine if it is currently on grassland or not. Let us assume the mower robot is equipped with an on-board camera which allows to take snapshots which are characterized by a feature vector $\mathbf{x}$ (see Figure 2.4). We could then define the label as either $y = 1$ if the snapshot suggests that the mower is on grassland and $y = -1$ if not. However, we might be interested in a finer-grained information about the floor type and define the label as $y = 1$ for grassland, $y = 0$ for soil and $y = -1$ for when the mower is on tiles. The latter problem is more difficult since we have to distinguish between three different types of floor ("grass" vs. "soil" vs. "tiles") whereas for the former problem we only have to distinguish between two types of floor ("grass" vs. "no grass").

7.7 Exercises

7.7.1 Ridge Regression as Quadratic Form

Consider linear hypothesis space consisting of linear maps parameterized by weights $\mathbf{w}$. We try to find the best linear map by minimizing the regularized average squared error loss

(empirical risk) incurred on some labeled training datapoints $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$. As regularizer we use $\|\mathbf{w}\|^2$, yielding the following learning problem

$$\min_{\mathbf{w}} f(\mathbf{w}) = \sum_{i=1}^{m} \ldots + \|\mathbf{w}\|_2^2.$$

Is it possible to write the objective function $f(\mathbf{w})$ as a convex quadratic form $f(\mathbf{w}) = \mathbf{w}^T\mathbf{C}\mathbf{w} + \mathbf{b}\mathbf{w} + c$? If this is possible, how are the matrix $\mathbf{C}$, vector $\mathbf{b}$ and constant c related to the feature vectors and labels of the training data ?

7.7.2 Regularization or Model Selection

Consider datapoints characterized by $n = 10$ features $\mathbf{x} \in \mathbb{R}^n$ and a single numeric label y. We want to learn a linear hypothesis $h(x) = \mathbf{w}^T\mathbf{x}$ by minimizing the average squared error on the training set $\mathcal{D}$ of size $m = 4$. We could learn such a hypothesis by two approaches. The first approach is split the dataset into training and validation set of the same size (two). Then we consider all models that are given by linear hypothesis with weight vectors have at most two non-zeros. We use the validation set to choose between these models. The second approach is learn a linear hypothesis with arbitrary weight vector $\mathbf{w}$ but using regularization (ridge).

Chapter 8

Clustering

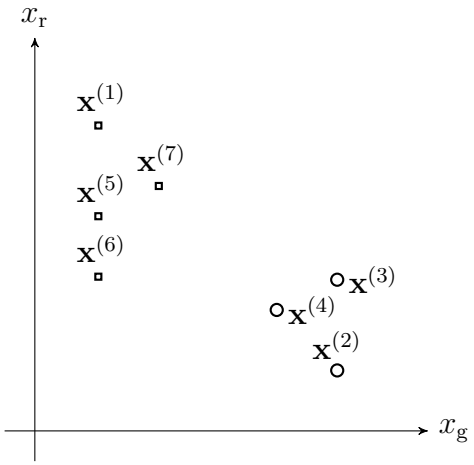


Figure 8.1: A scatterplot depicting some images. The i-th image is represented by the feature vector $\mathbf{x}^{(i)} = (x_{\rm r}^{(i)}, x_{\rm g}^{(i)})^T$ with the average redness $x_{\rm r}^{(i)}$ and average greenness $x_{\rm g}^{(i)}$ of the image.

Up to now, we mainly considered ML methods which required some labeled training data in order to learn a good predictor or classifier. We will now start to discuss ML methods which do not make use of labels. These methods are often referred to as "unsupervised" since they do not require a supervisor (or teacher) which provides the labels for datapoints in a training set.

An important class of unsupervised methods, known as clustering methods, aims at grouping datapoints into few subsets (or **clusters**). While there is no unique formal definition, we understand by cluster a subset of datapoints which are more similar to each other than

to the remaining datapoints (belonging to different clusters). Different clustering methods are obtained for different ways to measure the "similarity" between datapoints.

There are two main flavours of clustering methods:

- hard clustering (see Section 8.1)
- and soft clustering methods (see Section 8.2).

Within hard clustering, each datapoint $\mathbf{x}^{(i)}$ belongs to one and only one cluster. In contrast, soft clustering methods assign a datapoint $\mathbf{x}^{(i)}$ to several different clusters with varying degree of belonging (confidence).

Clustering methods determine for each datapoint $\mathbf{z}^{(i)}$ a cluster assignment $y^{(i)}$. The cluster assignment $y^{(i)}$ encodes the cluster to which the datapoint $\mathbf{x}^{(i)}$ is assigned. For hard clustering with a prescribed number of k clusters, the cluster assignments $y^{(i)} \in \{1, \ldots, k\}$ represent the index of the cluster to which $\mathbf{x}^{(i)}$ belongs.

In contrast, soft clustering methods allow each datapoint to belong to several different clusters. The degree with which datapoint $\mathbf{x}^{(i)}$ belongs to cluster $c \in \{1, \ldots, k\}$ is represented by the degree of belonging $y_c^{(i)} \in [0, 1]$, which we stack into the vector $\mathbf{y}^{(i)} = \left(y_1^{(i)}, \ldots, y_k^{(i)}\right)^T \in [0, 1]^k$. Thus, while hard clustering generates non-overlapping clusters, the clusters produced by soft clustering methods may overlap.

We intentionally used the same symbol $y^{(i)}$ for cluster assignments of a datapoint as we used to denote an associated label in classification problems. There is a strong conceptual link between clustering and classification. We can interpret clustering as an extreme case of classification without having access to any labeled training data, i.e., we do not know the label of any datapoint.To find the correct labels (cluster assignments) $y_c^{(i)}$, clustering method rely solely on the intrinsic geometry of the datapoints.

8.1 Hard Clustering with K-Means

In what follows we assume that datapoints $\mathbf{z}^{(i)}$, for $i = 1, \ldots, m$, are characterized by feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and measure similarity between datapoints using the Euclidean distance $\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|$. With a slight abuse of notation, we will occasionally denote a datapoint $\mathbf{z}^{(i)}$ using its feature vector $\mathbf{x}^{(i)}$. In general, the feature vector is only a (incomplete) representation of a datapoint but it is customary in many unsupervised ML methods to identify a datapoint with its features. Thus, we consider two datapoints $\mathbf{z}^{(i)}$ and $\mathbf{z}^{(j)}$ similar if $\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|$ is small. Moreover, we assume the number k of clusters prescribed.

A simple method for hard clustering is the "k-means" algorithm which requires the number k of clusters to specified before-hand. The idea underlying k-means is quite simple. First, given a current guess for the cluster assignments $y^{(i)}$, determine the cluster means $\mathbf{m}^{(c)} = \frac{1}{|\{i: y^{(i)}=c\}|} \sum_{i: y^{(i)}=c} \mathbf{x}^{(i)}$ for each cluster. Then, in a second step, update the cluster assignments $y^{(i)} \in \{1, \ldots, k\}$ for each datapoint $\mathbf{x}^{(i)}$ based on the nearest cluster mean. By iterating these two steps we obtain Algorithm 5.

Algorithm 5 "k-means"

Input: dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^m$; number k of clusters.

Initialize: choose initial cluster means $\mathbf{m}^{(c)}$ for $c = 1, \ldots, k$.

1: **repeat**

2: for each datapoint $\mathbf{x}^{(i)}$, $i = 1, \ldots, m$, do

$$y^{(i)} \in \operatorname*{argmin}_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\| \text{ (update cluster assignments)} \tag{8.1}$$

3: for each cluster $c = 1, \ldots, k$ do

$$\mathbf{m}^{(c)} = \frac{1}{|\{i : y^{(i)} = c\}|} \sum_{i: y^{(i)}=c} \mathbf{x}^{(i)} \text{ (update cluster means)} \tag{8.2}$$

4: **until** convergence

Output: cluster assignments $y^{(i)} \in \{1, \ldots, k\}$

In (8.1) we denote by $\operatorname*{argmin}_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\|$ the set of all cluster indices $c \in \{1, \ldots, k\}$ such that $\|\mathbf{x}^{(i)} - \mathbf{m}^{(c)}\| = \min_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\|$.

The k-means algorithm requires the specification of initial choices for the cluster means $\mathbf{m}^{(c)}$, for $c = 1, \ldots, k$. There is no unique optimal strategy for the initialization but several heuristic strategies can be used. One option is to initialize the cluster means with i.i.d.

realizations of a random vector $\mathbf{m}$ whose distribution is matched to the dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$, e.g., $\mathbf{m} \sim \mathcal{N}(\hat{\mathbf{m}}, \widehat{\mathbf{C}})$ with sample mean $\hat{\mathbf{m}} = (1/m)\sum_{i=1}^{m} \mathbf{x}^{(i)}$ and the sample covariance $\widehat{\mathbf{C}} = (1/m)\sum_{i=1}^{m}(\mathbf{x}^{(i)}-\hat{\mathbf{m}})(\mathbf{x}^{(i)}-\hat{\mathbf{m}})^T$. Another option is to choose the cluster means $\mathbf{m}^{(c)}$ by randomly selecting k different data points $\mathbf{x}^{(i)}$. The cluster means might also be chosen by evenly partitioning the principal component of the dataset (see Chapter 9).

