Machine Learning: The Basics

!! ROUGH DRAFT !!

Alexander Jung, August 27, 2021

To cite this version:

> A. Jung,"Machine Learning: The Basics," under preparation, 2021. available online at https://alexjungaalto.github.io/MLBasicsBook.pdf.

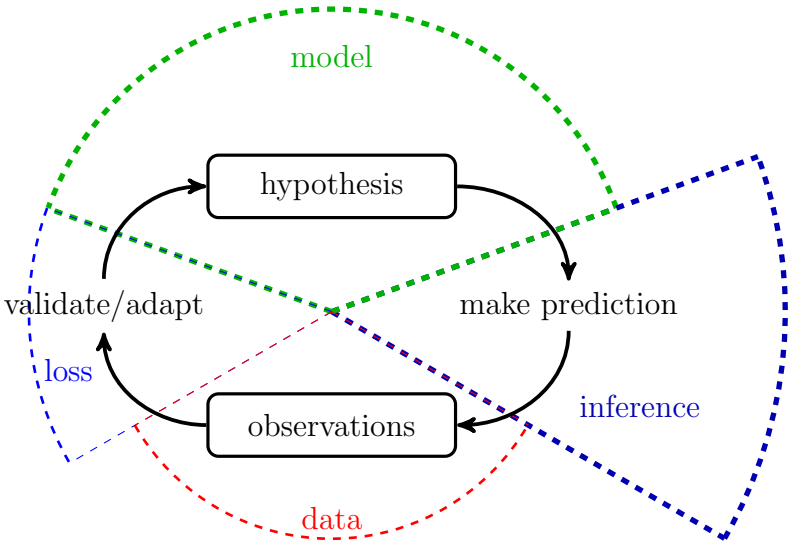


Figure 1: Machine learning combines three main components: data, model and loss. Machine learning methods implement the scientific principle of "trial and error". These methods continuously validate and refine a model based on the loss incurred by its predictions about a phenomenon that generates data.

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 the model or hypothesis space underlying 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.

We believe that thinking about ML as combinations of three components given by data, model and loss helps to navigate the steadily growing offer for ready-to-use ML methods [68]. 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. The regularization effect of early stopping in iterative methods is due to the shrinking of the effective hypothesis

space. Privacy-preserving ML is obtained by particular choices for the features of data points. Explainable ML methods are characterized by particular choices for the hypothesis space.

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 offers a higher-level view on the implementation of ML methods which is typically required to manage a team of ML engineers and data scientists.

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 one 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.

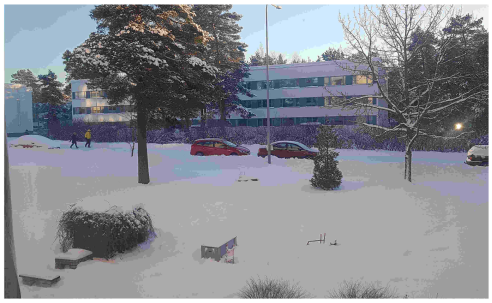


Figure 1.1: Looking outside the window during the morning of a winter day in Finland.

Let us show how ML can be used to learn a predictor for the maximum daytime temperature for the specific day depicted in Figure 1.1. The prediction shall be based solely on the minimum daytime temperature observed in the morning of that day. ML methods can learn a predictor in a data-driven fashion using historic weather observations provided by the **Finnish Meteorological Institute (FMI)**. Let us download the recordings of minimum and maximum daytime temperature for the most recent days and 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. We depict the data (1.1) in Figure 1.2. Each dot in Figure 1.2 represents a specific day with minimum daytime temperature x and maximum daytime temperature y.

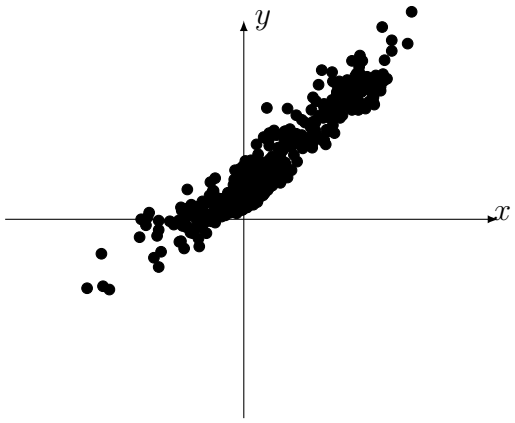


Figure 1.2: Dots represent days according to their minimum daytime temperature x and maximum daytime temperature y measured at some Finnish weather station.

ML methods learn a hypothesis $h(x)$, that reads in the minimum temperature x and delivers a prediction (forecast or approximation) $\hat{y} = h(x)$ for the maximum daytime temperature y. Different ML methods use different hypothesis spaces from which the hypothesis h is chosen.

Let us assume that the minimum and maximum daytime temperature of an arbitrary day are approximately related via

$$y \approx w_1 x + w_0 \text{ with some weights } w_1 \in \mathbb{R}_+, w_0 \in \mathbb{R}. \tag{1.2}$$

The assumption (1.2) reflects the intuition that the maximum daytime temperature y should be higher for days with a higher minimum daytime temperature x. The assumption (1.2) contains two weights (or parameters) w_1 and w_0. These weights are tuning parameters that allow for some flexibility in our assumption. We require the weight w_1 to be non-negative but otherwise leave these weights unspecified for the time being. The main subject of this

book is are ML methods that can be used to learn good values for the weights w_1 and w_0 in a data-driven fashion.

Before we detail how ML can be used to find or learn good values for the weights w_0 in w_1 in (1.2) let us interpret them. The weight w_1 in (1.2) can be interpreted as the relative increase in the maximum daytime temperature for an increased minimum daytime temperature. Consider an earlier day with recorded maximum daytime temperature of 10 degrees and minimum daytime temperature of 0 degrees. The assumption (1.2) then means that the maximum daytime temperature for another other day with minimum daytime temperature of +1 degrees would be $10+w_1$ degrees. The second weight w_0 in our assumption (1.2) can be interpreted as the maximum daytime temperature that we anticipate for a day with minimum daytime temperature equal to 0.

Given the assumption (1.2), it seems reasonable to restrict the ML method to only consider linear maps[1]

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

Since we require $w_1 \geq 0$, the map (2.5) is monotonically increasing with respect to the argument x. Therefore, the prediction $h(x)$ for the maximum daytime temperature becomes higher with higher minimum daytime temperature x.

The expression (2.5) 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 (2.5) 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 $\mathbf{w}$.

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 (2.5) for different weights is uncountably infinite.

To find, or **learn**, a good hypothesis out of the infinite set (2.5), we need to somehow assess the quality of a particular hypothesis map. ML methods use data and a loss function

[1]We describe our notation in Chapter 10.

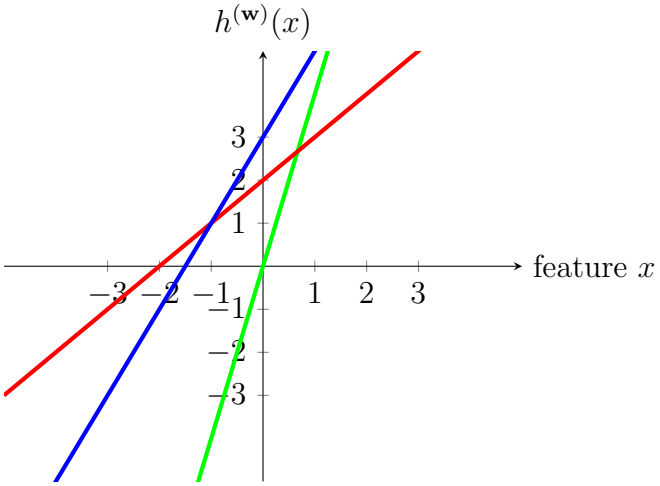


Figure 1.3: Three hypothesis maps of the form (2.5).

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 (2.5) by tuning w_1, w_0 to minimize the average loss

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

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 [57]. 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.

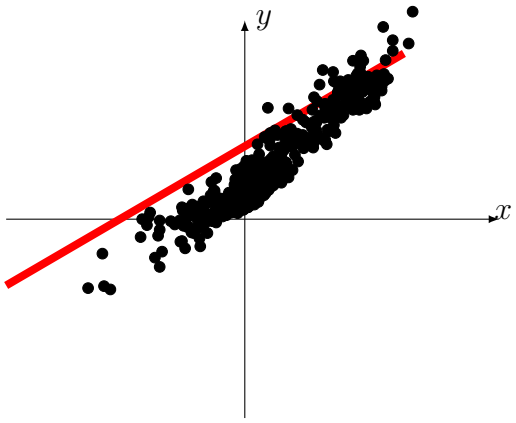


Figure 1.4: Dots represent days characterized by its minimum daytime temperature x and its maximum daytime temperature y. We also depict a straight line representing a linear predictor map. ML methods learn a predictor map with minimum discrepancy between predictor map and datapoints.

1.1 Relation to Other Fields

ML builds on concepts from several other scientific fields. Conversely, ML provides important tools for many 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 us 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 [80]. 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 [78]. 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 [80, 79].

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 us 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 might loose data since we cannot store it for future use. This lost data might have carried relevant information for the ML task at hand.

1.1.5 Probability Theory and 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 by adding a contour line that indicates the probability distribution $p(\mathbf{z})$ [8].

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 to 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** [34].

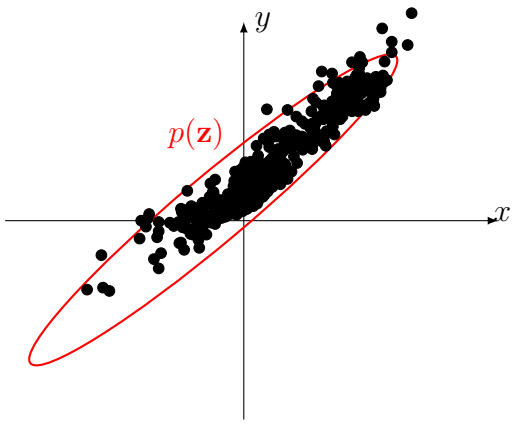


Figure 1.5: A scatterplot where each dot represents some day that is characterized by its minimum daytime temperature x and its maximum daytime temperature y.

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`).

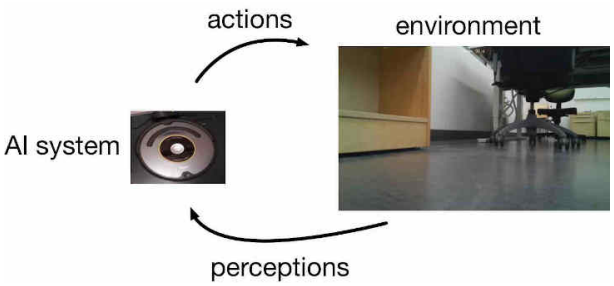


Figure 1.6: A cleaning robot chooses actions (moving directions) to maximize a long-term reward measured by the amount of cleaned floor area per day.

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 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 a consequent choice of actions that are predicted as optimal according to a 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 (or 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 component of any form of ML are collections of data points which can carry information about some phenomenon. An individual data point is characterized by various properties. We find it convenient to divide the properties of data points into two groups: features and labels (see Chapter 2.1). Features are properties that we measure or compute easily in an automated fashion. Labels are properties that cannot be measured easily and often represent some higher-level fact whose discovery often requires human experts.

ML aims at predicting (approximating or guessing) the label of a datapoint based solely on the features of this datapoint. Formally, the prediction is obtained as the function value of a hypothesis function (or map) which takes the features of a data point as its argument. Since any ML method must be implemented with finite computational resources, it can only consider a subset of all possible hypothesis maps. This subset is referred to as the hypothesis space or model underlying a ML method. Based on how ML methods assess the quality of different hypothesis maps we distinguish three main flavours of ML: supervised, unsupervised and reinforcement learning.

Supervised Learning. The main focus of this book is on **supervised** ML methods. These methods use a training set that consists of labeled data points. We refer to a data point as labeled if its label value is known. Such labeled data points can be obtained by some human annotating data points with their label values. There are even marketplaces for renting human labelling workforce [77]. Supervised ML searches for a hypothesis that can imitate the human annotator and allows to predict the label solely from the features of a data point.

Figure 1.7 illustrates the basic principle of supervised methods. These methods learn a hypothesis with minimum discrepancy between its predictions and the true labels of the datapoint in the training set. Loosely speaking, supervised ML fits a curve (the graph of the predictor map) to labeled datapoints in a training set. For the actual implementing of this curve fitting we need a loss function that quantifies the fitting error. Supervised ML method differ in their choice for a loss function to measure the discrepancy between predicted label and true label of a data point.

While the principle behind supervised ML sounds trivial, the challenge of modern ML applications is the sheer amount of datapoints and their complexity. 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 size and complexity

of datasets, another challenge for 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 [33].

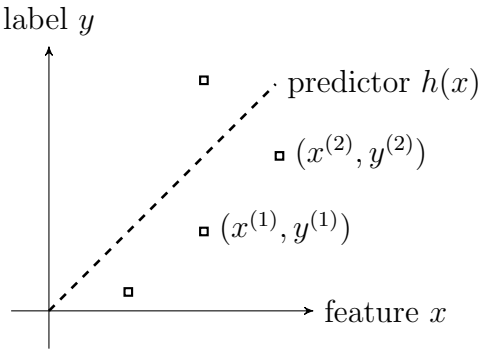


Figure 1.7: Supervised ML methods fit a curve to (a huge number of) datapoints.

Unsupervised Learning. Some ML methods do not require any labeled datapoint and are referred to as **unsupervised**. Loosely speaking, these methods only use the intrinsic structure of data points to learn a good hypothesis. They do not need a teacher or human expert which provides labels for the datapoints in a training set. Chapters 8 and 9 discuss two large families of unsupervised methods, clustering and feature learning methods.

Clustering methods group data points into few subsets such that data points within the same subset or cluster are more similar with each other than with data points outside the cluster. Feature learning methods determine numeric features such that data points can be processed efficiently using these features. Two important applications of feature learning are dimensionality reduction and data visualization.

Reinforcement Learning. ML methods use a loss function to evaluate and compare different hypotheses. The loss function assigns a (typically non-negative) loss value to a given pair of a data point and a hypothesis. Loosely speaking, ML is the search for a hypothesis that incurs minimum loss for any datapoint. Reinforcement learning (RL) studies applications where data point are generated sequentially. We could think of data points forming a time series, each time instant a new data point comes in. Based on the most recent data point coming in, RL methods determine a new candidate for an optimal hypothesis.

What sets RL methods apart from supervised and unsupervised methods is that they must learn a hypothesis in an online fashion. In each time instant, providing a new data point, RL methods must deliver a guess or estimate for the optimal hypothesis and apply it

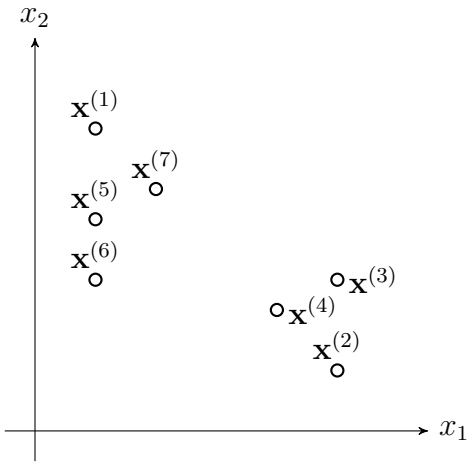


Figure 1.8: Clustering methods aim at learning to predict cluster or group memberships of data points based on their features. These methods are unsupervised as they do not require the knowledge of cluster membership of any data point.

to predict the label of the new data point. RL methods can only evaluate the loss value for the actually chosen hypothesis in each time instant. This loss value might be obtained by some sensing device [81].

One important application domain for RL methods is autonomous driving (see Figure 1.9). An self-driving car generates data points representing individual time instants during the car ride. The features of a data point are the pixel intensities of the on-board camera snapshot and its label is the optimal steering direction in order to maximize the distance between the car and any obstacle. The loss incurred by a particular hypothesis is determined from the measurement of a distance sensor after the car moved along the predicted direction. We can evaluate the loss only for the hypothesis that has actually been used to predict the optimal steering direction. It is impossible to evaluate the loss for other predictions of the optimal steering direction since the car already moved on.

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

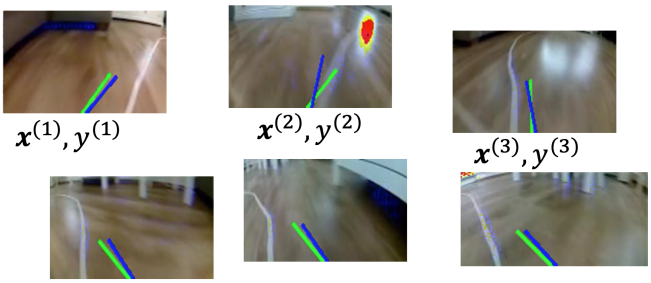


Figure 1.9: Autonomous driving requires to predict the optimal steering direction (label) based on an on-board camera snapshot (features) in each time instant. RL methods sequentially adjust a hypothesis for predicting the steering direction from the snapshot. The quality of the current hypothesis is evaluated by the measurement of a distance sensor (to avoid collisions with obstacles).

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 [33].

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.

Prerequisites. We assume some familiarity with basic concepts of linear algebra, real analysis, and probability theory. For a review of those concepts, we recommend [33, Chapter 2-4] and the references therein.

Notation. We mainly follow the notational conventions used in [33]. 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

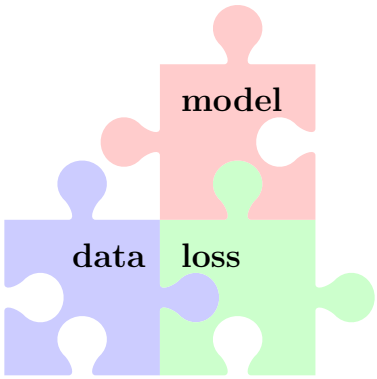


Figure 2.1: ML methods fit a model to data via minimizing a loss function.

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, including linear regression (see Section 3.1) as well as deep learning methods (see Section 3.11), are obtained by specific design choices for the three components. 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, random variables, videos within a movie database, cows within a herd, individual trees within a forest, individual forests within a collection of forests. Mountain hikers might be interested in datapoints that represent different hiking tours (see Figure 2.2).

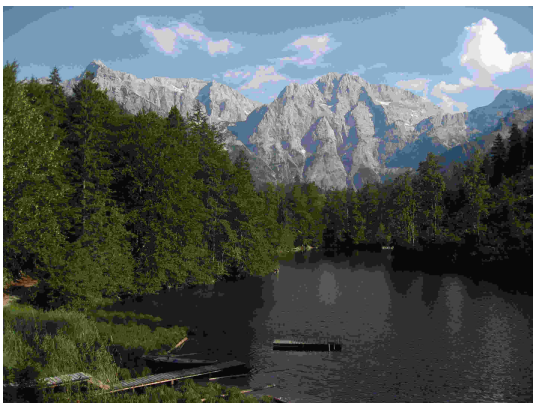


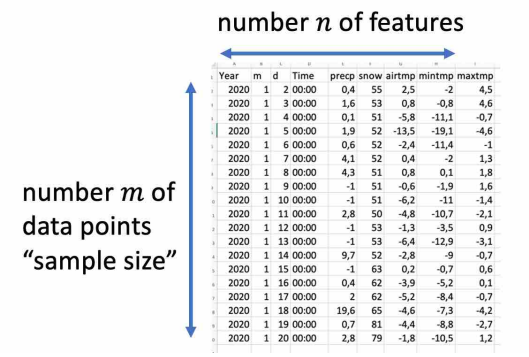
Figure 2.2: Snapshot taken at the beginning of a mountain hike.

We use the concept of datapoints in a very abstract and therefore highly flexible manner. Datapoints can represent very different types of objects that arise in fundamentally different application domains. For an image processing application it might be useful to define

datapoints as images. When developing a recommendation system we might define datapoints to represent customers. In the development of new drugs we might use data points to represent different diseases. The view in this book is that the meaning of definition of datapoints should be considered as a design choice. We might refer to the task of finding a useful definition of datapoints as **datapoint engineering**.

One practical requirement for a useful definition of datapoints is that we should have access to many of them. Many ML methods construct estimates for a quantity of interest (such as a prediction or forecast) by averaging over a set of reference (or training) datapoints. These estimates become more accurate for an increasing number of datapoints used for computing the average. 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 (such as memory size) 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. The behaviour of ML methods often depends crucially on the ratio m/n. In general, ML methods work best if m/n is much larger than one. We will study the relevance of the (informal) condition $mn \gg 1$ more precisely in Chapter 6.



Year	m	d	Time	precp	snow	airtmp	mintmp	maxtmp
2020	1	2	00:00	0,4	55	2,5	-2	4,5
2020	1	3	00:00	1,6	53	0,8	-0,8	4,6
2020	1	4	00:00	0,1	51	-5,8	-11,1	-0,7
2020	1	5	00:00	1,9	52	-13,5	-19,1	-4,6
2020	1	6	00:00	0,6	52	-2,4	-11,4	-1
2020	1	7	00:00	4,1	52	0,4	-2	1,3
2020	1	8	00:00	4,3	51	0,8	0,1	1,8
2020	1	9	00:00	-1	51	-0,6	-1,9	1,6
2020	1	10	00:00	-1	51	-6,2	-11	-1,4
2020	1	11	00:00	2,8	50	-4,8	-10,7	-2,1
2020	1	12	00:00	-1	53	-1,3	-3,5	0,9
2020	1	13	00:00	-1	53	-6,4	-12,9	-3,1
2020	1	14	00:00	9,7	52	-2,8	-9	-0,7
2020	1	15	00:00	-1	63	0,2	-0,7	0,6
2020	1	16	00:00	0,4	62	-3,9	-5,2	0,1
2020	1	17	00:00	2	62	-5,2	-8,4	-0,7
2020	1	18	00:00	19,6	65	-4,6	-7,3	-4,2
2020	1	19	00:00	0,7	81	-4,4	-8,8	-2,7
2020	1	20	00:00	2,8	79	-1,8	-10,5	1,2

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.

We find it 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 a **label**. Depending on the application domain, we might refer to labels also as a **target** or the **output variable**. The features of a datapoint might also be referred to as **input variables**.

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 us 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 their features is a design choice. Loosely speaking, features are properties or quantities that can be computed or measured easily. However, this is a highly informal characterization since there no universal criterion for the difficulty of computing of measuring a property of datapoints. The task of choosing which properties to use as features of data points might be the most challenging part in the application of ML methods. Chapter 9 discusses feature learning methods that automate (to some extend) the construction of good features.

In some application domains there is a rather natural choice for the features of a data point. For data points representing audio recording (of a given duration) we might use the signal amplitudes at regular sampling instants (e.g., using sampling frequency 44 kHz) as features. For data points representing images it seems natural to use the colour (red, green and blue) intensity levels of each pixel as a feature (see Figure 2.4).

The feature construction for images depicted in Figure 2.4 can be extended to other types of data points as long as they can be visualized efficiently. As a case in point, we might visualize an audio recording using an intensity plot of its spectrogram (see Figure 2.5). We can then use the pixel RGB intensities of this intensity plot as the features for an audio recording. Using this trick we can transform any ML method for image data to an ML method for audio data. We can use the scatter plot of a data set to use ML methods for image segmentation to cluster the dataset (see Chapter 8).

Many important ML application domains generate data points that are characterized by

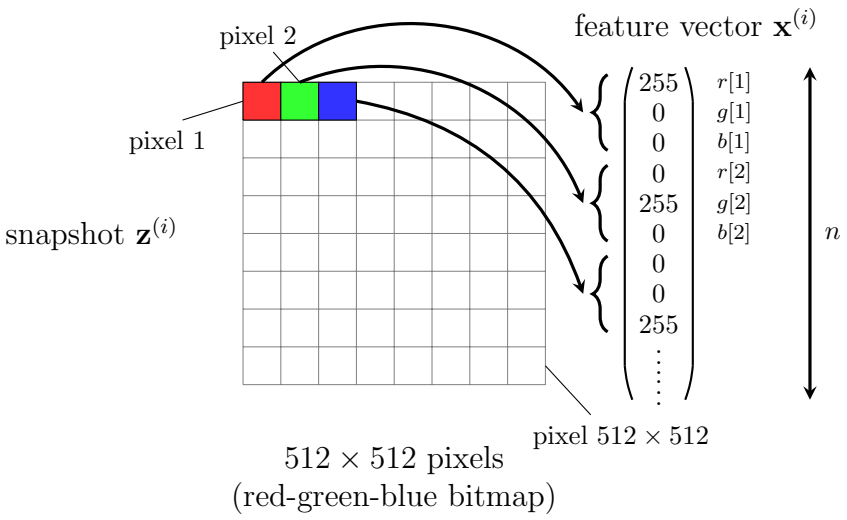


Figure 2.4: If the snapshot $\mathbf{z}^{(i)}$ is stored as a 512×512 RGB bitmap, we could use as features $\mathbf{x}^{(i)} \in \mathbb{R}^n$ the red-, green- and blue component of each pixel in the snapshot. The length of the feature vector would then be $n = 3 \cdot 512 \cdot 512 \approx 786000$.

several numeric features $x_1, \ldots, x_n \in \mathbb{R}$. Note that we represent features by real numbers which is might seem like an unnecessary restriction at this point. However for the purpose of this book this assumption means no significant loss of generality. Section 9.3 will discuss methods for constructing numeric features of data points whose natural representation is non-numeric.

We assume that data points arising in a given ML application are characterized by the same number n of individual features $x_1 \ldots, x_n$. We find it convenient to stack the individual features of a data point into a single feature vector

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

Each datapoint is then characterized by such a feature vector $\mathbf{x}$. Note that stacking the features of a datapoint into a column vector $\mathbf{x}$ is pure convention. We could also arrange the features as a row vector or even as a matrix, which might be even more natural for features obtained by the pixels of an image (see Figure 2.4).

We refer to the set of possible feature vectors of datapoints arising in some ML application as the **feature space** and denote it 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.

The Euclidean space $\mathbb{R}^n$ is an example of a feature space with a rich geometric and algebraic structure [71]. 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.

Modern information-technology, including smartphones or wearables, allows us 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 roughly five snapshots per day (sometimes more, e.g., during a mountain hike) resulting in more than 1000 snapshots per year. Each snapshot contains around 10^6 pixels whose greyscale levels we can use as features of the datapoint. This results in more than 10^9 features (per year!). If we stack all those features into a 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 [15, 88]. 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 concentrated near low-dimensional subspaces (or manifolds).

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 be detrimental for the performance of ML methods to use an excessive amount of (irrelevant) features. Computationally, using too many features might result in prohibitive computational resource requirements (such as processing time). Statistically, each additional feature typically introduces an additional amount of noise (due to measurement or modelling errors) which is harmful for the accuracy of the ML method.

It is difficult to give a precise and broadly applicable characterization of the maximum number of features that should be used to characterize the datapoints. As a rule of thumb, the number m of (labeled) datapoints used to train a ML method should be much larger than the number n of numeric features (see Figure 2.3). 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. We next discuss implementations for each of these two complementary approaches.

The acquisition of (labeled) datapoints might be costly, requiring human expert labour. Instead of collecting more raw data, it might be more efficient to generate synthetic data. Section 7.3 shows how intrinsic symmetries in the data 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. Chapter 7 interprets regularization techniques as an implicit form of data augmentation.

The informal condition $m/n \gg 1$ can also be ensured by reducing 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 discusses methods that learn, based on some given dataset, to determine a small number of relevant features of datapoints.

Beside the available computational infrastructure, also the statistical properties of datasets must be taken into account for the choices of the feature space. The linear algebraic structure of $\mathbb{R}^n$ allows us to efficiently represent and approximate datasets that are well aligned along linear subspaces. Section 9.2 discusses a basic method to optimally approximate datasets by linear subspaces of a given dimension. The geometric structure of $\mathbb{R}^n$ is also 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 $\mathbb{R}^n$ reflect the intrinsic structure of the data generated in many important application domains. This should not be too surprising as the Euclidean space has evolved as a useful mathematical abstraction of physical phenomena.

In general there is no mathematically correct choice for which properties of a data point to be used as its features. Most application domains allow for some design freedom in the choice of features. Let us illustrate this design freedom with a personalized health-care applications. This application involves data points that represent audio recordings with the

fixed duration of three seconds. These recordings are obtained via smartphone microphones and used to detect coughing [6].

Audio recordings are typically available a sequence of signal amplitudes a_t collected regularly at time instants $t = 1, \ldots, n$ with sampling frequency ≈ 44 kHz. From a signal processing perspective, it seems natural to directly use the signal amplitudes as features, $x_j = a_j$ for $j = 1, \ldots, n$. However, another choice for the features would be the pixel RGB values of some visualization of the audio recording. Figure 2.5 depicts two possible visualizations of an audio signal obtained from a line plot of the signal amplitudes (as a function of time index t) or an intensity plot of the spectrogram [12, 55].

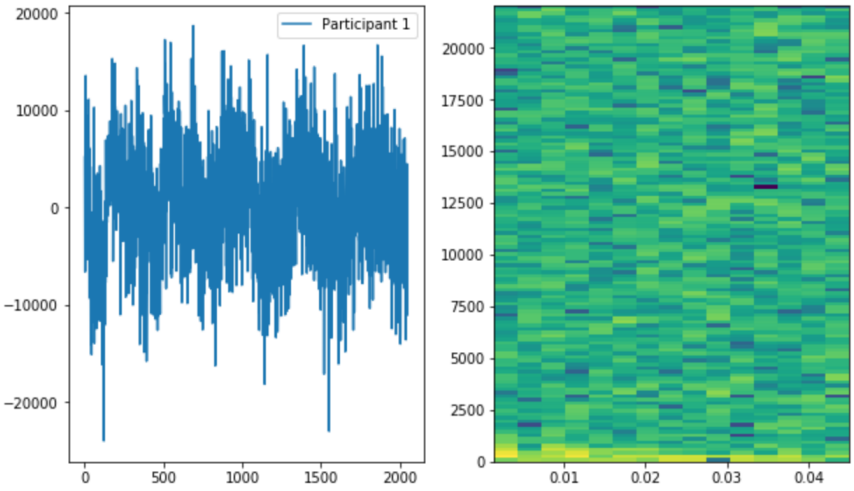


Figure 2.5: Two visualizations of an audio recording obtained from a line plot of the signal amplitudes and by the spectrogram of the recording.

2.1.2 Labels

Besides the features of a datapoint, there are other properties of a data point that represent some higher-level fact (or quantity of interest associated) with the datapoint. We refer to the higher level fact, 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 $y' < 0$.

We refer to a data point as being *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 [76], running a particle physics experiment at the European organization for nuclear research (CERN) [16], running animal testing in pharmacology [30].

There are also online market places for human labelling workforce [58]. 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 [36]. ML methods for speech recognition and machine translation can make use of massive labeled datasets that are freely available [47].

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 $h : \mathcal{X} \rightarrow \mathcal{Y}$ 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

A powerful idea in ML is to interpret each datapoints as the realization of a **random variable (RV)**. For ease of exposition let us consider datapoints that are characterized by a single feature x. The following concepts can be extended easily to datapoints characterized by a feature vector $\mathbf{x}$ and a label y.

One of the most basic examples of a probabilistic model for datapoints in ML is the **"independent and identically distributed" (i.i.d.)** assumption. This assumption interprets datapoints $x^{(1)}, \ldots, x^{(m)}$ as realizations of statistically independent RVs with the same probability distribution $p(x)$.[1] It might seem somewhat awkward to interpret datapoints as realizations of RVs with the common probability distribution $p(x)$. However, this interpretation allows us to use the properties of the probability distribution to characterize overall properties of (large) collections of datapoints.

The probability distribution $p(x)$ underlying the data points within the i.i.d. assumption is either known (based on some domain expertise) or estimated from data. It is often enough to estimate only some parameters of the distribution $p(x)$. Section 3.12 discusses a principled approach to estimate the parameters of a probability distribution from datapoints. This approach is sometimes referred to as maximum likelihood and aims at finding (parameter of) a probability distribution $p(x)$ such that the probability (density) of actually observing the given data points is maximized [52, 46, 8].

Two of the most basic and widely used parameters of a probability distribution $p(x)$ are the expected value or **mean**

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

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,

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

[1] We assume the reader is familiar with the concepts of a probability distribution which reduces to the concept of a probability mass function for discrete RVs [8].

The sample mean and variance (2.1) are the maximum likelihood estimates for the mean and variance of a normal (Gaussian) distribution $p(x)$ (see [11, Chapter 2.3.4].

Most of the ML methods discussed in this book are motivated by the i.i.d. assumption. It is important to note that this i.i.d. assumption is only a modelling hypothesis. There is no means by which we can verify if an arbitrary set of data points are "really" obtained from realizations of i.i.d. RVs. However, there are principled statistical methods (hypothesis tests) if a given set of data point can be well approximated as realizations of i.i.d. RVs [54]. The only way to ensure the i.i.d. assumption is to generate synthetic data using a random number generator. Such synthetic i.i.d. data points could be obtained by sampling algorithms that incrementally build a synthetic dataset by adding randomly chosen raw data points [24].

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**

$$\mathcal{R}^{(a)} := \{\mathbf{x} \in \mathbb{R}^n : \hat{y} = a\} \subseteq \mathcal{X}. \tag{2.3}$$

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, $\hat{y} = a \in \mathcal{Y}$.

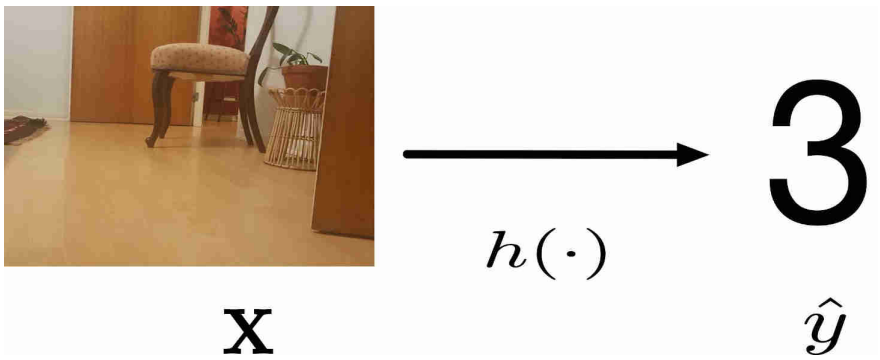


Figure 2.6: A predictor (hypothesis) h maps features $\mathbf{x} \in \mathcal{X}$, of an on-board camera snapshot, to the prediction $\hat{y} = h(\mathbf{x}) \in \mathcal{Y}$ for the coordinate of the current location of a cleaning robot. ML methods use data to learn predictors h such that $\hat{y} \approx y$ (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 2.10, 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.

We can implement ML methods using a spreadsheet software by using a hypothesis space of 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.

Since most desktop computers allow for efficient numerical linear algebra, some of the most popular ML methods use the hypothesis space

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

The hypothesis space (2.4) is constituted by linear maps (functions)

$$h^{(\mathbf{w})}\big(\mathbf{x}\big) : \mathbb{R}^n \to \mathbb{R} : \mathbf{x} \mapsto \mathbf{w}^T\mathbf{x}. \tag{2.5}$$

The function $h^{(\mathbf{w})}$ (2.5) maps, in a linear fashion, 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 2.8).