We now show that k-means can be interpreted as a variant of ERM. To this end we define the empirical risk as the **clustering error**,

$$\mathcal{E}\big(\{\mathbf{m}^{(c)}\}_{c=1}^{k}, \{y^{(i)}\}_{i=1}^{m} \mid \mathcal{D}\big) = (1/m)\sum_{i=1}^{m} \Big\|\mathbf{x}^{(i)} - \mathbf{m}^{(y^{(i)})}\Big\|^2. \tag{8.3}$$

Note that the empirical risk (8.3) depends on the current guess for the cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^{k}$ and cluster assignments $\{y^{(i)}\}_{i=1}^{m}$.

Finding the global optimum of the function (8.3), over all possible cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^{k}$ and cluster assignments $\{y^{(i)}\}_{i=1}^{m}$, is difficult as the function is non-convex. However, minimizing (8.3) only with respect to the cluster assignments $\{y^{(i)}\}_{i=1}^{m}$ but with the cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^{k}$ held fixed is easy. Similarly, minimizing (8.3) over the choices of cluster means with the cluster assignments held fixed is also straightforward. This observation is used by Algorithm 5: it is alternatively minimizing $\mathcal{E}$ over all cluster means with the assignments $\{y^{(i)}\}_{i=1}^{m}$ held fixed and minimizing $\mathcal{E}$ over all cluster assignments with the cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^{k}$ held fixed.

The interpretation of Algorithm 5 as a method for minimizing the cost function (8.3) is useful for convergence diagnosis. In particular, we might terminate Algorithm 5 if the decrease of the objective function $\mathcal{E}$ is below a prescribed (small) threshold.

A practical implementation of Algorithm 5 needs to fix three issues:

- Issue 1: We need to specify a "tie-breaking strategy" to handle the case when several different cluster indices $c \in \{1, \ldots, k\}$ achieve the minimum value in (8.1).

- Issue 2: We need to specify how to handle the situation when after a cluster assignment update (8.1), there is a cluster c with no datapoints are associated with it, i.e., $|\{i : y^{(i)} = c\}| = 0$. In this case, the cluster means update (8.2) would be not well defined for the cluster c.

- Issue 3: We need to specify a stopping criterion ("checking convergence").

The following algorithm fixes those three issues in a particular way [29].

Algorithm 6 "k-Means II" (slight variation of "Fixed Point Algorithm" in [29])

Input: dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^m$; number k of clusters; tolerance $\varepsilon \geq 0$.

Initialize: choose initial cluster means $\big\{\mathbf{m}^{(c)}\big\}_{c=1}^k$ and cluster assignments $\big\{y^{(i)}\big\}_{i=1}^m$; set iteration counter $k := 0$; compute $E^{(k)} = \mathcal{E}\big(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathcal{D}\big)$;

1: **repeat**

2: for all datapoints $i = 1, \ldots, m$, update cluster assignment

$$y^{(i)} := \min\{\operatorname*{argmin}_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\|\} \text{ (update cluster assignments)} \tag{8.4}$$

3: for all clusters $c = 1, \ldots, k$, update the activity indicator

$$b^{(c)} := \begin{cases} 1 & \text{if } |\{i : y^{(i)} = c\}| > 0 \\ 0 & \text{else.} \end{cases}$$

4: for all $c = 1, \ldots, k$ with $b^{(c)} = 1$, update cluster means

$$\mathbf{m}^{(c)} := \frac{1}{|\{i : y^{(i)} = c\}|} \sum_{i : y^{(i)} = c} \mathbf{x}^{(i)} \text{ (update cluster means)} \tag{8.5}$$

5: $k := k + 1$ (increment iteration counter)

6: $E^{(k)} = \mathcal{E}\big(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathcal{D}\big)$ (see (8.3))

7: **until** $E^{(k-1)} - E^{(k)} \leq \varepsilon$

Output: cluster assignments $y^{(i)} \in \{1, \ldots, k\}$ and cluster means $\mathbf{m}^{(c)}$

The variables $b^{(c)} \in \{0,1\}$ indicate if cluster c is active ($b^{(c)} = 1$) or cluster c is inactive ($b^{(c)} = 0$), in the sense of having no datapoints assigned to it during the preceding cluster assignment step (8.4). We use the cluster activity inductors $b^{(c)}$ to make sure that the mean update (8.5) only to clusters c with at least one datapoint $\mathbf{x}^{(i)}$.

It can be shown that Algorithm 6 amounts to a fixed-point iteration

$$\{y^{(i)}\}_{i=1}^{m} \mapsto \mathcal{P}\{y^{(i)}\}_{i=1}^{m} \tag{8.6}$$

with a particular operator $\mathcal{P}$ (which depends on the dataset $\mathcal{D}$).

Each iteration of Algorithm 6 updates the cluster assignments $y^{(i)}$ by applying the operator $\mathcal{P}$. By interpreting Algorithm 6 as a fixed-point iteration (8.6), the authors of [29, Thm. 2] present an elegant proof of the convergence of Algorithm 6 within a finite number of iterations (even for $\varepsilon = 0$). What is more, after running Algorithm 6 for a finite number of iterations the cluster assignments $\{y^{(i)}\}_{i=1}^{m}$ do not change any more.

We illustrate the operation of Algorithm 6 in Figure 8.2. Each column corresponds to one iteration of Algorithm 6. The upper picture in each column depicts the update of cluster means while the lower picture shows the update of the cluster assignments during each iteration.

While Algorithm 6 is guaranteed to terminate after a finite number of iterations, the delivered cluster assignments and cluster means might only be (approximations) of local minima of the clustering error (8.3) (see Figure 8.3).

To escape local minima, it is useful to run Algorithm 6 several times, using different initializations for the cluster means, and picking the cluster assignments $\{y^{(i)}\}_{i=1}^{m}$ with smallest clustering error (8.3).

Up till now, we have assumed the number k of clusters to be given before hand. In some applications it is unclear what a good choice for k is. The choice for the number k of clusters depends on how the clustering methods is used within an overall ML application. If the clustering method serves as a pre-processing for a supervised ML problem, we could try out different values of the number k and determine, for each choice k, the corresponding validation error. We pick then the value of k which results in the smallest validation error.

Another approach to choosing k is the so-called "elbow-method". We run the k-means Algorithm 6 for different choice for k. For each choice of k, Algorithm 6 delivers a different (approximate) optimum empirical error

$$\mathcal{E}^{(k)} = \mathcal{E}\big(\{\mathbf{m}^{(c)}\}_{c=1}^{k}, \{y^{(i)}\}_{i=1}^{m} \mid \mathcal{D}\big).$$

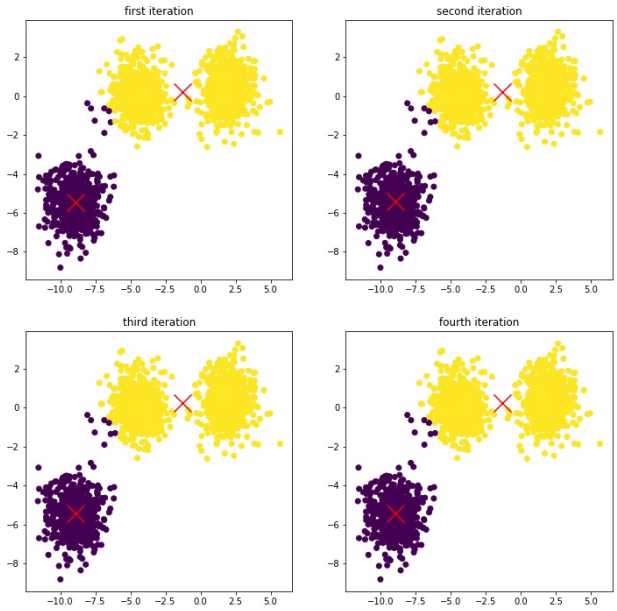


Figure 8.2: Evolution of cluster means and cluster assignments within k-means.

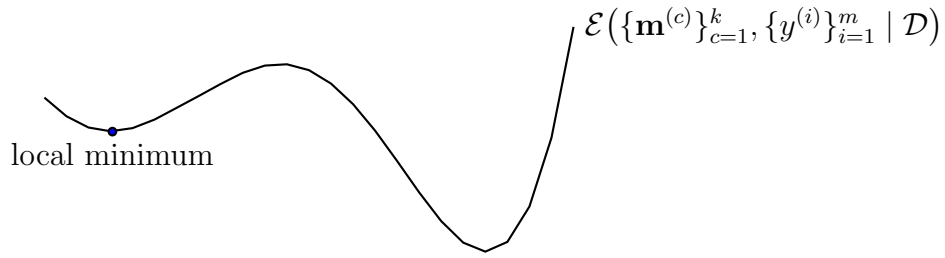


Figure 8.3: The clustering error $\mathcal{E}\big(\{\mathbf{m}^{(c)}\}_{c=1}^{k}, \{y^{(i)}\}_{i=1}^{m} \mid \mathcal{D}\big)$ (see (8.3)), which is minimized by k-means, is a non-convex function of the cluster means and assignments. The k-means algorithm can get trapped around a local minimum.

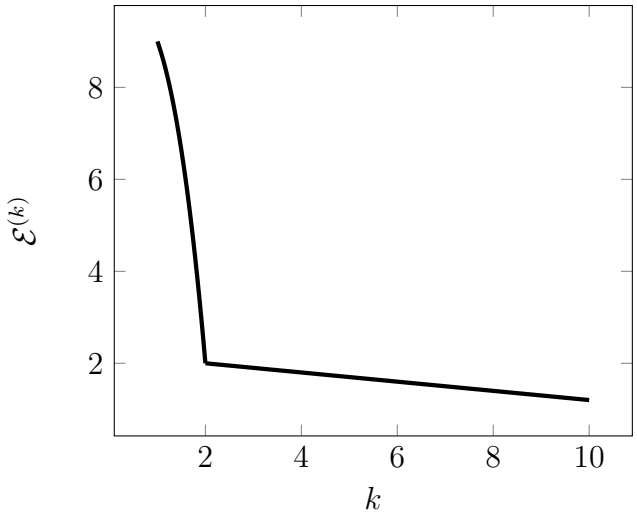


Figure 8.4: The clustering error $\mathcal{E}^{(k)}$ achieved by k-means for increasing number k of clusters.

We then plot the minimum empirical error $\mathcal{E}^{(k)}$ as a function of the number k of clusters. Figure 8.4 depicts an example for such a plot which typically starts with a steep decrease for increasing k and then flattening out for larger values of k. Finally, the choice of k might be guided by some probabilistic model which specifies a prior distribution of the values for k.

8.2 Soft Clustering with Gaussian Mixture Models

The cluster assignments obtained from hard-clustering methods, such as Algorithm 6, provide rather coarse-grained information. Indeed, even if two data points $\mathbf{x}^{(i)}, \mathbf{x}^{(j)}$ are assigned to the same cluster c, their distances to the cluster mean $\mathbf{m}^{(c)}$ might be very different. For some applications, we would like to have a more fine-grained information about the cluster assignments.

Soft-clustering methods provide such fine-grained information by explicitly modelling the degree (or confidence) by which a particular datapoint belongs to a particular cluster. More precisely, soft-clustering methods track for each datapoint $\mathbf{x}^{(i)}$ the degree of belonging to each of the clusters $c \in \{1, \ldots, k\}$.

A principled approach to modelling a degree of belonging to different clusters is based on a probabilistic (generative) model for the dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$. This approach identifies a cluster with a probability distribution. One popular choice for this distribution is the

multivariate normal distribution

$$\mathcal{N}(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{\sqrt{\det\{2\pi\boldsymbol{\Sigma}\}}} \exp\big(-(1/2)(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\big) \tag{8.7}$$

of a Gaussian random vector with mean $\boldsymbol{\mu}$ and (invertible) covariance matrix $\boldsymbol{\Sigma}$.[1]

Each cluster $c \in \{1,\ldots,k\}$ is represented by a distribution of the form (8.7) with a cluster-specific mean $\boldsymbol{\mu}^{(c)} \in \mathbb{R}^n$ and cluster-specific covariance matrix $\boldsymbol{\Sigma}^{(c)} \in \mathbb{R}^{n\times n}$.

Since we do not know before-hand the cluster assignment $c^{(i)}$ of the datapoint $\mathbf{x}^{(i)}$, we model $c^{(i)}$ as a random variable with probability distribution

$$p_c := \mathrm{P}(c^{(i)} = c) \text{ for } c = 1,\ldots,k. \tag{8.8}$$

The (prior) probabilities p_c, for $c = 1,\ldots,k$, are either assumed to be known or estimated from data, e.g., using a maximum likelihood method.

The random cluster assignment $c^{(i)}$ selects the cluster-specific distribution (8.7) of the random datapoint $\mathbf{x}^{(i)}$,

$$\mathrm{P}(\mathbf{x}^{(i)}|c^{(i)}) = \mathcal{N}(\mathbf{x};\boldsymbol{\mu}^{(c^{(i)})},\boldsymbol{\Sigma}^{(c^{(i)})}) \tag{8.9}$$

with mean vector $\boldsymbol{\mu}^{(c)}$ and covariance matrix $\boldsymbol{\Sigma}^{(c)}$.

The modelling of cluster assignments $c^{(i)}$ as (unobserved) random variables suggests a natural definition for degree $y_c^{(i)}$ by which datapoint $\mathbf{x}^{(i)}$ belongs to cluster c. We define the degree $y_c^{(i)}$ of datapoint $\mathbf{x}^{(i)}$ belonging to cluster c as the "a-posteriori" probability of the cluster assignment $c^{(i)}$ being equal to $c \in \{1,\ldots,k\}$:

$$y_c^{(i)} := \mathrm{P}(c^{(i)} = c|\mathcal{D}). \tag{8.10}$$

By their very definition (8.10), the degrees of belonging $y_c^{(i)}$ always sum to one,

$$\sum_{c=1}^{k} y_c^{(i)} = 1 \text{ for each } i = 1,\ldots,m. \tag{8.11}$$

It is important to note that we use the conditional cluster probability (8.10), conditioned on the dataset $\mathcal{D}$, for defining the degree of belonging $y_c^{(i)}$. This is reasonable since the degree of belonging $y_c^{(i)}$ depends on the overall (cluster) geometry of the data set $\mathcal{D}$.

A probabilistic model for the observed datapoints $\mathbf{x}^{(i)}$ is obtained by considering each

[1] Note that the distribution (8.7) is only defined for an invertible (non-singular) covariance matrix $\boldsymbol{\Sigma}$.

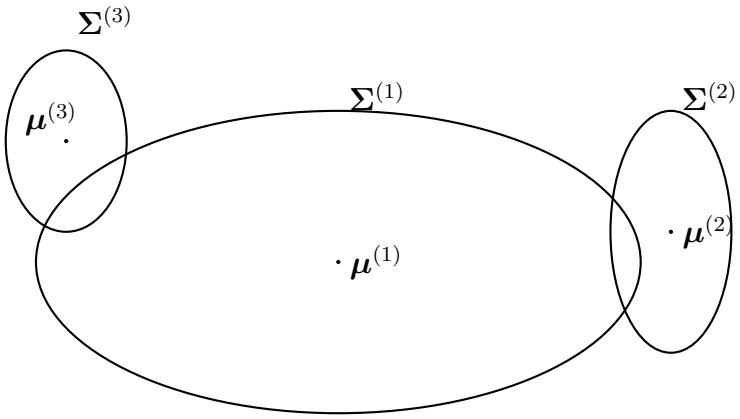


Figure 8.5: The GMM (8.12) yields a probability density function which is a weighted sum of multivariate normal distributions $\mathcal{N}(\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)})$. The weight of the c-th component is the cluster probability $\mathrm{P}(c^{(i)} = c)$.

datapoint $\mathbf{x}^{(i)}$ as a random draw from the distribution $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}^{(c^{(i)})}, \boldsymbol{\Sigma}^{(c^{(i)})})$ with some cluster $c^{(i)}$.

Since the cluster indices $c^{(i)}$ are unknown,[2] we model them as random variables. In particular, we model the cluster indices $c^{(i)}$ as i.i.d. with probabilities$p_c = \mathrm{P}(c^{(i)} = c)$.

The overall probabilistic model (8.9), (8.8) amounts to a **Gaussian mixture model** (GMM). The marginal distribution $\mathrm{P}(\mathbf{x}^{(i)})$, which is the same for all datapoints $\mathbf{x}^{(i)}$, is a (additive) mixture of multivariate Gaussian distributions,

$$\mathrm{P}(\mathbf{x}^{(i)}) = \sum_{c=1}^{k} \underbrace{\mathrm{P}(c^{(i)} = c)}_{p_c} \underbrace{\mathrm{P}(\mathbf{x}^{(i)} | c^{(i)} = c)}_{\mathcal{N}(\mathbf{x}^{(i)}; \boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)})}. \tag{8.12}$$

The cluster assignments $c^{(i)}$ are hidden (unobserved) random variables. We thus have to infer or estimate these variables from the observed datapoints $\mathbf{x}^{(i)}$ which are i.i.d. realizations of the GMM (8.12).

Using the GMM (8.12) for explaining the observed datapoints $\mathbf{x}^{(i)}$ turns the clustering problem into a **statistical inference** or **parameter estimation problem** [40, 47]. The estimation problem is to estimate the true underlying cluster probabilities p_c (see (8.8)), cluster means $\boldsymbol{\mu}^{(c)}$ and cluster covariance matrices $\boldsymbol{\Sigma}^{(c)}$ (see (8.9)) from the observed datapoints $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$. The datapoints $\mathbf{x}^{(i)}$ are realizations of i.i.d. random vectors with the common

[2] After all, the goal of soft-clustering is to estimate the cluster indices $c^{(i)}$.

probability distribution (8.12).

We denote the estimates for the GMM parameters by $\hat{p}_c (\approx p_c)$, $\mathbf{m}^{(c)} (\approx \boldsymbol{\mu}^{(c)})$ and $\mathbf{C}^{(c)} (\approx \boldsymbol{\Sigma}^{(c)})$, respectively. Based on these estimates, we can then compute an estimate $\hat{y}_c^{(i)}$ of the (a-posterior) probability

$$y_c^{(i)} = \mathrm{P}(c^{(i)} = c \mid \mathcal{D}) \tag{8.13}$$

of the i-th datapoint $\mathbf{x}^{(i)}$ belonging to cluster c, given the observed dataset $\mathcal{D}$.