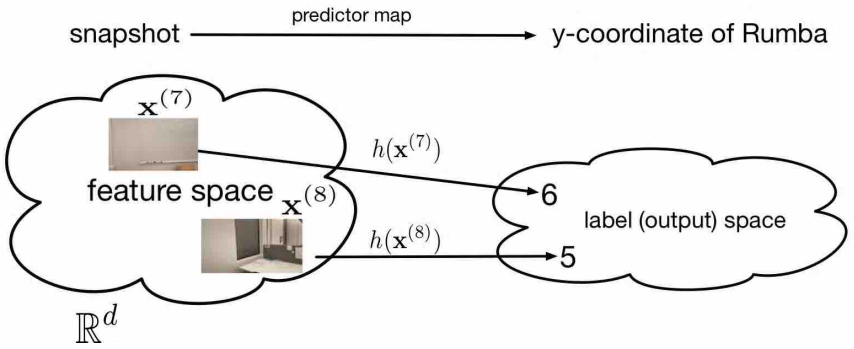


Figure 2.7: A predictor (hypothesis) $h : \mathcal{X} \to \mathcal{Y}$ takes the feature vector $\mathbf{x}^{(t)} \in \mathcal{X}$ (e.g., representing the snapshot taken by Rumba at time t) as input and outputs a predicted label $\hat{y}_t = h(\mathbf{x}^{(t)})$ (e.g., the predicted y-coordinate of Rumba 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 $\mathbf{w} \in \mathbb{R}^n$. This parametrization of the hypothesis space $\mathcal{H}$ allows to process and manipulate hypothesis maps by vector operations. In particular, instead of searching over the function space $\mathcal{H}$ (its elements are functions!) to find a good hypothesis, 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

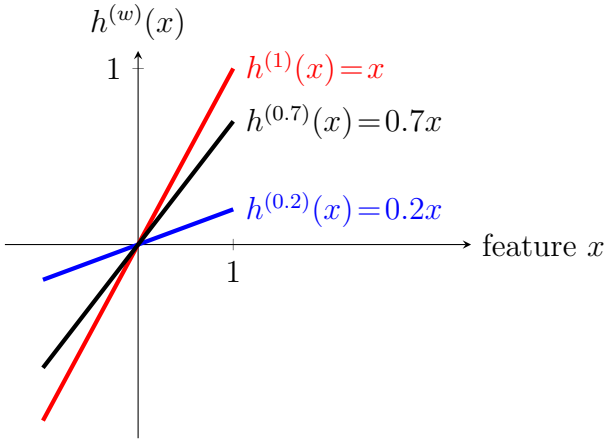


Figure 2.8: Three particular members of the hypothesis space $\mathcal{H} = \{h^{(w)} : \mathbb{R} \to \mathbb{R}, h^{(w)}(x) = w \cdot x\}$ which consists of all linear functions of the scalar feature x. We can parametrize this hypothesis space conveniently using the weight $w \in \mathbb{R}$ as $h^{(w)}(x) = w \cdot x$.

algebraic structure that allows us 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. We refer to a classifier that determine the predicted label solely based on the value of a linear map as a **linear classifier**.

Figure 2.9 illustrates the decision regions (2.3) of a **linear classifier** for binary labels. The decision regions are half-spaces and, in turn, the decision boundary is a hyperplane $\{\mathbf{x} : \mathbf{w}^T\mathbf{x} = b\}$. Note that each **linear classifier** corresponds to a particular linear hypothesis map from the hypothesis space (2.4). However, we can use different loss functions to measure the quality of a **linear classifier**. Three widely-used examples for ML methods that learn a **linear classifier** are logistic regression (see Section 3.6), the support vector machine (SVM) (see Section 3.7) and the naive Bayes' classifier (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

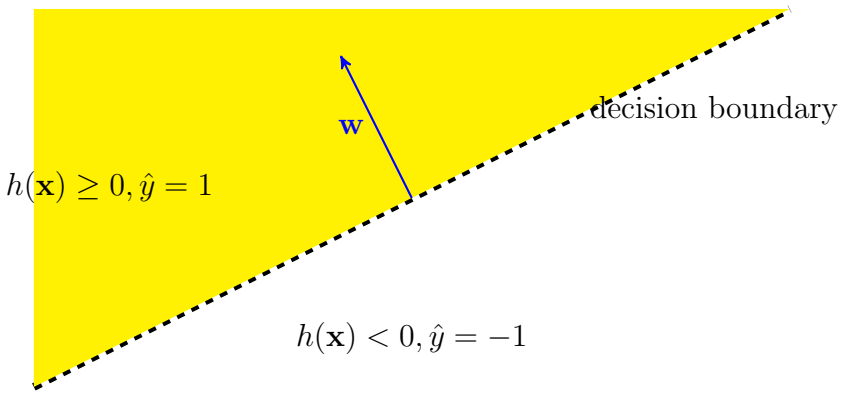


Figure 2.9: A hypothesis $h : \mathcal{X} \to \mathcal{Y}$ for a binary classification problem, with label space $\mathcal{Y} = \{-1, 1\}$ and feature space $\mathcal{X} = \mathbb{R}^2$, can be represented conveniently via the decision boundary (dashed line) which separates all feature vectors $\mathbf{x}$ 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**.

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

$$\hat{y} = \mathbf{w}^T\mathbf{z} = \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}). \tag{2.6}$$

Section 3.9 discusses the family of kernel methods which are based on the concatenation (2.6) 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 [33, 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 [4]. 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. For example, consider datapoints represented by $100 \times 10 = 1000$ black-and-white pixels 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 use the number of its elements as a measure for its size. Indeed, for an infinite hypothesis space, the number of elements is not well-defined. Therefore, we measure the size of a hypothesis space $\mathcal{H}$ using its **effective dimension** $d_{\text{eff}}(\mathcal{H})$.

Consider a hypothesis space $\mathcal{H}$ consisting of maps $h : \mathcal{X} \to \mathcal{Y}$ that read in the features $\mathbf{x} \in \mathcal{X}$ and output an predicted label $\hat{y} = h(\mathbf{x}) \in \mathcal{Y}$. We define the effective dimension $d_{\text{eff}}(\mathcal{H})$ of $\mathcal{H}$ as the maximum number $D \in \mathbb{N}$ such that for any set $\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(D)}, y^{(D)}\big)\big\}$ of D data points with different features, we can always find a hypothesis $h \in \mathcal{H}$ that perfectly fits the labels, $y^{(i)} = h\big(\mathbf{x}^{(i)}\big)$ for $i = 1, \ldots, D$.

The effective dimension of a hypothesis space is closely related to the **Vapnik–Chervonenkis (VC) dimension** [83]. The VC dimension is maybe the most widely used concept for measuring the size of infinite hypothesis spaces. However, the precise definition of the VC dimension are beyond the scope of this book. Moreover, the effective dimension captures most of the relevant properties of the VC dimension for our purposes. For a precise definition of the VC dimension and discussion of its applications in ML we refer to [73].

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 us 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 $\widehat{\mathbf{w}}$ such that $y^{(i)} = \widehat{\mathbf{w}}^T\mathbf{x}^{(i)}$ for all $i = 1, \ldots, m$. The effective dimension 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 effective dimension of the hypothesis space $\mathcal{H}^{(n)}_{\text{poly}}$ is $D = n$. Section 3.2 discusses polynomial regression in more detail.

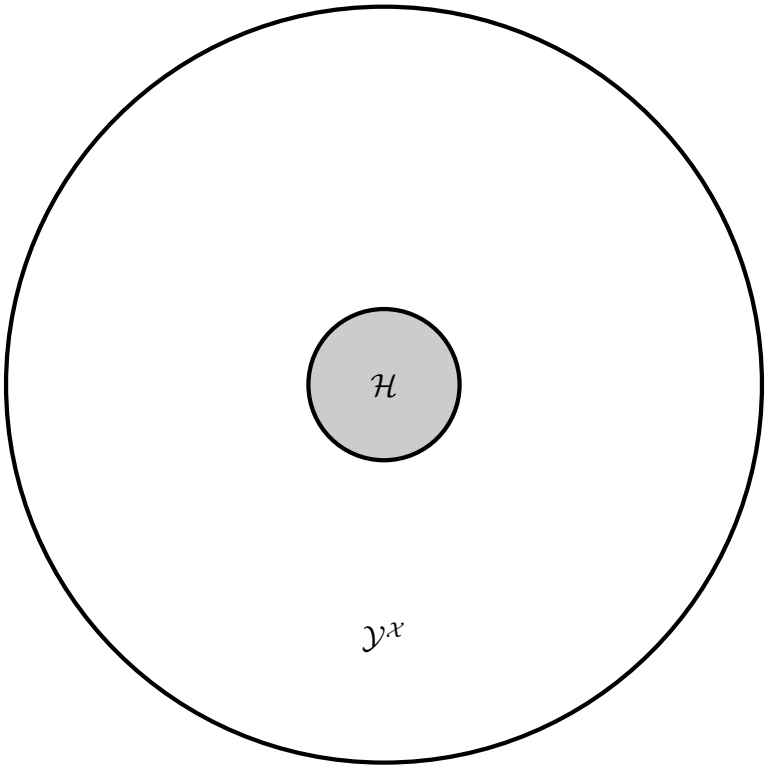


Figure 2.10: The hypothesis space $\mathcal{H}$ is a (typically very small) subset of the (typically very large) set $\mathcal{Y}^{\mathcal{X}}$ of all possible maps from feature space $\mathcal{X}$ into the label space $\mathcal{Y}$.

feature x	prediction $h(x)$
0	0
1/10	10
2/10	3
⋮	⋮
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.

2.3 The Loss

Every ML method uses a (more of less explicit) 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? To answer this questions, ML methods use **loss functions**. A loss function $\mathcal{L} : \mathcal{X} \times \mathcal{Y} \times \mathcal{H} \to \mathbb{R}$ measures the loss (or error) $\mathcal{L}((\mathbf{x}, y), h)$ incurred by predicting the label y of a datapoint using the function value $h(\mathbf{x})$ obtained by applying the hypothesis map to the feature vector $\mathbf{x}$.

Throughout this book we mainly consider loss function that are non-negative, $\mathcal{L}((\mathbf{x}, y), h) \geq 0$. These loss functions are small only if the hypothesis h allows to predict the label y of a datapoint with features $\mathbf{x}$. The basic idea behind most ML methods is quite simple: learn (find) the particular hypothesis out of a given hypothesis space $\mathcal{H}$ that incurs a small loss $\mathcal{L}((\mathbf{x}, y), h)$ for any datapoint (see Chapter 4). Like the hypothesis space $\mathcal{H}$ used in a ML method, also the loss function is a design choice. We will discuss some widely used examples for loss function in Section 2.3.1 and Section 2.3.2.

The choice for the loss function should take into account the computational complexity of searching the hypothesis space for a hypothesis with minimum loss. Consider a ML method that uses a hypothesis space parametrized by a weight vector and a loss function that is a convex and differentiable (smooth) function of the weight vector. In this case, searching for a hypothesis with small loss can be done efficiently using the gradient-based methods discussed in Chapter 5. The minimization of loss functions that are either non-convex or non-differentiable is typically computationally much more expensive. We further discuss the computational complexities of different types of loss functions in Section 4.2.

Beside their computational complexity, loss functions differ also in their statistical properties. We will see that some loss functions allow to make a ML method robust against outliers (see Section 3.3). More generally, there is a system approach to choose useful loss functions using a probabilistic model for the datapoints arising in a ML application. Section 3.12 details how the maximum likelihood principle of statistical inference provides an explicit construction of loss functions in terms of an (assumed) probability distribution for datapoints.

2.3.1 Loss Functions for Numeric Labels

For ML problems involving data points with numeric labels $y \in \mathbb{R}$, i.e., for regression problems (see Section 2.1.2), a widely used (first) choice for the loss function can be the

squared error loss

$$\mathcal{L}((\mathbf{x}, y), h) := \big(y - \underbrace{h(\mathbf{x})}_{=\hat{y}}\big)^2. \tag{2.7}$$

The squared error loss (2.7) 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 features $\mathbf{x}$ are required to determine the squared error loss. We will slightly abuse notation and use the shorthand $\mathcal{L}(y, \hat{y})$ for any loss function that depends on the features $\mathbf{x}$ only via the predicted label $\hat{y} = h(\mathbf{x})$. Figure 2.11 depicts the squared error loss as a function of the prediction error $y - \hat{y}$.

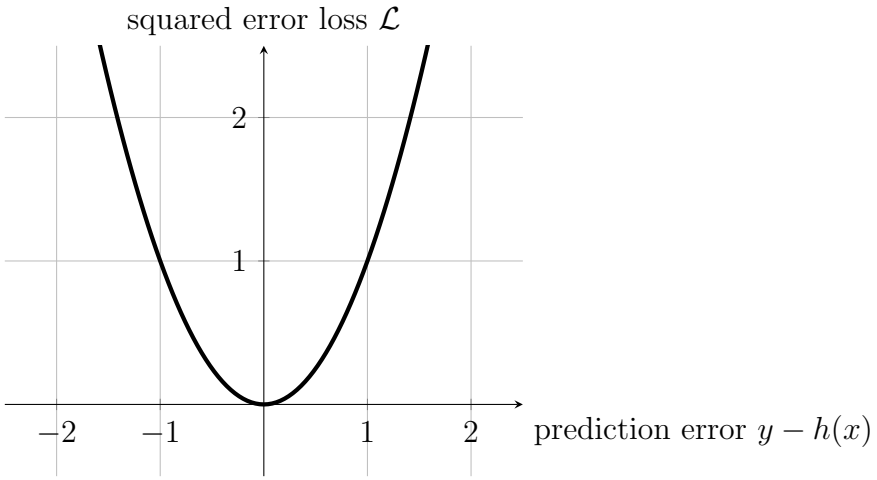


Figure 2.11: A widely used choice for the loss function in regression problems (with label space $\mathcal{Y} = \mathbb{R}$) is the squared error loss (2.7). Note that, for a given hypothesis h, we can evaluate the squared error loss only if we know the features $\mathbf{x}$ and the label y of the data point.

The squared error loss (2.7) 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 [37, Sec. 2.6.3].

Another loss function used in regression problems is the absolute error loss $|\hat{y} - y|$. Using this loss function to guide the learning of a predictor results in methods that are

robust against few outliers in the training set (see Section 3.3). However, this improved robustness comes at the expense of increased computational complexity of minimizing the (non-differentiable) absolute error loss compared to the (differentiable) squared error loss (2.7).

2.3.2 Loss Functions for Categorical Labels

Classification problems involve data points whose labels take on values from a discrete label space $\mathcal{Y}$. In what follows, unless stated otherwise, we focus on binary classification problems. Moreover, without loss of generality we assume that labels values are $\mathcal{Y} = \{-1, 1\}$. Classification methods aim at learning a classifier that maps the features $\mathbf{x}$ of a data point to a predicted label $\hat{y} \in \mathcal{Y}$.

We implement a classifier by thresholding the value $h(\mathbf{x}) \in \mathbb{R}$ of a hypothesis that can deliver arbitrary real numbers. We then classify a data point as $\hat{y} = 1$ if $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ otherwise. Thus, the predicted label is obtained from the sign of the value $h(\mathbf{x})$. While the sign of $h(\mathbf{x})$ determines the classification result, i.e., the predicted label $\hat{y}$, we interpret the absolute value $|h(\mathbf{x})|$ as the confidence in this classification.

It might seem sensible to measure the quality of a hypothesis when used to classify data points using the squared error loss (2.7). However, the squared error is typically a bad measure for the quality of a hypothesis $h(\mathbf{x})$ used to classify a data point with binary label $y \in \{-1, 1\}$. Figure 2.12 illustrates how the squared error loss of a hypothesis, which is used for classification, can be misleading.

Figure 2.12 depicts a dataset consisting of $m = 4$ data points with binary labels $y^{(i)} \in \{-1, 1\}$, for $i = 1, \ldots, m$. The figure also depicts two candidate hypotheses $h^{(1)}(x)$ and $h^{(2)}(x)$ that can be used for classifying data points. The classifications $\hat{y}$ obtained with the hypothesis $h^{(2)}(x)$ would perfectly match the labels of the four training datapoints since $h^{(2)}\big(x^{(i)}\big) \geq 0$ if and if only if $y^{(i)} = 1$. In contrast, the classifications $\hat{y}^{(i)}$ obtained by thresholding $h^{(1)}(x)$ are wrong for data points with $y = -1$. Thus, based on the training data, we would prefer using $h^{(2)}(x)$ over $h^{(1)}$ to classify data points. However, the squared error loss incurred by the (reasonable) classifier $h^{(2)}$ is much larger than the squared error loss incurred by the (poor) classifier $h^{(1)}$. 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.

Generally speaking, we want the loss function to punish (deliver large values for) a hypothesis that is very confident ($|h(\mathbf{x})|$ is large) in a wrong classification ($\hat{y} \neq y$). Moreover,

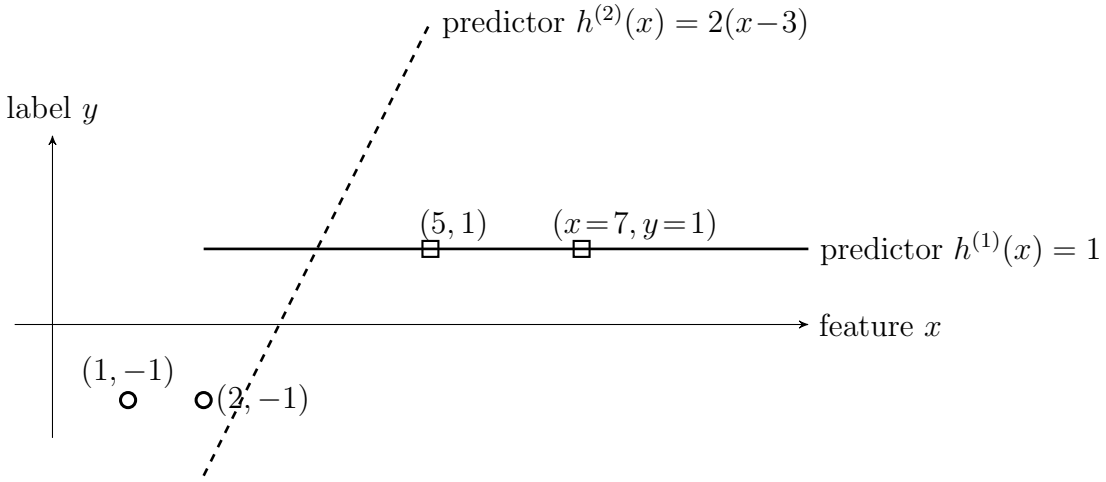


Figure 2.12: Training set consisting of four data points with binary labels $\hat{y}^{(i)} \in \{-1, 1\}$. Minimizing the squared error loss (2.7) would prefer the (poor) classifier $h^{(1)}$ over the (reasonable) classifier $h^{(2)}$.

a good loss function should not punish (deliver small values for) a hypothesis is very confident ($|h(\mathbf{x})|$ is large) in a correct classification ($\hat{y} = y$). However, by its very definition, the squared loss yields large values if the confidence $|h(\mathbf{x})|$ is large, no matter if the resulting classification is correct or wrong.

We now discuss some loss functions which have proven useful for assessing the quality of a hypothesis used to classify data points. It is important to note that some of the formulas for these loss function only apply if the label values are the real numbers -1 and 1. However in some binary classification problems there might be different natural choices for label values (e.g. $\{0, 1\}$ or $\{\square, \triangle\}$. It is then necessary to first map these label values to -1 and 1 so that the formulas for the loss functions below can be used.

The first loss function that we discuss is essentially a formalization of the natural requirement for a hypothesis to result on correct classifications, i.e., $\hat{y} = y$ for any datapoint. This suggests to learn a hypothesis $h(\mathbf{x})$ by minimizing the 0/1 loss

$$\mathcal{L}((\mathbf{x}, y), h) := \begin{cases} 1 & \text{if } y \neq \hat{y} \\ 0 & \text{else,} \end{cases} \text{ with } \hat{y} = 1 \text{ for } h(\mathbf{x}) \geq 0, \text{ and } \hat{y} = -1 \text{ for } h(\mathbf{x}) < 0. \quad (2.8)$$

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 value $h(\mathbf{x})$. The 0/1 loss is equal to zero if the hypothesis yields a correct classification $\hat{y} = y$ (see (**??**)) and equal to one if it yields a wrong classification

$\hat{y} \neq y$.

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

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

with high probability for sufficiently large sample size m. A precise formulation of the approximation (2.9) can be obtained from the **law of large numbers** [10, 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. RVs. The average 0/1 loss on the left-hand side of (2.9) is referred to as the **accuracy** of the hypothesis h.

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).

To avoid the non-convexity of the 0/1 loss we can approximate it by a convex loss function. One popular convex approximation of the 0/1 loss is 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. Non-differentiable loss functions are typically harder to minimize, implying a higher computational complexity of the ML method using such a loss.

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

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

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

Consider a specific datapoint with the feature vector $\mathbf{x} \in \mathbb{R}^n$ and a binary label $y \in \{-1, 1\}$. Let us use a linear hypothesis $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$, with some weight vector, to predict

the label based on the features $\mathbf{x}$ according to $\hat{y} = 1$ if $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} > 0$ and $\hat{y} = -1$ otherwise. Then, both the hinge loss (2.10) and the logistic loss (2.11) are **convex functions** of the weight vector $\mathbf{w} \in \mathbb{R}^n$. The logistic loss (2.11) depends **smoothly** on $\mathbf{w}$. It is a differentiable function in the sense of allowing to determine a gradient with respect to $\mathbf{w}$. In contrast, the hinge loss (2.10) is **non-smooth** which makes it more difficult to minimize.

ML methods that use the convex and differentiable logistic loss function, such as logistic regression in Section 3.6, can apply simple **gradient based methods** such as GD (see Chapter 5) to minimize the average loss. In contrast, we cannot use gradient based methods to minimize the hinge loss since it is not differentiable. However, we can apply a generalization of GD which is known as **subgradient descent** [14]. Loosely speaking, subgradient descent is obtained from GD by replacing the concept of a gradient with the concept of a subgradient.

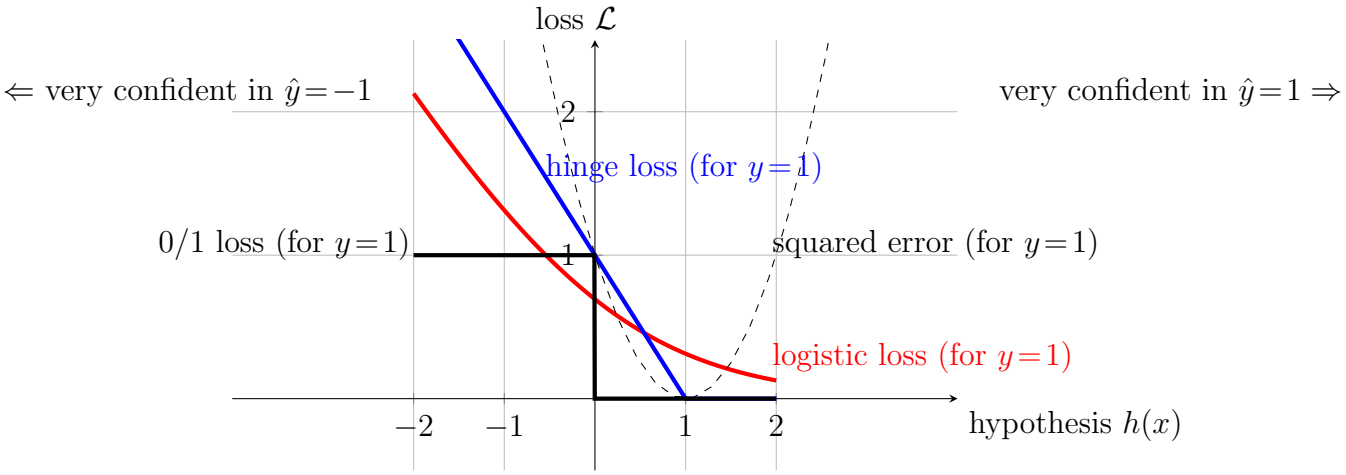


Figure 2.13: The solid curves depict three widely-used loss functions of a hypothesis h used in binary classification. A data points is classified as $\hat{y} = 1$ if $h(x) \geq 0$ and classified as $\hat{y} = -1$ if $h(x) < 0$. We can interpret the absolute value $|h(x)|$ as the confidence in the classification. 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 dashed curve depicts the squared error loss (2.7), which increases for increasing $h(x)$.

2.3.3 Empirical 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. Many ML methods approximate the expected value of the loss incurred by a given hypothesis by an empirical (sample) average over a finite set of labeled datapoints $\mathcal{D} = \big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)$.

We define the **empirical risk** of a hypothesis $h \in \mathcal{H}$ for a dataset $\mathcal{D}$ as

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

The empirical risk of $h \in \mathcal{H}$ is the average loss on the datapoints in $\mathcal{D}$. To ease notational burden and if the dataset $\mathcal{D}$ is clear from the context, we use $\mathcal{E}(h)$ as a shorthand for $\mathcal{E}(h|\mathcal{D})$. Note that in general the empirical risk depends on both, the hypothesis h and the properties of the dataset $\mathcal{D}$.

As discussed in Section 4.1, the empirical risk (2.12) can be interpreted as an approximation of the expected loss obtained for a random data point with probability distribution $p(\mathbf{x}, y)$. This approximation is valid if the data points are realizations of i.i.d. RVs whose common distribution is $p(\mathbf{x}, y)$. We hope that a hypothesis with small empirical risk (2.12) will also result in a small expected loss. The minimum possible expected loss is achieved by the Bayes' estimator of the label y, given the features $\mathbf{x}$. However, l to actually compute this optimal estimator we would need to know the probability distribution $p(\mathbf{x}, y)$.

Confusion Matrix

Consider a dataset $\mathcal{D}$ consisting of data point characterized by feature vectors $\mathbf{x}^{(i)}$ and labels $y^{(i)} \in \{1, \ldots, k\}$. We might interpret the label value of a data point as the index of a category or class to which the data point belongs to. Multi-class classification problems aim at learning a hypothesis h such that $h(\mathbf{x}) \approx y$ for any data point. In principle, we could measure the quality of a given hypothesis h by the average 0/1 loss incurred on the labeled data points in (the training set) $\mathcal{D}$. However, if the dataset $\mathcal{D}$ dominated by data points having one specific label value the average 0/1 loss might obscure the performance of h for datapoints having rare label values. Even if the average 0/1 loss is very small, the hypothesis might perform poorly for data points of a minority category.

The confusion matrix generalizes the concept of the 0/1 loss to application domains where the relative frequency (fraction) of data points depends significantly on their label values (imbalanced data). Instead of considering only the average 0/1 loss incurred by a hypothesis on a dataset $\mathcal{D}$, we use a whole family of loss functions. In particular, for each

pair of label values $p, q \in \{1, \ldots, k\}$, we define the loss

$$\mathcal{L}^{(p \to q)}\big(\big(\mathbf{x}, y\big), h\big) := \begin{cases} 1 & \text{if } y = p \text{ and } h(\mathbf{x}) = q \\ 0 & \text{otherwise.} \end{cases} \tag{2.13}$$

We then evaluate a classifier by computing the average loss (2.13) incurred on the dataset $\mathcal{D}$,

$$\mathcal{E}^{(p \to q)}(h|\mathcal{D}) := (1/m) \sum_{i=1}^{m} \mathcal{L}^{(p \to q)}\big((\mathbf{x}^{(i)}, y^{(i)}), h\big) \text{ for } p, q \in \{1, \ldots, k\}. \tag{2.14}$$

It is convenient to arrange the values (2.14) as a matrix whose rows correspond to potential label values p of a data point and columns correspond to potential values q delivered by the hypothesis $h(\mathbf{x})$. .

Precision, Recall and F-Measure

Consider an object detection application where data points represent images. The label of data points might indicate the presence ($y = 1$)or absence ($y = -1$) of an object, it is then customary to define the [5]

$$\text{recall } := \mathcal{E}^{(1 \to 1)}(h|\mathcal{D}), \text{ and the precision } := \frac{\mathcal{E}^{(1 \to 1)}(h|\mathcal{D})}{\mathcal{E}^{(1 \to 1)}(h|\mathcal{D}) + \mathcal{E}^{(-1 \to 1)}(h|\mathcal{D})}. \tag{2.15}$$

Clearly, we would like to find a hypothesis with both, large recall and large precision. However, these two goals are typically conflicting, a hypothesis with a high recall will have small precision. Depending on the application, we might prefer having a high recall and tolerate a lower precision.

It might be convenient to combine the recall and precision of a hypothesis into a single quantity,

$$F_1 := 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.16}$$

The F measure (2.16) is the harmonic mean [1] of the precision and recall of a hypothesis h. It is a special case of the F_β-score

$$F_\beta := \big(1 + \beta^2\big) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \text{precision} + \text{recall}}. \tag{2.17}$$

The F measure (2.16) is obtained from (2.17) for the choice $\beta = 1$. It is therefore customary

to refer to (2.16) as the F_1-score of a hypothesis h.

Ordinal Label Values

Some application domains involve data points whose different label values have a natural ordering. Consider data points representing areal images of rectangular areas of size 1 km by 1 km. We characterize each data point (rectangular area) by the feature vector $\mathbf{x}$ obtained by stacking the RGB values of each image pixel (see Figure 2.4). Beside the feature vector, each rectangular area is characterized by a label $y \in \{1, 2, 3\}$ where

- $y = 1$ means that the area contains no trees.
- $y = 2$ means that the area is partially covered by trees.
- $y = 3$ means that the are is entirely covered by trees.

Thus we might say that label value $y = 2$ is "larger" than label value $y = 1$ and label value $y = 3$ is "larger" than label value $y = 2$. It might be useful to take the ordering of label values into account when evaluating the quality of the predictions obtained by a hypothesis $h(\mathbf{x})$.

Consider a data point with feature vector $\mathbf{x}$ and label $y = 1$ as well as two different hypotheses $h^{(a)}, h^{(b)} \in \mathcal{H}$. The hypothesis $h^{(a)}$ delivers the predicted label $\hat{y}^{(a)} = h^{(a)}(\mathbf{x}) = 2$, while the other hypothesis $h^{(b)}$ delivers the predicted label $\hat{y}^{(a)} = h^{(a)}(\mathbf{x}) = 3$. Both predictions are wrong, since they are different form the true label value $y = 1$. It seems reasonable to consider the prediction $\hat{y}^{(a)}$ to be less wrong than the prediction $\hat{y}^{(b)}$ and therefore we would prefer the hypothesis $h^{(a)}$ over $h^{(b)}$. However, the 0/1 loss is the same for $h^{(a)}$ and $h^{(b)}$ and therefore does not reflect our preference for $h^{(a)}$. We need to modify (or tailor) the 0/1 loss to take into account the application-specific ordering of label values. For the above application, we might define a loss function via

$$\mathcal{L}((\mathbf{x}, y), h) := \begin{cases} 0 & \text{, when } y = h(\mathbf{x}) \\ 10 & \text{, when } |y - h(\mathbf{x})| = 1 \\ 100 & \text{otherwise.} \end{cases} \tag{2.18}$$

???? define Bayes risk ????

2.3.4 Regret

In some ML applications, 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 [39]. This difference, which is referred to as the **regret**, measures by how much we regret to have used the prediction $h(\mathbf{x})$ instead of using (following) the prediction of the expert. The goal of regret minimization is to learn a hypothesis with a small regret compared to all considered experts.

The concept of regret minimization is useful when we do not make any probabilistic assumptions (see Section 2.1.4) about the data. Without a probabilistic model we cannot use the Bayes risk, which is the risk of the Bayes optimal estimator, as a benchmark.

Regret minimization techniques can be designed and analyzed without any such probabilistic model for the data [18]. This approach replaces the Bayes risk with the regret relative to given reference predictors (experts) as the benchmark.

2.3.5 Rewards as Partial Feedback

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 evaluate the loss function for different choices for the hypothesis. Indeed, the evaluation of the loss function typically amounts to measuring the distance between predicted label and true label of a data point. Instead of evaluating a loss function, we must rely on some indirect feedback or "reward" that indicates the usefulness of a particular prediction [18, 81].

Consider the ML problem of predicting the optimal steering directions for an autonomous car. The prediction has to be recalculated for each new state of the 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). Unless the car circles around in small area with fixed obstacles, we have no access to labeled datapoints or reference driving scenes for which we already know the optimum steering direction. Instead, the car (control unit) needs to learn the hypothesis $h(\mathbf{x})$ based solely on the feedback signals obtained from various sensing devices (cameras, distance sensors).

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

$$\big(x^{(1)}, y^{(1)}\big), \ldots, \big(x^{(m)}, y^{(m)}\big) \tag{2.19}$$

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

$$h^{(w_0,w_1)}(x) = w_1 x + w_0. \tag{2.20}$$

The predictor (2.20) 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 (2.7))

$$\begin{aligned} f(w_0, w_1) &:= (1/m) \sum_{i=1}^{m} \big(y^{(i)} - h^{(w_0,w_1)}(x^{(i)})\big)^2 \\ &\overset{(2.20)}{=} (1/m) \sum_{i=1}^{m} \big(y^{(i)} - (w_1 x^{(i)} + w_0)\big)^2. \end{aligned} \tag{2.21}$$

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 (2.19).

It seems natural to learn a good predictor (2.20) by choosing the weights w_0, w_1 to

minimize the training error

$$\min_{w_1,w_0\in\mathbb{R}} f(w_0,w_1) \stackrel{(2.21)}{=} \min_{w_1,w_0\in\mathbb{R}} (1/m)\sum_{i=1}^{m}\big(y^{(i)}-(w_1x^{(i)}+w_0)\big)^2. \tag{2.22}$$

The optimal weights w_0', w_1' are characterized by the **zero-gradient condition**,[2]

$$\frac{\partial f(w_0',w_1')}{\partial w_0}=0, \text{ and } \frac{\partial f(w_0',w_1')}{\partial w_1}=0. \tag{2.23}$$

Inserting (2.21) into (2.23) and by using basic rules for calculating derivatives, we obtain the following optimality conditions

$$(1/m)\sum_{i=1}^{m}\big(y^{(i)}-(w_1'x^{(i)}+w_0')\big)=0, \text{ and } (1/m)\sum_{i=1}^{m}x^{(i)}\big(y^{(i)}-(w_1'x^{(i)}+w_0')\big)=0. \tag{2.24}$$

Any weights w_0', w_1' that satisfy (2.24) 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 (2.24) using matrices and vectors. To this end, we first rewrite the predictor (2.20) 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 (2.19) into the feature matrix and label vector,

$$\mathbf{X}=\big(\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(m)}\big)^T\in\mathbb{R}^{m\times 2}, \mathbf{y}=\big(y^{(1)},\ldots,y^{(m)}\big)^T\in\mathbb{R}^m. \tag{2.25}$$

We can then reformulate (2.24) as

$$\mathbf{X}^T\big(\mathbf{y}-\mathbf{X}\mathbf{w}'\big)=\mathbf{0}. \tag{2.26}$$

The entries of any weight vector $\mathbf{w}'=\big(w_0',w_1'\big)$ that satisfies (2.26) are solutions to (2.24).

[2] A necessary and sufficient condition for $\mathbf{w}'$ to minimize a convex differentiable function $f(\mathbf{w})$ is $\nabla f(\mathbf{w}')=\mathbf{0}$ [13, Sec. 4.2.3].