This estimation problem becomes significantly easier by operating in an alternating fashion. In each iteration, we first compute a new estimate $\hat{p}_c$ of the cluster probabilities p_c, given the current estimate $\mathbf{m}^{(c)}, \mathbf{C}^{(c)}$ for the cluster means and covariance matrices. Then, using this new estimate $\hat{p}_c$ for the cluster probabilities, we update the estimates $\mathbf{m}^{(c)}, \mathbf{C}^{(c)}$ of the cluster means and covariance matrices. Then, using the new estimates $\mathbf{m}^{(c)}, \mathbf{C}^{(c)}$, we compute a new estimate $\hat{p}_c$ and so on. By repeating these two steps, we obtain an iterative soft-clustering method which is summarized in Algorithm 7.

Algorithm 7 "A Soft-Clustering Algorithm" [10]

Input: dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$; number k of clusters.

Initialize: use initial guess for GMM parameters $\{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^{k}$

1: **repeat**

2: for each datapoint $\mathbf{x}^{(i)}$ and cluster $c \in \{1, \ldots, k\}$, update degrees of belonging

$$y_c^{(i)} = \frac{\hat{p}_c \mathcal{N}(\mathbf{x}^{(i)}; \mathbf{m}^{(c)}, \mathbf{C}^{(c)})}{\sum_{c'=1}^{k} \hat{p}_{c'} \mathcal{N}(\mathbf{x}^{(i)}; \mathbf{m}^{(c')}, \mathbf{C}^{(c')})} \tag{8.14}$$

3: for each cluster $c \in \{1, \ldots, k\}$, update estimates of GMM parameters:

- cluster probability $\hat{p}_c = m_c/m$, with effective cluster size $m_c = \sum_{i=1}^{m} y_c^{(i)}$
- cluster mean $\mathbf{m}^{(c)} = (1/m_c) \sum_{i=1}^{m} y_c^{(i)} \mathbf{x}^{(i)}$
- cluster covariance matrix $\mathbf{C}^{(c)} = (1/m_c) \sum_{i=1}^{m} y_c^{(i)} \big(\mathbf{x}^{(i)} - \mathbf{m}^{(c)}\big)\big(\mathbf{x}^{(i)} - \mathbf{m}^{(c)}\big)^T$

4: **until** convergence

Output: soft cluster assignments $\mathbf{y}^{(i)} = (y_1^{(i)}, \ldots, y_k^{(i)})^T$ for each datapoint $\mathbf{x}^{(i)}$

As for k-means, we can interpret the soft clustering problem as an instance of the ERM principle discussed in Chapter 4. Indeed, Algorithm 7 aims at minimizing the empirical risk

$$\mathcal{E}\big(\{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^{k} \mid \mathcal{D}\big) = -\log \mathrm{Prob}\big\{\mathcal{D}; \{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^{k}\big\}. \tag{8.15}$$

The interpretation of Algorithm 7 as a method for minimizing the empirical risk (8.15) suggests a stopping criterion. We can monitor the decrease of the empirical risk

$$-\log \mathrm{Prob}\big\{\mathcal{D}; \{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^{k}\big\}$$

to decide when to stop iterating (see step 4 of Algorithm 7).

Similar to k-means Algorithm 5, also the soft clustering Algorithm 7 suffers from the problem of getting stuck in local minima of the empirical risk (8.15). As for k-means, we can avoid local minima by running Algorithm 7 several times, each time with a different initialization for the GMM parameter estimates $\{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^{k}$ and then picking the result which yields the smallest empirical risk (8.15).

The empirical risk (8.15) underlying the soft-clustering Algorithm 7 is essentially a **log-likelihood function**. Thus, Algorithm 7 can be interpreted as an **approximate maximum likelihood** estimator for the true underlying GMM parameters $\{\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)}, p_c\}_{c=1}^{k}$. In particular, Algorithm 7 is an instance of a generic approximate maximum likelihood technique referred to as **expectation maximization** (EM) (see [31, Chap. 8.5] for more details). The interpretation of Algorithm 7 as a special case of EM allows to characterize the behaviour of Algorithm 7 using existing convergence results for EM methods [75].

There is an interesting link between the soft-clustering Algorithm 7 and k-means. The k-means algorithm can be interpreted as an extreme case of soft-clustering Algorithm 7. Consider fixing the cluster covariance matrices $\boldsymbol{\Sigma}^{(c)}$ within the GMM (8.9) to be the scaled identity:

$$\boldsymbol{\Sigma}^{(c)} = \sigma^2 \mathbf{I} \text{ for all } c \in \{1, \ldots, k\}. \tag{8.16}$$

We assume the covariance matrix (8.16), with a particular value for σ^2, to be the actual "correct" covariance matrix for cluster c. Thus, we replace the covariance matrix updates in Algorithm 7 with $\mathbf{C}^{(c)} := \boldsymbol{\Sigma}^{(c)}$.

When choosing a very small variance σ^2 in (8.16), the update (8.14) tends to enforce $y_c^{(i)} \in \{0, 1\}$. Thus, each datapoint $\mathbf{x}^{(i)}$ is associated exclusively to the cluster c whose cluster mean $\mathbf{m}^{(c)}$ has minimum Euclidean distance to the datapoint $\mathbf{x}^{(i)}$. To summarize, for $\sigma^2 \to 0$, the soft-clustering update (8.14) reduces to the hard cluster assignment update (8.1) of the k-means Algorithm 5.

8.3 Density Based Clustering with DBSCAN

Both k-means and GMM cluster datapoints using the Euclidean distance, which is a natural measure of similarity in many cases. However, in some applications, the data conforms to a different non-Euclidean structure. One example for a non-Euclidean structure is a graph or network structure. Here, two datapoints are considered similar if they can be reached by intermediate datapoints that have a small Euclidean distance. Thus, two datapoints can be similar in terms of connectivity, even if their Euclidean distance is large.Density-based spatial clustering of applications with noise (DBSCAN) is a hard clustering method that uses a connectivity-based similarity measure. In contrast to k-means and the GMM, DBSCAN does not require the number of clusters to be pre-defined - the number will depend on its parameters. Moreover, DBSCAN detects outliers that are interpreted as degenerated clusters consisting of a single datapoint. For a detailed discussion of how DBSCAN works, we refer to https://en.wikipedia.org/wiki/DBSCAN. DBSCAN

8.4 Exercises

8.4.1 Image Compression with k-means

use k-means to compress a RGB bitmap image. Instead of RGB values we need to store only cluster index and the cluster means.

8.4.2 Compression with k-means

Consider $m = 10000$ datapoints are characterized by two floating point numbers (32 bit). We apply k-means to cluster the data set into two clusters. How many bits do we need to store the clustering ?

Chapter 9

Feature Learning

"Solving Problems By Changing the Viewpoint."

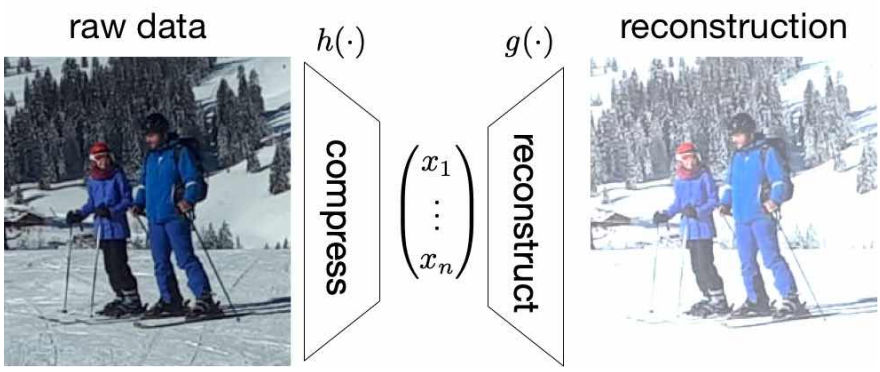


Figure 9.1: Dimensionality reduction methods aim at finding a map h which maximally compresses the raw data while still allowing to accurately reconstruct the original datapoint from a small number of features $x_1, \ldots, x_n$.

Chapter 2 discussed features as those properties of a datapoint that can be measured or computed easily. Sometimes the choice of features follows naturally from the available hard and software. As an example we might use the measurements delivered by sensing devices as features. Beside direct sensor readings, we might construct new features by simple computations.

Consider datapoints representing ski days which are characterized by the minimum daytime temperature as their feature x. Then we might construct new features of a datapoint by computing powers x^2 and x^3. Another feature is obtained by $x + 5$. Each of these computations produces a new feature. Which of these additional features are most useful?

Feature learning methods automate the choice of finding a good features. These methods learn a map that read in the original raw features and transform them to a set of new features. A subclass of feature learning methods are dimensionality reduction methods, where the new feature space has a (much) smaller dimension than the original feature space (see Section 9.1). Sometimes it might be useful to change to a higher-dimensional feature space. Section 9.6 discusses feature learning method that result in new feature vectors which are longer than the raw feature vector

??? Develop feature learning as an approximation problem. The raw data is the vector to be approximated. The approximation has to be in a (small) subspace which is spanned by all possible low-dimensional feature vectors???

9.1 Dimensionality Reduction

Consider a ML method that aims at predicting the label y of a datapoint $\mathbf{z}$ based on some features $\mathbf{x}$ which characterize the datapoint $\mathbf{z}$. Intuitively, it should be beneficial to use as many features as possible. Indeed, the more features of a datapoint we know, the more we should know about its label y.

There are, however, two pitfalls in using an unnecessarily large number of features. The first one is a **computational pitfall** and the second one is a **statistical pitfall**. The larger the feature vector $\mathbf{x} \in \mathbb{R}^n$ (with large n), the more computation (and storage) is required for executing the resulting ML method. Moreover, using a large number of features makes the resulting ML methods more prone to overfitting. Indeed, linear regression will overfit when using feature vectors $\mathbf{x} \in \mathbb{R}^n$ whose length n exceeds the number m of labeled datapoints used for training (see Chapter 7).

Thus, both from a computational and statistical perspective, it is beneficial to use only the maximum necessary amount of relevant features. A key challenge here is to select those features which carry most of the relevant information required for the prediction of the label y. Beside coping with overfitting and limited computational resources, dimensionality reduction can also be useful for data visualization. Indeed, if the resulting feature vector has length $d = 2$, we can use scatter plots to depict datasets.

The basic idea behind most dimensionality reduction methods is quite simple. As illustrated in Figure 9.1, these methods aim at learning (finding) a "compression" map that transforms a raw datapoint $\mathbf{z}$ to a (short) feature vector $\mathbf{x} = (x_1, \ldots, x_n)^T$ in such a way that it is possible to find (learn) a "reconstruction" map which allows to accurately reconstruct the

original datapoint from the features $\mathbf{x}$. The compression and reconstruction map is typically constrained to belong some set of computationally feasible maps or hypothesis space (see Chapter 3 for different examples of hypothesis spaces). In what follows we restrict ourselves to using only linear maps for compression and reconstruction leading to principal component analysis. The extension to non-linear maps using deep neural networks is known as **deep autoencoders** [27, Ch. 14].

9.2 Principal Component Analysis

Consider a datapoint $\mathbf{z} \in \mathbb{R}^D$ which is represented by a (typically very long) vector of length D. The length D of the raw feature vector might be easily of the order of millions. To obtain a small set of relevant features $\mathbf{x} \in \mathbb{R}^n$, we apply a linear transformation to the datapoint:

$$\mathbf{x} = \mathbf{W}\mathbf{z}. \tag{9.1}$$

Here, the "compression" matrix $\mathbf{W} \in \mathbb{R}^{n \times D}$ maps (in a linear fashion) the large vector $\mathbf{z} \in \mathbb{R}^D$ to a smaller feature vector $\mathbf{x} \in \mathbb{R}^n$.

It is reasonable to choose the compression matrix $\mathbf{W} \in \mathbb{R}^{n \times D}$ in (9.1) such that the resulting features $\mathbf{x} \in \mathbb{R}^n$ allow to approximate the original datapoint $\mathbf{z} \in \mathbb{R}^D$ as accurate as possible. We can approximate (or recover) the datapoint $\mathbf{z} \in \mathbb{R}^D$ back from the features $\mathbf{x}$ by applying a reconstruction operator $\mathbf{R} \in \mathbb{R}^{D \times n}$, which is chosen such that

$$\mathbf{z} \approx \mathbf{R}\mathbf{x} \overset{(9.1)}{=} \mathbf{R}\mathbf{W}\mathbf{z}. \tag{9.2}$$

The approximation error $\mathcal{E}\big(\mathbf{W}, \mathbf{R} \mid \mathcal{D}\big)$ resulting when (9.2) is applied to each datapoint in a dataset $\mathcal{D} = \{\mathbf{z}^{(i)}\}_{i=1}^m$ is then

$$\mathcal{E}\big(\mathbf{W}, \mathbf{R} \mid \mathcal{D}\big) = (1/m) \sum_{i=1}^{m} \|\mathbf{z}^{(i)} - \mathbf{R}\mathbf{W}\mathbf{z}^{(i)}\|. \tag{9.3}$$

One can verify that the approximation error $\mathcal{E}\big(\mathbf{W}, \mathbf{R} \mid \mathcal{D}\big)$ can only by minimal if the compression matrix $\mathbf{W}$ is of the form

$$\mathbf{W} = \mathbf{W}_{\rm PCA} := \big(\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}\big)^T \in \mathbb{R}^{n \times D}, \tag{9.4}$$

with n orthonormal vectors $\mathbf{u}^{(l)}$ which correspond to the n largest eigenvalues of the **sample**

covariance matrix

$$\mathbf{Q} := (1/m)\mathbf{Z}^T\mathbf{Z} \in \mathbb{R}^{D\times D} \tag{9.5}$$

with data matrix $\mathbf{Z} = \big(\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(m)}\big)^T \in \mathbb{R}^{m\times D}$. [1] By its very definition (9.5), the matrix $\mathbf{Q}$ is positive semi-definite so that it allows for an eigenvalue decomposition (EVD) of the form [66]

$$\mathbf{Q} = \big(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(D)}\big) \begin{pmatrix} \lambda^{(1)} & \ldots & 0 \\ 0 & \ddots & 0 \\ 0 & \ldots & \lambda^{(D)} \end{pmatrix} \big(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(D)}\big)^T$$

with real-valued eigenvalues $\lambda^{(1)} \geq \lambda^{(2)} \geq \ldots \geq \lambda^{(D)} \geq 0$ and orthonormal eigenvectors $\{\mathbf{u}_r\}_{r=1}^{D}$.

The features $\mathbf{x}^{(i)}$, obtained by applying the compression matrix $\mathbf{W}_{\rm PCA}$ (9.4) to the raw datapoints $\mathbf{z}^{(i)}$, are referred to as **principal components (PC)**. The overall procedure of determining the compression matrix (9.4) and, in turn, computing the PC vectors $\mathbf{x}^{(i)}$ is known as **principal component analysis (PCA)** and summarized in Algorithm 8. Note

Algorithm 8 Principal Component Analysis (PCA)

Input: dataset $\mathcal{D} = \{\mathbf{z}^{(i)} \in \mathbb{R}^D\}_{i=1}^m$; number n of PCs.

1: compute EVD (9.6) to obtain orthonormal eigenvectors $\big(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(D)}\big)$ corresponding to (decreasingly ordered) eigenvalues $\lambda^{(1)} \geq \lambda^{(2)} \geq \ldots \geq \lambda^{(D)} \geq 0$

2: construct compression matrix $\mathbf{W}_{\rm PCA} := \big(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(n)}\big)^T \in \mathbb{R}^{n\times D}$

3: compute feature vector $\mathbf{x}^{(i)} = \mathbf{W}_{\rm PCA}\mathbf{z}^{(i)}$ whose entries are PC of $\mathbf{z}^{(i)}$

4: compute approximation error $\mathcal{E}^{(\rm PCA)} = \sum_{r=n+1}^{D} \lambda^{(r)}$ (see (9.6)).

Output: $\mathbf{x}^{(i)}$, for $i = 1,\ldots,m$, and the approximation error $\mathcal{E}^{(\rm PCA)}$.

that the length n of the feature vectors $\mathbf{x}$, which is also the number of PCs used, is an input parameter of Algorithm 8. The number n can be chosen between $n = 0$ and $n = D$. However, it can be shown that PCA for $n > m$ is not well-defined. In particular, the orthonormal eigenvectors $\mathbf{u}^{(n+1)},\ldots,\mathbf{u}^{(D)}$ are not unique.

From a computational perspective, Algorithm 8 essentially amounts to performing an EVD of the sample covariance matrix $\mathbf{Q}$ (see (9.5)). Indeed, the EVD of $\mathbf{Q}$ provides not only the optimal compression matrix $\mathbf{W}_{\rm PCA}$ but also the measure $\mathcal{E}^{(\rm PCA)}$ for the information loss incurred by replacing the original datapoints $\mathbf{z}^{(i)} \in \mathbb{R}^D$ with the smaller feature vector

[1] Some authors define the data matrix as $\mathbf{Z} = \big(\widetilde{\mathbf{z}}^{(1)},\ldots,\widetilde{\mathbf{z}}^{(m)}\big)^T \in \mathbb{R}^{m\times D}$ using "centered" datapoints $\widetilde{\mathbf{z}}^{(i)} - \widehat{\mathbf{m}}$ obtained by subtracting the average $\widehat{\mathbf{m}} = (1/m)\sum_{i=1}^m \mathbf{z}^{(i)}$.

$\mathbf{x}^{(i)} \in \mathbb{R}^n$. In particular, this information loss is measured by the approximation error (obtained for the optimal reconstruction matrix $\mathbf{R}_{\rm opt} = \mathbf{W}_{\rm PCA}^T$)

$$\mathcal{E}^{(\rm PCA)} := \mathcal{E}\big(\mathbf{W}_{\rm PCA}, \underbrace{\mathbf{R}_{\rm opt}}_{=\mathbf{W}_{\rm PCA}^T} \mid \mathcal{D}\big) = \sum_{r=n+1}^{D} \lambda^{(r)}. \tag{9.6}$$

As depicted in Figure 9.2, the approximation error $\mathcal{E}^{(\rm PCA)}$ decreases with increasing number n of PCs used for the new features (9.1). The maximum error $\mathcal{E}^{(\rm PCA)} = (1/m)\sum_{i=1}^{m} \|\mathbf{z}^{(i)}\|^2$ is obtained for $n = 0$, which amounts to completely ignoring the datapoints $\mathbf{z}^{(i)}$. In the other extreme case where $n = D$ and $\mathbf{x}^{(i)} = \mathbf{z}^{(i)}$, which amounts to no compression at all, the approximation error is zero $\mathcal{E}^{(\rm PCA)} = 0$.