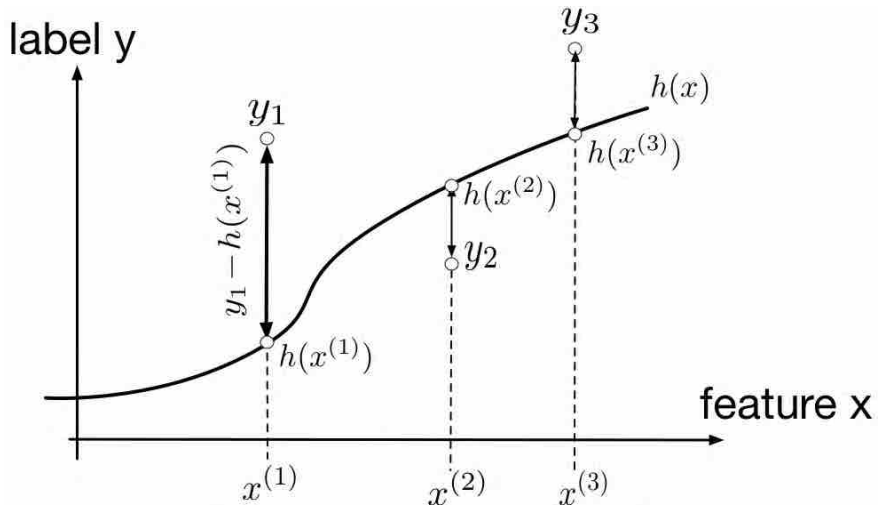


Figure 2.14: We can evaluate the quality of a particular predictor $h \in \mathcal{H}$ by measuring the prediction error $y - h(x)$ obtained for a labeled datapoint (x, y).

2.5 Exercises

2.5.1 Perfect Prediction

Consider data points (x, y) characterized by a single numeric feature $x \in \mathbb{R}$ and a numeric label $y \in \mathbb{R}$. We use a ML method to learn a hypothesis map $h(x)$ based on a training set consisting of three data points

$$(x^{(1)} = 1, y^{(1)} = 3), (x^{(2)} = 4, y^{(2)} = -1), (x^{(3)} = 1, y^{(3)} = 5).$$

Is there any chance for the ML method to learn a hypothesis map that perfectly fits the training data points such that $h\big(x^{(i)}\big) = y^{(i)}$ for $i = 1, \ldots, 3$. Hint: Try to visualize the data points in a scatter plot and various hypothesis maps (see Figure 1.3).

2.5.2 Temperature Data

Consider a dataset of daily air temperatures $x^{(1)}, \ldots, x^{(m)}$ measured at the observation station Utsjoki Nuorgam between 01.12.2019 and 29.02.2020. Thus, $x^{(1)}$ is the daily temperature measured on 01.12.2019, $x^{(2)}$ is the daily temperature measure don 02.12.2019, and $x^{(m)}$ is the daily temperature measured on 29.02.2020. You can download this dataset from the link `https://en.ilmatieteenlaitos.fi/download-observations`. ML methods often determine few parameters to characterize large collections of data points. Compute, for the

above temperature measurement dataset, the following parameters

- the **minimum** $A := \min_{i=1,\ldots,m} x^{(i)}$
- the **maximum** $B := \max_{i=1,\ldots,m} x^{(i)}$
- the average $C := (1/m) \sum_{i=1,\ldots,m} x^{(i)}$
- the standard deviation $D := (1/m) \sum_{i=1,\ldots,m} \left(x^{(i)} - C\right)^2$

2.5.3 Deep Learning on Raspberry PI

Consider the tiny desktop computer "RaspberryPI" equipped with a total of 8 Gigabytes memory (RAM) [23]. On that computer, we want implement a ML algorithm that learns a hypothesis map that is represented by a deep neural network involving $n = 10^6$ numeric weights (or parameters). Each hypothesis is represented by the specification of the values of all the weights. Let us assume that each weight is quantized using 8 bits (= 1 Byte). How many different hypotheses can we store on a RaspberryPI computer? (You can assume that 1Gigabyte = 10^9Bytes.)

2.5.4 How Many Features?

Consider the ML problem underlying a music information retrieval smartphone app [89]. 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?

2.5.5 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

$$\mathbf{h}(\mathbf{x}) = \mathbf{W}\mathbf{x} \text{ with some matrix } \mathbf{W} \in \mathbb{R}^{30\times 10}. \tag{2.27}$$

How many different linear predictors (2.27) are there ? 10, 30,40, infinite.

2.5.6 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 ?

2.5.7 Find Labeled Data for Given Empirical Risk

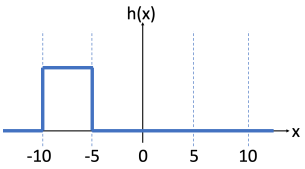
Consider linear hypothesis space consisting of linear maps $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ that are parameterized by the weight vector $\mathbf{w}$. We learn a good choice for the weight vector by minimizing the average squared error loss $f(\mathbf{w}) = \mathcal{E}\big(h^{(\mathbf{w})}|\mathcal{D}\big)$ incurred by $h^{(\mathbf{w})}(\mathbf{x})$ on the training set $\mathcal{D} = \big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)$. Is it possible to reconstruct the training data $\mathcal{D}$ just from knowing the function $f(\mathbf{w})$?. Is the resulting labeled training data unique or are there different training sets that could have resulted in the same empirical risk function? Hint: Write down the training error $f(\mathbf{w})$ in the form $f(\mathbf{w}) = \mathbf{w}^T\mathbf{Q}\mathbf{w} + c + \mathbf{b}^T\mathbf{w}$ with some matrix $\mathbf{Q}$, vector $\mathbf{b}$ and scalar c that might depend on the features and labels of the training datapoints.

2.5.8 Dummy Feature Instead of Intercept

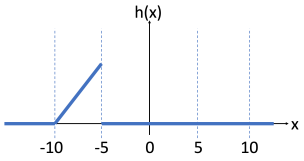
Show that any predictor of the form $h(x) = w_1 x + w_0$ can be emulated by combining a feature map $x \mapsto \mathbf{z}$ with a predictor of the form $\mathbf{w}^T\mathbf{z}$.

2.5.9 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)

2.5.10 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.

2.5.11 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 ?

2.5.12 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?

2.5.13 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 $(1,-1)$.

2.5.14 Classification Loss

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

2.5.15 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 the 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 $h(3)$?

2.5.16 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?

2.5.17 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 $\hat{y} = h(x)$?

2.5.18 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?

2.5.19 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

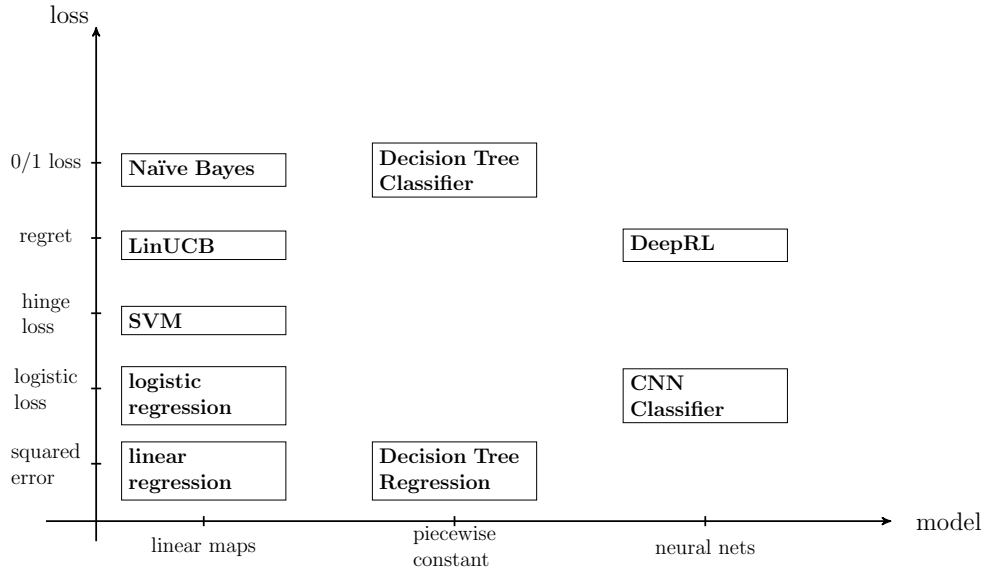


Figure 3.1: ML methods fit a model to data by minimizing a loss function. Different ML methods use different design choices for model, data and loss.

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

- data points characterized by **features** and **labels**
- a **model** or **hypothesis space** $\mathcal{H}$ of hypotheses $h \in \mathcal{H}$.
- a **loss function** to measure the quality of a particular hypothesis h.

Each of these three components involves design choices for the representation of data, their features and labels, the model and loss function. This chapter details the high-level design

choices used by some of the most popular ML methods. However, we do not discuss their detailed implementation in the form of algorithms (pseudo-code). Loosely speaking we only provide the high-level specifications of some widely used ML methods. These specifications amount to specific design choices for data representation, the model and the loss function.

To obtain a practical ML method we also need to combine the above components. The basic principle of any ML method is to search the model for a hypothesis that incurs minimum loss on any data point. Chapter 4 will then discuss a principled way to turn this informal statement into actual ML algorithms that could be implemented on a computer.

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 (2.7). Using labeled training data $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, linear regression learns a predictor $\hat{h}$ which minimizes the average squared error loss, or **mean squared error**, (see (2.7))

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

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 $\mathbf{w}$:

$$\begin{aligned} \widehat{\mathbf{w}} &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} (1/m)\sum_{i=1}^m (y^{(i)} - h^{(\mathbf{w})}(\mathbf{x}^{(i)}))^2 \\ &\overset{h^{(\mathbf{w})}(\mathbf{x})=\mathbf{w}^T\mathbf{x}}{=} \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} (1/m)\sum_{i=1}^m (y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})^2. \end{aligned} \tag{3.3}$$

The optimization problems (3.2) and (3.3) are equivalent in the following sense: Any optimal

weight vector $\widehat{\mathbf{w}}$ which solves (3.3), can be used to construct an optimal predictor $\hat{h}$, which solves (3.2), via $\hat{h}(\mathbf{x}) = h^{(\widehat{\mathbf{w}})}(\mathbf{x}) = \big(\widehat{\mathbf{w}}\big)^T \mathbf{x}$.

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.

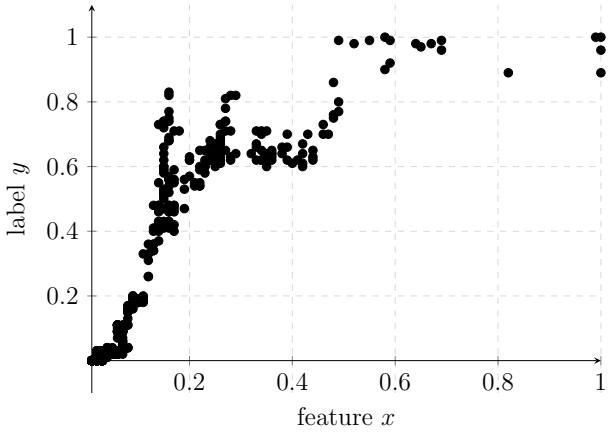


Figure 3.2: A scatterplot of some datapoints $(x^{(i)}, y^{(i)})$.

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 maps

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

We can approximate any non-linear relation $y = h(x)$ with any desired level of accuracy using a polynomial $\sum_{r=1}^{n} w_r x^{r-1}$ 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 (2.7). 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

[1] The precise formulation of this statement is known as the "Stone-Weierstrass Theorem" [71, Thm. 7.26].

error loss (mean squared error) (see (2.7)):

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

We can 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 $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}_{\rm poly}$ 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

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

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 $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}_{\rm poly}$ 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** [42]

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

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 (2.7) 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 a RV.

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 $\mathbb{R}^n$.

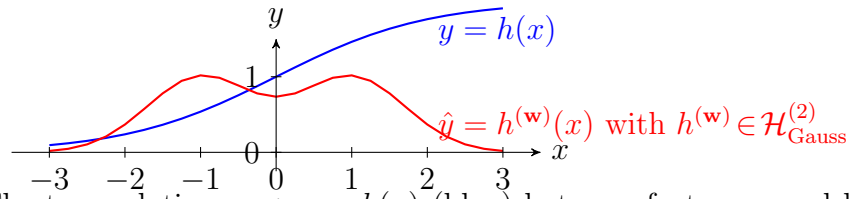


Figure 3.3: The true relation $x \mapsto y = h(x)$ (blue) between feature x and label y is highly non-linear. We might predict the label using a non-linear predictor $\hat{y} = h^{(\mathbf{w})}(x)$ with some weight vector $\mathbf{w} \in \mathbb{R}^2$ and $h^{(\mathbf{w})} \in \mathcal{H}^{(2)}_{\text{Gauss}}$.

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 $\hat{y}$.

[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 $\mathcal{Y} = \{0, 1\}$.

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.

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 2.9). 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 RVs with the probability

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

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 $p(y = \hat{y}; \mathbf{w})$.

Since $p(y=1)+p(y=-1)=1$,

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

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

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

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

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

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 [50, 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 [50, 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 the **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 2.9). 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 $p(\hat{y} \neq y)$ (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

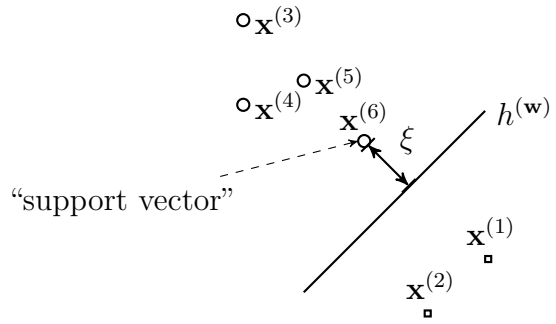


Figure 3.4: The SVM aims at a classifier $h^{(\mathbf{w})}$ with small hinge loss (2.10). Minimizing hinge loss of a classifier is the same as maximizing the margin ξ between the decision boundary (of the classifier) and each class of the training set.

methods. In contrast, the logistic loss (2.11) is convex and differentiable. We can therefore use gradient based methods to minimize the average logistic loss incurred on a training set (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 $P_{\rm err} = p(y \neq h(\mathbf{x}))$.

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 2.9).

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 us 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 $\mathcal{X}'$ [50]. 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 $\hat{\mathbf{x}} = \phi(\mathbf{x})$.

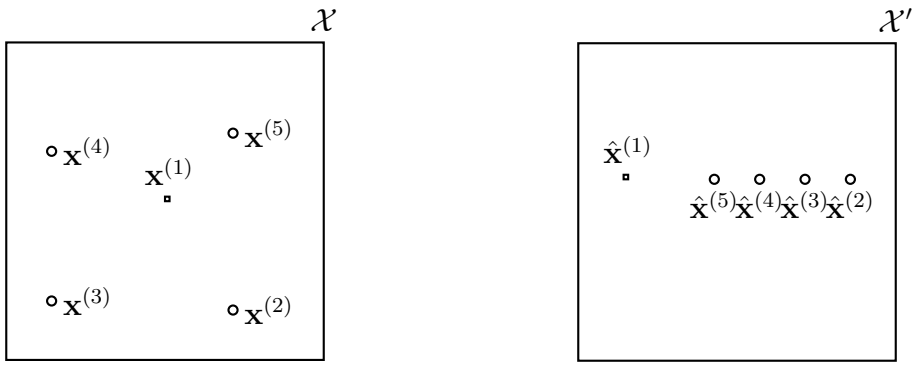


Figure 3.5: The data set $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{5}$ consists of 5 datapoints with features $\mathbf{x}^{(i)}$ and binary labels $y^{(i)}$. Left: In the original feature space $\mathcal{X}$, the datapoints cannot be separated perfectly by any linear classifier. Right: The 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 $h(\mathbf{x}) \in \mathcal{Y}$ [37]. 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 $\mathcal{Y} = \mathbb{R}$.

Figure 3.6 depicts an example for a decision tree. A decision tree consists of nodes which are connected by directed edges. We can 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 use 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 a particular decision tree we can use various loss functions. 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).

Decision tree methods use as a hypothesis space the set of all hypotheses which represented by some collection of decision trees. Figure 3.7 depicts a collection of decision trees which are characterized by having depth at most two. These methods search for a decision trees such that the corresponding hypothesis has minimum average loss on some labeled training data (see Section 4.4).

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 one of the elementary tests.

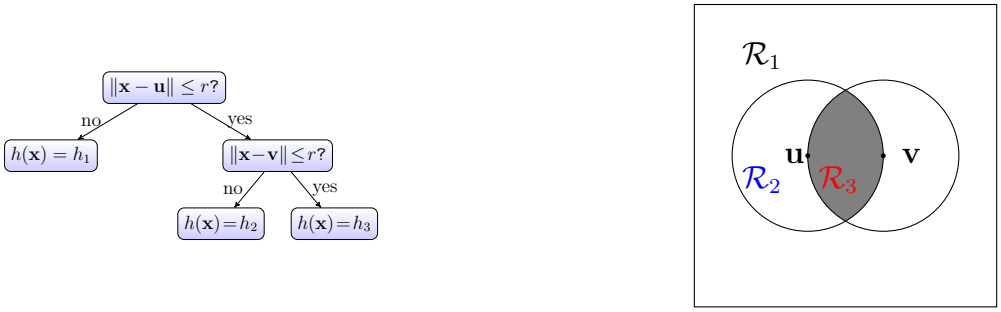


Figure 3.6: A decision tree represents a hypothesis h which is constant on subsets $\mathcal{R}_m$, i.e., $h(\mathbf{x}) = h_m$ for all $\mathbf{x} \in \mathcal{R}_m$. Each subset $\mathcal{R}_m \subseteq \mathcal{X}$ corresponds to a leaf node in the decision tree.

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

[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.

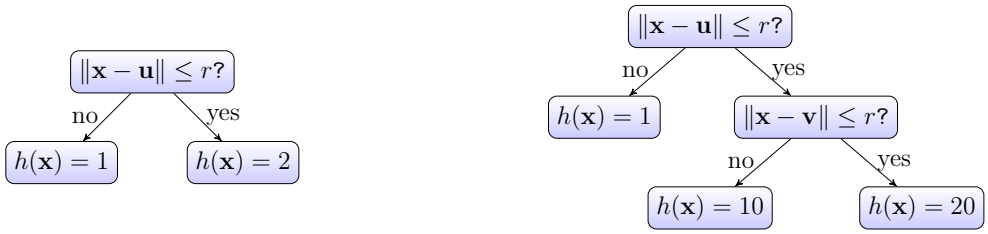


Figure 3.7: A hypothesis space $\mathcal{H}$ consisting of two decision trees with depth at most 2 and using the tests $\|\mathbf{x}-\mathbf{u}\| \leq r$ and $\|\mathbf{x}-\mathbf{v}\| \leq r$ with a fixed radius r and vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$.

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 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 2.9).

In contrast, the shape of a map represented by a decision tree can be much more complicated. Using a sufficiently large (deep) decision tree, we can obtain a hypothesis map that closely approximates any given non-linear map. Using sufficiently deep decision trees for classification problems allows for highly irregular decision regions.

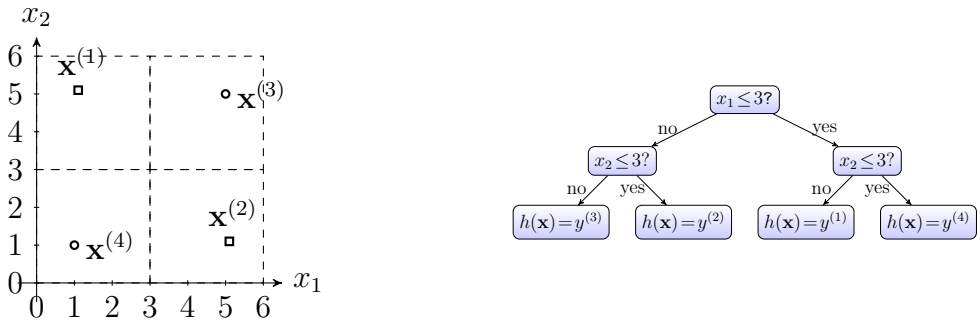


Figure 3.8: Using a sufficiently large (deep) decision tree, we can construct a map h that perfectly fits any given labeled dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ such that $h(\mathbf{x}^{(i)}) = y^{(i)}$ for $i = 1, \ldots, m$.

3.11 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

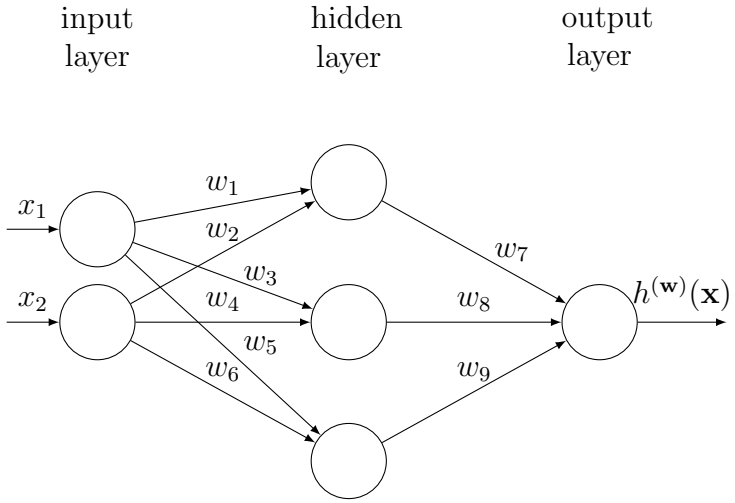


Figure 3.9: ANN representation of a predictor $h^{(\mathbf{w})}(\mathbf{x})$ which maps the input (feature) vector $\mathbf{x} = (x_1, x_2)^T$ to a predicted label (output) $h^{(\mathbf{w})}(\mathbf{x})$.

fed into the input units, each of which reads in one single feature $x_r \in \mathbb{R}$. The features x_r are then multiplied with the weights $w_{j,r}$ 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 $s_j = g(\sum_{r=1}^{n} w_{j,r} x_r)$ 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 $\sum_{r=1}^{n} w_{j,r} s_r$ of the outputs s_r 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 [33] 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 us 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 [33, 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 [21]. 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 [25].

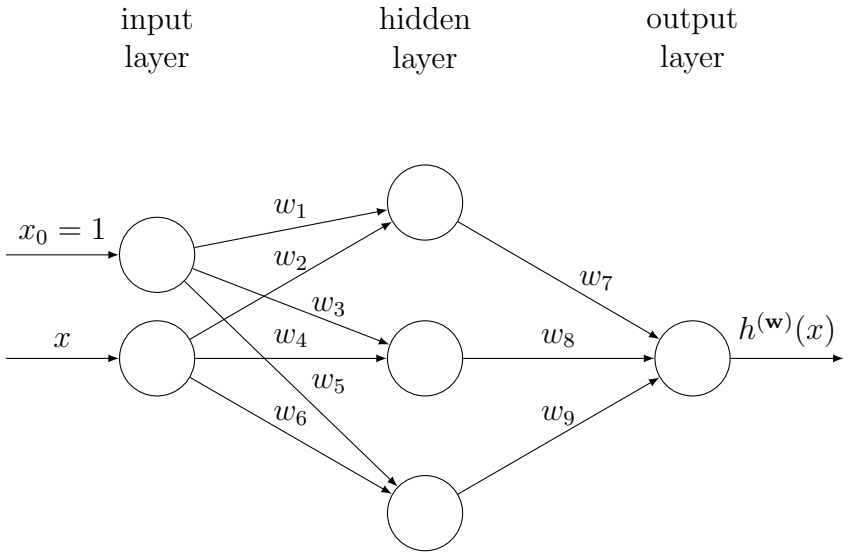


Figure 3.10: This ANN with one hidden layer defines a hypothesis space consisting of all maps $h^{(\mathbf{w})}(x)$ obtained by implementing the ANN with different weight vectors $\mathbf{w} = (w_1, \ldots, w_9)^T$.

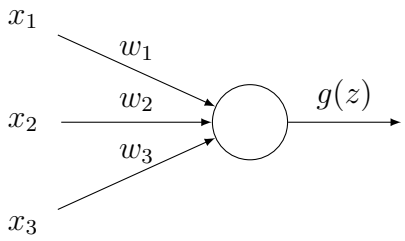


Figure 3.11: Each single neuron of the ANN depicted in Figure 3.10 implements a weighted summation $z = \sum_i w_i x_i$ of its inputs x_i followed by applying a non-linear activation function $g(z)$.

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 $g(z) = 10 \cdot z$?

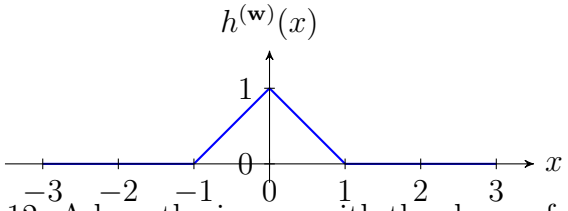


Figure 3.12: A hypothesis map with the shape of a triangle.

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 us to efficiently compute the gradient of the loss function via a (highly scalable) message passing procedure known as **back-propagation** [33].

3.12 Maximum Likelihood

For many applications it is useful to model the observed datapoints $\mathbf{z}^{(i)}$ as realizations of a RV $\mathbf{z}$ with probability distribution $p(\mathbf{z}; \mathbf{w})$ 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 $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)} \sim p(\mathbf{z}; \mathbf{w})$ is **maximum likelihood estimation** [52].

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 the 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 $p(\mathbf{z}; \mathbf{w})$ 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 $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)} \sim p(\mathbf{z}; \mathbf{w})$, 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 all choices for the mean vector $\boldsymbol{\mu} \in \mathbb{R}^n$ and the covariance matrix $\boldsymbol{\Sigma} \in \mathbb{S}^n_+$. Here, $\mathbb{S}^n_+$ denotes the set of all **positive semi-definite (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 [71]. 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 $h : \mathcal{X} \to \mathcal{Y}$ 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 $\mathcal{D}$.

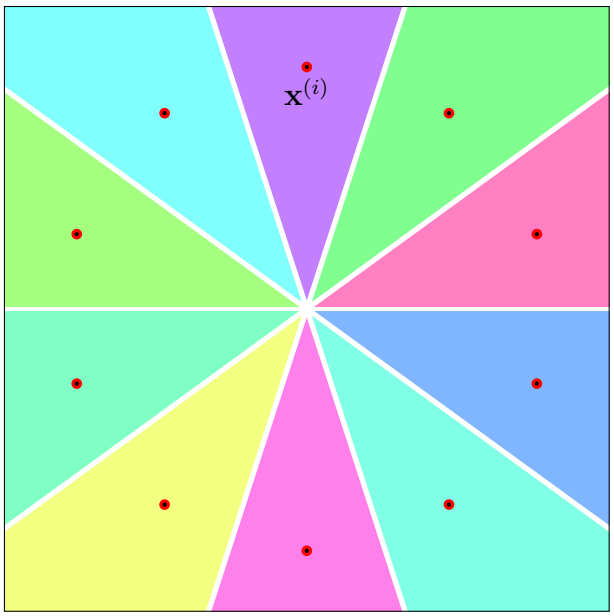


Figure 3.13: A hypothesis map h for k-NN with $k = 1$ and feature space $\mathcal{X} = \mathbb{R}^2$. The hypothesis map is constant over regions (indicated by the coloured areas) located around feature vectors $\mathbf{x}^{(i)}$ (indicated by a dot) of a dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}$.

3.14 Deep Reinforcement Learning

Deep reinforcement learning (DRL) refers to a subset of ML problems and methods that revolve around the control of dynamic systems such as autonomous driving cars or cleaning robots [75, 81, 63]. A DRL problem typically involves data points that represent the states of

a dynamic system at different time instant t. The data points representing the state at some time instant t is characterized by the feature vector $\mathbf{x}^{(t)}$. The Entries of this feature vector, i.e., the individual features of the state at time t, might be obtained via hardware sensors, onboard-cameras or positioning systems. The label $y^{(t)}$ of a data point might represent the optimal steering angle at time t.

DRL methods learn a hypothesis h that delivers optimal predictions $\hat{y}^{(t)} := h\big(\mathbf{x}^{(t)}\big)$ for the optimal steering angle $y^{(t)}$. As their name indicates, DRL methods use hypothesis spaces obtained from deep ANN (see Section 3.11). The quality of the prediction $\hat{y}^{(t)}$ obtained from a hypothesis is measured by the loss $\mathcal{L}(\big(\mathbf{x}^{(t)}, y^{(t)}\big), h) := -r^{(t)}$ with a reward signal $r^{(t)}$ that is typically obtained by some sensing device (such as collision avoidance sensor).

The reward signal $-r^{(t)}$ typically depends on the feature vector $\mathbf{x}^{(t)}$ and the discrepancy between optimal steering direction $y^{(t)}$ (which is unknown) and its prediction $\hat{y}^{(t)} := h\big(\mathbf{x}^{(t)}\big)$. However, what sets DRL methods apart from other ML methods such as linear regression (see Section 3.1) or logistic regression (see Section 3.6) is that they evaluate the loss function only point-wise $\mathcal{L}(\big(\mathbf{x}^{(t)}, y^{(t)}\big), h)$ for the specific hypothesis h that has been used to compute the prediction $\hat{y}^{(t)} := h\big(\mathbf{x}^{(t)}\big)$ at time instant t. This is fundamentally different from linear regression that uses the squared error loss (2.7) which can be evaluated for every possible hypothesis $h \in \mathcal{H}$.

3.15 LinUCB

ML methods are instrumental for various recommender systems [53]. A basic form of a recommender system amount to chose at some time instant t the most suitable item (product, song, movie) among a finite set of alternatives $a = 1, \ldots, A$. Each alternative is characterized by a feature vector $\mathbf{x}^{(t,a)}$ that varies between different time instants.

The data points arising in recommender systems typically represent different time instants t at which recommendations are computed. The data point at time t is characterized by a feature vector

$$\mathbf{x}^{(t)} = \big(\big(\mathbf{x}^{(t,1)}\big)^T, \ldots, \big(\mathbf{x}^{(t,A)}\big)^T\big)^T. \tag{3.25}$$

The feature vector $\mathbf{x}^{(t)}$ is obtained by stacking the feature vectors of alternatives at time t into a single long feature vector. The label of the data point t is a vector of rewards $\mathbf{y}^{(t)} := \big(r_1^{(t)}, \ldots, r_A^{(t)}\big)^T \in \mathbb{R}^A$. The entry $r_a^{(t)}$ represents the reward obtained by choosing (recommending) alternative a (with features $\mathbf{x}^{(t,a)}$) at time t. We might interpret the reward $r^{(t,a)}$ as an indicator if the costumer actually buys the product corresponding to

the recommended alternative a.

The ML method LinUCB (the name seems to be inspired by the terms "linear" and "upper confidence bound" (UCB)) aims at learning a hypothesis h that allows to predict the rewards $\mathbf{y}^{(i)}$ based on the feature vector $\mathbf{x}^{(t)}$ (3.25). As its hypothesis space $\mathcal{H}$, LinUCB uses the space of linear maps from the stacked feature vectors $\mathbb{R}^{nA}$ to the space of reward vectors $\mathbb{R}^{A}$. This hypothesis space can be parametrized by matrices $\mathbf{W} \in \mathbb{R}^{A \times nA}$. Thus, LinUCB learns a hypothesis that computes predicted rewards via

$$\widehat{\mathbf{y}}^{(t)} := \mathbf{W}\mathbf{x}^{(t)}. \tag{3.26}$$

The entries of $\widehat{\mathbf{y}}^{(t)} = \big(\hat{r}_1^{(t)}, \ldots, \hat{r}_A^{(t)}\big)$ are predictions of the individual rewards $r^{(t,a)}$. It seems natural to recommend at time t the alternative a whose predicted reward is maximum. However, it turns out that this approach is sub-optimal as it prevents the recommender system from learning the optimal predictor map $\mathbf{W}$.

Loosely speaking, LinUCB tries out (explores) each alternative $a \in \{1, \ldots, A\}$ sufficiently often to obtain a sufficient amount of training data for learning a good weight matrix $\mathbf{W}$. Therefore, at time t, the LInUCB method chooses the alternative $a^{(t)}$ that maximizes the quantity

$$\hat{r}_a^{(t)} + R(t,a) \ , \ a = 1, \ldots, A. \tag{3.27}$$

The quantity $R(t,a)$, which depends on the feature vectors $\mathbf{x}^{(t',a)}$ of the alternative a at previous time instants $t' < t$, is chosen such that (3.27) provides an upper bound on the actual reward $r_a^{(t)}$ with a prescribed level of confidence. Thus, at each time instant t, LinUCB chooses the alternative that results in the largest upper confidence bound (UCB) (3.27) on the reward (hence the "UCB" in LinUCB). For more details on the LinUCB method and its analysis, we refer to [53].

3.16 Exercises

3.16.1 Logistic Loss and Accuracy

Section 3.6 discussed logistic regression as a ML method that learns a linear hypothesis map by minimizing the logistic loss (3.15). The logistic loss has computationally pleasant properties as it is smooth and convex. However, in some applications we might be ultimately interested in the accuracy or (equivalently) the average 0/1 loss (2.8). Can we upper bound

the average 0/1 loss using the average logistic loss incurred by a given hypothesis on a given training set.

3.16.2 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?

3.16.3 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?

3.16.4 Data Dependent Hypothesis Space

Consider a ML application involving data points that are characterized by feature vectors $\mathbf{x} \in \mathbb{R}^6$ and a numeric label y. We learn a hypothesis by minimizing the average loss incurred on a training set $\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}$. Which of the following ML methods uses a hypothesis space that depends on the dataset $\mathcal{D}$?

- logistic regression
- linear regression
- k-NN

Chapter 4

Empirical Risk Minimization

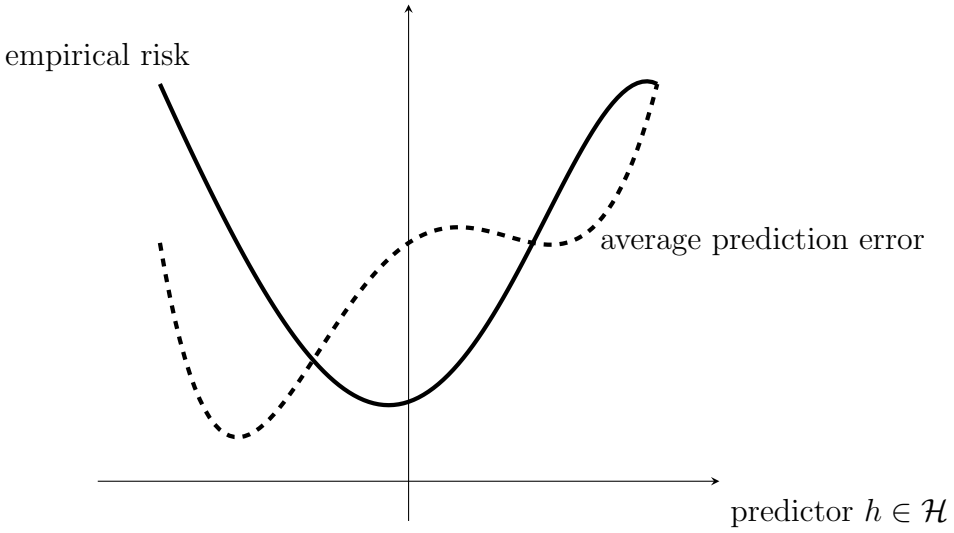


Figure 4.1: ML methods learn a hypothesis $h \in \mathcal{H}$ that incur small loss when predicting the label y of datapoint based on its features $\mathbf{x}$. Empirical risk minimization approximates the expected loss or risk with the empirical risk (solid curve) incurred on a finite set of labeled 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. RVs with the common 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 us 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 $p(\mathbf{x}, y)$ 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) defines the risk of a hypothesis and motivates **empirical risk minimization (ERM)** by approximating the risk using an (empirical) average over 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.

Section 4.3 discusses ERM for linear regression (see Section 3.1). Here, 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 [43].