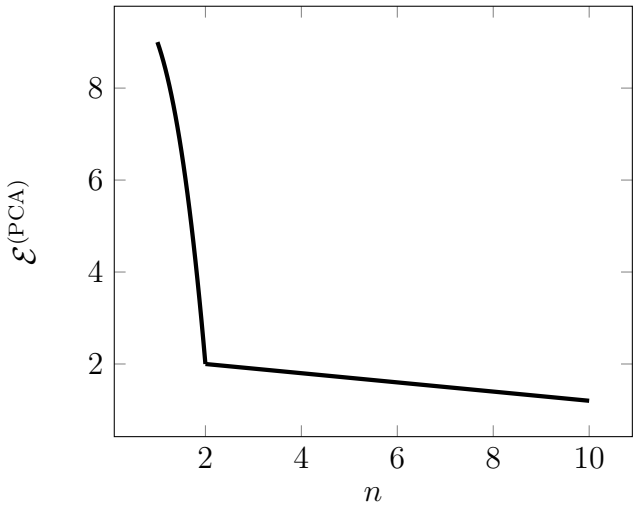


Figure 9.2: Reconstruction error $\mathcal{E}^{(\rm PCA)}$ (see (9.6)) of PCA for varying number n of PCs.

9.2.1 Combining PCA with Linear Regression

One important use case of PCA is as a pre-processing step within an overall ML problem such as linear regression (see Section 3.1). As discussed in Chapter 7, linear regression methods are prone to overfitting whenever the datapoints are characterized by feature vectors whose length D exceeds the number m of labeled datapoints used for training. One simple but powerful strategy to avoid overfitting is to preprocess the original feature vectors (they are considered as the raw datapoints $\mathbf{z}^{(i)} \in \mathbb{R}^D$) by applying PCA in order to obtain smaller feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ with $n < m$.

9.2.2 How To Choose Number of PC?

There are several aspects which can guide the choice for the number n of PCs to be used as features.

- for data visualization: use either $n = 2$ or $n = 3$
- computational budget: choose n sufficiently small such that the computational complexity of the overall ML method does not exceed the available computational resources.
- statistical budget: consider using PCA as a pre-processing step within a linear regression problem (see Section 3.1). Thus, we use the output $\mathbf{x}^{(i)}$ of PCA as the feature vectors in linear regression. In order to avoid overfitting, we should choose $n < m$ (see Chapter 7).
- elbow method: choose n large enough such that approximation error $\mathcal{E}^{(\mathrm{PCA})}$ is reasonably small (see Figure 9.2).

9.2.3 Data Visualisation

If we use PCA with $n = 2$ PC, we obtain feature vectors $\mathbf{x}^{(i)} = \mathbf{W}\mathbf{z}^{(i)}$ (see (9.1)) which can be depicted as points in a scatter plot (see Section 2.1.3). As an example, consider datapoints $\mathbf{z}^{(i)}$ obtained from historic recordings of Bitcoin statistics. Each datapoint $\mathbf{z}^{(i)} \in \mathbb{R}^6$ is a vector of length $D = 6$. It is difficult to visualise points in an Euclidean space $\mathbb{R}^D$ of dimension $D > 2$. We apply PCA with $n = 2$ which results in feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^2$. Such feature vectors can be depicted conveniently as a scatter plot (see Figure 9.3).

9.2.4 Extensions of PCA

There have been proposed several extensions of the basic PCA method:

- **kernel PCA [31, Ch.14.5.4]:** combines PCA with a non-linear feature map (see Section 3.9).
- **robust PCA [74]:** modifies PCA to better cope with **outliers** in the dataset.
- **sparse PCA [31, Ch.14.5.5]:** requires each PC to depend only on a small number of data attributes z_j.

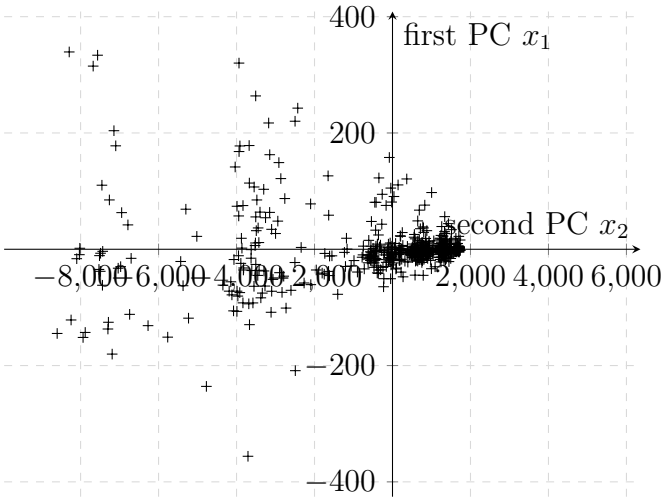


Figure 9.3: A scatter plot of feature vectors $\mathbf{x}^{(i)} = \big(x_1^{(i)}, x_2^{(i)}\big)^T$ whose entries are the first two PCs of the Bitcoin statistics $\mathbf{z}^{(i)}$ of the i-th day.

- **probabilistic PCA [60, 69]:** generalizes PCA by using a **probabilistic (generative) model** for the data.

9.3 Linear Discriminant Analysis

Dimensionality reduction is typically used as a preprocessing step within some overall ML problem such as regression or classification. It can then be useful to exploit the availability of labeled data for the design of the compression matrix $\mathbf{W}$ in (9.1). However, plain PCA (see Algorithm 8) does not make use of any label information provided additionally for the raw datapoints $\mathbf{z}^{(i)} \in \mathbb{R}^D$. Therefore, the compression matrix $\mathbf{W}_{\rm PCA}$ delivered by PCA can be highly suboptimal as a pre-processing step for labeled datapoints. A principled approach for choosing the compression matrix $\mathbf{W}$ such that datapoints with different labels are well separated is **linear discriminant analysis** [31].

9.4 Random Projections

Note that PCA amounts to computing an EVD of the sample covariance matrix $\mathbf{Q} = (1/m)\mathbf{Z}\mathbf{Z}^T$ with the data matrix $\mathbf{Z} = \big(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\big)^T$ containing the datapoints $\mathbf{z}^{(i)} \in \mathbb{R}^D$ as its columns. The computational complexity (amount of multiplications and additions)

for computing this PCA is lower bounded by $\min\{D^2, m^2\}$ [21, 62]. This computational complexity can be prohibitive for ML applications with n and m being of the order of millions (which is already the case if the features are pixel values of a 512×512 RGB bitmap, see Section 2.1.1). There is a surprisingly cheap alternative to PCA for finding a good choice for the compression matrix $\mathbf{W}$ in (9.1). Indeed, a randomly chosen matrix $\mathbf{W}$ with entries drawn i.i.d. from a suitable probability distribution (such as Bernoulli or Gaussian) yields a good compression matrix $\mathbf{W}$ (see (9.1)) with high probability [9, 39].

9.5 Information Bottleneck

We can use information bottleneck for feature learning. Using Gaussian process model, we even get closed-form solutions of Gaussian Information Bottleneck.

9.6 Dimensionality Increase

? Discuss kernel methods; and polynomial regression as special case???

Feature learning methods are mainly dimensionality reduction methods. However, it might be beneficial to also consider feature learning methods that produce new feature vectors which are longer than the raw feature vectors. An extreme example for such a feature map are kernel methods which map finite length vector to infinite dimensional spaces.

Mapping raw feature vectors into higher-dimensional spaces might be useful if the intrinsic geometry of the datapoints is simpler when looked at in the higher-dimensional space. Consider a binary classification problem where datapoints are highly inter-winded in the original feature space. By mapping into higher-dimensional feature space we might "even-out" this non-linear geometry such that we can use linear classifiers in the higher-dimensional space.

Chapter 10

Privacy-Preserving ML

Many ML applications involve datapoints representing individual humans. These datapoints might include sensitive data, such as medical records, which is subject to privacy protection. This chapter discusses some techniques for preprocessing the raw data to protect privacy of individuals while still allowing to solve the overall ML task. We will illustrate these techniques using a stylized healthcare application.

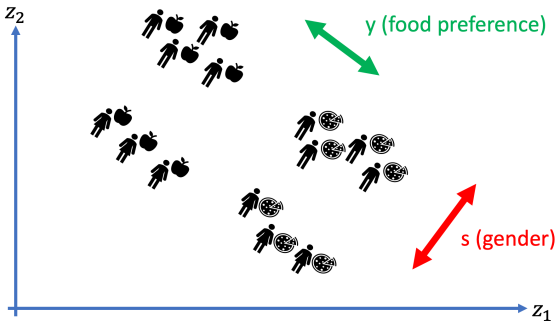


Figure 10.1: datapoints represent humans. We are interested in the fruit preference of humans. Their gender is considered sensitive information and should not be revealed to ML methods.

A key challenge for health-care are pandemics. To optimally manage pandemics it is important to have accurate information about the dynamics. We can model this as a ML problem with datapoints representing humans. One key feature of datapoints is if it represents an infected human or not. This data is sensitive and typically only available to public health-care institutes.

Consider the patient database of a hospital which should provide information about the average number of infected patients. Instead of directly forward the patient files, the hospital

must only forward the fraction of infected patients. This is an example of privacy-preserving data processing. For a sufficiently large number of patients at the hospital (say, more than 1000), we cannot infer much about individual patients just form the fraction of infected patients treated in that hospital.

10.1 Privacy-Preserving Feature Learning (Operating on level of individual datapoints)

Privacy-preserving ML can be implemented using modification of feature learning methods discussed in Chapter 9. Generic feature learning methods aim at learning a compressed representation of the raw datapoints which contain as much information as possible about the quantity of interest. In contrast, privacy-preserving ML does not aim at compression but rather obscuring the raw data such that it does not reveal sensible information about datapoints.

10.1.1 Privacy-Preserving Information Bottleneck

10.1.2 Privacy-Preserving Feature Selection

?? ignore features which are sensitive (name, social ID) but not very relevant for actual task (e.g. predicting income). ???

10.1.3 Privacy-Preserving Random Projections

?? cheap form: random projections/compressed sensing. random projections blur features of individual datapoints but still allow to learn a sparse linear model using e.g. Lasso ???

10.2 Exercises

10.2.1 Where are you?

Consider a ML method that uses FMI data for temperature forecasts. The ML methods downloads the following sequence of daily temperatures: ??,???,???,??. What is the most likely nearest observation station to the ML user ?

10.3 Federated Learning (Operates on level of local datasets)

FL method only exchange model parameter updates; no raw local data is revealed;

Chapter 11

Explainable ML

A key challenge for the successful deployment of ML methods to many (critical) application domain is their explainability. Human users of ML seem to have a strong desire to get explanations that resolve the uncertainty about predictions and decisions obtained from ML methods. Explainable ML enables the user to better predict the outcomes of ML methods.

Explainable ML is challenging since explanations must be tailored (personalized) to individual users with varying backgrounds. Some users might have received university-level education in ML, while other users might have no formal training in linear algebra. Linear regression with few features might be perfectly interpretable for the first group but might be considered a black-box by the latter.

?????? discuss relation between finding good explanations and active learning. Active learning aims at finding datapoints (by their features) which provide most information about the true model parameters. XML aims at finding explanations (e.g. datapoints from training set) which provide most information about the prediction provided by some black-box ML method. ????????????? discuss relation between XML and feature learning. XML can be obtained from feature learning methods by learning those subset of features which provide most information about the prediction (not about the label itself) ?????????????

11.1 A Model Agnostic Method

We propose a simple probabilistic model for the predictions and user knowledge. This model allows to study explainable ML using information theory. Explaining is here considered as the task of reducing the "surprise" incurred by a prediction. We quantify the effect of an explanation by the conditional mutual information between the explanation and prediction,

given the user background.

11.2 Explainable Empirical Risk Minimization

The approach discussed in Section 11.1 constructs explanations for any given ML method such that the user is able to better predict the outcome of this ML method. Instead of providing an explanation we could also try to make the ML method itself more predictable for a user.

Chapter 12

Lists of Symbols

12.1 Sets

$\mathbb{R}$	The set of real numbers x.
$\mathbb{R}_+$	The set of non-negative real numbers $x \geq 0$.

12.2 Matrices and Vectors

$\mathbf{I}$	The identity matrix having ones on the main diagonal and zeros off diagonal.
$\mathbf{R}^n$	The set of all vectors constituted by n real-valued entries.
$\mathbf{x} = \big(x_1, \ldots, x_n\big)^T$	Some vector of length n. The jth entry of the vector is denoted x_j.
$\|\mathbf{x}\|_2$	The Euclidean norm of the vector $\mathbf{x}$, $\|\mathbf{x}\|_2 := \sqrt{\sum_{j=1}^{n} x_j^2}$

12.3 Machine Learning

t	A discrete time index.
i	Generic index used to enumerate datapoints in a list of datapoints.
m	The number of different datapoints in the training set.
$h(\cdot)$	A predictor that maps a feature vector $\mathbf{x}$ of a datapoint to a predicted label $\hat{y} = h(\mathbf{x})$.
y	The label of some datapoint.
$\big(\mathbf{x}^{(i)}, y^{(i)}\big)$	The i-th datapoint within an indexed set of datapoints.
$y^{(i)}$	The label of the ith datapoint.
$\mathbf{x}$	Feature vector whose entries are the features of some datapoint.
$\mathbf{x}^{(i)}$	Feature vector whose entries are the features of the ith datapoint.
n	The number of (real-valued) features of a single datapoint.
x_j	The jth entry of a vector $\mathbf{x} = \big(x_1, \ldots, x_n\big)^T$.

Chapter 13

Glossary

- **a sample**: a sequence of datapoints $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(i)}$ which are interpreted as the realizations of i i.i.d. random variables with the same probability distribution $p(\mathbf{z})$. The length m of the list is also known as the **sample size**.
- **classification problem**: A ML problem involving a discrete label space $\mathcal{Y}$ such as $\mathcal{Y} = \{-1, 1\}$ for binary classification, or $\mathcal{Y} = \{1, 2, \ldots, K\}$ with $K > 2$ for multi-class classification.
- **classifier**. a hypothesis map $h : \mathcal{X} \to \mathcal{Y}$ with discrete label space (e.g., $\mathcal{Y} = \{-1, 1\}$).
- **condition number**. $\kappa(\mathbf{Q})$ of a matrix $\mathbf{Q}$: the ratio of largest to smallest eigenvalue of a psd matrix $\mathbf{Q}$.
- **datapoint**: an elementary unit of information such as a single pixel, a single image, a particular audio recording, a letter, a text document or an entire social network user profile.
- **labeled datapoint**. a datapoint for which we know the value of its label.
- **dataset**: a collection (set or list) of datapoints.
- **eigenvalue/eigenvector**: for a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ we call a non-zero vector $\mathbf{x} \in \mathbb{R}^n$ an eigenvector of $\mathbf{A}$ if $\mathbf{A}\mathbf{x} = \lambda \mathbf{x}$ with some $\lambda \in \mathbb{R}$, which we call an eigenvalue of $\mathbf{A}$.
- **features**: any measurements (or quantities) used to characterize a datapoint (e.g., the maximum amplitude of a sound recoding or the greenness of an RGB image). In

principle, we can use as a feature any quantity which can be measured or computed easily in an automated fashion.

- **hypothesis map**: a map (or function) $h : \mathcal{X} \to \mathcal{Y}$ from the feature space $\mathcal{X}$ to the label space $\mathcal{Y}$. Given a datapoint with features $\mathbf{x}$ we use a hypothesis map to estimate (or approximate) the label y using the predicted label $\hat{y} = h(\mathbf{x})$. ML is about automating the search for a good hypothesis map such that the error $y - h(\mathbf{x})$ is small.
- **hypothesis space**: a set of computationally feasible (predictor) maps $h : \mathcal{X} \to \mathcal{Y}$.
- **i.i.d.**: independent and identically distributed; e.g., "x, y, z are i.i.d. random variables" means that the joint probability distribution $p(x, y, z)$ of the random variables x, y, z factors into the product $p(x)p(y)p(z)$ of the marginal probability distributions of the variables x, y, z which are identical.
- **label**: some property of a datapoint which is of interest, such as the fact if a webcam snapshot shows a forest fire or not. In contrast to features, labels are properties of datapoints that cannot be measured or computed easily in an automated fashion. Instead, acquiring accurate label information often involves human expert labour. Many ML methods aim at learning accurate predictor maps that allow to guess or approximate the label of a datapoint based on its features.
- **loss function**: a function which associates a given datapoint $(\mathbf{x}, y)$ with features $\mathbf{x}$ and label y and hypothesis map h a number that quantifies the prediction error $y - h(\mathbf{x})$.
- **positive semi-definite**: a positive semi-definite matrix $\mathbf{Q}$ is a symmetric matrix $\mathbf{Q} = \mathbf{Q}^T$ such that $\mathbf{x}^T \mathbf{Q} \mathbf{x} \geq 0$ for every vector $\mathbf{x}$.
- **predictor**: a hypothesis map $h : \mathcal{X} \to \mathcal{Y}$ with continuous label space (e.g., $\mathcal{Y} = \mathbb{R}$).
- **psd**: positive semi-definite.
- **regression problem**: an ML problem involving a continuous label space $\mathcal{Y}$ (such as $\mathcal{Y} = \mathbb{R}$).
- **training data**: a dataset which is used for finding a good hypothesis map $h \in \mathcal{H}$ out of a hypothesis space $\mathcal{H}$, e.g., via empirical risk minimization (see Chapter 4).
- **validation data**: a dataset which is used for evaluating the quality of a predictor which has been learnt using some other (training) data.

Bibliography

[1] A. E. Alaoui, X. Cheng, A. Ramdas, M. J. Wainwright, and M. I. Jordan. Asymptotic behavior of ℓ_p-based Laplacian regularization in semi-supervised learning. In *Conf. on Learn. Th.*, pages 879–906, June 2016.