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 The Basic Idea of Empirical Risk Minimization

Consider some ML application that generates datapoints, each of which is characterized by a feature vector $\mathbf{x}$ and a label y. It is often useful to interpret datapoints as realizations of i.i.d. RVs with common 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 $\mathbf{x}$. However, in practice we do not know the probability distribution $p(\mathbf{x}, y)$ and therefore cannot evaluate the expectation in (4.1).[1] **ERM** replaces the expectation in (4.1)

[1] One exception to this rule is if the datapoints are synthetically generated by drawing realizations from a given probability distribution $p(\mathbf{x}, y)$.

with an average over a given set of labeled datapoints,

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

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,

$$\widehat{\mathbf{w}} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} f(\mathbf{w}) \text{ with } f(\mathbf{w}) := (1/m) \sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h^{(\mathbf{w})}). \tag{4.3}$$

The objective function $f(\mathbf{w})$ in (4.3) is the empirical risk $\mathcal{E}\big(h^{(\mathbf{w})}|\mathcal{D}\big)$ incurred by the hypothesis $h^{(\mathbf{w})}$ 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 $\widehat{\mathbf{w}}$ solving (4.3), the predictor $h^{(\widehat{\mathbf{w}})}$ 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 gradient based 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 [67].

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 [33]. Even more challenging is the ERM obtained for decision trees or Bayes' classifiers. These ML problems involve non-differentiable and non-convex objective functions.

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 (2.7). For linear regression, the ERM problem (4.3)

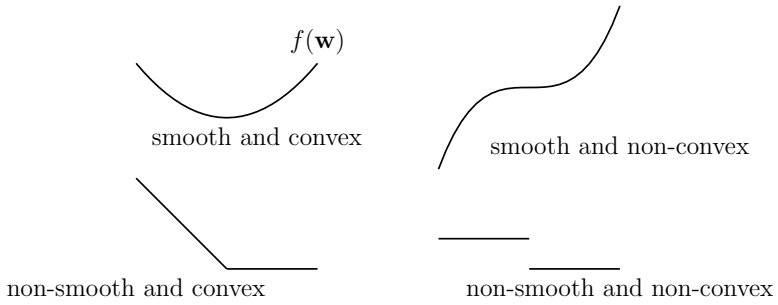


Figure 4.2: Different types of objective functions obtained for ERM in different settings.

becomes

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

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.

We can rewrite the ERM problem (4.4) more concisely 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 us 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), resulting in the following form of the ERM for linear regression:

$$\widehat{\mathbf{w}} = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} (1/m)\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2. \tag{4.7}$$

The formulation (4.7) allows for an interesting geometric interpretation of linear regression. Solving (4.7) amounts to finding a vector $\mathbf{X}\mathbf{w}$, with fixed feature matrix $\mathbf{X}$ (see (4.5)), that

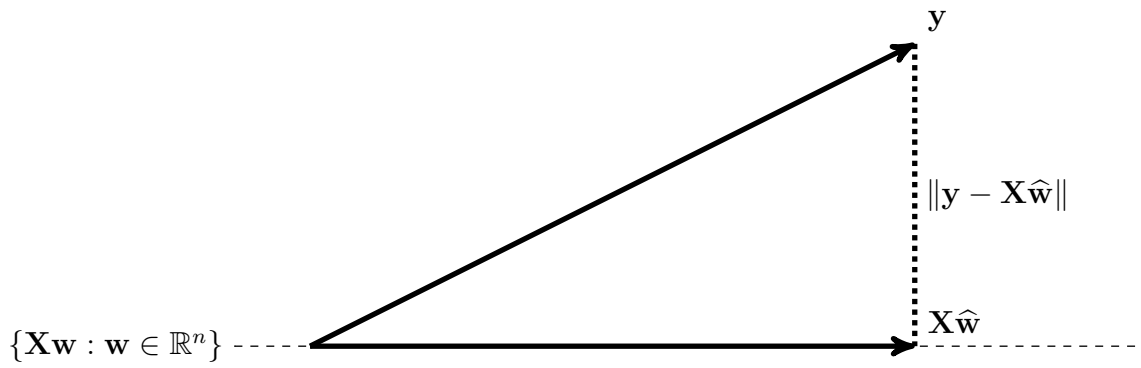


Figure 4.3: The ERM (4.7) for linear regression amounts to an orthogonal projection of the label vector $\mathbf{y} = \big(y^{(1)}, \ldots, y^{(m)}\big)^T$ on the subspace spanned by the columns of the feature matrix $\mathbf{X} = \big(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\big)^T$.

is closest (in the Euclidean norm) to the given label vector $\mathbf{y} \in \mathbb{R}^m$ (see (4.5)). The solution to this approximation problem is precisely the orthogonal projection of the vector $\mathbf{y}$ onto the subspace of $\mathbf{R}^m$ that is spanned by the columns of the feature matrix $\mathbf{X}$.

To solve the optimization problem (4.7), it is convenient to rewrite it as the quadratic problem

$$\min_{\mathbf{w} \in \mathbb{R}^n} \underbrace{(1/2)\mathbf{w}^T\mathbf{Q}\mathbf{w} - \mathbf{q}^T\mathbf{w}}_{=f(\mathbf{w})}$$
$$\text{with } \mathbf{Q} = (1/m)\mathbf{X}^T\mathbf{X}, \mathbf{q} = (1/m)\mathbf{X}^T\mathbf{y}. \tag{4.8}$$

Since $f(\mathbf{w})$ is a differentiable and convex function, a necessary and sufficient condition for $\widehat{\mathbf{w}}$ to be a minimizer $f(\widehat{\mathbf{w}}) = \min_{\mathbf{w} \in \mathbb{R}^n} f(\mathbf{w})$ is the **zero-gradient condition** [13, Sec. 4.2.3]

$$\nabla f(\widehat{\mathbf{w}}) = \mathbf{0}. \tag{4.9}$$

Combining (4.8) with (4.9), yields the following necessary and sufficient condition for a weight vector $\widehat{\mathbf{w}}$ to solve the ERM (4.4),

$$(1/m)\mathbf{X}^T\mathbf{X}\widehat{\mathbf{w}} = (1/m)\mathbf{X}^T\mathbf{y}. \tag{4.10}$$

This condition can be rewritten as

$$(1/m)\mathbf{X}^T\big(\mathbf{y} - \mathbf{X}\widehat{\mathbf{w}}\big) = \mathbf{0}. \tag{4.11}$$

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

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

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 $\widehat{\mathbf{w}}$ which solves (4.10). 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 $\widehat{\mathbf{w}}$ which solves (4.10) achieves the same minimum empirical risk

$$\mathcal{E}(h^{(\widehat{\mathbf{w}})} \mid \mathcal{D}) = \min_{\mathbf{w} \in \mathbb{R}^n} \mathcal{E}(h^{(\mathbf{w})} \mid \mathcal{D}) = \|(\mathbf{I} - \mathbf{P})\mathbf{y}\|^2. \tag{4.12}$$

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)).[2]

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 (4.10) is then unique and given by

$$\widehat{\mathbf{w}} = \big(\mathbf{X}^T\mathbf{X}\big)^{-1}\mathbf{X}^T\mathbf{y}. \tag{4.13}$$

The closed-form solution (4.13) 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 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 $\widehat{\mathbf{w}}$ which does not require any matrix inversion. This method, referred to as **gradient descent** (GD), constructs a sequence $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \ldots$ of increasingly accurate approximations of $\widehat{\mathbf{w}}$. This iterative method has two major benefits compared to evaluating the formula (4.13) using

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

direct matrix inversion, such as Gauss-Jordan elimination [32]. 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 (4.13) 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 [44, 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 4.4 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 4.4. Then we try out growing the tree by replacing a leaf node with a decision node. According to Figure 4.4, replacing the right leaf node results in a decision tree which is able to perfectly represent the training dataset (it achieves zero empirical risk).

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

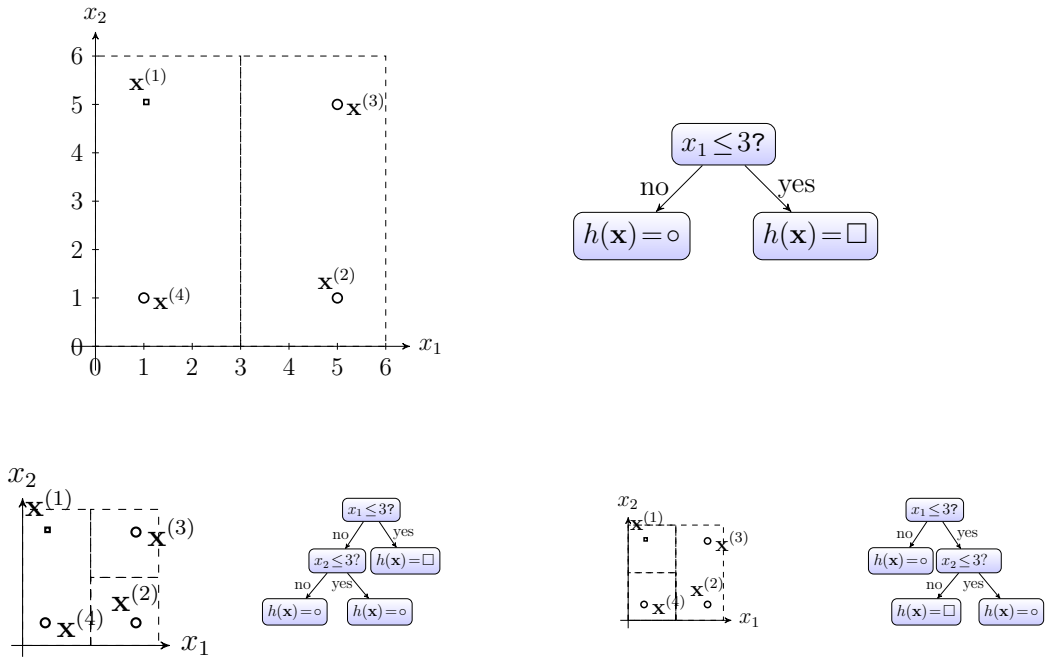


Figure 4.4: Consider a given labeled dataset and the decision tree in the top row. We then grow the decision tree by expanding one of its two leaf nodes. The bottom row shows the resulting decision trees, along with their decision boundaries. Each decision tree in the bottom row is obtained by expanding a different leaf node of the decision tree in the top row.

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.

ML methods that allow for very deep decision trees, which represent highly complicated maps, tend to overfit the training set (see Figure 3.8 and Chapter 7). Even if a deep decision tree incurs small average loss on the training set, it might incur large loss when predicting the labels of datapoints outside the training set.

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

$$\begin{aligned} \hat{h} &= \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) \\ &\overset{(2.8)}{=} \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} \mathcal{I}(h(\mathbf{x}^{(i)}) \neq y^{(i)}). \end{aligned} \tag{4.14}$$

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 (4.14).

We will now approach the ERM (4.14) via a different route by interpreting the datapoints $(\mathbf{x}^{(i)}, y^{(i)})$ as realizations of i.i.d. RVs with the common probability distribution $p(\mathbf{x}, y)$.

As discussed in Section 2.3, the empirical risk obtained using 0/1 loss approximates the error probability $p(\hat{y} \neq y)$ 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 (4.14) as

$$\hat{h} \overset{(2.9)}{\approx} \operatorname*{argmin}_{h \in \mathcal{H}} p(\hat{y} \neq y). \tag{4.15}$$

Note that the hypothesis h, which is the optimization variable in (4.15), enters into the objective function of (4.15) 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 $p(\hat{y} \neq y)$, the solution of (4.15) can be found easily via elementary Bayesian decision theory [69]. 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. RVs 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 $\mathbf{\Sigma}$, i.e.,

$$p(\mathbf{x}|y) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_y, \mathbf{\Sigma}).^{3} \tag{4.16}$$

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 $\mathbf{\Sigma} = \mathbb{E}\{(\mathbf{x} - \boldsymbol{\mu}_y)(\mathbf{x} - \boldsymbol{\mu}_y)^T|y\}$ 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 $p(\hat{y} \neq y)$ is $\hat{y} = 1$ for $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ for $h(\mathbf{x}) \leq 0$ using the classifier map

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with } \mathbf{w} = \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1}). \tag{4.17}$$

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

We cannot implement the classifier (4.17) directly, since we do not know the true values of the class-specific mean vectors $\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_{-1}$ and covariance matrix $\mathbf{\Sigma}$. Therefore, we have to replace those unknown parameters with some estimates $\hat{\boldsymbol{\mu}}_1$, $\hat{\boldsymbol{\mu}}_{-1}$ and $\widehat{\mathbf{\Sigma}}$. A principled

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

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

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

approach is to use the maximum likelihood estimates (see (3.24))

$$\begin{aligned}\hat{\boldsymbol{\mu}}_1 &= (1/m_1)\sum_{i=1}^{m}\mathcal{I}(y^{(i)}=1)\mathbf{x}^{(i)},\\ \hat{\boldsymbol{\mu}}_{-1} &= (1/m_{-1})\sum_{i=1}^{m}\mathcal{I}(y^{(i)}=-1)\mathbf{x}^{(i)},\\ \hat{\boldsymbol{\mu}} &= (1/m)\sum_{i=1}^{m}\mathbf{x}^{(i)},\\ \text{and } \widehat{\boldsymbol{\Sigma}} &= (1/m)\sum_{i=1}^{m}(\mathbf{z}^{(i)}-\hat{\boldsymbol{\mu}})(\mathbf{z}^{(i)}-\hat{\boldsymbol{\mu}})^T, \end{aligned} \tag{4.18}$$

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 (4.18) into (4.17) yields the implementable classifier

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \text{ with } \mathbf{w} = \widehat{\boldsymbol{\Sigma}}^{-1}(\hat{\boldsymbol{\mu}}_1 - \hat{\boldsymbol{\mu}}_{-1}). \tag{4.19}$$

We highlight that the classifier (4.19) is only well-defined if the estimated covariance matrix $\widehat{\boldsymbol{\Sigma}}$ (4.18) is invertible. This requires to use a sufficiently large number of training datapoints such that $m \geq n$.

We derived the classifier (4.19) as an approximate solution to the ERM (4.14). The classifier (4.19) partitions the feature space $\mathbb{R}^n$ into two half-spaces. One half-space consists of feature vectors $\mathbf{x}$ for which the hypothesis (4.19) is non-negative and, in turn, $\hat{y}=1$. The other half-space is constituted by feature vectors $\mathbf{x}$ for which the hypothesis (4.19) is negative and, in turn, $\hat{y}=-1$. Figure 2.9 illustrates these two half-spaces and the decision boundary between them.

The Bayes' classifier (4.19) 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 2.9). 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 $\widehat{\boldsymbol{\Sigma}}$ (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 $\widehat{\boldsymbol{\Sigma}}$ (4.18) is not invertible whenever $m < n$. In this case,

the expression (4.19) becomes useless. To cope with small sample size $m < n$ we can simplify the model (4.16) by requiring the covariance to be diagonal $\mathbf{\Sigma} = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_n^2)$. 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 (4.19) is obtained using the generative model (4.16) 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 (4.16) allows us to synthetically generate more labeled data by generating random features and labels according to the probability distribution (4.16). We refer to [64] 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 hypothesis is characterized by (2.26).

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 $\mathbf{y}^{(m)}$:

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

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

$$m = 3: \quad \mathbf{X}^{(3)} = \big(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}\big)^T, \quad \mathbf{y}^{(3} = \big(y^{(1)}, y^{(2)}, y^{(3)}\big)^T. \tag{4.22}$$

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 (2.26) for each time step m. However, this approach does not reuse computations already invested in solving (2.26) 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 [31]? 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 standard normal RV 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 normal 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 [33]. 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))

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

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 $\mathbf{w}_0$) [71]

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

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.

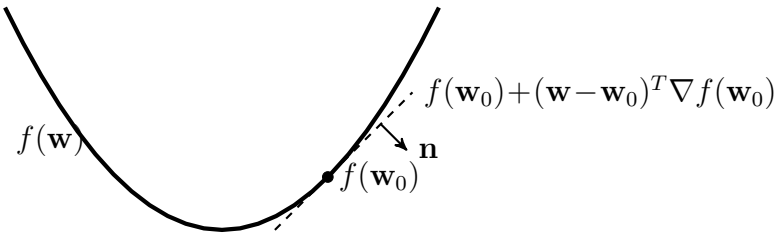


Figure 5.1: A smooth function $f(\mathbf{w})$ can be approximated locally around a point $\mathbf{w}_0$ using a hyperplane whose normal vector $\mathbf{n} = (\nabla f(\mathbf{w}_0), -1)$ is determined by the gradient $\nabla f(\mathbf{w}_0)$.

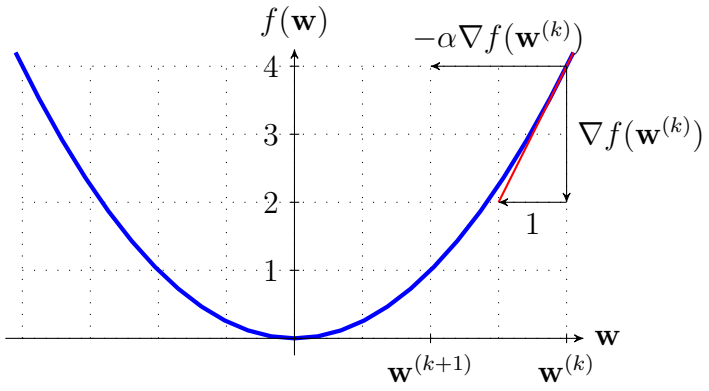


Figure 5.2: The GD step (5.4) amounts to a shift by $-\alpha\nabla f(\mathbf{w}^{(k)})$.

5.1 The Basic GD Step

We now discuss a very simple, yet quite powerful, algorithm for finding the weight vector $\widehat{\mathbf{w}}$ 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 $\widehat{\mathbf{w}}$ 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., $\lim_{k\to\infty} f(\mathbf{w}^{(k)}) = f\big(\widehat{\mathbf{w}}\big)$ (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** [33].

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

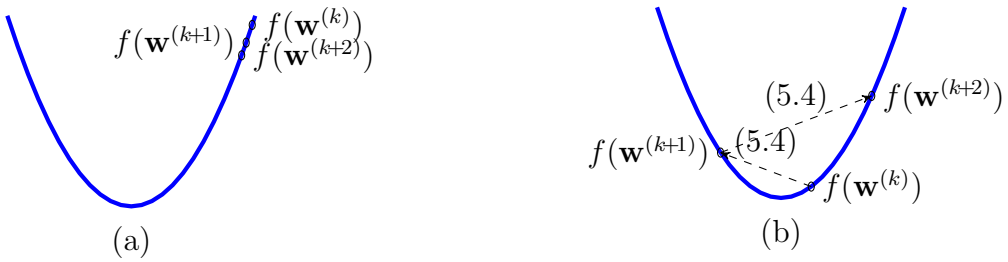


Figure 5.3: Effect of choosing learning rate α in GD step (5.4) too small (a) or too large (b). If the steps size α in the GD step (5.4) is chosen too small, the iterations make very little progress towards the optimum or even fail to reach the optimum at all. If the learning rate α is chosen too large,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

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 $\mathbf{w}^{(\mathrm{opt})}$ (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 $\widehat{\mathbf{w}}$. 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 $\widehat{\mathbf{w}}$ 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 [33, 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 $\widehat{\mathbf{w}}$ for any step size α satisfying [61]

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

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

Time-Data Tradeoffs The number of iteration required by GD to achieve a given sub-optimality depends on the condition number of $\mathbf{X}^T\mathbf{X}$. What can we say about the condition number? In general, we have not control over this quantity as the matrix $\mathbf{X}$ consists of the feature vectors of arbitrary datapoints. However, it is often useful to model the feature vectors as realizations of i.i.d. random vectors. We can then study the probability of a very small condition number.

??? Can we obtain Time-Data Tradeoffs already for simple linear regression with gradient descent see [65]. ???

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 (2.7)

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

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 $\widehat{\mathbf{w}}$ for (5.6) is characterized as

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

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 $\widehat{\mathbf{w}}$. 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: $k := k + 1$ (increase iteration counter)

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

4: **until** stopping criterion met

Output: $\mathbf{w}^{(k)}$ (which approximates $\widehat{\mathbf{w}}$ 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 $\widehat{\mathbf{w}}$ 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 $U \geq$

[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 (4.10), which we want to avoid by using the "cheaper" GD algorithm.

$\lambda_{\max}\big((1/m)\mathbf{X}^T\mathbf{X}\big)$, 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 $h^{(\widehat{\mathbf{w}})}$ 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 smooth optimization problem (5.2),

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

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 $r := 0$

1: **repeat**

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

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

4: **until** stopping criterion met

Output: $\mathbf{w}^{(r)}$, which approximates a solution $\widehat{\mathbf{w}}$ 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 $\widehat{\mathbf{w}}$ 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 $\mathbf{x}^{(i)}$. The feature vector $\mathbf{x}^{(i)}$ 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 $h^{(\mathbf{w}^{(k-1)})}(\mathbf{x}^{(i)}) = \left(\mathbf{w}^{(k-1)}\right)^T\mathbf{x}^{(i)}$ 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 [37, 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 $\mathbf{X} = \left(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\right)^T \in \mathbb{R}^{m \times n}$ (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) of the objective function arising in linear regression, the Hessian (5.17) of logistic regression 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 *datapoin* 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 $\kappa(\mathbf{X}^T\mathbf{X})$ [45]. 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 $\bar{\mathbf{x}} = (1/m)\sum_{i=1}^m \mathbf{x}^{(i)}$ 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 $\big(\widehat{\mathbf{X}}\big)^T\widehat{\mathbf{X}}$ 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 [48]. 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)) by 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

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

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 r, $\alpha = \alpha_r$. The sequence α_r of step-sizes is referred to as a **learning rate schedule** [33, Chapter 8]. One popular choice for the learning rate schedule is $\alpha = 1/r$ [60]. 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, $f[r'] \geq f[r]$ for any $r' \geq r$ and $f \in \mathcal{P}$
- a change point r, where $f[r] \neq f[r+1]$ can only be at integer multiples of 100, e.g., $r = 100$ or $r = 300$.

Given some unknown function $f \in \mathcal{P}$ and starting point r_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 r to obtain the values $f[r], f[r-1]$ and $f[r+1]$.

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 $r = 1, \ldots,$, we collect a new datapoint $\big(\mathbf{x}^{(r)}, y^{(r)}\big)$. 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 noise $\varepsilon \sim \mathcal{N}(0, 1)$ following a standard normal distribution. We use SGD to learn the weight vector $\mathbf{w}$ of a linear hypothesis,

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

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

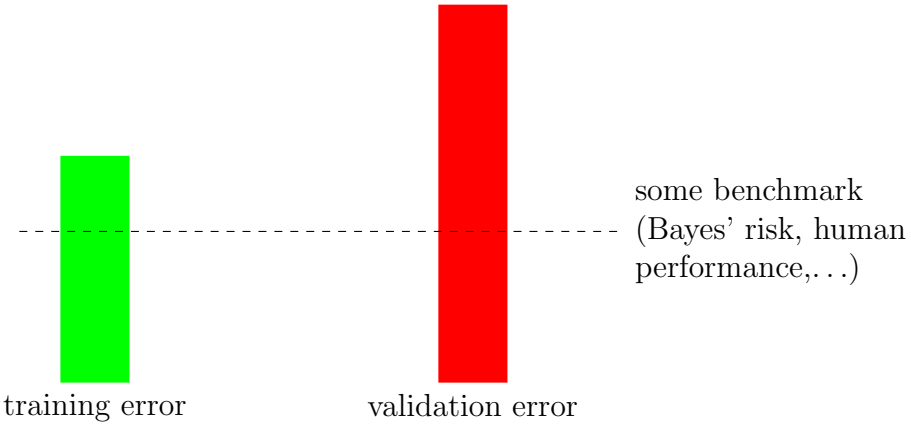


Figure 6.1: To diagnose ML methods we compare the training with validation error. Ideally both errors are on the same level as a relevant benchmark.

Chapter 4 discussed ERM as a principled approach to learning a good hypothesis out of a hypothesis space or model. ERM-based methods learn a hypothesis $\hat{h} \in \mathcal{H}$ that incurs minimum average loss on some labeled datapoints that serve as the **training set**.[1] We refer to the average loss incurred by a hypothesis on the training set as the **training error**. The minimum average loss achieved by a hypothesis that solves the ERM might be referred to as the training error of the overall ML method.

ERM makes sense only if the training error of a hypothesis is a good indicator for its loss incurred on datapoints outside the training set. Whether the training error of a hypothesis

[1] In statistics, the training set is also referred to as a **sample**.

is a reliable indicator for its performance outside the training set depends on the statistical properties of the datapoints and on the hypothesis space used by the ML method.

ML methods often use hypothesis spaces with a large effective dimension (see Section 2.2). As an example consider linear regression (see Section 3.1) with datapoints having a vast number n of features. The effective dimension of the linear hypothesis space (3.1), which is used by linear regression, is equal to the number n of features. Modern technology allows to collect a huge number of features about individual datapoints which implies, in turn, that the effective dimension of (3.1) is huge. Another example of high-dimensional hypothesis spaces are deep learning methods whose hypothesis spaces are constituted by all maps represented by some ANN with **billions of tunable weights**.

A high-dimensional hypothesis space is typically very likely to contain a hypothesis that fits perfectly any given training set. Such a hypothesis achieves a very small training error but might incur a large loss when predicting the labels of datapoints outside the training data. The (minimum) training error achieved by a hypothesis learnt by ERM can be highly misleading. We say that a ML method, such as linear regression using too many features, overfits the training data when it learns a hypothesis (e.g., via ERM) that has small training error but incurs much larger loss outside the training set.

Section 6.1 shows why linear regression will most likely overfit as soon as the number of features of a datapoint exceeds the size of the training set. Section 6.2 demonstrates 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. Thus, we can detect if a ML method overfits by comparing its training and validation errors (see Figure 6.1).

We can use the validation error not only to detect if a ML method overfits. The validation error can also be used as a quality measure for an entire hypothesis space or model. This is analogous to the concept of a loss function that allows us to evaluate the quality of a hypothesis $h \in \mathcal{H}$. Section 6.3 shows how to do model selection based on comparing the validation errors obtained for different candidate models (hypothesis spaces).

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. The analysis of the probabilistic model reveals the interplay between the data, the hypothesis space and the resulting training and validation error of a ML method.

Section 6.5 presents the **bootstrap** method as a simulation-based alternative to the

analysis of Section 6.4. While Section 6.4 assumes a particular probability distribution of datapoints, the bootstrap method does not require the specification of a probability distribution that underlies the data. The bootstrap method allows us to analyze statistical fluctuations in the learning process that arise from using different training sets.

As indicated in Figure 6.1, for some ML applications, we might have a benchmark level for the error of ML methods. Such a benchmark might be obtained from existing ML methods, human performance levels or from a probabilistic model (see Section 6.4). Section 6.6 details how the comparison between training and validation error with some benchmark error level informs possible improvements of the ML method. These improvements might be obtained by collecting more datapoints, using more features of datapoints or by changing the model (hypothesis space). Having a benchmark level also allows us to tell if a ML method already provides satisfactory results. If the training and validation error of a ML method are on the same level as the error of the theoretically optimal Bayes' estimator, there is little point in modifying the ML method as it already performs (nearly) optimal.

6.1 Overfitting

We now have a closer look at the occurrence of overfitting in linear regression which is one of the ML method discussed in Section 3.1. Linear regression methods learn a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ which is parametrized by the weight vector $\mathbf{w} \in \mathbb{R}^n$. The learnt hypothesis is then used to predict the numeric label $y \in \mathbb{R}$ of a datapoint based on its feature vector $\mathbf{x} \in \mathbb{R}^n$.

Linear regression aims at finding a weight vector $\widehat{\mathbf{w}}$ with minimum average squared error loss incurred on a training set

$$\mathcal{D} = \big\{ \big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big) \big\}.$$

The training set consists of m datapoints $\big(\mathbf{x}^{(i)}, y^{(i)}\big)$, for $i = 1, \ldots, m$, with known label values $y^{(i)}$. We stack the feature vectors $\mathbf{x}^{(i)}$ and labels $y^{(i)}$ of the training data into the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})^T$ and label vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(m)})^T$.

The ERM (4.12) of linear regression is solved by any weight vector $\widehat{\mathbf{w}}$ that solves (4.10).

The (minimum) training error of the hypothesis $h^{(\widehat{\mathbf{w}})}$ is obtained as

$$\mathcal{E}(h^{(\widehat{\mathbf{w}})} \mid \mathcal{D}) \stackrel{(4.3)}{=} \min_{\mathbf{w} \in \mathbb{R}^n} \mathcal{E}(h^{(\mathbf{w})}|\mathcal{D}) \stackrel{(4.12)}{=} \|(\mathbf{I}-\mathbf{P})\mathbf{y}\|^2. \tag{6.1}$$

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

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

of the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})^T \in \mathbb{R}^{m \times n}$.

In many ML applications, we have access to a huge number of individual datapoints. If a datapoint represents a snapshot obtained from a smartphone camera, we can use millions of pixel colour intensities as its features. Therefore, it is common to have more features for datapoints than the size of the training set,

$$n \geq m. \tag{6.2}$$

Whenever (6.2) holds, the feature vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)} \in \mathbb{R}^n$ of the datapoints in $\mathcal{D}$ are typically linearly independent. As a case in point, 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, these vectors are linearly independent with probability one [59].

If the feature vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)} \in \mathbb{R}^n$ are linearly independent, the span of the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})^T$ coincides with $\mathbb{R}^m$ which implies, in turn, $\mathbf{P} = \mathbf{I}$. Inserting $\mathbf{P} = \mathbf{I}$ into (4.12) yields

$$\mathcal{E}(h^{(\widehat{\mathbf{w}})} \mid \mathcal{D}) = 0. \tag{6.3}$$

As soon as the number $m = |\mathcal{D}|$ of training datapoints does not exceed the number n of features that characterize datapoints, there is (with probability one) a linear predictor $h^{(\widehat{\mathbf{w}})}$ achieving **zero training error**.

While the hypothesis $h^{(\widehat{\mathbf{w}})}$ achieves zero training error, it will typically incur a non-zero average prediction error $y - h^{(\widehat{\mathbf{w}})}(\mathbf{x})$ on datapoints $(\mathbf{x}, y)$ outside the training set (see Figure 6.2). Section 6.4 will make this statement more precise by using a probabilistic model for the datapoints within and outside the training set.

Note that (6.3) also applies if the features $\mathbf{x}$ and labels y of datapoints are completely unrelated. Consider a ML problem with datapoints whose labels y and features are realizations

of a RV that are statistically independent. Thus, in a very strong sense, the features $\mathbf{x}$ contain no information about the label of a datapoint. Nevertheless, as soon as the number of features exceeds the size of the training set, such that (6.2) holds, linear regression will learn a hypothesis with zero training error.

We can easily extend the above discussion about the occurrence of overfitting in linear regression to other methods that combine linear regression with a feature map. Polynomial regression, using datapoints with a single feature z, combines linear regression with the feature map $z \mapsto \mathbf{\Phi}(z) := \big(z^0, \ldots, z^{n-1}\big)^T$ as discussed in Section 3.2.

It can be shown that whenever (6.2) holds and the features $z^{(1)}, \ldots, z^{(m)}$ of the training data are all different, the feature vectors $\mathbf{x}^{(1)} = \mathbf{\Phi}\big(z^{(1)}\big), \ldots, \mathbf{x}^{(m)} = \mathbf{\Phi}\big(z^{(m)}\big)$ are linearly independent. This implies, in turn, that polynomial regression is guaranteed to find a hypothesis with zero training error whenever $m \leq n$ and the training datapoints have different feature values.

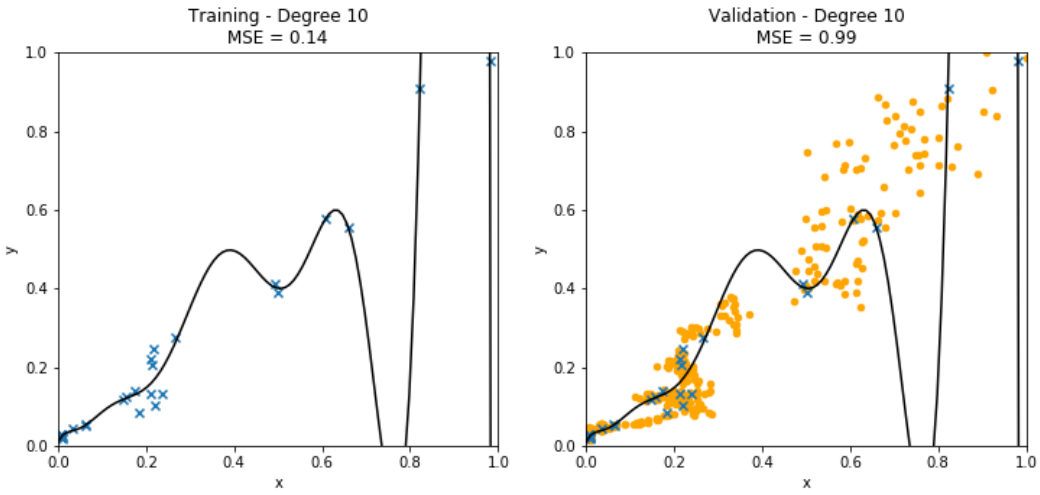


Figure 6.2: Polynomial regression learns a polynomial map with degree $n-1$ by minimizing its average loss on a training set (blue crosses). Using high-degree polynomials (large n) results in a small training error. However, the learnt high-degree polynomial performs poorly on datapoints outside the training set (orange dots).

6.2 Validation

Consider an ML method that uses ERM (4.2) to learn a hypothesis $\hat{h} \in \mathcal{H}$ out of the hypothesis space $\mathcal{H}$. The discussion in Section 6.1 revealed that the training error of a learnt

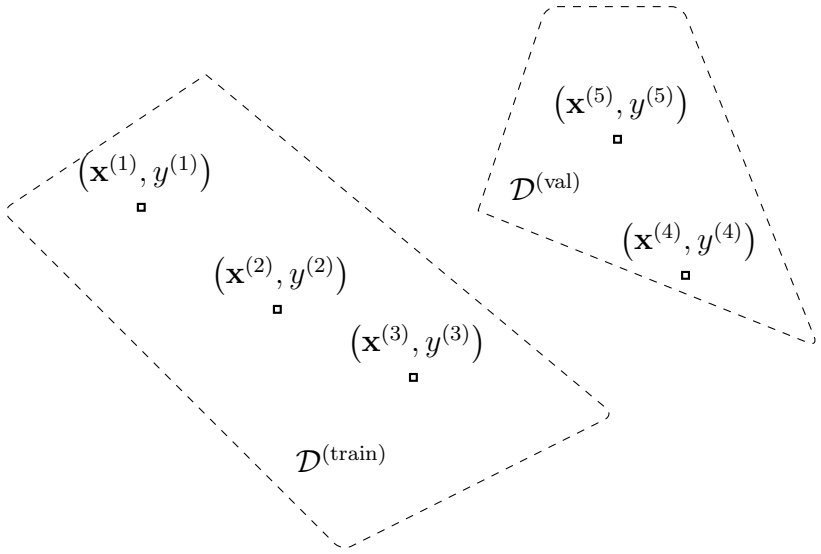


Figure 6.3: We split the dataset $\mathcal{D}$ into two subsets, a **training set** $\mathcal{D}^{(\text{train})}$ and a **validation set** $\mathcal{D}^{(\text{val})}$. We use the training set to learn (find) the hypothesis $\hat{h}$ with minimum empirical risk $\mathcal{E}(\hat{h}|\mathcal{D}^{(\text{train})})$ 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 $\mathcal{D}^{(\text{val})}$.

hypothesis $\hat{h}$ can be a poor indicator for the performance of $\hat{h}$ for datapoints outside the training set. The hypothesis $\hat{h}$ tends to "look better" on the training set over which it has been tuned within ERM. The basic idea of validating the predictor $\hat{h}$ is simple: after learning $\hat{h}$ using ERM on a training set, compute its average loss on datapoints which have not been used in ERM. By **validation** we refer to the computation of the average loss on datapoints that have not been used in ERM.

Assume we have access to a dataset of m datapoints,

$$\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}.$$

Each datapoint is characterized by a feature vector $\mathbf{x}^{(i)}$ and a label $y^{(i)}$. Algorithm 4 outlines how to learn and validate a hypothesis $h \in \mathcal{H}$ by splitting the dataset $\mathcal{D}$ into a **training set** and a **validation set**.

Algorithm 4 Validated ERM

Input: model $\mathcal{H}$, loss function $\mathcal{L}$, dataset $\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}$; split ratio ρ

1: randomly shuffle the datapoints in $\mathcal{D}$

2: create the **training set** $\mathcal{D}^{(\text{train})}$ using the first $m_t = \lceil \rho m \rceil$ datapoints,

$$\mathcal{D}^{(\text{train})} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m_t)}, y^{(m_t)}\big)\big\}.$$

3: create the **validation set** $\mathcal{D}^{(\text{val})}$ by the $m_v = m - m_t$ remaining datapoints,

$$\mathcal{D}^{(\text{val})} = \big\{\big(\mathbf{x}^{(m_t+1)}, y^{(m_t+1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}.$$

4: learn hypothesis $\hat{h}$ via ERM on the training set,

$$\hat{h} := \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{E}\big(h|\mathcal{D}^{(\text{train})}\big) \tag{6.4}$$

5: compute **training error**

$$E_t := \mathcal{E}\big(\hat{h}|\mathcal{D}^{(\text{train})}\big) = (1/m_t) \sum_{i=1}^{m_t} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}). \tag{6.5}$$

6: compute **validation error**

$$E_v := \mathcal{E}\big(\hat{h}|\mathcal{D}^{(\text{val})}\big) = (1/m_v) \sum_{i=m_t+1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}). \tag{6.6}$$

Output: learnt hypothesis $\hat{h}$, training error E_t, validation error E_v

The random shuffling in step 1 of Algorithm 4 ensures that the order of the datapoints has no meaning. This is important in applications where the datapoints are collected sequentially over time and consecutive datapoints might be correlated. We could avoid the shuffling step, if we construct the training set by randomly choosing a subset of size m_t instead of using the first m_t datapoints.

6.2.1 The Size of the Validation Set

The choice of the split ratio $\rho \approx m_t/m$ in Algorithm 4 is often based on trial and error. We try out different choices for the split ratio and pick the one resulting in the smallest validation error. It is difficult to make a precise statement on how to choose the split ratio which applies broadly [51]. This difficulty stems from the fact that the optimal choice for ρ depends on the precise statistical properties of the datapoints.

To obtain a lower bound on the required size of the validation set, we need a probabilistic model for the datapoints. Let us assume that datapoints are realizations of i.i.d. RVs with the same probability distribution $p(\mathbf{x}, y)$. Then the validation error E_v (6.6) becomes a realization of a RV. The expectation (or mean) $\mathbb{E}\{E_v\}$ of this RV is precisely the risk $\mathbb{E}\{\mathcal{L}((\mathbf{x}, y), \hat{h})\}$ of $\hat{h}$ (see (4.1)).

The random validation error E_v fluctuates around its mean. We can quantify this fluctuations using the variance

$$\sigma_{E_v}^2 := \mathbb{E}\big\{\big(E_v - \mathbb{E}\{E_v\}\big)^2\big\}.$$

Note that the validation error is the average of the realizations $\mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h})$ of i.i.d. RVs. The probability distribution of the RV $\mathcal{L}((\mathbf{x}, y), \hat{h})$ is determined by the probability distribution $p(\mathbf{x}, y)$, the choice of loss function and the hypothesis $\hat{h}$. In general, we do not know $p(\mathbf{x}, y)$ and, in turn, also do not know the probability distribution of $\mathcal{L}((\mathbf{x}, y), \hat{h})$.

If we know an upper bound U on the variance of the (random) loss $\mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h})$, we can bound the variance of E_v as

$$\sigma_{E_v}^2 \leq U/m_v.$$

We can then, in turn, ensure that the variance $\sigma_{E_v}^2$ of the validation error E_v does not exceed a given threshold η, say $\eta = (1/100)E_t^2$, by using a validation set of size

$$m_v \geq U/\eta. \tag{6.7}$$

The lower bound (6.7) is only useful if we can determine an upper bound U on the variance of the RV $\mathcal{L}((\mathbf{x},y),\hat{h})$ where $\big(\mathbf{x},y\big)$ is a RV with probability distribution $p(\mathbf{x},y)$. An upper bound on the variance of $\mathcal{L}((\mathbf{x},y),\hat{h})$ can be derived using probability theory if we know an accurate probabilistic model $p(\mathbf{x},y)$ for the datapoints. Such a probabilistic model might be provided by application-specific scientific fields such as biology or psychology. Another option is to estimate the variance of $\mathcal{L}((\mathbf{x},y),\hat{h})$ using the sample variance of the actual loss values $\mathcal{L}((\mathbf{x}^{(1)},y^{(1)}),\hat{h}),\ldots,\mathcal{L}((\mathbf{x}^{(m)},y^{(m)}),\hat{h})$ obtained for the dataset $\mathcal{D}$.

6.2.2 k-Fold Cross Validation

Algorithm 4 uses the most basic form of splitting a given dataset $\mathcal{D}$ into a training and a validation set. Many variations and extensions of this basic splitting approach have been proposed and studied (see [24] and Section 6.5). One very popular extension of the single split into training and validation set is known as **k-fold cross-validation** (CV) [37, Sec. 7.10]. We summarize k-fold CV in Algorithm 5 below.

Figure 6.4 illustrates the key principle behind k-fold CV which is to divide the entire dataset evenly into k subsets which are referred to as **folds**. The learning (via ERM) and validation of a hypothesis out of a given hypothesis space $\mathcal{H}$ is then repeated k times. During each repetition, we use one fold as the validation set and the remaining $k-1$ folds as a training set. We then average the training and validation error over all repetitions.

The average (over all k folds) validation error delivered by k-fold CV is a more robust estimator for the expected loss or risk (4.1) compared to the validation error obtained from a single split. Consider a small dataset and using a single split into training and validation set. We might then be very unlucky and choose datapoints for the validation set which are outliers and not representative of the overall distribution of the data.

6.2.3 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. We might then be interested in having a good estimate of the conditional risks $\mathbb{E}\{\mathcal{L}((\mathbf{x},y),h)|y=y'\}$ where y' is one of the rare label values. This is more than requiring a good estimate for the risk $\mathbb{E}\{\mathcal{L}((\mathbf{x},y),h)\}$.

Consider datapoints characterized by a feature vector $\mathbf{x}$ and binary label $y \in \{-1,1\}$.

Algorithm 5 k-fold CV ERM

Input: model $\mathcal{H}$, loss function $\mathcal{L}$, dataset $\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}$; number k of folds

1: randomly shuffle the datapoints in $\mathcal{D}$

2: divide the shuffled dataset $\mathcal{D}$ into k folds $\mathcal{D}_1, \ldots, \mathcal{D}_k$ of size $B = \lceil m/k \rceil$,

$$\mathcal{D}_1 = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(B)}, y^{(B)}\big)\big\}, \ldots, \mathcal{D}_k = \big\{\big(\mathbf{x}^{((k-1)B+1)}, y^{((k-1)B+1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\} \quad (6.8)$$

3: **for** fold index $b = 1, \ldots, k$ **do**

4: use bth fold as the validation set $\mathcal{D}^{(\text{val})} = \mathcal{D}_b$

5: use rest as the **training set** $\mathcal{D}^{(\text{train})} = \mathcal{D} \setminus \mathcal{D}_b$

6: learn hypothesis $\hat{h}$ via ERM on the training set,

$$\hat{h}^{(b)} := \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{E}\big(h | \mathcal{D}^{(\text{train})}\big) \quad (6.9)$$

7: compute **training error**

$$E_t^{(b)} := \mathcal{E}\big(\hat{h} | \mathcal{D}^{(\text{train})}\big) = (1/\big|\mathcal{D}^{(\text{train})}\big|) \sum_{i \in \mathcal{D}^{(\text{train})}} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h). \quad (6.10)$$

8: compute **validation error**

$$E_v^{(b)} := \mathcal{E}\big(\hat{h} | \mathcal{D}^{(\text{val})}\big) = (1/\big|\mathcal{D}^{(\text{val})}\big|) \sum_{i \in \mathcal{D}^{(\text{val})}} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}). \quad (6.11)$$

9: **end for**

10: compute average training and validation errors

$$E_t := (1/k) \sum_{b=1}^{k} E_t^{(b)}, \text{ and } E_v := (1/k) \sum_{b=1}^{k} E_v^{(b)}$$

11: pick a learnt hypothesis $\hat{h} := \hat{h}^{(b)}$ for some $b \in \{1, \ldots, k\}$

Output: learnt hypothesis $\hat{h}$; average training error E_t; average validation error E_v

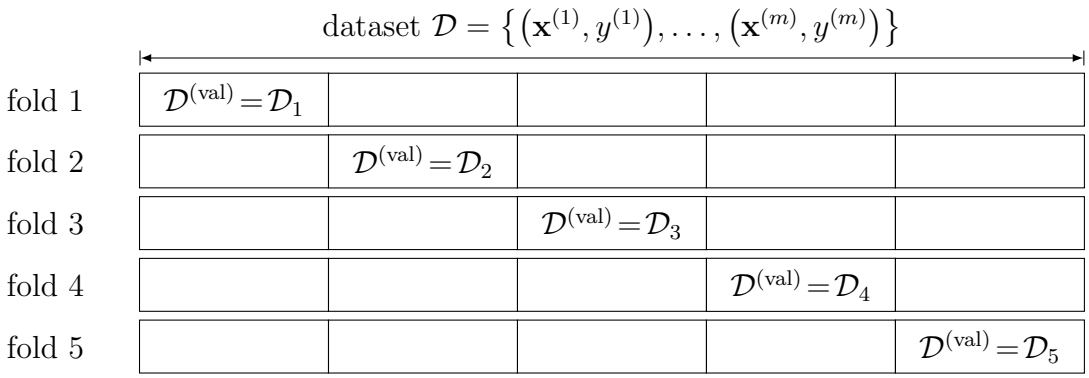


Figure 6.4: Illustration of k-fold CV for $k = 5$. We evenly partition the entire dataset $\mathcal{D}$ into $k = 5$ subsets (or folds) $\mathcal{D}_1, \ldots, \mathcal{D}_5$. We then repeat the validated ERM Algorithm 4 for $k = 5$ times. The bth repetition uses the bth fold $\mathcal{D}_b$ as the validation set and the remaining $k-1(= 4)$ folds as the training set for ERM (4.2).

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 in one of the validation folds. However, even 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 loss of $h(\mathbf{x})$ incurred on datapoints with the (majority) label $y = 1$.

When learning and validating a hypothesis using imbalanced data, it might be useful to to generate synthetic datapoints to enlarge the minority class. This can be done using data augmentation techniques which we discuss in Section 7.3. Another option is to use a loss function that takes the different frequency of label values into account.

Consider an imbalanced dataset of size $m = 100$, which contains 90 datapoints with label $y = 1$ but only 10 datapoints with label $y = -1$. We might then put more weight on wrong predictions obtained for the minority class (of datapoints with $y = -1$). This can be done by using a much larger value for the loss $\mathcal{L}((\mathbf{x}, y = -1), h(\mathbf{x}) = 1)$ than for the loss $\mathcal{L}((\mathbf{x}, y = 1), h(\mathbf{x}) = -1)$. Remember, the loss function is a design choice and can been freely set by the ML engineer.

6.3 Model Selection

Chapter 3 illustrated how many well-known ML methods are obtained by different combinations of a hypothesis space or model, loss function and data representation. While for many ML applications there is often a natural choice for the loss function and data representation, the right choice for the model is typically less obvious. This chapter shows how to use the validation methods of Section 6.2 to choose between different candidate models.

Consider datapoints characterized by a single numeric feature $x \in \mathbb{R}$ and numeric label $y \in \mathbb{R}$. If we suspect that the relation between feature x and label y is non-linear, we might use polynomial regression which is discussed in Section 3.2. Polynomial regression uses the hypothesis space $\mathcal{H}^{(n)}_{\rm poly}$ with some maximum degree n. Different choices for the maximum degree n yield a different hypothesis space: $\mathcal{H}^{(1)} = \mathcal{H}^{(0)}_{\rm poly}, \mathcal{H}^{(2)} = \mathcal{H}^{(1)}_{\rm poly}, \ldots, \mathcal{H}^{(M)} = \mathcal{H}^{(M-1)}_{\rm poly}$.

Another ML method that learns non-linear hypothesis map is Gaussian basis regression (see Section 3.5). Here, different choices for the variance σ and shifts μ of the Gaussian basis function (3.12) result in different hypothesis spaces. For example, $\mathcal{H}^{(1)} = \mathcal{H}^{(2)}_{\rm Gauss}$ with $\sigma = 1$ and $\mu_1 = 1$ and $\mu_2 = 2$, $\mathcal{H}^{(2)} = \mathcal{H}^{(2)}_{\rm Gauss}$ with $\sigma = 1/10$, $\mu_1 = 10$, $\mu_2 = 20$.

Algorithm 6 summarizes a simple method to choose between different candidate models $\mathcal{H}^{(1)}, \mathcal{H}^{(2)}, \ldots, \mathcal{H}^{(M)}$. The idea is to first learn and validate a hypothesis $\hat{h}^{(l)}$ separately for each model $\mathcal{H}^{(l)}$ using Algorithm 5. For each model $\mathcal{H}^{(l)}$, we learn the hypothesis $\hat{h}^{(l)}$ via ERM (6.4) and then compute its validation error $E^{(l)}_v$ (6.6). We then choose the hypothesis $\hat{h}^{(\hat{l})}$ from those model $\mathcal{H}^{(\hat{l})}$ which resulted in the smallest validation error $E^{(\hat{l})}_v = \min_{l=1,\ldots,M} E^{(l)}_v$.

The "work-flow" of Algorithm 6 is quite similar to the work-flow of ERM. The idea of ERM is to learn a hypothesis out of a set of different candidates (the hypothesis space). The quality of a particular hypothesis h is measured using the (average) loss incurred on some training set. We use a similar principle for model selection but on a higher level. Instead of learning a hypothesis within a hypothesis space, we choose (or learn) a hypothesis space within a set of candidate hypothesis spaces. The quality of a given hypothesis space is measured by the validation error (6.6). To determine the validation error of a hypothesis space, we first learn the hypothesis $\hat{h} \in \mathcal{H}$ via ERM (6.4) on the training set. Then, we obtain the validation error as the average loss of $\hat{h}$ on the validation set.

The final hypothesis $\hat{h}$ delivered by the model selection Algorithm 6 not only depends on the training set used in ERM (see (6.9)). This hypothesis $\hat{h}$ has also been chosen based on its validation error which is the average loss on the validation set in (6.11). Indeed, we compared this validation error with the validation errors of other models to pick the model $\mathcal{H}^{(\hat{l})}$ (see step 10) which contains $\hat{h}$. Since we used the validation error (6.11) of $\hat{h}$ to learn

it, we cannot use this validation error as a good indicator for the general performance of $\hat{h}$.

To estimate the general performance of the final hypothesis $\hat{h}$ delivered by Algorithm 6 we must try it out on a test set. The test set, which is constructed in step 3 of Algorithm 6, consists of datapoints that have neither been used within training (6.9) or validation (6.11) of the candidate models $\mathcal{H}^{(1)}, \ldots, \mathcal{H}^{(M)}$. The average loss of the final hypothesis on the test set is referred to as the test error. The test error is computed in the step 12 of Algorithm 6.

Algorithm 6 Model Selection

Input: list of candidate models $\mathcal{H}^{(1)}, \ldots, \mathcal{H}^{(M)}$, loss function $\mathcal{L}$, dataset $\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}$; number k of folds, test fraction ρ

1: randomly shuffle the datapoints in $\mathcal{D}$
2: determine size $m' := \lceil \rho m \rceil$ of test set
3: construct **test set**

$$\mathcal{D}^{(\text{test})} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m')}, y^{(m')}\big)\big\}$$

4: construct the set used for training and validation,

$$\mathcal{D}^{(\text{trainval})} = \big\{\big(\mathbf{x}^{(m'+1)}, y^{(m'+1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}$$

5: **for** model index $l = 1, \ldots, M$ **do**
6: run Algorithm 5 using $\mathcal{H} = \mathcal{H}^{(l)}$, dataset $\mathcal{D} = \mathcal{D}^{(\text{trainval})}$, loss function $\mathcal{L}$ and k folds
7: Algorithm 5 delivers hypothesis $\hat{h}$ and validation error E_v
8: store learnt hypothesis $\hat{h}^{(l)} := \hat{h}$ and validation error $E_v^{(l)} := E_v$
9: **end for**
10: pick model $\mathcal{H}^{(\hat{l})}$ with minimum validation error $E_v^{(\hat{l})} = \min_{l=1,\ldots,M} E_v^{(l)}$
11: define optimal hypothesis $\hat{h} = \hat{h}^{(\hat{l})}$
12: compute **test error**

$$E^{(\text{test})} := \mathcal{E}\big(\hat{h} | \mathcal{D}^{(\text{test})}\big) = (1/\big|\mathcal{D}^{(\text{test})}\big|) \sum_{i \in \mathcal{D}^{(\text{test})}} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}). \tag{6.12}$$

Output: hypothesis $\hat{h}$; training error $E_t^{(\hat{l})}$; validation error $E_v^{(\hat{l})}$, test error $E^{(\text{test})}$.

Sometimes it is beneficial to use different loss functions for the training and the validation of a hypothesis. As an example, consider the ML methods logistic regression and SVM which have been discussed in Sections 3.6 and 3.7, respectively. Both methods use the same model which is the space of linear hypothesis maps $h(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$. The main difference between these two methods is the choice for the loss function used to measure the quality of a hypothesis. Logistic regression minimizes the (average) logistic loss (2.11) on the training set to learn

the hypothesis $h^{(1)}(\mathbf{x}) = \left(\mathbf{w}^{(1)}\right)^T \mathbf{x}$ with a weight vector $\mathbf{w}^{(1)}$. The SVM instead minimizes the (average) hinge loss (2.10) on the training set to learn the hypothesis $h^{(2)}(\mathbf{x}) = \left(\mathbf{w}^{(2)}\right)^T \mathbf{x}$ with a weight vector $\mathbf{w}^{(2)}$. It would be difficult to compare the hypotheses $h^{(1)}(\mathbf{x})$ and $h^{(2)}(\mathbf{x})$ using different loss functions to compute their validation errors. For a comparison, we could instead compute the validation errors for $h^{(1)}(\mathbf{x})$ and $h^{(2)}(\mathbf{x})$ using the average 0/1 loss (2.8) ("accuracy").

Algorithm 6 requires as one of its inputs a given list of candidate models. The longer this list, the more computation is required from Algorithm 6. Sometimes it is possible to prune the list of candidate models by removing models that are very unlikely to have minimum validation error.

Consider polynomial regression which uses as the model the space $\mathcal{H}^{(r)}_{\rm poly}$ of polynomials with maximum degree r (see (3.4)). For $r = 1$, $\mathcal{H}^{(r)}_{\rm poly}$ is the space of polynomials with maximum degree one (which are linear maps), $h(x) = w_2 x + w_1$. For $r = 2$, $\mathcal{H}^{(r)}_{\rm poly}$ is the space of polynomials with maximum degree two, $h(x) = w_3 x^2 + w_2 x + w_1$.

The polynomial degree r parametrizes a nested set of models,

$$\mathcal{H}^{(1)}_{\rm poly} \subset \mathcal{H}^{(2)}_{\rm poly} \subset \ldots .$$

For each degree r, we learn a hypothesis $h^{(r)} \in \mathcal{H}^{(r)}_{\rm poly}$ with minimum average loss (training error) $E^{(r)}_t$ on a training set (see (6.5)). To validate the learnt hypothesis $h^{(r)}$, we compute its average loss (validation error) $E^{(r)}_v$ on a validation set (see (6.6)).

Figure 6.5 depicts the typical dependency of the training and validation errors on the polynomial degree r. The training error $E^{(r)}_t$ decreases monotonically with increasing degree r. To understand why this is the case, consider the two specific choices $r = 3$ and $r = 5$ with corresponding models $\mathcal{H}^{(3)}_{\rm poly}$ and $\mathcal{H}^{(5)}_{\rm poly}$. Note that $\mathcal{H}^{(3)}_{\rm poly} \subset \mathcal{H}^{(5)}_{\rm poly}$ since any polynomial with degree not exceeding 3 is also a polynomial with degree not exceeding 5. Therefore, the training error (6.5) obtained when minimizing over the larger model $\mathcal{H}^{(5)}_{\rm poly}$ can only decrease but never increase compared to (6.5) using the smaller model $\mathcal{H}^{(3)}_{\rm poly}$

Figure 6.5 indicates that the validation error $E^{(r)}_v$ (see (6.6)) behaves very different compared to the training error $E^{(r)}_t$. Starting with degree $r = 0$, the validation error first decreases with increasing degree r. As soon as the degree r is increased beyond a critical value, the validation error starts to increase with increasing r. For very large values of r, the training error becomes almost negligible while the validation error becomes very large. In this regime, polynomial regression overfits the training data.

Figure 6.6 illustrates the overfitting of polynomial regression when using a maximum

degree that is too large. In particular, Figure 6.6 depicts a learnt hypothesis which is a degree 9 polynomial that fits very well the training set, resulting in a very small training error. To achieve this low training error the resulting polynomial has an unreasonable high rate of change for feature values $x \approx 0$. This results in large prediction errors for validation datapoints with feature values $x \approx 0$.

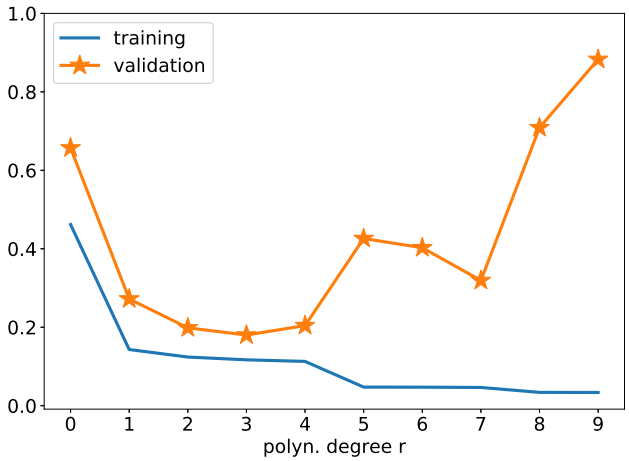


Figure 6.5: The training error and validation error obtained from polynomial regression using different values r for the maximum polynomial degree.

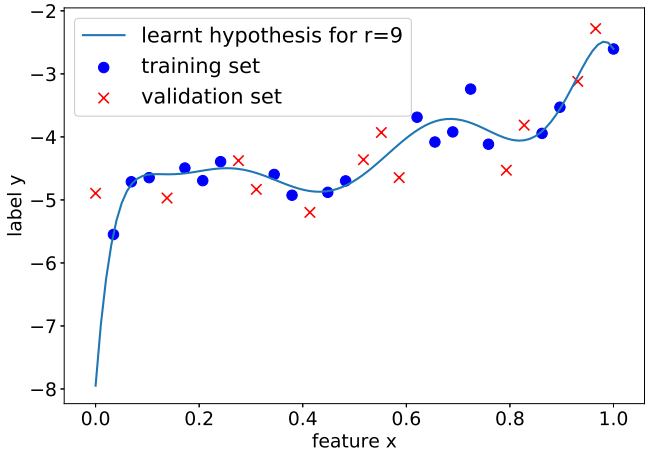


Figure 6.6: A hypothesis $\hat{h}$ which is a polynomial with degree not larger than $r = 9$. The hypothesis has been learnt by minimizing the average loss on the training set. Note the fast rate of the change of $\hat{h}$ for feature values $x \approx 0$.

6.4 A Probabilistic Analysis of Generalization

More Data Beats Clever Algorithms ?; More Data Beats Clever Feature Selection?

A key challenge in ML is to ensure that a hypothesis that predicts well the labels on a training set (which has been used to learn that hypothesis) will also predict well the labels of datapoints outside the training set. We say that a ML method generalizes if a small loss on the training set implies small loss on datapoints outside the training set.

To study generalization within a linear regression problem (see Section 3.1), we will use a **probabilistic model** for the data. We interpret datapoints as i.i.d. realizations of RVs that have the same distribution as a random datapoint $\mathbf{z} = (\mathbf{x}, y)$. The random 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 random datapoint is related to its features $\mathbf{x}$ via a **linear Gaussian model**

$$y = \bar{\mathbf{w}}^T \mathbf{x} + \varepsilon, \text{ with noise } \varepsilon \sim \mathcal{N}(0, \sigma^2). \tag{6.13}$$

We assume the noise variance σ^2 fixed and known. This is a simplifying assumption as in practice, we would need to estimate the noise variance from data [20]. Note that, within our probabilistic model, the error component ε in (6.13) is intrinsic to the data and cannot be overcome by any ML method. We highlight that the probabilistic model for the observed data points is just a modelling assumption. This assumption allows us to study some fundamental

behaviour of ML methods. There are principled methods ("tests") that allow to determine if a given dataset can be accurately modelled using (6.13) [41].

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

$$\mathcal{H}^{(r)} = \{h^{(\mathbf{w})}(\mathbf{x}) = (\mathbf{w}^T, \mathbf{0})\mathbf{x} \text{ with } \mathbf{w} \in \mathbb{R}^r\}. \tag{6.14}$$

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 on the labeled training set $\mathcal{D}^{(\text{train})} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m_\text{t}}$. Within our toy model (see (6.13), (6.15) and (6.16)), 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,

$$\mathbf{x}, \mathbf{x}^{(i)} \text{ i.i.d. with } \mathbf{x}, \mathbf{x}^{(i)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{6.15}$$

and the labels $y^{(i)}, y$ are obtained as

$$y^{(i)} = \bar{\mathbf{w}}^T \mathbf{x}^{(i)} + \varepsilon^{(i)}, \text{ and } y = \bar{\mathbf{w}}^T \mathbf{x} + \varepsilon \tag{6.16}$$

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 $h^{(\widehat{\mathbf{w}})}(\mathbf{x}) = \widehat{\mathbf{w}}^T \mathbf{I}_{r \times n} \mathbf{x}$, with weight vector

$$\widehat{\mathbf{w}} = (\mathbf{X}_r^T \mathbf{X}_r)^{-1} \mathbf{X}_r^T \mathbf{y} \tag{6.17}$$

with feature matrix $\mathbf{X}_r$ and label vector $\mathbf{y}$ defined as

$$\begin{aligned} \mathbf{X}_r &= (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m_\text{t})})^T \mathbf{I}_{n \times r} \in \mathbb{R}^{m_\text{t} \times r}, \text{ and} \\ \mathbf{y} &= \big(y^{(1)}, \ldots, y^{(m_\text{t})}\big)^T \in \mathbb{R}^{m_\text{t}}. \end{aligned} \tag{6.18}$$

It will be convenient to tolerate a slight abuse of notation and denote both, the length-r vector (6.17) 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

$$h^{(\widehat{\mathbf{w}})}(\mathbf{x}) = \widehat{\mathbf{w}}^T \mathbf{x}. \tag{6.19}$$

We highlight that the formula (6.17) 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 (6.15)), this is true with probability one whenever $m_{\rm t} \geq r$. In what follows, we will consider the case of having more training samples than the dimension of the hypothesis space, i.e., $m_{\rm t} > r$ such that the formula (6.17) is valid (with probability one). The case $m_{\rm t} \leq r$ will be studied in Chapter 7.

The optimal weight vector $\widehat{\mathbf{w}}$ (see (6.17)) depends on the training data $\mathcal{D}^{(\rm train)}$ via the feature matrix $\mathbf{X}_r$ and label vector $\mathbf{y}$ (see (6.18)). Therefore, since we model the training data as random, the weight vector $\widehat{\mathbf{w}}$ (6.17) 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 (6.13) 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**

$$\Delta \mathbf{w} := \widehat{\mathbf{w}} - \bar{\mathbf{w}}. \tag{6.20}$$

The estimation error (6.20) is the realization of a RV since the learnt weight vector $\widehat{\mathbf{w}}$ (see (6.17)) is itself a realization of a RV.

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)**

$$\mathcal{E}_{\rm est} := \mathbb{E}\{\|\Delta \mathbf{w}\|_2^2\} = \mathbb{E}\big\{\big\|\widehat{\mathbf{w}} - \bar{\mathbf{w}}\big\|_2^2\big\}. \tag{6.21}$$

We can decompose the MSE $\mathcal{E}_{\rm est}$ into two components. The first component is the **bias** which characterizes the average behaviour, over all different realizations of training sets, of the learnt hypothesis. 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

$$\mathcal{E}_{\text{est}} = \underbrace{\|\bar{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2}_{\text{“bias”}\, B^2} + \underbrace{\mathbb{E}\|\widehat{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2}_{\text{“variance”}\, V} \tag{6.22}$$

The bias term in (6.22), which can be computed as

$$B^2 = \|\bar{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2 = \sum_{l=r+1}^{n} \bar{w}_l^2, \tag{6.23}$$

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 (6.14)) of the linear regression problem.

The bias (6.23) 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 (6.13) 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 (6.13). In general, the bias term decreases with an increasing model size r (see Figure 6.7). We highlight that the bias term does not depend on the variance σ^2 of the noise ε in our toy model (6.13).

Let us now consider the variance term in (6.22). Using the properties of our toy model (see (6.13), (6.15) and (6.16))

$$V = \mathbb{E}\{\|\widehat{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2\} = \sigma^2 \text{tr}\big\{\mathbb{E}\{(\mathbf{X}_r^T\mathbf{X}_r)^{-1}\}\big\}. \tag{6.24}$$

By (6.15), the matrix $(\mathbf{X}_r^T\mathbf{X}_r)^{-1}$ is random and distributed according to an **inverse Wishart distribution** [56]. For $m_{\text{t}} > r+1$, its expectation is given as

$$\mathbb{E}\{(\mathbf{X}_r^T\mathbf{X}_r)^{-1}\} = 1/(m_{\text{t}} - r - 1)\mathbf{I}_{r\times r}. \tag{6.25}$$

By inserting (6.25) and $\text{tr}\{\mathbf{I}_{r\times r}\} = r$ into (6.24),

$$V = \mathbb{E}\{\|\widehat{\mathbf{w}} - \mathbb{E}\{\widehat{\mathbf{w}}\}\|_2^2\} = \sigma^2 r/(m_{\text{t}} - r - 1). \tag{6.26}$$

As indicated by (6.26), the variance term increases with increasing model complexity r (see Figure 6.7). 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 (6.14))

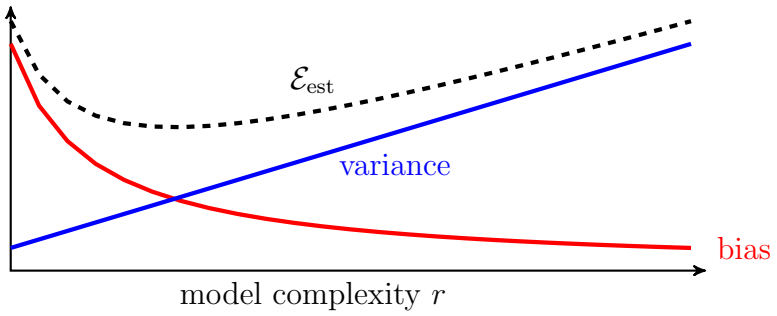


Figure 6.7: The estimation error $\mathcal{E}_{\rm est}$ incurred by linear regression can be decomposed into a bias term B^2 and a variance term V (see (6.22)). These two components depend on the model complexity r in an opposite manner resulting in a bias-variance trade-off.

has to balance between a small variance and a small bias.

Generalization. Consider the linear hypothesis $h(\mathbf{x}) = \widehat{\mathbf{w}}^T\mathbf{x}$ with the weight vector (6.17) 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

$$\begin{aligned}
\mathcal{E}_{\rm pred} &= \mathbb{E}\big\{\big(y-\hat{y}\big)^2\big\} \\
&\overset{(6.13)}{=} \mathbb{E}\{\Delta\mathbf{w}^T\mathbf{x}\mathbf{x}^T\Delta\mathbf{w}\} + \sigma^2 \\
&\overset{(a)}{=} \mathbb{E}\{\mathbb{E}\{\Delta\mathbf{w}^T\mathbf{x}\mathbf{x}^T\Delta\mathbf{w} \mid \mathcal{D}\}\} + \sigma^2 \\
&\overset{(b)}{=} \mathbb{E}\{\Delta\mathbf{w}^T\Delta\mathbf{w}\} + \sigma^2 \\
&\overset{(6.20),(6.21)}{=} \mathcal{E}_{\rm est} + \sigma^2 \\
&\overset{(6.22)}{=} B^2 + V + \sigma^2. \qquad (6.27)
\end{aligned}$$

Step (a) uses the law of total expectation [10] 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 (6.15)).

According to (6.27), 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 6.7 illustrates the typical dependency of the bias and variance on the model, which is parametrized by the model complexity r. which also coincides with our notion of effective model dimension (see Section 2.2).

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 (6.13) 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 (6.23), variance (6.26) and the average prediction error (6.27) only applies if the observed datapoints are well modelled as realizations of random vectors according to (6.13), (6.15) and (6.16). The usefulness of this model for the data arising in a particular application has to be verified in practice by statistical model validation techniques [92, 84].

The qualitative behaviour of estimation error in Figure 6.7 depends on the definition for the model complexity. Our concept of effective dimension (see Section 2.2) coincides with most other notions of model complexity for the linear hypothesis space (6.14). However, for more complicated models such as deep nets it is often not obvious how the model complexity is related to more tangible quantities such as total number of tunable weights or neurons. In general, the model complexity or effective model dimension is not directly proportional to number of tunable weights but also depends on the specific learning algorithm such as SGD. Therefore, for deep nets, if we would plot estimation error against number of tunable weights we might observe a behaviour of estimation error fundamentally different from the shape in Figure 6.7. One example for such un-intuitive behaviour is known as "double descent phenomena" [7].

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 [2]. Using these i.i.d. copies, we can replace exact computations (expectations) by empirical approximations (sample averages).

6.5 The Bootstrap

basic idea of bootstrap: use empirical distribution (histogram) of data points as their probability distribution; we can then sample any amount of new data points from that distribution (using sampling with replacement)

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. RVs. Let use denote the (common) probability distribution of these RVs by $p(\mathbf{x}, y)$.

If we interpret the datapoints $\big(\mathbf{x}^{(i))}, y^{(i)}\big)$ as realizations of RVs, also the learnt hypothesis

$\hat{h}$ is a realization of a RV. Indeed, the hypothesis $\hat{h}$ is obtained by solving an optimization problem (4.2) that involves realizations of RVs. The bootstrap is a method for estimating (parameters of) the probability distribution $p(\hat{h})$ [37].