[2] H. Ambos, N. Tran, and A. Jung. Classifying big data over networks via the logistic network lasso. In *Proc. 52nd Asilomar Conf. Signals, Systems, Computers*, Oct./Nov. 2018.

[3] H. Ambos, N. Tran, and A. Jung. The logistic network lasso. *arXiv*, 2018.

[4] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5 – 43, 2003.

[5] P. Austin, P. Kaski, and K. Kubjas. Tensor network complexity of multilinear maps. *arXiv*, 2018.

[6] M. Belkin, I. Matveeva, and P. Niyogi. Regularization and semi-supervised learning on large graphs. In *COLT*, volume 3120, pages 624–638. Springer, 2004.

[7] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 2nd edition, June 1999.

[8] P. Billingsley. *Probability and Measure*. Wiley, New York, 3 edition, 1995.

[9] E. Bingham and H. Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Knowledge Discovery and Data Mining*, pages 245–250. ACM Press, 2001.

[10] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[11] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers*, volume 3. Now Publishers, Hanover, MA, 2010.

[12] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge Univ. Press, Cambridge, UK, 2004.

[13] S. Bubeck. Convex optimization. algorithms and complexity. In *Foundations and Trends in Machine Learning*, volume 8. Now Publishers, 2015.

[14] P. Bühlmann and S. van de Geer. *Statistics for High-Dimensional Data*. Springer, New York, 2011.

[15] S. Carrazza. Machine learning challenges in theoretical HEP. *arXiv*, 2018.

[16] N. Cesa-Bianchi and G. Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, New York, NY, USA, 2006.

[17] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. The MIT Press, Cambridge, Massachusetts, 2006.

[18] S. Chen, A. Sandryhaila, J. M. F. Moura, and J. Kovačević. Signal recovery on graphs: Variation minimization. *IEEE Trans. Signal Processing*, 63(17):4609–4624, Sept. 2015.

[19] S. Chen, R. Varma, A. Sandryhaila, and J. Kovačević. Discrete signal processing on graphs: Sampling theory. *IEEE Trans. Signal Processing*, 63(24):6510–6523, Dec. 2015.

[20] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*, (4):303–314, 1989.

[21] Q. Du and J. Fowler. Low-complexity principal component analysis for hyperspectral image compression. *Int. J. High Performance Comput. Appl*, pages 438–448, 2008.

[22] R. Eldan and O. Shamir. The power of depth for feedforward neural networks. *CoRR*, abs/1512.03965, 2015.

[23] R. Fergus, Y. Weiss, and A. Torralba. Semi-supervised learning in gigantic image collections. In *Proceedings of the 22Nd International Conference on Neural Information Processing Systems*, NIPS'09, pages 522–530, USA, 2009. Curran Associates Inc.

[24] M. Gao, H. Igata, A. Takeuchi, K. Sato, and Y. Ikegaya. Machine learning-based prediction of adverse drug effects: An example of seizure-inducing compounds. *Journal of Pharmacological Sciences*, 133(2):70 – 78, 2017.

[25] W. Gautschi and G. Inglese. Lower bounds for the condition number of vandermonde matrices. *Numer. Math.*, 52:241 – 250, 1988.

[26] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[27] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[28] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proc. Neural Inf. Proc. Syst. (NIPS)*, 2014.

[29] R. Gray, J. Kieffer, and Y. Linde. Locally optimal block quantizer design. *Information and Control*, 45:178 – 198, 1980.

[30] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, March/April 2009.

[31] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, New York, NY, USA, 2001.

[32] E. Hazan. *Introduction to Online Convex Optimization*. Now Publishers Inc., 2016.

[33] P. J. Huber. *Robust Statistics*. Wiley, New York, 1981.

[34] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer, 2013.

[35] A. Jung. A fixed-point of view on gradient methods for big data. *Frontiers in Applied Mathematics and Statistics*, 3, 2017.

[36] A. Jung, A. O. Hero, A. Mara, S. Jahromi, A. Heimowitz, and Y. Eldar. Semi-supervised learning in network-structured data via total variation minimization. *IEEE Trans. Signal Processing*, 67(24), Dec. 2019.

[37] A. Jung and M. Hulsebos. The network nullspace property for compressed sensing of big data over networks. *Front. Appl. Math. Stat.*, Apr. 2018.

[38] A. Jung, N. Quang, and A. Mara. When is Network Lasso Accurate? *Front. Appl. Math. Stat.*, 3, Jan. 2018.

[39] A. Jung, G. Tauböck, and F. Hlawatsch. Compressive spectral estimation for nonstationary random processes. *IEEE Trans. Inf. Theory*, 59(5):3117–3138, May 2013.

[40] S. M. Kay. *Fundamentals of Statistical Signal Processing: Estimation Theory*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[41] P. Koehn. Europarl: A parallel corpus for statistical machine translation. In *The 10th Machine Translation Summit, page 79–86., AAMT,*, Phuket, Thailand, 2005.

[42] D. Koller, N., and Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, 2009.

[43] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems, NIPS*, 2012.

[44] C. Lampert. Kernel methods in computer vision. *Foundations and Trends in Computer Graphics and Vision*, 2009.

[45] J. Larsen and C. Goutte. On optimal data split for generalization estimation and model selection. In *IEEE Workshop on Neural Networks for Signal Process*, 1999.

[46] S. L. Lauritzen. *Graphical Models*. Clarendon Press, Oxford, UK, 1996.

[47] E. L. Lehmann and G. Casella. *Theory of Point Estimation*. Springer, New York, 2nd edition, 1998.

[48] V. Lempitsky, P. Kohli, C. Rother, and T. Sharp. Image segmentation with a bounding box prior. In *2009 IEEE 12th International Conference on Computer Vision*, pages 277–284, Sept 2009.

[49] K. V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Academic Press, 1979.

[50] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations invector space. In *ICLR (Workshop Poster)*, 2013.

[51] T. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR 5-110,, Rutgers University, New Brunswick, New Jersey, USA, 1980.

[52] K. Mortensen and T. Hughes. Comparing amazon's mechanical turk platform to conventional data collection methods in the health and medical research literature. *J. Gen. Intern Med.*, 33(4):533–538, 2018.

[53] N. Murata. A statistical study on on-line learning. In D. Saad, editor, *On-line Learning in Neural Networks*, pages 63–92. Cambridge University Press, New York, NY, USA, 1998.

[54] B. Nadler, N. Srebro, and X. Zhou. Statistical analysis of semi-supervised learning: The limit of infinite unlabelled data. In *Advances in Neural Information Processing Systems 22*, pages 1330–1338. 2009.

[55] Y. Nesterov. *Introductory lectures on convex optimization*, volume 87 of *Applied Optimization*. Kluwer Academic Publishers, Boston, MA, 2004. A basic course.

[56] M. E. J. Newman. *Networks: An Introduction*. Oxford Univ. Press, 2010.

[57] A. Y. Ng and M. I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 841–848. MIT Press, 2002.

[58] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):123–231, 2013.

[59] H. Poor. *An Introduction to Signal Detection and Estimation*. Springer, 2 edition, 1994.

[60] S. Roweis. EM Algorithms for PCA and SPCA. In *Advances in Neural Information Processing Systems*, pages 626–632. MIT Press, 1998.

[61] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, New York, 3 edition, 1976.

[62] A. Sharma and K. Paliwal. Fast principal component analysis using fixed-point analysis. *Pattern Recognition Letters*, 28:1151 – 1155, 2007.

[63] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, Aug. 2000.

[64] S. Smoliski and K. Radtke. Spatial prediction of demersal fish diversity in the baltic sea: comparison of machine learning and regression-based techniques. *ICES Journal of Marine Science*, 74(1):102–111, 2017.

[65] S. Sra, S. Nowozin, and S. J. Wright, editors. *Optimization for Machine Learning*. MIT Press, 2012.

[66] G. Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, MA, 2007.

[67] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, MA, 5 edition, 2016.

[68] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. draft in progress, available online at http://www.incompleteideas.net/book/bookdraft2017nov5.pdf, 2017.

[69] M. E. Tipping and C. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 21/3:611–622, January 1999.

[70] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1999.

[71] O. Vasicek. A test for normality based on sample entropy. *Journal of the Royal Statistical Society. Series B (Methodological)*, 38(1):54–59, 1976.

[72] M. Wainwright. *High-Dimensional Statistics: A Non-Asymptotic Viewpoint*. Cambridge: Cambridge University Press, 2019.

[73] A. Wang. An industrial-strength audio search algorithm. In *International Symposium on Music Information Retrieval*, Baltimore, MD, 2003.

[74] J. Wright, Y. Peng, Y. Ma, A. Ganesh, and S. Rao. Robust principal component analysis: Exact recovery of corrupted low-rank matrices by convex optimization. In *Neural Information Processing Systems, NIPS 2009*, 2009.

[75] L. Xu and M. Jordan. On convergence properties of the EM algorithm for Gaussian mixtures. *Neural Computation*, 8(1):129–151, 1996.

[76] Y. Yamaguchi and K. Hayashi. When does label propagation fail? a view from a network generative model. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3224–3230, 2017.

[77] K. Young. Bayesian diagnostics for checking assumptions of normality. *Journal of Statistical Computation and Simulation*, 47(3–4):167 – 180, 1993.

[78] W. W. Zachary. An information flow model for conflict and fission in small groups. *J. Anthro. Res.*, 33(4), 1977.