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 (6.15), (6.16), 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 $\mathcal{D}$. For each dataset $\mathcal{D}^{(b)}$, we solve a separate ERM (4.2) to obtain the hypothesis $\hat{h}^{(b)}$. The hypothesis $\hat{h}^{(b)}$ is a realization of a RV whose distribution is determined by the empirical distribution $\hat{p}(\mathbf{z})$ as well as the hypothesis space and the 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)

In what follows, we assume that datapoints can (to a good approximation) be interpreted as realizations of i.i.d. RVs (see Section 2.1.4). This "i.i.d. assumption" underlies ERM (4.2) as the guiding principle for learning a hypothesis with small risk (4.1). This assumption also motivates to use the average loss (6.6) on a validation set as an estimate for the risk.

Consider a ML method which uses Algorithm 4 (or Algorithm 5) to learn and validate the hypothesis $\hat{h} \in \mathcal{H}$. Besides the learnt hypothesis $\hat{h}$, these algorithms also deliver the training error E_t and the validation error E_v. As we will see shortly, we can diagnose ML methods to some extend just by comparing training with validation errors. This diagnosis is further enabled if we know a benchmark (or reference) error level $E^{(\mathrm{ref})}$.

One important source of a benchmark error level $E^{(\mathrm{ref})}$ are probabilistic models for the datapoints (see Section 6.4). Given a probabilistic model, which specifies the probability distribution $p(\mathbf{x}, y)$ of the features and label of datapoints, we can compute the minimum achievable expected loss or risk (4.1). Indeed, the minimum achievable risk is precisely the expected loss of the Bayes' estimator $\hat{h}(x)$ of the label y, given the features $\mathbf{x}$ of a datapoint. The Bayes' estimator $\hat{h}(x)$ is fully determined by the probability distribution $p(\mathbf{x}, y)$ of the

features and label of a (random) datapoint [52, Chapter 4].

Example. Let us derive the minimum achievable risk for datapoints with single numeric feature and label and being realizations of a Gaussian random vector $\mathbf{z} \sim \mathcal{N}(0, \mathbf{C})$. Here, the optimal estimator of the label y given the feature x is the conditional expectation of the (unobserved) label y given the (observed) feature x. The resulting MSE is equal to the posterior variance of y, given x which is given by the $K_{y,y}^{-1}$ with the entry $K_{y,y}$ of the precision matrix $\mathbf{K} = \mathbf{C}^{-1}$.

A further potential source for a benchmark error level $E^{(\text{ref})}$ is another ML method. This other ML method might be computationally too expensive to be used for a ML application. However, we could still use its error level measured in illustrative test scenarios as a benchmark.

Finally, a benchmark can be obtained from the performance of human experts. If we want to develop a ML method that detects certain type of skin cancers from images of the skin, a benchmark might be the current classification accuracy achieved by experienced dermatologists [28].

We can diagnose a ML method by comparing the training error E_t with the validation error E_v and (if available) the benchmark $E^{(\text{ref})}$.

- $E_t \approx E_v \approx E^{(\text{ref})}$: The training error is on the same level as the validation error and the benchmark error. 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. It seems we have obtained a ML method that achieves the benchmark error level.

- $E_v \gg E_t$: The validation error is significantly larger than the training error. It seems that the ERM (4.2) results in a hypothesis $\hat{h}$ that overfits the training set. The loss incurred by $\hat{h}$ on data points outside the training set, such as those in the validation set, is significantly worse. This is an indicator for overfitting which can be addressed either by reducing the effective size of the hypothesis space or by increasing the effective number of training data points. To reduce the effective hypothesis space we can either use a smaller hypothesis space, e.g., using fewer features in a linear model (3.1), using smaller maximum depth of decision trees (Section 3.10) or by using a smaller ANN (Section 3.11). Another way to reduce the effective size of a hypothesis space is to use regularization techniques from Chapter 7.

- $E_t \approx E_v \gg E^{(\text{ref})}$: The training error is on the same level as the validation error

and both are significantly larger than the benchmark error. Since the training error is not much smaller than the validation error, the learnt hypothesis seems to not overfit the training data. However, the training error achieved by the learnt hypothesis is significantly larger than the benchmark error level. There can be several reasons for this to happen. First, it might be that the hypothesis space used by the ML method is too small, i.e., it does not include a hypothesis that provides a good approximation for the relation between features and label of a datapoint. The remedy for this situation is to use a larger hypothesis space, e.g., by including more features in a linear model, using higher polynomial degrees in polynomial regression, using deeper decision trees or having larger ANNs ("deep learning"). Another reason for the training error being too large is that the optimization algorithms used to solve the ERM (4.2) is not working properly. When using gradient based optimization (see Section 5.4) to solve ERM, one reason for $E_t \gg E^{(\mathrm{ref})}$ could be that the step size α in the GD step (5.4) is chosen too small or too large (see Figure 5.3-(b)). This can be solved by adjusting the step-size by trying out several different values and using the one resulting in minimum training and validation error. Another option is to use some probabilistic model for data points derive optimal values for the step-size based on such a model (see Section 6.4).

- $E_v \gg E_t$: The validation error is significantly larger than the training error. The idea of the ERM (4.2) principle is to approximate the risk (4.1) of a hypothesis by its average loss on a training set $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$. The mathematical underpinning for this approximation is the **law of large numbers**which characterizes the average of i.i.d. RVs. The quality of this approximation requires two conditions. First, the data points used for computing the average loss should be such that they would be typically obtained as realizations of i.i.d. RVs with a common probability distribution. Second the number of data points used for computing the average loss must be sufficiently large. Thus, if data points cannot be modelled as realizations of i.i.d. RVs or if the size of the training or validation set is too small, the validation and training errors are unrelated. In particular, it might then be that the validation set consists of data points for which any predictor incurs small average loss. Here, we might try to increase training and validation sets by collecting more labeled data points or using data augmentation (see Section 7.3). If we already have quite large training and validation sets, it might be a good idea to verify if data points conform to the i.i.d. assumption underlying ERM. There are principled methods to test if an i.i.d. assumption is satisfied (see [54] and references therein).

???See mycourses quiz??? Another reason why the validation and training errors can show erratic behavior is if the training and validation sets are too small. The

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

Keywords: Data Augmentation. Robustness. Semi-Supervised Learning. Transfer Learning. Multitask Learning.

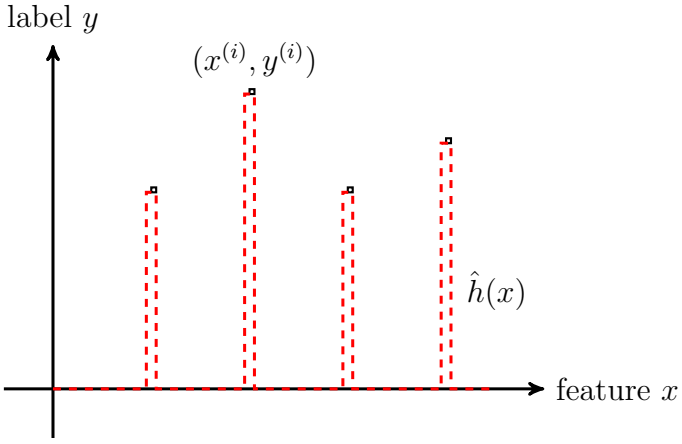


Figure 7.1: The non-linear hypothesis map $\hat{h}$ perfectly fits the training set and has vanishing training error. Despite perfectly fitting the training set, the hypothesis $\hat{h}$ delivers the trivial (and useless) prediction $\hat{y} = \hat{h}(x) = 0$ for any datapoint that is not in the vicinity of the datapoints in 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 its loss incurred outside the training set.

Figure 7.1 illustrates a typical scenario for a modern ML method which uses a large hypothesis space. This large hypothesis space includes highly non-linear maps which can

perfectly resemble any dataset of modest size. However, there might be non-linear maps for which a small training error does not guarantee accurate predictions for the labels of datapoints outside the training set.

Chapter 6 discussed validation techniques to verify if a hypothesis with small training error will predict also well the labels of datapoints outside the training set. These validation techniques, including Algorithm 4 and Algorithm 5, probe the hypothesis $\hat{h} \in \mathcal{H}$ delivered by ERM on a validation set. The validation set consists of datapoints which have not been used for the training set of ERM (4.2). The validation error, which is the average loss of the hypothesis on the datapoints in the validation set, serves as an estimate for the average error or risk (4.1) of the hypothesis $\hat{h}$.

This chapter discusses **regularization** as an alternative to validation techniques. In contrast to validation, regularization techniques do not require having a separate validation set which is not used for the ERM (4.2). This makes regularization attractive for applications where obtaining a separate validation set is difficult or costly (where labelled data is scarce).

Instead of probing a hypothesis $\hat{h}$ on a validation set, regularization techniques compute estimate the loss increase when applying $\hat{h}$ to datapoints outside the training set. The loss increase is estimated by adding a regularization term to the training error in ERM (4.2).

Section 7.1 discusses the resulting regularized ERM, which we will refer to as **structural risk minimization (SRM)**. It turns out that the SRM is equivalent to ERM using a smaller (pruned) hypothesis space. The amount of pruning depends on the weight of the regularization term relative to the training error. For an increasing weight of the regularization term, we obtain a stronger pruning resulting in a smaller effective hypothesis space.

Section 7.2 constructs a regularization terms by requiring the resulting ML method to be robust against (small) random perturbations to the training data. Conceptually, we replace each datapoint of a training set by a RV that fluctuates around this datapoint. This construction allows to interpret regularization as a (implicit) form of **data augmentation**.

Section 7.3 discusses **data augmentation** methods as a simulation-based implementation of regularization. Data augmentation adds a certain number of perturbed copies to each datapoint in the training set. One way to construct perturbed copies of a datapoint is to add (the realization of) a random vector to its features.

Section 7.4 analyzes the effect of regularization for linear regression using a simple probabilistic model for data points. This analysis parallels our previous study of the validation error of linear regression in Section 6.4. Similar to Section 6.4, we reveal a **trade-off**

between the bias and variance of the hypothesis learnt by regularized linear regression. This trade off was traced out by a discrete model parameter (the effective dimension)in Section 6.4. In contrast, regularization offers a continuous trade-off between bias and variance via a real-valued regularization parameter.

Semi-supervised learning (SSL) allows to use (relatively large amounts of) unlabeled data to support the learning of a hypothesis using ERM with (relatively small amounts of) labeled data [19]. Section 7.5 discusses SSL methods that use the statistical properties of unlabeled datapoints, for which we only know the features, to construct useful regularization terms. These regularization terms are then used in SRM with a (typically small) set of labeled datapoints, for which we also know the label values.

Multitask learning exploits similarities between different but related learning tasks [17]. A learning task amounts to a particular choice for the loss function (see Section 2.3) used to score the quality of predictions for the label. The loss function also encapsulates the choice for the label of a datapoint. For learning tasks defined for a single underlying data generation process it is reasonable to assume that the same subset of features is relevant for those learning tasks. Section 7.6 shows how multitask learning can be implemented using regularization methods. The loss incurred in different learning tasks serves mutual regularization terms in a joint SRM for all learning tasks.

Section 7.7 shows how regularization can be used for **transfer learning**. Like multitask learning also transfer learning exploits relations between different learning tasks. In contrast to multitask learning, which jointly solves the individual learning tasks, transfer learning solves the learning tasks sequentially. The most basic form of transfer learning is to fine tune a pre-trained model. A pre-trained model can be obtained via ERM (4.2) in a ("source") learning task for which we have a large amount of labeled training data. The fine-tuning is then obtained via ERM (4.2) in the ("target") learning task of interest for which we might have only a small amount of labeled training data.

7.1 Structural Risk Minimization

Section 2.2 defined the effective dimension $d_{\text{eff}}(\mathcal{H})$ of a hypothesis space $\mathcal{H}$ as the maximum number of datapoints that can be perfectly fit by some hypothesis $h \in \mathcal{H}$. As soon as the effective dimension of the hypothesis space in (4.2) exceeds the number m of training data points, we can find a hypothesis that perfectly fits the training data. However, a hypothesis that perfectly fits the training data might deliver poor predictions for datapoints outside the

training set (see Figure 7.1).

Modern ML methods use hypothesis spaces with large effective dimensions [88, 15]. Two well-known examples for such methods is linear regression (see Section 3.1) using a large number of features and deep learning with ANNs using a large number (billions) of artificial neurons (see Section 3.11). The effective dimension of these methods can be easily on the order of billions (10^9) if not larger [73]. To avoid overfitting during the naive use of ERM (4.2) we would require a training set containing at least as many datapoints as the effective dimension of the hypothesis space.However, in practice we often do not have access to training sets containing billions of labeled datapoints.

It seems natural to combat overfitting of a ML method by pruning its hypothesis space $\mathcal{H}$. We prune $\mathcal{H}$ by removing some of the hypothesis in $\mathcal{H}$ to obtain the smaller hypothesis space $\mathcal{H}' \subset \mathcal{H}$. We then replace ERM (4.2) with the **restricted ERM**

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}'} \mathcal{E}(h|\mathcal{D}) \text{ with pruned hypothesis space } \mathcal{H}' \subset \mathcal{H}. \tag{7.1}$$

The effective dimension of the pruned hypothesis space $\mathcal{H}'$ is typically much smaller than the effective dimension of the original (large) hypothesis space $\mathcal{H}$, $d_{\text{eff}}(\mathcal{H}') \ll d_{\text{eff}}(\mathcal{H})$. For a given size m of the training set, the risk of overfitting in (7.1) is much smaller than the risk of overfitting in (4.2).

Example. Consider linear regression which the hypothesis space (3.1) constituted by linear maps $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$. The effective dimension of (3.1) is equal to the number of features, $d(\mathcal{H}) = n$. The hypothesis space $\mathcal{H}$ might be too large if we use a large number n of features, leading to overfitting. We prune (3.1) by retaining only linear hypotheses $h(\mathbf{x}) = \left(\mathbf{w}'\right)^T\mathbf{x}$ with weight vectors satisfying $w'_3 = w'_4 = \ldots = w'_n = 0$. Thus, the hypothesis space $\mathcal{H}'$ is constituted by all linear maps that only depend on the first two features x_1, x_2 of a datapoint. The effective dimension of $\mathcal{H}'$ is dimension is $d_{\text{eff}}(\mathcal{H}') = 2$ instead of $d_{\text{eff}}(\mathcal{H}) = n$.

Pruning the hypothesis space is a special case of a more general strategy which we refer to as **structural risk minimization** (SRM) [83]. The idea behind SRM is to modify the training error in ERM (4.2) to favour hypotheses which are more smooth or regular in a specific sense. By enforcing a smooth hypothesis, a ML methods becomes less sensitive, or more robust, to small perturbations of the training datapoints. Section 7.2 discusses the intimate relation between the robustness (against perturbations of the training set) of a ML method and its ability to generalize to datapoints outside the training set.

We measure the smoothness of a hypothesis using a **regularizer** $\mathcal{R}(h) \in \mathbb{R}_+$. Roughly

speaking, the value $\mathcal{R}(h)$ measures the irregularity or variation of a predictor map h. The (design) choice for the regularizer depends on the precise definition of what is meant by regularity or variation of a hypothesis. Section 7.3 discusses how a natural choice for the regularizer $\mathcal{R}(h)$ can arise from a probabilistic model for datapoints.

The SRM approach is obtained from ERM (4.2) by adding the scaled regularizer $\lambda\mathcal{R}(h)$,

$$\begin{aligned}\hat{h} &= \operatorname*{argmin}_{h\in\mathcal{H}} \big[\mathcal{E}(h|\mathcal{D}) + \lambda\mathcal{R}(h)\big] \\ &\overset{(2.12)}{=} \operatorname*{argmin}_{h\in\mathcal{H}} \big[(1/m)\sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) + \lambda\mathcal{R}(h)\big]. \end{aligned} \tag{7.2}$$

We can interpret the regularization term $\lambda\mathcal{R}(h)$ in (7.2) as an estimate (or approximation) for the increase, relative to the training error on $\mathcal{D}$, of the average loss of a hypothesis $\hat{h}$ when it is applied to datapoints outside $\mathcal{D}$. Another interpretation of the term $\lambda\mathcal{R}(h)$ will be discussed in Section 7.3.

The **regularization parameter** λ allows us to trade between a small training error $\mathcal{E}(h^{(\mathbf{w})}|\mathcal{D})$ and small regularization term $\mathcal{R}(h)$, which enforces smoothness or regularity of h. If we choose a large value for λ, irregular or hypotheses h, with large $\mathcal{R}(h)$, are heavily "punished" in (7.2). Thus, increasing the value of λ results in the solution (minimizer) of (7.2) having smaller $\mathcal{R}(h)$. On the other hand, choosing a small value for λ in (7.2) puts more emphasis on obtaining a hypothesis h incurring a small training error. For the extreme case $\lambda = 0$, the SRM (7.2) reduces to ERM (4.2).

The pruning approach (7.1) is intimately related to the SRM (7.2). They are, in a certain sense, **dual** to each other. First, note that (7.2) reduces to the pruning approach (7.1) when using the regularizer $\mathcal{R}(h) = 0$ for all $h \in \mathcal{H}'$, and $\mathcal{R}(h) = \infty$ otherwise, in (7.2). In the other direction, for many important choices for the regularizer $\mathcal{R}(h)$, there is a restriction $\mathcal{H}^{(\lambda)} \subset \mathcal{H}$ such that the solutions of (7.1) and (7.2) coincide (see Figure 7.2). The relation between the optimization problems (7.1) and (7.2) can be made precise using the theory of convex duality (see [13, Ch. 5] and [9]).

For a hypothesis space $\mathcal{H}$ whose elements $h \in \mathcal{H}$ are parameterized by a weight vector $\mathbf{w} \in \mathbb{R}^n$, we can rewrite SRM (7.2) as

$$\begin{aligned}\widehat{\mathbf{w}}^{(\lambda)} &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} \big[\mathcal{E}(h^{(\mathbf{w})}|\mathcal{D}) + \lambda\mathcal{R}(\mathbf{w})\big] \\ &= \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^n} \big[(1/m)\sum_{i=1}^{m} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h^{(\mathbf{w})}) + \lambda\mathcal{R}(\mathbf{w})\big]. \end{aligned} \tag{7.3}$$

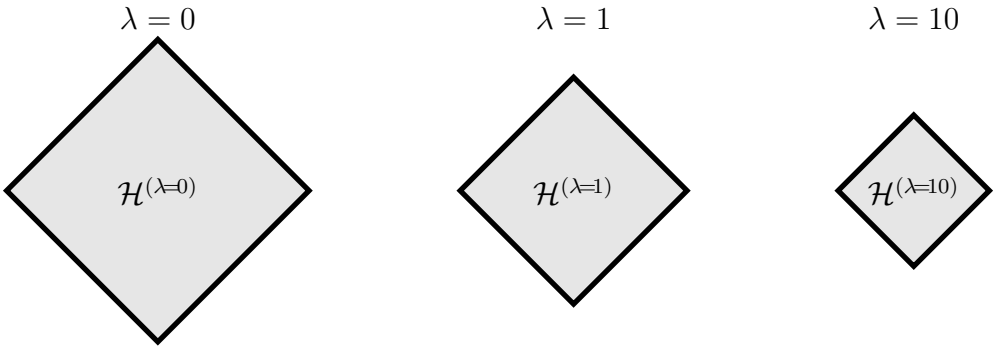


Figure 7.2: Adding the scaled regularizer $\lambda \mathcal{R}(h)$ to the training error in the objective function of SRM (7.2) is equivalent to solving ERM (7.1) with a pruned hypothesis space $\mathcal{H}^{(\lambda)}$.

For the particular choice of squared error loss (2.7), linear hypothesis space (3.1) and regularizer $\mathcal{R}(\mathbf{w}) = \|\mathbf{w}\|_2^2$, SRM (7.3) specializes to

$$\widehat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} \big[(1/m) \sum_{i=1}^{m} \big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)} \big)^2 + \lambda \|\mathbf{w}\|_2^2 \big]. \tag{7.4}$$

The special case (7.4) of SRM (7.3) is known as ridge regression [37]. Another popular special of SRM (7.3) is obtained for the regularizer $\mathcal{R}(\mathbf{w}) = \|\mathbf{w}\|_1$, known as the Lasso [38]

$$\widehat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} \big[(1/m) \sum_{i=1}^{m} \big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)} \big)^2 + \lambda \|\mathbf{w}\|_1 \big]. \tag{7.5}$$

Ridge regression (7.4) and the Lasso (7.5) have fundamentally different computational and statistical properties. Involving a smooth and convex objective function, ridge regression (7.4) can be implemented using efficient GD methods. The objective function of Lasso (7.5) is also convex but non-smooth and therefore requires advanced optimization methods. The increased computational complexity of Lasso (7.5) comes at the benefit of typically delivering a hypothesis with a smaller risk than those obtained from ridge regression [15, 38].

7.2 Robustness

Section 7.1 motivates regularization as a soft variant of model selection. Indeed, the regularization term in SRM (7.2) is equivalent to ERM (7.1) using a pruned (reducing) hypothesis space. We now discuss an alternative view on regularization as a means to make ML methods

robust.

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 labeled datapoints (the training set). These methods learn a hypothesis $h \in \mathcal{H}$ with minimum average loss (training error) incurred for data points in the training set. In practice, the acquisition of label and feature values might be prone to errors. These errors might stem from the measurement device itself (hardware failures or thermal noise) or might be due to human mistakes such as labelling errors.

Let us assume for the sake of exposition that the label values $y^{(i)}$ in the training set are accurate but that the features $\mathbf{x}^{(i)}$ are a perturbed version of the true features of the ith data point. Thus, instead of having observed the data point $\big(\mathbf{x}^{(i)}, y^{(i)}\big)$ we could have equally well observed the data point $\big(\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}, y^{(i)}\big)$ in the training set. Here, we have modelled the perturbations in the features using a RV $\boldsymbol{\varepsilon}$. The probability distribution of the perturbation $\boldsymbol{\varepsilon}$ is a design parameter that controls robustness properties of the overall ML method. We will study a particular choice for this distribution in Section 7.3.

A robust ML method should learn a hypothesis that incurs a small loss not only for a specific data point $\big(\mathbf{x}^{(i)}, y^{(i)}\big)$ but also for perturbed data points $\big(\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}, y^{(i)}\big)$. Therefore, it seems natural to replace the loss $\mathcal{L}(\big(\mathbf{x}^{(i)}, y^{(i)}\big), h)$, incurred on the ith data point in the training set, with the expectation

$$\mathbb{E}\big\{\mathcal{L}(\big(\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}, y^{(i)}\big), h)\big\}. \tag{7.6}$$

The expectation (7.6) is computed using the probability distribution of the perturbation $\boldsymbol{\varepsilon}$. We will show in Section 7.3 that minimizing the average of the expectation (7.6), for $i = 1, \ldots, m$, results precisely in the SRM (7.2).

Using the expected loss (7.6) is not the only approach to make a ML method robust. Another approach to make ML methods robust, referred to as **bagging**, creates a finite number of perturbed copies $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(B)}$ of the original training set $\mathcal{D}$. Such perturbed copies can be obtained by perturbing the features or labels of data points in $\mathcal{D}$ or using the bootstrap method (see Section 6.5 and [37, Ch. 8]). We then use ERM to learn a hypothesis $h^{(b)}$ separately for reach version $\mathcal{D}^{(b)}$, $b = 1, \ldots, B$. A robust ML method can then be obtained by combining or aggregating (e.g., using the average) the predictions $h^{(b)}\big(\mathbf{x}\big)$ delivered by each of the hypotheses $h^{(b)}$, for $b = 1, \ldots, B$. This approach is

7.3 Data Augmentation

ML methods using ERM (4.2) are prone to overfitting as soon as the effective dimension of the hypothesis space $\mathcal{H}$ exceeds the number m of training datapoints. Section 6.3 and Section 7.1 approached this by modifying either the model or the loss function by adding a regularization term. Both approaches prune the hypothesis space $\mathcal{H}$ underlying a ML method to reduce the effective dimension $d_{\rm eff}(\mathcal{H})$. Model selection does this reduction in a discrete fashion while regularization implements a soft "shrinking" of the hypothesis space.

Instead of trying to reduce the effective dimension we could also try to increase the number m of training datapoints used in ERM (4.2). We now discuss how to synthetically generate new labeled datapoints by exploiting known structures that are inherent to a given application domain.

The data arising in many ML applications exhibit intrinsic symmetries and invariances at least in some approximation. The rotated image of a cat still shows a cat. The temperature measurement taken at a given location will be similar to another measurement taken 10 milliseconds later. Data augmentation exploits such symmetries and invariances to augment the raw data with additional synthetic data.

Let us illustrate data augmentation using an application that involves data points characterized by features $\mathbf{x} \in \mathbb{R}^n$ and number labels $y \in \mathbb{R}$. We assume that the data generating process is such that data points with close feature values have the same label. Equivalently, this assumption is requiring the resulting ML method to be robust against small perturbations of the feature values (see Section 7.2). This suggests to augment a data point $\big(\mathbf{x}, y\big)$ by several synthetic data points

$$\big(\mathbf{x} + \boldsymbol{\varepsilon}^{(1)}, y\big), \ldots, \big(\mathbf{x} + \boldsymbol{\varepsilon}^{(B)}, y\big), \tag{7.7}$$

with $\boldsymbol{\varepsilon}^{(1)}, \ldots, \boldsymbol{\varepsilon}^{(B)}$ being realizations of i.i.d. random vectors with the same probability distribution $p(\boldsymbol{\varepsilon})$.

Given a (raw) dataset $\mathcal{D} = \big\{\big(\mathbf{x}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(m)}, y^{(m)}\big)\big\}$ we denote the associated augmented dataset by

$$\begin{aligned}\mathcal{D}' = \big\{&\big(\mathbf{x}^{(1,1)}, y^{(1)}\big), \ldots, \big(\mathbf{x}^{(1,B)}, y^{(1)}\big),\\ &\big(\mathbf{x}^{(2,1)}, y^{(2)}\big), \ldots, \big(\mathbf{x}^{(2,B)}, y^{(2)}\big),\\ &\ldots\\ &\big(\mathbf{x}^{(m,1)}, y^{(m)}\big), \ldots, \big(\mathbf{x}^{(m,B)}, y^{(m)}\big)\big\}.\end{aligned} \tag{7.8}$$

The size of the augmented dataset $\mathcal{D}'$ is $m' = B \times m$. For a sufficiently large augmentation parameter B, the augmented sample size m' is larger than the effective dimension n of the hypothesis space $\mathcal{H}$. We then learn a hypothesis via ERM on the augmented dataset,

$$\begin{aligned}\hat{h} &= \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{E}(h|\mathcal{D}') \\ &\overset{(7.8)}{=} \operatorname*{argmin}_{h \in \mathcal{H}} (1/m') \sum_{i=1}^{m} \sum_{b=1}^{B} \mathcal{L}((\mathbf{x}^{(i,b)}, y^{(i,b)}), h) \\ &\overset{(7.7)}{=} \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} (1/B) \sum_{b=1}^{B} \mathcal{L}((\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}^{(b)}, y^{(i)}), h). \end{aligned} \tag{7.9}$$

We can interpret data-augmented ERM (7.9) as a data-driven form of regularization (see Section 7.1). The regularization is implemented by replacing, for each data point $\big(\mathbf{x}^{(i)}, y^{(i)}\big) \in \mathcal{D}$, the loss $\mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h)$ with the average loss $(1/B) \sum_{b=1}^{B} \mathcal{L}((\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}^{(b)}, y^{(i)}), h)$ over the augmented datapoints that accompany $\big(\mathbf{x}^{(i)}, y^{(i)}\big) \in \mathcal{D}$.

Note that in order to implement (7.9) we need to first generate B realizations $\boldsymbol{\varepsilon}^{(b)} \in \mathbb{R}^n$ of i.i.d. random vectors with common probability distribution $p(\boldsymbol{\varepsilon})$. This might be computationally costly for a large B, n. However, when using a large augmentation parameter B, we might use the approximation

$$(1/B) \sum_{b=1}^{B} \mathcal{L}((\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}^{(b)}, y^{(i)}), h) \approx \mathbb{E}\big\{\mathcal{L}((\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}, y^{(i)}), h)\big\}. \tag{7.10}$$

This approximation is made precise by a key result of probability theory, known as the **law of large numbers**. We obtain an instance of ERM by inserting (7.10) into (7.9),

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^{m} \mathbb{E}\big\{\mathcal{L}((\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}, y^{(i)}), h)\big\}. \tag{7.11}$$

The usefulness of (7.11) as an approximation to the augmented ERM (7.9) depends on the difficulty of computing the expectation $\mathbb{E}\big\{\mathcal{L}((\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}, y^{(i)}), h)\big\}$. The complexity of computing this expectation depends on the choice of loss function and the choice for the probability distribution $p(\boldsymbol{\varepsilon})$.

Let us study (7.11) for the special case linear regression with squared error loss (2.7) and

linear hypothesis space (3.1),

$$\hat{h} = \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}^{(n)}} (1/m) \sum_{i=1}^{m} \mathbb{E}\big\{ \big(y^{(i)} - \mathbf{w}^T \big(\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}\big)\big)^2 \big\}. \tag{7.12}$$

We use perturbations $\boldsymbol{\varepsilon}$ drawn a multivariate normal distribution with zero mean and covariance matrix $\sigma^2 \mathbf{I}$,

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I}). \tag{7.13}$$

We develop (7.12) further by using

$$\mathbb{E}\{\big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)}\big)\boldsymbol{\varepsilon}\} = \mathbf{0}. \tag{7.14}$$

The identity (7.14) uses that the datapoints $\big(\mathbf{x}^{(i)}, y^{(i)}\big)$ are fixed and known (deterministic) while $\boldsymbol{\varepsilon}$ is a zero-mean random vector. Combining (7.14) with (7.12),

$$\begin{aligned} \mathbb{E}\big\{ \big(y^{(i)} - \mathbf{w}^T \big(\mathbf{x}^{(i)} + \boldsymbol{\varepsilon}\big)\big)^2 \big\} &= \big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)}\big)^2 + \big\|\mathbf{w}\big\|_2^2 \mathbb{E}\big\{ \big\|\boldsymbol{\varepsilon}\big\|_2^2 \big\} \\ &= \big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)}\big)^2 + n\big\|\mathbf{w}\big\|^2 \sigma^2. \end{aligned} \tag{7.15}$$

where the last step used $\mathbb{E}\big\{ \big\|\boldsymbol{\varepsilon}\big\|_2^2 \big\} \overset{(7.13)}{=} n\sigma^2$. Inserting (7.15) into (7.12),

$$\hat{h} = \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}^{(n)}} (1/m) \sum_{i=1}^{m} \big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)}\big)^2 + n\big\|\mathbf{w}\big\|^2 \sigma^2. \tag{7.16}$$

We have obtained (7.16) as an approximation of the augmented ERM (7.9) for the special case of squared error loss (2.7) and the linear hypothesis space (3.1). This approximation uses the **law of large numbers** (7.10) and becomes more accurate for increasing augmentation parameter B.

Note that (7.16) is nothing but ridge regression (7.4) using the regularization parameter $\lambda = n\sigma^2$. Thus, we can interpret ridge regression as implicit data augmentation (7.8) by applying random perturbations (7.7) to the feature vectors in the original training set $\mathcal{D}$.

The regularizer $\mathcal{R}(\mathbf{w}) = \|\mathbf{w}\|_2^2$ in (7.16) arose naturally from the specific choice for the probability distribution (7.13) of the random perturbation $\boldsymbol{\varepsilon}^{(i)}$ in (7.7) and using the squared error loss. Other choices for this probability distribution or the loss function result in different regularizers.

Augmenting data points with random perturbations distributed according (7.13) treat the

features of a data point independently. For application domains that generate data points with highly correlated features it might be useful to augment data points using random perturbations $\boldsymbol{\varepsilon}$ (see (7.7)) distributed as

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{C}). \tag{7.17}$$

The covariance matrix $\mathbf{C}$ of the perturbation $\boldsymbol{\varepsilon}$ can be chosen using domain expertise or estimated (see Section 7.5). Inserting the distribution (7.17) into (7.11),

$$\hat{h} = \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}^{(n)}} \left[(1/m) \sum_{i=1}^{m} \left(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)} \right)^2 + \mathbf{w}^T \mathbf{C} \mathbf{w} \right]. \tag{7.18}$$

Note that (7.18) reduces to ordinary ridge regression (7.16) for the choice $\mathbf{C} = \sigma^2 \mathbf{I}$.

7.4 A Probabilistic Analysis of Regularization

The goal of this section is to develop a better understanding for the effect of the regularization term in SRM (7.3). We will analyze the solutions of ridge regression (7.4) which is the special case of SRM using the linear hypothesis space (3.1) and squared error loss (2.7). Using the feature matrix $\mathbf{X} = \left(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\right)^T$ and label vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(m)})^T$, we can rewrite (7.4) more compactly as

$$\widehat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} \left[(1/m) \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 \right]. \tag{7.19}$$

The solution of (7.19) is given by

$$\widehat{\mathbf{w}}^{(\lambda)} = (1/m)((1/m)\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}. \tag{7.20}$$

For $\lambda = 0$, (7.20) reduces to the formula (6.17) for the optimal weights in linear regression (see (7.4) and (4.4)). Note that for $\lambda > 0$, the formula (7.20) is always valid, even when $\mathbf{X}^T\mathbf{X}$ is singular (not invertible). For $\lambda > 0$ the optimization problem (7.19) (and (7.4)) has the unique solution (7.20).

To study the statistical properties of the predictor $h^{(\widehat{\mathbf{w}}^{(\lambda)})}(\mathbf{x}) = \left(\widehat{\mathbf{w}}^{(\lambda)}\right)^T \mathbf{x}$ (see (7.20)) we use the probabilistic toy model (6.13), (6.15) and (6.16) that we used already in Section 6.4. We interpret the training data $\mathcal{D}^{(\text{train})} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ as realizations of i.i.d. RVs whose distribution is defined by (6.13), (6.15) and (6.16).

We can then define the average prediction error of ridge regression as

$$\mathcal{E}_{\text{pred}}^{(\lambda)} := \mathbb{E}\Big\{\Big(y - h^{(\widehat{\mathbf{w}}^{(\lambda)})}(\mathbf{x})\Big)^2\Big\}. \tag{7.21}$$

As shown in Section 6.4, the error $\mathcal{E}_{\text{pred}}^{(\lambda)}$ is the sum of three components: the bias, the variance and the noise variance σ^2 (see (6.27)). The bias of $\widehat{\mathbf{w}}^{(\lambda)}$ is

$$B^2 = \big\|(\mathbf{I} - \mathbb{E}\{(\mathbf{X}^T\mathbf{X} + m\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{X}\})\overline{\mathbf{w}}\big\|_2^2. \tag{7.22}$$

For sufficiently large size m of the training set, we can use the approximation

$$\mathbf{X}^T\mathbf{X} \approx m\mathbf{I} \tag{7.23}$$

such that (7.22) can be approximated as

$$\begin{aligned} B^2 &\approx \big\|(\mathbf{I}-(\mathbf{I}+\lambda\mathbf{I})^{-1})\overline{\mathbf{w}}\big\|_2^2 \\ &= \sum_{l=1}^{n} \frac{\lambda}{1+\lambda}\overline{w}_l^2. \end{aligned} \tag{7.24}$$

Let us compare the (approximate) bias term (7.24) of regularized linear regression with the bias term (6.23) of ordinary linear regression (which is the extreme case of ridge regression with $\lambda = 0$). The bias term (7.24) increases with increasing regularization parameter λ in ridge regression (7.4). In many relevant settings, the increase in bias is outweighed by the reduction in variance. The variance typically decreases with increasing λ as shown next.

The variance of ridge regression (7.4) satisfies

$$\begin{aligned} V = (\sigma^2/m^2)\times \\ \text{tr}\big\{\mathbb{E}\{((1/m)\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{X}((1/m)\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\}\big\}. \end{aligned} \tag{7.25}$$

Inserting the approximation (7.23) into (7.25),

$$V \approx \sigma^2(1/m)(n/(1+\lambda)). \tag{7.26}$$

According to (7.26), the variance of $\widehat{\mathbf{w}}^{(\lambda)}$ decreases with increasing regularization parameter λ of ridge regression (7.4). This is the opposite behaviour as observed for the bias (7.24), which increases with increasing λ. The approximate variance formula (7.26) suggests to

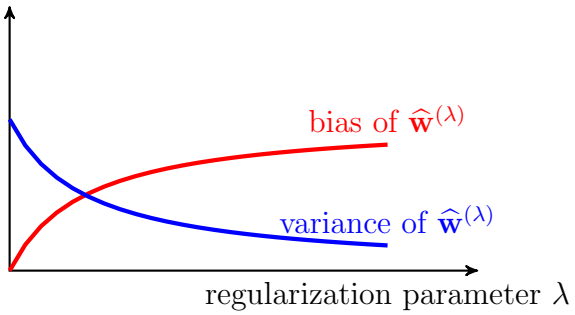


Figure 7.3: The bias and variance of regularized linear regression depend on the regularization parameter λ in an opposite manner resulting in a bias-variance trade-off.

interpret the ratio $(n/(1+\lambda))$ as the effective number of features used by ridge regression. Increasing the regularization parameter λ decreases the effective number of features.

Figure 7.3 illustrates the trade-off between the bias B^2 (7.24) of ridge regression, which increases for increasing λ, and the variance V (7.26) which decreases with increasing λ. Note that we have seen another example for a bias-variance trade-off in Section 6.4. This trade-off was traced out by a discrete (model complexity) parameter $r \in \{1, 2, \ldots\}$ (see (6.14)). In stark contrast to discrete model selection, the bias-variance trade-off for ridge regression is traced out by the continuous regularization parameter $\lambda \in \mathbb{R}_+$.

The main statistical effect of the regularization term in ridge regression is to balance the bias with the variance to minimize the average prediction error of the learnt hypothesis. There is also a computational effect or adding a regularization term. Roughly speaking, the regularization term serves as a pre-conditioning of the optimization problem and, in turn, reduces the computational complexity of solving ridge regression (7.19).

The objective function in (7.19) is a smooth (infinitely often differentiable) convex function. We can therefore use GD to solve (7.19) efficiently (see Chapter 5). Algorithm 7 summarizes the application of GD to (7.19). The computational complexity of Algorithm 7 depends crucially on the number of GD iterations required to reach a sufficiently small neighbourhood of the solutions to (7.19). 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 (7.19) as the quadratic

problem

$$\min_{\mathbf{w}\in\mathbb{R}^n} \underbrace{(1/2)\mathbf{w}^T\mathbf{Q}\mathbf{w} - \mathbf{q}^T\mathbf{w}}_{=f(\mathbf{w})}$$
$$\text{with } \mathbf{Q} = (1/m)\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}, \mathbf{q} = (1/m)\mathbf{X}^T\mathbf{y}. \tag{7.27}$$

This is similar to the quadratic optimization problem (4.8) underlying linear regression but with a different matrix $\mathbf{Q}$. The computational complexity (number of iterations) required by GD (see (5.4)) applied to solve (7.27) up to a prescribed accuracy depends crucially on the condition number $\kappa(\mathbf{Q}) \geq 1$ of the **psd** matrix $\mathbf{Q}$ [45]. The smaller the condition number $\kappa(\mathbf{Q})$, the fewer iterations are required by GD. A matrix with small condition number is also referred to as being "well-conditioned".

The condition number of the matrix $\mathbf{Q}$ in (7.27) is given by

$$\kappa(\mathbf{Q}) = \frac{\lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X}) + \lambda}{\lambda_{\min}((1/m)\mathbf{X}^T\mathbf{X}) + \lambda}. \tag{7.28}$$

According to (7.28), the condition number tends to one for increasing regularization parameter λ,

$$\lim_{\lambda\to\infty} \frac{\lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X}) + \lambda}{\lambda_{\min}((1/m)\mathbf{X}^T\mathbf{X}) + \lambda} = 1. \tag{7.29}$$

Thus, the number of required GD iterations in Algorithm 7 decreases with increasing regularization parameter λ.

Algorithm 7 "Regularized Linear Regression via GD"

Input: dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$; 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** stopping criterion met
Output: $\mathbf{w}^{(k)}$ (which approximates $\widehat{\mathbf{w}}^{(\lambda)}$ in (7.19))

Let us finally point out a close relation between regularization, which amounts to including the additive term $\lambda\|\mathbf{w}\|_2^2$ to the objective function in (7.3), and model selection (see Section

6.3). The regularized ERM (7.3) can be shown (see [9, Ch. 5]) to be equivalent to

$$\widehat{\mathbf{w}}^{(\lambda)} = \operatorname*{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}^{(\lambda)}} (1/m) \sum_{i=1}^{m} (y^{(i)} - h^{(\mathbf{w})}(\mathbf{x}^{(i)}))^2 \tag{7.30}$$

with the restricted hypothesis space

$$\begin{aligned} \mathcal{H}^{(\lambda)} := \{ & h^{(\mathbf{w})} : \mathbb{R}^n \to \mathbb{R} : h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \\ & \text{, with some weights } \mathbf{w} \text{ satisfying } \|\mathbf{w}\|_2^2 \leq C(\lambda) \} \subset \mathcal{H}^{(n)}. \end{aligned} \tag{7.31}$$

For any given value λ, there is a number $C(\lambda)$ such that solutions of (7.3) coincide with the solutions of (7.30). We can interpret regularized ERM (7.3) as a form of model selection using a continuous ensemble of hypothesis spaces $\mathcal{H}^{(\lambda)}$ (7.31). In contrast, the simple model selection strategy considered in Section 6.3 uses a discrete sequence of hypothesis spaces.

7.5 Semi-Supervised Learning

Consider the task of predicting the numeric label y of a data point $\mathbf{z} = \big(\mathbf{x}, y\big)$ based on its feature vector $\mathbf{x} = \big(x_1, \ldots, x_n\big)^T \in \mathbb{R}^n$. At our disposal are two datasets $\mathcal{D}^{(u)}$ and $\mathcal{D}^{(l)}$. For each datapoint in $\mathcal{D}^{(u)}$ we only know the feature vector. We therefore refer to $\mathcal{D}^{(u)}$ as "unlabelled data". For each datapoint in $\mathcal{D}^{(l)}$ we know both, the feature vector $\mathbf{x}$ and the label y. We therefore refer to $\mathcal{D}^{(l)}$ as "labeled data".

SSL methods exploit the information provided by unlabelled data $\mathcal{D}^{(u)}$ to support the learning of a hypothesis based on minimizing its empirical risk on the labelled (training) data $\mathcal{D}^{(l)}$. The success of SSL methods depends on the statistical properties of the data generated within a given application domain. Loosely speaking, the information provided by the probability distribution of the features must be relevant for the ultimate task of predicting the label y from the the features $\mathbf{x}$ [19].

Let us design a SSL method, summarized in Algorithm 8 below, using the data augmentation perspective from Section 7.3. The idea is the augment the (small) labeled dataset $\mathcal{D}^{(l)}$ by adding random perturbations fo the features vectors of data point in $\mathcal{D}^{(l)}$. This is reasonable for applications where feature vectors are subject to inherent measurement or modelling errors. Given a data point with vector $\mathbf{x}$ we could have equally well observed a feature vector $\mathbf{x} + \boldsymbol{\varepsilon}$ with some small random perturbation $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{C})$. To estimate the covariance matrix $\mathbf{C}$, we use the sample covariance matrix of the feature vectors in the (large) unlabelled

dataset $\mathcal{D}^{(u)}$. We then learn a hypothesis using the augmented (regularized) ERM (7.18).

Algorithm 8 A Semi-Supervised Learning Algorithm

Input: labeled dataset $\mathcal{D}^{(l)} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$; unlabeled dataset $\mathcal{D}^{(u)} = \{\widetilde{\mathbf{x}}^{(i)}\}_{i=1}^{m'}$

1: compute $\mathbf{C}$ via sample covariance on $\mathcal{D}^{(u)}$,

$$\mathbf{C} := (1/m') \sum_{i=1}^{m'} \big(\widetilde{\mathbf{x}}^{(i)} - \widehat{\mathbf{x}}\big)\big(\widetilde{\mathbf{x}}^{(i)} - \widehat{\mathbf{x}}\big)^T \text{ with } \widehat{\mathbf{x}} := (1/m') \sum_{i=1}^{m'} \widetilde{\mathbf{x}}^{(i)}. \tag{7.32}$$

2: compute (e.g. using GD)

$$\widehat{\mathbf{w}} := \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^n} \Big[(1/m) \sum_{i=1}^{m} \big(y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)}\big)^2 + \mathbf{w}^T \mathbf{C} \mathbf{w} \Big]. \tag{7.33}$$

Output: hypothesis $\hat{h}(\mathbf{x}) = \big(\widehat{\mathbf{w}}\big)^T \mathbf{x}$

7.6 Multitask Learning

We can identify a learning task with the loss function $\mathcal{L}((\mathbf{x}, y), h)$ that is used to measure the quality of a particular hypothesis $h \in \mathcal{H}$. Note that the loss obtained for a given datapoint also depends on the definition for the label of a datapoint. For the same datapoints, we obtain different learning tasks from different choices or definitions for the label of a data point. Multitask learning exploits the similarities between different learning tasks to jointly solve them.

Example. Consider a data point $\mathbf{z}$ representing a hand-drawing that is collected via the online game `https://quickdraw.withgoogle.com/`. The features of a data point are the pixel intensities of the bitmap which is used to store the hand-drawing. As label we could use the fact if a hand-drawing shows an apple or not. This results in the learning task $\mathcal{L}^{(1)}$. Another choice for the label of a hand-drawing could be the fact if a hand-drawing shows a fruit at all or not. This results in another learning task $\mathcal{L}^{(2)}$ which is similar but different from the task $\mathcal{L}^{(1)}$.

The idea of multitask learning is that a reasonable hypothesis h for a learning task should also do well for a related learning tasks. Thus, we can use the loss incurred on similar learning tasks as a regularization term for learning a hypothesis for the learning task at hand. Algorithm 9 is a straightforward implementation of this idea for a given dataset that gives rise to T related learning tasks $\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(T)}$. For each individual learning task $\mathcal{L}^{(t')}$

it uses the loss on the remaining learning tasks $\mathcal{L}^{(t)}$, with $t \neq t'$, as regularization term in (7.35).

Algorithm 9 A Multitask Learning Algorithm

Input: dataset $\mathcal{D} = \{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ with T associated learning tasks with loss functions $\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(T)}$, hypothesis space $\mathcal{H}$

1: learn a hypothesis $\hat{h}$ via

$$\hat{h} := \operatorname*{argmin}_{h \in \mathcal{H}} \sum_{t=1}^{T} \sum_{i=1}^{m} \mathcal{L}^{(t)}\big(\mathbf{z}^{(i)}, h\big). \tag{7.34}$$

Output: hypothesis $\hat{h}$

The applicability of Algorithm 9 is somewhat limited as it aims at finding a single hypothesis that does well for all T learning tasks simultaneously. For certain application domains it might be more reasonable to not learn a single hypothesis for all learning tasks but to learn a separate hypothesis $h^{(t)}$ for each learning task $t = 1, \ldots, T$. However, these separate hypotheses typically might still share some structural similarities.[1] We can enforce different notion of similarities between the hypotheses $h^{(t)}$ by adding a regularization term to the loss functions of the tasks.

Algorithm 10 generalizes Algorithms 9 by learning a separate hypothesis for each task t while requiring these hypotheses to be structurally similar. The structural (dis-)similarity between the hypotheses is measured by a regularization term $\mathcal{R}$.

Algorithm 10 A Multitask Learning Algorithm

Input: dataset $\mathcal{D} = \{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ with T associated learning tasks with loss functions $\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(T)}$, hypothesis space $\mathcal{H}$

1: learn a hypothesis $\hat{h}$ via

$$\hat{h}^{(1)}, \ldots, \hat{h}^{(T)} := \operatorname*{argmin}_{h^{(1)}, \ldots, h^{(T)} \in \mathcal{H}} \sum_{t=1}^{T} \sum_{i=1}^{m} \mathcal{L}^{(t)}\big(\mathbf{z}^{(i)}, h^{(t)}\big) + \lambda \mathcal{R}\big(h^{(1)}, \ldots, h^{(T)}\big). \tag{7.35}$$

Output: hypotheses $\hat{h}^{(1)}, \ldots, \hat{h}^{(T)}$

[1] One important example for such a structural similarity in the case of linear predictors $h^{(t)}(\mathbf{x}) = \big(\mathbf{w}^{(t)}\big)^T \mathbf{x}$is that the weight vectors $\mathbf{w}^{(T)}$ have a small joint support. Requiring the weight vectors to have a small joint support is equivalent to requiring the stacked vector $\widetilde{\mathbf{w}} = \big(\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(T)}\big)$ to be block (group) sparse [26].

7.7 Transfer Learning

Regularization is also instrumental for transfer learning to capitalize on synergies between different related learning tasks [66, 40]. Transfer learning is enabled by constructing regularization terms for a learning task by using the result of a previous leaning task. While multitask learning methods solve many related learning tasks simultaneously, transfer learning methods operate in a sequential fashion.

To illustrate the idea of transfer learning consider two learning tasks which differ in their intrinsic difficulty. We consider a learning task to be easy if it involves if we can easily gather large amounts of labeled (training) data for that task. Consider the learning task $\mathcal{L}^{(1)}$ of predicting whether an image shows a cat or not. For this learning task we can easily gather a large training set $\mathcal{D}^{(1)}$ using via image collections of animals. Another (related) learning task $\mathcal{L}^{(2)}$ is to predicting whether an image shows a cat of a particular breed, with a particular body height and with a specific age, we might not be able to collect many labeled datapoints.

7.8 Exercises

7.8.1 Ridge Regression as Quadratic Form

Consider the 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 the 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 ?

7.8.2 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 to split the dataset into training and validation set of the same size (two). Then we consider all models that consists of linear hypotheses with weight vectors having at most two non-zero weights. Each of these models corresponds to a different subset of two weights that might be non-zero. Each of these models corresponds to a different subset of two features that are linearly combined to obtain a predicted label value. We can use their validation errors (see Algorithm 4) to choose between these models. The second approach is to learn a linear hypothesis with an arbitrary weight vector $\mathbf{w}$ but using regularization (ridge).

Chapter 8

Clustering

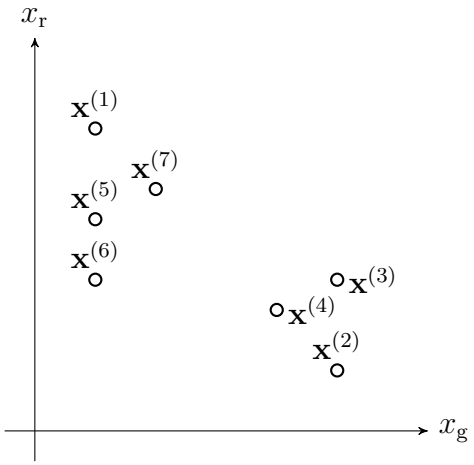


Figure 8.1: Each circle represents an image which is characterized by its average redness $x_{\rm r}$ and average greenness $x_{\rm g}$. The i-th image is depicted by a circle located at the point $\mathbf{x}^{(i)} = \big(x_{\rm r}^{(i)}, x_{\rm g}^{(i)}\big)^T \in \mathbb{R}^2$. It seems that the images can be grouped into two clusters.

So far we focused on ML methods that use the ERM principle and lean a hypothesis by minimizing the discrepancy between its predictions and the true labels on a training set. These methods are referred to as supervised methods as they require labeled datapoints for which the true label values have been determined by some human (who serves as a "supervisor"). This and the following chapter discus ML methods which do not require any labeled datapoint. These methods are often referred to as "unsupervised" since they do not require a supervisor to provide the label values for any datapoint.

The basic idea of clustering is that the data points arising in a ML application can be

decomposed into few subsets which we refer to as **clusters**. Clustering methods learn a hypothesis for assigning each data point either to one cluster (see Section 8.1) or several clusters with different degrees of belonging (see Section 8.2). Two data points are assigned to the same cluster if they are similar to each other. Different clustering methods use different measures for the "similarity" between data points. For data points characterized by (numeric) Euclidean feature vectors, the similarity between data points can be naturally defined in terms of the Euclidean distance between feature vectors. Section 8.3 discusses clustering methods that use notions of similarity which do not require to characterize data points by Euclidean feature vectors.

There is a strong conceptual link between clustering methods and the classification methods discussed in Chapter 3. Both type of methods learn a hypothesis that reads in the features of a data point an outputs a prediction for some quantity of interest. In classification methods, this quantity of interest is some generic label of a data point. For clustering methods, this quantity of interest is the index of the cluster to which a data point belongs to. A main difference between clustering and classification is that clustering methods do not require knowledge of the true label (cluster index) of any data point.

Classification methods learn a good hypothesis via minimizing their average loss incurred on a training set of labeled data points. In contrast, clustering methods do not have access to a single labeled data point. To find the correct labels (cluster assignments) clustering methods rely solely on the intrinsic geometry of the datapoints. We will see that clustering methods use this intrinsic geometry to define an empirical risk incurred by a candidate hypothesis. Like classification methods, also clustering methods use an instance of the ERM principle (see Chapter 4) to find a good hypothesis (clustering).

This chapter discusses two main flavours of clustering methods:

- hard clustering (see Section 8.1)
- and soft clustering methods (see Section 8.2).

Hard clustering methods learn a hypothesis h that reads in the feature vector $\mathbf{x}$ of a datapoint and delivers a predicted cluster index $\hat{y} = h(\mathbf{x}) \in \{1, \ldots, k\}$. Thus, hard clustering assigns each data point to one single cluster. Section 8.1 will discuss one of the most widely-used hard clustering algorithms which is known as k-means.

In contrast to hard clustering methods, soft clustering methods assign each data point to several clusters with different degrees of belonging. These methods learn a hypothesis that delivers a vector $\hat{\mathbf{y}} = \big(\hat{y}_1, \ldots, \hat{y}_k\big)^T$ with entry $\hat{y}_c \in [0,1]$ being the predicted degree of

the datapoint belonging to the cluster with index c. Hard clustering is an extreme case of soft-clustering with requiring degrees of belonging taking values in $\{0, 1\}$ and allowing only one of them to be non-zero.

The main focus of this chapter is on methods that require data points being represented by numeric feature vectors (see Sections 8.1 and 8.2). These methods define the similarity between data points using the Euclidean distance between their feature vectors. Some applications generate data points for which it is not obvious how to obtain numeric feature vectors such that their Euclidean distances reflect the similarity between data points. It is then desirable to use a more flexible notion of similarity which does not require to determine (useful) numeric feature vectors of data points. Maybe the most fundamental concept to represent similarities between data points is a similarity graph. The nodes of the similarity graph are the individual data points of a dataset. Similar data points are connected by edges (links) that might be assigned some weight that quantities the amount of similarity. Section 8.3 discusses clustering methods that use a graph to represent similarities between data points.

8.1 Hard Clustering with k-Means

Consider a dataset $\mathcal{D}$ which consists m datapoints indexed by $i = 1, \ldots, m$. We can access the data points only via their numeric feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$, for $i = 1, \ldots, m$. It is convenient to identify a data point with its feature vector, so that we refer by $\mathbf{x}^{(i)}$ to the i-th data point. The goal of hard clustering is to decompose the dataset into a given number k of different clusters $\mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(k)}$. Hard clustering aims at assigning each data point $\mathbf{x}^{(i)}$ to one and only one cluster $\mathcal{C}^{(c)}$ with some cluster index $c \in \{1, \ldots, k\}$.

Let us define for each data point its label $y^{(i)} \in \{1, \ldots, k\}$ as the index of the cluster to which the ith datapoint belongs to. The cth cluster consists of all datapoints with $y^{(i)} = c$,

$$\mathcal{C}^{(c)} := \big\{ i \in \{1, \ldots, m\} : y^{(i)} = c \big\}. \tag{8.1}$$

We now discuss a clustering method that computes predictions $\hat{y}^{(i)}$ for the cluster assignments $y^{(i)}$. The predicted cluster assignments result in the delivered clusters

$$\widehat{\mathcal{C}}^{(c)} := \big\{ i \in \{1, \ldots, m\} : \hat{y}^{(i)} = c \big\}, \text{ for } c = 1, \ldots, k. \tag{8.2}$$

This clustering method, known as **k-means**, does not require the knowledge of the label or (true) cluster assignment $y^{(i)}$ for any datapoint in $\mathcal{D}$. The predictions $\hat{y}^{(i)}$ are determined solely from the intrinsic geometry of the feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ for all $i = 1, \ldots, m$. Note that k-means requires the number k of clusters to be given as an input (or hyper) parameter.

The key idea of k-means is to represent the c-th cluster $\widehat{\mathcal{C}}^{(c)}$ by a representative feature vector $\boldsymbol{\mu}^{(c)} \in \mathbb{R}^n$. It seems reasonable to assign datapoints in $\mathcal{D}$ to clusters $\widehat{\mathcal{C}}^{(c)}$ such that they are well concentrated around the cluster representatives $\boldsymbol{\mu}^{(c)}$. We make this informal requirement precise by defining the **clustering error**

$$\mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}, \{\hat{y}^{(i)}\}_{i=1}^{m} \mid \mathcal{D}\big) = (1/m) \sum_{i=1}^{m} \Big\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(\hat{y}^{(i)})} \Big\|^2. \tag{8.3}$$

Note that the clustering error $\mathcal{E}$ (8.3) depends on both, the cluster assignments $\hat{y}^{(i)}$, which define the cluster (8.2), and the cluster representatives $\boldsymbol{\mu}^{(c)}$, for $c = 1, \ldots, k$.

Finding the optimal cluster means $\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}$ and cluster assignments $\{\hat{y}^{(i)}\}_{i=1}^{m}$ that minimize the clustering error (8.3) is computationally challenging. The difficulty stems from the fact that the clustering error is a non-convex function of the cluster means and assignments. While jointly optimizing the cluster means and assignments is hard, separately optimizing

either the cluster means for given assignments or vice-versa is easy. In what follows, we present simple closed-form solutions for these sub-problems. The k-means method simply combines these solutions in an alternating fashion.

It can be shown that for given predictions (cluster assignments) $\hat{y}^{(i)}$, the clustering error (8.3) is minimized by setting the cluster representatives equal to the **cluster means** [11]

$$\boldsymbol{\mu}^{(c)} := \big(1/|\widehat{\mathcal{C}}^{(c)}|\big) \sum_{\hat{y}^{(i)}=c} \mathbf{x}^{(i)}. \tag{8.4}$$

To evaluate (8.4) we need to know the predicted cluster assignments $\hat{y}^{(i)}$. The crux is that the optimal predictions $\hat{y}^{(i)}$, in the sense of minimizing clustering error (8.3), depend themselves on the choice for the cluster representatives $\boldsymbol{\mu}^{(c)}$. In particular, for given cluster representative $\boldsymbol{\mu}^{(c)}$ with $c = 1, \ldots, k$, the clustering error is minimized by the cluster assignments

$$\hat{y}^{(i)} \in \operatorname*{argmin}_{c \in \{1,\ldots,k\}} \big\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c)}\big\|. \tag{8.5}$$

Here, we denote by $\operatorname*{argmin}_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c')}\|$ the set of all cluster indices $c \in \{1, \ldots, k\}$ such that $\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c)}\| = \min_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c')}\|$.

Note that (8.5) assigns the ith datapoint to those cluster $\mathcal{C}^{(c)}$ whose cluster mean $\boldsymbol{\mu}^{(c)}$ is nearest (in Euclidean distance) to $\mathbf{x}^{(i)}$. Thus, if we knew the optimal cluster representatives, we could predict the cluster assignments using (8.5). However, we do not know the optimal cluster representatives unless we have found good predictions for the cluster assignments $\hat{y}^{(i)}$ (see (8.4)).

To recap: We have characterized the optimal choice (8.4) for the cluster representatives for given cluster assignments and the optimal choice (8.5) for the cluster assignments for given cluster representatives. It seems natural, starting from some initial guess for the cluster representatives, to alternate between the cluster assignment update (8.5) and the update (8.4) for the cluster means. This alternating optimization strategy is illustrated in Figure 8.2 and summarized in Algorithm 11. Note that Algorithm 11, which is maybe the most basic variant of k-means, simply alternates between the two updates (8.4) and (8.5) until some stopping criterion is satisfied.

Algorithm 11 requires the specification of the number k of clusters and initial choices for the cluster means $\boldsymbol{\mu}^{(c)}$, for $c = 1, \ldots, k$. Those quantities are hyper-parameters that must be tuned to the specific geometry of the given dataset $\mathcal{D}$. This tuning can be based on probabilistic models for the dataset and its cluster structure (see Section 2.1.4 and [49, 87]).

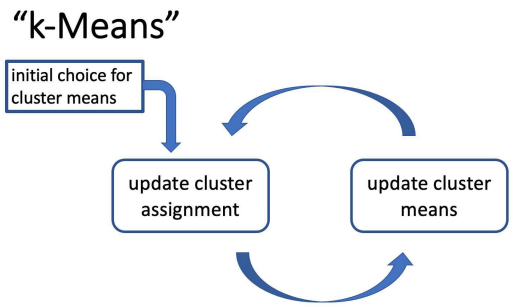


Figure 8.2: The flow of k-means. Starting from an initial guess or estimate for the cluster means, the cluster assignments and cluster means are updated (improved) in an alternating fashion.

Algorithm 11 “k-means”

Input: dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$; number k of clusters; initial cluster means $\boldsymbol{\mu}^{(c)}$ for $c = 1, \ldots, k$.

1: **repeat**

2: for each datapoint $\mathbf{x}^{(i)}$, $i=1, \ldots, m$, do

$$\hat{y}^{(i)} := \underset{c' \in \{1,\ldots,k\}}{\operatorname{argmin}} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c')}\| \quad \text{(update cluster assignments)} \tag{8.6}$$

3: for each cluster $c=1, \ldots, k$ do

$$\boldsymbol{\mu}^{(c)} := \frac{1}{|\{i : \hat{y}^{(i)} = c\}|} \sum_{i:\hat{y}^{(i)}=c} \mathbf{x}^{(i)} \quad \text{(update cluster means)} \tag{8.7}$$

4: **until** stopping criterion is met

5: compute final clustering error $\mathcal{E}^{(k)} := (1/m) \sum_{i=1}^{m} \left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(\hat{y}^{(i)})} \right\|^2$

Output: cluster means $\boldsymbol{\mu}^{(c)}$, for $c = 1, \ldots, k$, cluster assignments $\hat{y}^{(i)} \in \{1, \ldots, k\}$, for $i = 1, \ldots, m$, final clustering error $\mathcal{E}^{(k)}$

Alternatively, if Algorithm 11 is used as pre-processing within an overall supervised ML method (see Chapter 3), the validation error (see Section 6.3) of the overall method might guide the choice of the number k of clusters.

Choosing Number of Clusters. The choice for the number k of clusters typically depends on the role of the clustering method 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 then pick the value of k which results in the smallest validation error. If the clustering method is mainly used as a tool for data visualization, we might prefer a small number of clusters. The choice for the number k of clusters can also be guided by the so-called "elbow-method". Here, we run the k-means Algorithm 11 for several different choices of k. For each k, Algorithm 11 delivers a clustering with clustering error

$$\mathcal{E}^{(k)} = \mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}, \{\hat{y}^{(i)}\}_{i=1}^{m} \mid \mathcal{D}\big).$$

We then plot the minimum empirical error $\mathcal{E}^{(k)}$ as a function of the number k of clusters. Figure 8.3 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. Note that for $k \geq m$ we can achieve zero clustering error since each datapoint $\mathbf{x}^{(i)}$ can be assigned to a separate cluster $\mathcal{C}^{(c)}$ whose mean coincides with that datapoint, $\mathbf{x}^{(i)} = \boldsymbol{\mu}^{(c)}$.

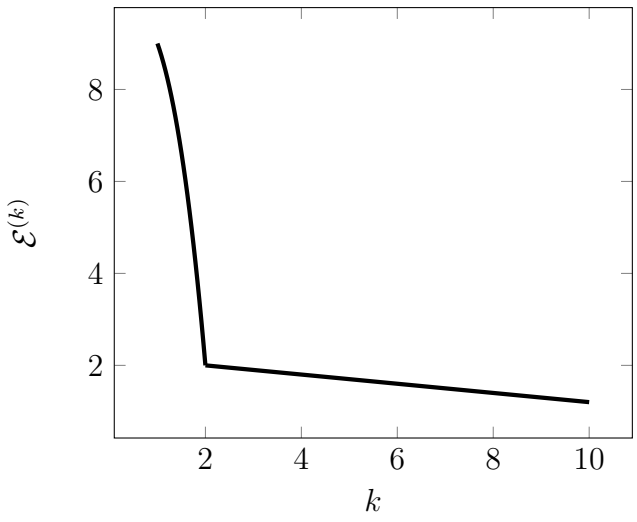


Figure 8.3: The clustering error $\mathcal{E}^{(k)}$ achieved by k-means for increasing number k of clusters.

Cluster-Means Initialization. We briefly mention some popular strategies for choosing

the initial cluster means in Algorithm 11. One option is to initialize the cluster means with realizations of i.i.d. random vectors whose probability distribution is matched to the dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$ (see Section 3.12). For example, we could use a multivariate normal distribution $\mathcal{N}(\mathbf{x}; \widehat{\boldsymbol{\mu}}, \widehat{\boldsymbol{\Sigma}})$ with the sample mean $\widehat{\boldsymbol{\mu}} = (1/m) \sum_{i=1}^{m} \mathbf{x}^{(i)}$ and the sample covariance $\widehat{\boldsymbol{\Sigma}} = (1/m) \sum_{i=1}^{m} (\mathbf{x}^{(i)} - \widehat{\boldsymbol{\mu}})(\mathbf{x}^{(i)} - \widehat{\boldsymbol{\mu}})^{T}$. Alternatively, we could choose the initial cluster means $\boldsymbol{\mu}^{(c)}$ by selecting k different data points $\mathbf{x}^{(i)}$ from $\mathcal{D}$. This selection process might combine random choices with an optimization of the distances between cluster means [3]. Finally, the cluster means might also be chosen by evenly partitioning the principal component of the dataset (see Chapter 9).

Interpretation as ERM. For a practical implementation of Algorithm 11 we need to decide when to stop updating the cluster means and assignments (see (8.6) and (8.7)). To this end it is useful to interpret Algorithm 11 as a method for iteratively minimizing the clustering error (8.3). As can be verified easily, the updates (8.6) and (8.7) always modify (update) the cluster means or assignments in such a way that the clustering error (8.3) is never increased. Thus, each new iteration of Algorithm 11 results in cluster means and assignments with a smaller (or the same) clustering error compared to the cluster means and assignments obtained after the previous iteration. Algorithm 11 implements a form of ERM (see Chapter 4) using the clustering error (8.3) as the empirical risk incurred by the predicted cluster assignments $\hat{y}^{(i)}$. Note that after completing a full iteration of Algorithm 11, the cluster means $\left\{\boldsymbol{\mu}^{(c)}\right\}_{c=1}^{k}$ are fully determined by the cluster assignments $\left\{\hat{y}^{(i)}\right\}_{i=1}^{m}$ via (8.7). It seems natural to terminate Algorithm 11 if the decrease in the clustering error achieved by the most recent iteration is below a prescribed (small) threshold.

Clustering and Classification. There is a strong conceptual link between Algorithm 11 and classification methods (see e.g. Section 3.13). Both methods essentially learn a hypothesis $h(\mathbf{x})$ that maps the feature vector $\mathbf{x}$ to a predicted label $\hat{y} = h(\mathbf{x})$ from a finite set. The practical meaning of the label values is different for Algorithm 11 and classification methods. For classification methods, the meaning of the label values is essentially defined by the training set (of labeled data points) used for ERM (4.2). On the other hand, clustering methods use the predicted label $\hat{y} = h(\mathbf{x})$ as a cluster index.

Another main difference between Algorithm 11 and most classification methods is the choice for the empirical risk used to evaluate the quality or usefulness of a given hypothesis $h(\cdot)$. Classification methods typically use an average loss over labeled datapoints in a training set as empirical risk. In contrast, Algorithm 11 uses the clustering error (8.3) as a form of empirical risk. Consider a hypothesis that resembles the cluster assignments $\hat{y}^{(i)}$ obtained

after completing an iteration in Algorithm 11, $\hat{y}^{(i)} = h\big(\mathbf{x}^{(i)}\big)$. Then we can rewrite the resulting clustering error achieved after this iteration as

$$\mathcal{E}\big(h|\mathcal{D}\big) = (1/m)\sum_{i=1}^{m}\left\|\mathbf{x}^{(i)} - \frac{\sum_{i':h(\mathbf{x}^{(i)})=h(\mathbf{x}^{(i')})}\mathbf{x}^{(i')}}{\sum_{i':h(\mathbf{x}^{(i)})=h(\mathbf{x}^{(i')})}}\right\|^2. \tag{8.8}$$

Note that the i-th summand in (8.8) depends on the entire dataset $\mathcal{D}$ and not only on the feature vector $\mathbf{x}^{(i)}$.

Some Practicalities. For a practical implementation of Algorithm 11 we need to fix three issues.

- Issue 1 ("tie-breaking"): We need to specify what to do if several different cluster indices $c \in \{1,\ldots,k\}$ achieve the minimum value in the cluster assignment update (8.6) during step 2.

- Issue 2 ("empty cluster"): The cluster assignment update (8.6) in step 3 of Algorithm 11 might result in a cluster c with no datapoints associated with it, $|\{i : \hat{y}^{(i)} = c\}| = 0$. For such a cluster c, the update (8.7) is not well-defined.

- Issue 3 ("stopping criterion"): We need to specify a criterion used in step 4 of Algorithm 11 to decide when to stop iterating.

Algorithm 12 is obtained from Algorithm 11 by fixing those three issues [35]. Step 3 of Algorithm 12 solves the first issue mentioned above ("tie breaking"), arising when there are several cluster clusters whose means have minimum distance to a data point $\mathbf{x}^{(i)}$, by assigning $\mathbf{x}^{(i)}$ to the cluster with smallest cluster index (see (8.9)). Step 4 of Algorithm 12 resolves the "empty cluster" issue by computing the variables $b^{(c)} \in \{0,1\}$ for $c = 1,\ldots,k$. The variable $b^{(c)}$ indicates if the cluster with index c is active ($b^{(c)} = 1$) or the cluster c is inactive ($b^{(c)} = 0$). The cluster c is defined to be inactive if there are no datapoints assigned to it during the preceding cluster assignment step (8.9). The cluster activity indicators $b^{(c)}$ allows to restrict the cluster mean updates (8.10) only to the clusters c with at least one datapoint $\mathbf{x}^{(i)}$. To obtain a stopping criterion, step 7 Algorithm 12 monitors the clustering error $E^{(r)}$ incurred by the cluster means and assignments obtained after r iterations. Algorithm 12 continues updating cluster assignments (8.9) and cluster means (8.10) as long as the decrease is above a given threshold $\varepsilon \geq 0$.

For Algorithm 12 to be useful we must ensure that the stopping criterion is met within a finite number of iterations. In other words, we must ensure that the clustering error decrease

Algorithm 12 "k-Means II" (slight variation of "Fixed Point Algorithm" in [35])

Input: dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^m$; number k of clusters; tolerance $\varepsilon \geq 0$; initial cluster means $\{\boldsymbol{\mu}^{(c)}\}_{c=1}^k$

1: **Initialize.** set iteration counter $r := 0$; $E^{(0)} := 0$

2: **repeat**

3: for all datapoints $i = 1, \ldots, m$,

$$\hat{y}^{(i)} := \min\{ \operatorname*{argmin}_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c')}\| \} \quad \text{(update cluster assignments)} \tag{8.9}$$

4: for all clusters $c = 1, \ldots, k$, update the activity indicator

$$b^{(c)} := \begin{cases} 1 & \text{if } |\{i : \hat{y}^{(i)} = c\}| > 0 \\ 0 & \text{else.} \end{cases}$$

5: for all $c = 1, \ldots, k$ with $b^{(c)} = 1$,

$$\boldsymbol{\mu}^{(c)} := \frac{1}{|\{i : \hat{y}^{(i)} = c\}|} \sum_{\{i : \hat{y}^{(i)} = c\}} \mathbf{x}^{(i)} \quad \text{(update cluster means)} \tag{8.10}$$

6: $r := r + 1$ (increment iteration counter)

7: $E^{(r)} := \mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^k, \{\hat{y}^{(i)}\}_{i=1}^m \mid \mathcal{D}\big)$ (evaluate clustering error (8.3))

8: **until** $r > 1$ and $E^{(r-1)} - E^{(r)} \leq \varepsilon$ (check for sufficient decrease in clustering error)

9: $\mathcal{E}^{(k)} := (1/m) \sum_{i=1}^m \left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(\hat{y}^{(i)})} \right\|^2$ (compute final clustering error)

Output: cluster assignments $\hat{y}^{(i)} \in \{1, \ldots, k\}$, cluster means $\boldsymbol{\mu}^{(c)}$, clustering error $\mathcal{E}^{(k)}$.

can be made arbitrarily small within a sufficiently large (but finite) number of iterations. To this end, it is useful to represent Algorithm 12 as a fixed-point iteration

$$\{\hat{y}^{(i)}\}_{i=1}^{m} \mapsto \mathcal{P}\{\hat{y}^{(i)}\}_{m=1}^{m}. \tag{8.11}$$

The operator $\mathcal{P}$, which depends on the dataset $\mathcal{D}$, reads in a list of cluster assignments and delivers an improved list of cluster assignments aiming at reducing the associated clustering error (8.3). Each iteration of Algorithm 12 updates the cluster assignments $\hat{y}^{(i)}$ by applying the operator $\mathcal{P}$. Representing Algorithm 12 as a fixed-point iteration (8.11) allows for an elegant proof of the convergence of Algorithm 12 within a finite number of iterations (even for $\varepsilon = 0$) [35, Thm. 2].

Figure 8.4 depicts the evolution of the cluster assignments and cluster means during the iterations Algorithm 12. Each column corresponds to one iteration of Algorithm 12. The upper part in each column depicts the update of cluster means while the lower part shows the update of the cluster assignments during each iteration.

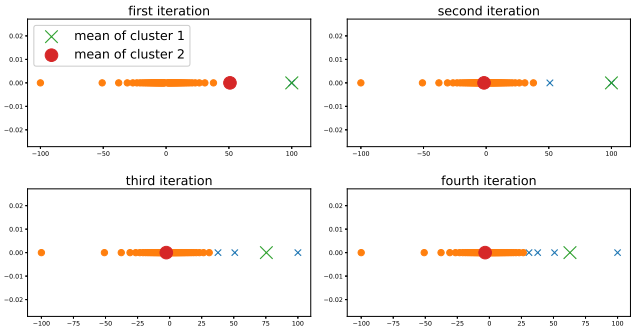


Figure 8.4: Evolution of cluster means (large dot and large cross) and cluster assignments during first four iterations of k-means.

Consider running Algorithm 12 with tolerance $\varepsilon = 0$ (see step 8) such that the iterations are continued until there is no decrease in the clustering error $E^{(r)}$ (see step 7 of Algorithm 12). As discussed above, Algorithm 12 will terminate after a finite number of iterations. Moreover, for $\varepsilon = 0$, the delivered cluster assignments $\left\{\hat{y}^{(i)}\right\}_{i=1}^{m}$ are fully determined by the delivered clustered means $\left\{\boldsymbol{\mu}^{(c)}\right\}_{c=1}^{k}$,

$$\hat{y}^{(i)} = \min\{ \operatorname*{argmin}_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c')}\|\}. \tag{8.12}$$

Indeed, if (8.12) does not hold one can show the final iteration r would still decrease the

clustering error and the stopping criterion in step 8 would not be met.

If cluster assignments and cluster means satisfy the condition (8.12), we can rewrite the clustering error (8.3) as a function of the cluster means solely,

$$\mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}|\mathcal{D}\big) := (1/m)\sum_{i=1}^{m} \min_{c' \in \{1,\ldots,k\}} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c')}\|^2. \tag{8.13}$$

Even for cluster assignments and cluster means that do not satisfy (8.12), we can still use (8.13) to lower bound the clustering error (8.3),

$$\mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}|\mathcal{D}\big) \leq \mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}, \{\hat{y}^{(i)}\}_{i=1}^{m} \mid \mathcal{D}\big)$$

.

Algorithm 12 iteratively improves the cluster means in order to minimize (8.13). Ideally, we would like Algorithm 12 to deliver cluster means that achieve the global minimum of (8.13) (see Figure 8.5). However, for some combination of dataset $\mathcal{D}$ and initial cluster means, Algorithm 12 delivers cluster means that form only a local optimum of $\mathcal{E}\big(\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}|\mathcal{D}\big)$ which is strictly worse (larger) than its global optimum (see Figure 8.5).

The tendency of Algorithm 12 to get trapped around a local minimum of (8.13) depends on the initial choice for cluster means. Therefore, it is often useful to repeat Algorithm 12 several times, with each repetition using a different initial choice for the cluster means. We then pick the cluster assignments $\{\hat{y}^{(i)}\}_{i=1}^{m}$ obtained for the repetition that resulted in the smallest clustering error $\mathcal{E}^{(k)}$ (see step 9).

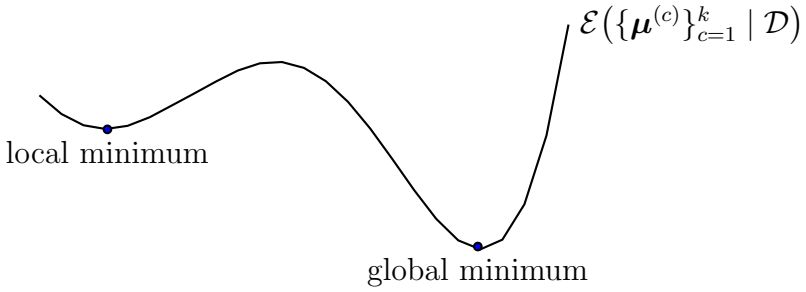


Figure 8.5: The clustering error (8.13) is a non-convex function of the cluster means $\{\boldsymbol{\mu}^{(c)}\}_{c=1}^{k}$. Algorithm 12 iteratively updates cluster means to minimize the clustering error but might get trapped around one of its local minimum.

Combining k-means with linear regression. Let us now sketch how Algorithm 11

could be combined with linear regression methods. The idea is to first group data points into a given number k of clusters and then learn separate linear predictors $h^{(c)}(\mathbf{x}) = \left(\mathbf{w}^{(c)}\right)^T \mathbf{x}$ for each cluster $c = 1, \ldots, k$. To predict the label of a new data point with features $\mathbf{x}$, we first assign to the cluster c' with the nearest cluster mean. We then use the linear predictor $h^{(c')}$ assigned to cluster c' to compute the predicted label $\hat{y} = h^{(c')}(\mathbf{x})$.

8.2 Soft Clustering with Gaussian Mixture Models

Consider a dataset $\mathcal{D} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ that we wish to group into a given number of k different clusters. The hard clustering methods of Section 8.1 deliver (predicted) cluster assignments $\hat{y}^{(i)}$ as the index of the cluster to which data point $\mathbf{x}^{(i)}$ is assigned to. These cluster assignments $\hat{y}$ provide rather coarse-grained information. Two data points $\mathbf{x}^{(i)}, \mathbf{x}^{(j)}$ might be assigned to the same cluster c although their distances to the cluster mean $\boldsymbol{\mu}^{(c)}$ might be very different. Intuitively, these two data points have a different degree of belonging to the cluster c.

For some clustering applications it is desirable to quantify the degree by which a datapoint belongs to a cluster. Soft clustering methods use a continues range, such as the closed interval $[0,1]$, of possible values for the degree of belonging. In contrast, hard clustering methods use only two possible degrees of belonging, either full belonging or no belonging to a cluster. While hard clustering methods assign a given datapoint to precisely one cluster, soft clustering methods typically assign a datapoint to several different clusters with non-zero degree of belonging.

This chapter discusses soft clustering methods that compute, for each data point $\mathbf{x}^{(i)}$ in the dataset $\mathcal{D}$, a vector $\widehat{\mathbf{y}}^{(i)} = \big(\hat{y}_1^{(i)}, \ldots, \hat{y}_k^{(i)}\big)^T$. We can interpret the entry $\hat{y}_c^{(i)} \in [0,1]$ as the degree by which the data point $\mathbf{x}^{(i)}$ belongs to the cluster $\mathcal{C}^{(c)}$. For $\hat{y}_c^{(i)} \approx 1$, we are quite confident in the data point $\mathbf{x}^{(i)}$ belonging to cluster $\mathcal{C}^{(c)}$. In contrast, for $\hat{y}_c^{(i)} \approx 0$, we are quite confident in the data point $\mathbf{x}^{(i)}$ being outside the cluster $\mathcal{C}^{(c)}$.

A widely used soft-clustering method uses a probabilistic model for the data points $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^m$. Within this model, each cluster $\mathcal{C}^{(c)}$, for $c = 1, \ldots, k$, is represented by a multivariate normal distributions [8]

$$\mathcal{N}\big(\mathbf{x}; \boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)}\big) = \frac{1}{\sqrt{\det\{2\pi\boldsymbol{\Sigma}\}}} \exp\big(-(1/2)\big(\mathbf{x}-\boldsymbol{\mu}^{(c)}\big)^T \big(\boldsymbol{\Sigma}^{(c)}\big)^{-1}\big(\mathbf{x}-\boldsymbol{\mu}^{(c)}\big)\big), \text{ for } c = 1, \ldots, k. \tag{8.14}$$

The probability distribution (8.14) is parameterized by a cluster-specific mean vector $\boldsymbol{\mu}^{(c)}$ and an (invertible) cluster-specific covariance matrix $\boldsymbol{\Sigma}^{(c)}$.[1] We interpret a specific data point $\mathbf{x}^{(i)}$ as a realization drawn from the probability distribution (8.14) of a specific cluster $c^{(i)}$,

$$\mathbf{x}^{(i)} \sim \mathcal{N}\big(\mathbf{x}; \boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)}\big) \text{ with cluster index } c = c^{(i)}. \tag{8.15}$$

We can think of $c^{(i)}$ as the true index of the cluster to which the data point $\mathbf{x}^{(i)}$ belongs to.

[1] Note that the expression (8.14) is only valid for an invertible (non-singular) covariance matrix $\boldsymbol{\Sigma}$.

The variable $c^{(i)}$ selects the cluster distributions (8.14) from which the feature vector $\mathbf{x}^{(i)}$ has been generated (drawn). We will therefore refer to the variable $c^{(i)}$ as the (true) cluster assignment for the ith datapoint. Similar to the feature vectors $\mathbf{x}^{(i)}$ we also interpret the cluster assignments $c^{(i)}$, for $i = 1, \ldots, m$ as realizations of i.i.d. RVs.

In contrast to the feature vectors $\mathbf{x}^{(i)}$, we do not observe (know) the true cluster indices $c^{(i)}$. After all, the goal of soft clustering is to estimate the cluster indices $c^{(i)}$. We obtain a soft-clustering method by estimating the cluster indices $c^{(i)}$ based solely on the data points in $\mathcal{D}$. To compute these estimates we assume that the (true) cluster indices $c^{(i)}$ are realizations of i.i.d. RVs with the common probability distribution (or probability mass function)

$$p_c := p(c^{(i)} = c) \text{ for } c = 1, \ldots, k. \tag{8.16}$$

The (prior) probabilities p_c, for $c = 1, \ldots, k$, are either assumed known or estimated from data [52, 8]. The choice for the probabilities p_c could reflect some prior knowledge about different sizes of the clusters. If cluster $\mathcal{C}^{(1)}$ is known to be larger than cluster $\mathcal{C}^{(2)}$, we might choose the prior probabilities such that $p_1 > p_2$.

The probabilistic model given by (8.15), (8.16) amounts to a **Gaussian mixture model** (GMM). This name is quite natural as the common marginal distribution for the feature vectors $\mathbf{x}^{(i)}$, for $i = 1, \ldots, m$, is a (additive) mixture of multivariate normal (Gaussian) distributions,

$$p(\mathbf{x}^{(i)}) = \sum_{c=1}^{k} \underbrace{p(c^{(i)} = c)}_{p_c} \underbrace{p(\mathbf{x}^{(i)} | c^{(i)} = c)}_{\mathcal{N}(\mathbf{x}^{(i)}; \boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)})}. \tag{8.17}$$

As already mentioned, the cluster assignments $c^{(i)}$ are hidden (unobserved) RVs. We thus have to infer or estimate these variables from the observed datapoints $\mathbf{x}^{(i)}$ which realizations or i.i.d. random vector with the common distribution (8.17).

The GMM (see (8.15) and (8.16)) lends naturally to a rigorous definition for the degree $y_c^{(i)}$ by which datapoint $\mathbf{x}^{(i)}$ belongs to cluster c.[2] Let us define the label value $y_c^{(i)}$ as the "a-posteriori" probability of the cluster assignment $c^{(i)}$ being equal to $c \in \{1, \ldots, k\}$:

$$y_c^{(i)} := p(c^{(i)} = c | \mathcal{D}). \tag{8.18}$$

[2]Remember that the degree of belongings $y_c^{(i)}$ are considered as (unknown) label values for datapoints. The choice or definition for the labels of datapoints is a design choice. In particular, we can define the labels of datapoints using a hypothetical probabilistic model such as the GMM.

By their very definition (8.18), the degrees of belonging $y_c^{(i)}$ always sum to one,

$$\sum_{c=1}^{k} y_c^{(i)} = 1 \text{ for each } i = 1, \ldots, m. \tag{8.19}$$

We emphasize that we use the conditional cluster probability (8.18), 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 $\mathcal{D}$.

The definition (8.18) for the label values (degrees of belonging) $y_c^{(i)}$ involves the GMM parameters $\{\boldsymbol{\mu}^{(c)}, \boldsymbol{\mu}^{(c)}, p_c\}_{c=1}^k$. Since we do not know these parameters beforehand we cannot evaluate the conditional probability in (8.18). A principled approach to solve this problem is to evaluate (8.18) with the true GMM parameters replaced by some estimates $\{\widehat{\boldsymbol{\mu}}^{(c)}, \widehat{\boldsymbol{\Sigma}}^{(c)}, \hat{p}_c\}_{c=1}^k$. Plugging in the GMM parameter estimates into (8.18) provides us with predictions $\hat{y}_c^{(i)}$ for the degrees of belonging. However, to compute the GMM parameter estimates we would have already needed the degrees of belonging $y_c^{(i)}$. This dilemma is similar to the dilemma of jointly optimizing cluster means and assignments in hard clustering (see Section 8.1).

Similar to the spirit of Algorithm 4 for hard clustering, we solve the above dilemma of soft clustering by an alternating optimization scheme. In particular, we alternate between updating (optimizing) the predicted degrees of belonging $\hat{y}_c^{(i)}$, for $i = 1, \ldots, m$ and $c = 1, \ldots, k$, given the current GMM parameter estimates $\{\widehat{\boldsymbol{\mu}}^{(c)}, \widehat{\boldsymbol{\Sigma}}^{(c)}, \hat{p}_c\}_{c=1}^k$ and then updating (optimizing) these GMM parameter estimates based on the updated predictions $\hat{y}_c^{(i)}$. We summarize the resulting soft clustering method in Algorithm 4. Each iteration of Algorithm 4 consists of an update (8.22) for the degrees of belonging followed by an update (step 3) for the GMM parameters.

To analyze Algorithm 13 it is helpful to interpret data points $\mathbf{x}^{(i)}$ as realizations of i.i.d. RVs distributed according to a GMM (8.15)-(8.16). We can then understand Algorithm 13 as a method for estimating the GMM parameters based on observing realizations drawn from the GMM (8.15)-(8.16). A principled approach to estimating the parameters of a probability distribution is the maximum likelihood method (see Section 3.12 and [46, 52]). The idea is to estimate the GMM parameters by maximizing the probability (density)

$$p\big(\mathcal{D}; \{\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)}, p_c\}_{c=1}^k\big) \tag{8.20}$$

of actually observing the data point in the dataset $\mathcal{D}$.

It can be shown that Algorithm 13 is an instance of a generic approximate maximum

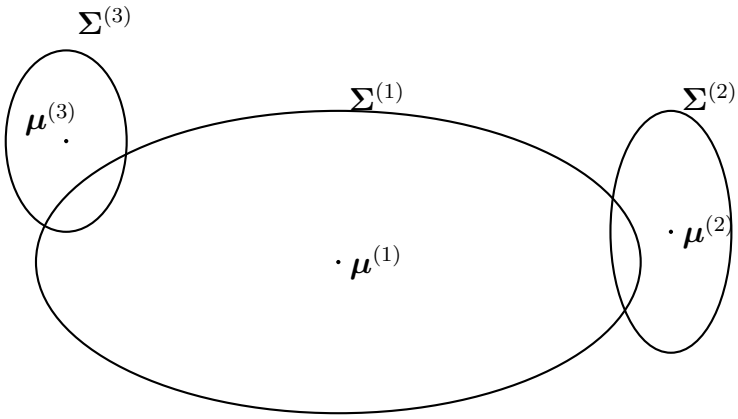


Figure 8.6: The GMM (8.15), (8.16) yields a probability density function (8.17) which is a weighted sum of multivariate normal distributions $\mathcal{N}(\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)})$. The weight of the c-th component is the cluster probability $p(c^{(i)} = c)$.

likelihood technique referred to as **expectation maximization** (EM) (see [37, Chap. 8.5] for more details). In particular, each iteration of Algorithm 13 updates the GMM parameter estimates such that the corresponding probability density (8.20) does not decrease [91]. If we denote the GMM parameter estimate obtained after r iterations of Algorithm 13 by $\boldsymbol{\theta}^{(r)}$ [37, Sec. 8.5.2],

$$p\big(\mathcal{D}; \boldsymbol{\theta}^{(r+1)}\big) \geq p\big(\mathcal{D}; \boldsymbol{\theta}^{(r)}\big) \tag{8.21}$$

As for Algorithm 11, we can also interpret Algorithm 4 as an instance of the ERM principle discussed in Chapter 4. Indeed, maximizing the probability density (8.20) is equivalent to minimizing the empirical risk

$$\mathcal{E}\big(\boldsymbol{\theta} \mid \mathcal{D}\big) := -\log p\big(\mathcal{D}; \boldsymbol{\theta}\big) \text{ with GMM paramters } \boldsymbol{\theta} := \{\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)}, p_c\}_{c=1}^{k} \tag{8.23}$$

The empirical risk (8.23) is the negative logarithm of the probability (density) (8.20) of observing the dataset $\mathcal{D}$ as i.i.d. realizations of the GMM (8.17). The monotone increase in the probability density (8.21) achieved by the iterations of Algorithm 4 translate into a monotone decrease of the empirical risk,

$$\mathcal{E}\big(\boldsymbol{\theta}^{(r)} \mid \mathcal{D}\big) \leq \mathcal{E}\big(\boldsymbol{\theta}^{(r-1)} \mid \mathcal{D}\big) \text{ with iteration counter } r. \tag{8.24}$$

The monotone decrease (8.24) in the empirical risk (8.23) achieved by the iterations of Algorithm 13 naturally lends to a stopping criterion. Let $E^{(r)}$ denote the empirical risk (8.23)

Algorithm 13 "A Soft-Clustering Algorithm" [11]

Input: dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{m}$; number k of clusters, initial GMM parameter estimates $\{\widehat{\boldsymbol{\mu}}^{(c)}, \widehat{\boldsymbol{\Sigma}}^{(c)}, \hat{p}_c\}_{c=1}^{k}$

1: **repeat**

2: for each $i = 1, \ldots, m$ and $c = 1, \ldots, k$, update degrees of belonging

$$\hat{y}_c^{(i)} := \frac{\hat{p}_c \mathcal{N}(\mathbf{x}^{(i)}; \widehat{\boldsymbol{\mu}}^{(c)}, \widehat{\boldsymbol{\Sigma}}^{(c)})}{\sum_{c'=1}^{k} \hat{p}_{c'} \mathcal{N}(\mathbf{x}^{(i)}; \widehat{\boldsymbol{\mu}}^{(c')}, \widehat{\boldsymbol{\Sigma}}^{(c')})} \tag{8.22}$$

3: for each $c \in \{1, \ldots, k\}$, **update GMM parameter estimates**:

- $\hat{p}_c := m_c/m$ with **effective cluster size** $m_c := \sum_{i=1}^{m} \hat{y}_c^{(i)}$ (cluster probability)
- $\widehat{\boldsymbol{\mu}}^{(c)} := (1/m_c) \sum_{i=1}^{m} \hat{y}_c^{(i)} \mathbf{x}^{(i)}$ (cluster mean)
- $\widehat{\boldsymbol{\Sigma}}^{(c)} := (1/m_c) \sum_{i=1}^{m} \hat{y}_c^{(i)} \big(\mathbf{x}^{(i)} - \widehat{\boldsymbol{\mu}}^{(c)}\big)\big(\mathbf{x}^{(i)} - \widehat{\boldsymbol{\mu}}^{(c)}\big)^T$ (cluster covariance matrix)

4: **until** stopping criterion met

Output: predicted degrees of belonging $\widehat{\mathbf{y}}^{(i)} = (\hat{y}_1^{(i)}, \ldots, \hat{y}_k^{(i)})^T$ for $i = 1, \ldots, m$.

achieved by the GMM parameter estimates $\boldsymbol{\theta}^{(r)}$ obtained after r iterations in Algorithm 13. We stop iterating as soon as the decrease $E^{(r)} - E^{(r+1)}$ achieved by the $r+1$-th iteration of Algorithm 4 falls below a given (positive) threshold $\varepsilon > 0$.

Similar to Algorithm 11, also Algorithm 13 might get trapped in local minima of the underlying empirical risk. The GMM parameters delivered by Algorithm 13 might only be a local minimum of (8.23) but not the global minimum (see Figure 8.5 for the analogous situation in hard clustering). As for hard clustering Algorithm 11, we typically repeat Algorithm 13 several times. During each repetition of Algorithm 13, we use a different (randomly chosen) initialization for the GMM parameter estimates $\boldsymbol{\theta} = \{\widehat{\boldsymbol{\mu}}^{(c)}, \widehat{\boldsymbol{\Sigma}}^{(c)}, \hat{p}_c\}_{c=1}^{k}$. Each repetition of Algorithm 13 results in a potentially different set of GMM parameter estimates and degrees of belongings $\hat{y}_c^{(i)}$. We then use the results for that repetition that achieves the smallest empirical risk (8.23).

Let us point out an interesting link between soft clustering using GMM (see Algorithm 13) and hard clustering with k-means (see Algorithm 11). Consider the GMM (8.15) with prescribed cluster covariance matrices

$$\boldsymbol{\Sigma}^{(c)} = \sigma^2 \mathbf{I} \text{ for all } c \in \{1, \ldots, k\}, \tag{8.25}$$

with some given variance $\sigma^2 > 0$. We assume the cluster covariance matrices in the GMM to be given by (8.25) and therefore can replace the covariance matrix updates in Algorithm 13 with the assignment $\widehat{\mathbf{\Sigma}}^{(c)} := \sigma^2\mathbf{I}$. It can be verified easily that for sufficiently small variance σ^2 in (8.25), the update (8.22) tends to enforce $\hat{y}_c^{(i)} \in \{0,1\}$. In other words, each datapoint $\mathbf{x}^{(i)}$ becomes then effectively associated with exactly one single cluster c whose cluster mean $\widehat{\boldsymbol{\mu}}^{(c)}$ is nearest to $\mathbf{x}^{(i)}$. For $\sigma^2 \to 0$, the soft-clustering update (8.22) in Algorithm 13 reduces to the (hard) cluster assignment update (8.6) in k-means Algorithm 11. We can interpret Algorithm 11 as an extreme case of Algorithm 13 that is obtained by fixing the covariance matrices in the GMM to $\sigma^2\mathbf{I}$ with a sufficiently small σ^2.

Combining GMM with Linear Regression. Let us now sketch how Algorithm 13 could be combined with linear regression methods. The idea is to first compute the degree of belongings to the clusters for each data point. We then learn separate linear predictors for each cluster using the degree of belongings as weights for the individual loss terms in the training error. To predict the label of a new data point, we first compute the predictions obtained for each cluster-specific linear hypothesis and then average them using the degree of the new data point belonging to each cluster.

8.3 Connectivity-based Clustering

The clustering methods discussed in Sections 8.1 and 8.2 can only be applied to data points which are characterized by numeric feature vectors. These methods define the similarity between data points using the Euclidean distance between the feature vectors of these data points. As illustrated in Figure 8.7, these methods can only produce "Euclidean shaped" clusters that are contained either within hyper-spheres (Algorithm 11) or hyper-ellipsoids (Algorithm 13).

Some applications generate data points for which the construction of useful numeric features is difficult. Consider the task of contact tracing during a pandemic. Here, data points might represent human individuals and we would like to find clusters of humans that are at risk ob being infected by a person that has been diagnosed as infected. Ideally we would have location traces for each individual and then determine if two persons have been close-by for a sufficient duration. However, getting access to high-resolution location traces is difficult. Instead contact tracing works by manually determining the similarity between humans using few rules. from the viewpoint of contact tracing,two humans might be considered similar if they share an office or if they live in the same household.

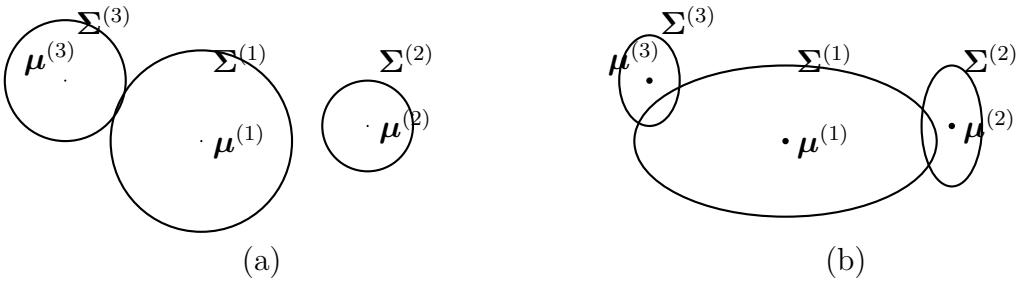


Figure 8.7: (a): Cartoon of typical cluster shapes delivered by k-means Algorithm 12. (b): Cartoon of typical cluster shapes delivered by soft clustering Algorithm 13.

Even if we can easily obtain numeric features of data points, the Euclidean distances between the resulting feature vectors might not reflect the actual similarities between data points. As a case in point, groups of similar text documents might form highly complicated shapes in the feature space that cannot be grouped within hyper-ellipsoids. For datasets with such "non-Euclidean" cluster shapes, k-means or GMM are not suitable as clustering methods. Instead of distances between Euclidean feature vectors we should then use a different measure for the similarity between data points.

Connectivity-based clustering methods do not require any numeric features of data points. These methods cluster data points based on explicitly specifying, for any pair of data point, if data points are similar and to what extend. A convenient mathematical tool to represent similarities between data points of a dataset $\mathcal{D}$ is a weighted undirected graph $\mathcal{G} = \big(\mathcal{V}, \mathcal{E}\big)$. We refer to this graph as the similarity graph of the dataset $\mathcal{D}$ (see Figure 8.8). The nodes $\mathcal{V}$ in this similarity graph $\mathcal{G}$ represent data points in $\mathcal{D}$ and the undirected edges connect nodes that represent similar data points. In some application we might have some quantitative measure for the similarity between data points. These similarity measures can be incorporated into the graph as weights $W_{i,j}$ for each edge $\{i,j\} \in \mathcal{E}$.

Given a similarity graph $\mathcal{G}$ of a set of data points, connectivity-based clustering methods determine clusters as subsets of nodes that are strongly connected within the cluster but weakly connected between different clusters. Different concepts for quantifying the connectivity between nodes in a graph yield different clustering methods. The density-based clustering algorithm DBSCAN considers data points as connected if there is a path between them such that each node on this path has a sufficiently large number of neighbours (such nodes are referred to as "core nodes") [27].

Density-based spatial clustering of applications with noise (DBSCAN) is a hard clustering

method that uses a notion of similarity that is based on **connectivity**. DBSCAN considers two datapoints as similar if they can be reached by intermediate datapoints that have a small Euclidean distance. Two datapoints can be similar in terms of connectivity, even if their Euclidean distance is large.

In contrast to k-means and GMM, DBSCAN does not require the number of clusters to be pre-defined. The number of clusters delivered by DBSCAN is determined by the choice of the parameters. DBSCAN also performs an implicit outlier detection. The outliers delivered by DBSCAN are those clusters which contain a single datapoint only. For a detailed discussion of how DBSCAN works, we refer to https://en.wikipedia.org/wiki/DBSCAN.

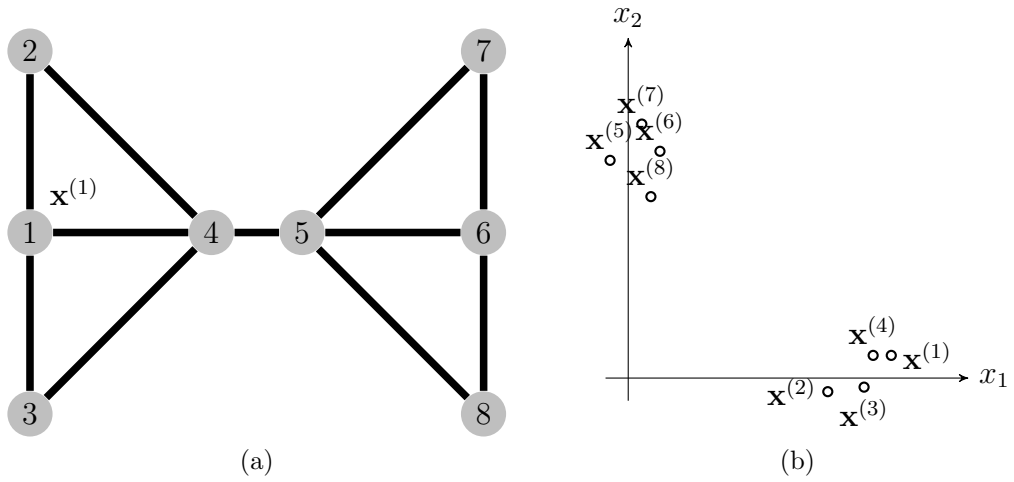


Figure 8.8: Connectivity-based clustering can be obtained by constructing features $\mathbf{x}^{(i)}$ that are (approximately) identical for well-connected data points. (a): A similarity graph for a dataset $\mathcal{D}$ consists of nodes representing individual data points and edges that connect similar data points. (b) Feature vectors of well-connected data points have small Euclidean distance.

8.4 Exercises

8.4.1 k-means updates

Show that the cluster means and assignment updates of k-means never increase the clustering error (8.3).

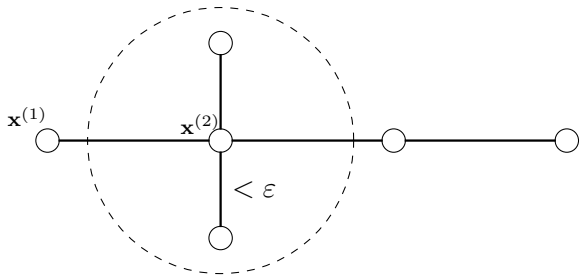


Figure 8.9: DBSCAN assigns two data points to the same cluster if they are reachable. Two data points $\mathbf{x}^{(i)}, \mathbf{x}^{(i')}$ are reachable if there is a path of data points from $\mathbf{x}^{(i')}$ to $\mathbf{x}^{(i)}$. This path consists of a sequence of data points that are within a distance of ε. Moreover, each data point on this path must be a core point which has at least a given number of neighbouring data points within the distance ε.

8.4.2 How to choose k?

Discuss strategies for choosing the number k of clusters used for k-means.

8.4.3 Local Minima.

Consider applying the hard clustering Algorithm 12 to the dataset $(-10, 1), (10, 1), (-10, -1), (10, -1)$ with initial cluster means $(0, 1), (0, -1)$ and tolerance $\varepsilon = 0$. Does Algorithm 12 get trapped in a local minimum of (8.13)?

8.4.4 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.

8.4.5 Compression with k-means

Consider $m = 10000$ datapoints $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$ which are represented by numeric features vectors of length two. We apply k-means to cluster the data set into two clusters. How many bits do we need to store the clustering? For simplicity, we assume that any real number can stored perfectly as a floating point numbers (32 bit).

Chapter 9

Feature Learning

"Solving Problems By Changing the Viewpoint."

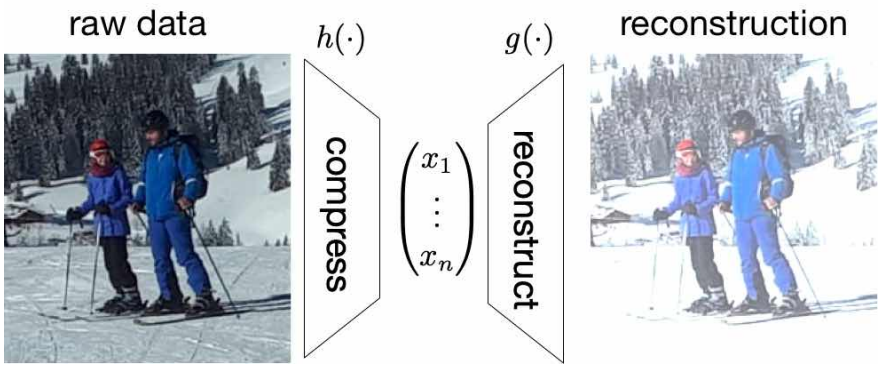


Figure 9.1: Dimensionality reduction methods aim at finding a map h which maximally compresses the raw data while still allowing to accurately reconstruct the original datapoint from a small number of features $x_1, \ldots, x_n$.

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. For example, we might use the numeric measurement $z \in \mathbb{R}$ delivered by a sensing device as a feature. However, we could augment this single feature with new features such as the powers z^2 and z^3 or adding a constant $z+5$. Each of these computations produces a new feature. Which of these additional features are most useful?

Feature learning methods automate the choice of finding good features. These methods learn a hypothesis map that reads in some representation of a data point and transforms it to a set of features. Feature learning methods differ in the precise format of the original

data representation as well as the format of the delivered features. The focus of this chapter in on feature learning methods that require data points being represented by d numeric raw features and deliver a set of n new numeric features. We will denote the set of raw and new features by $\mathbf{z} = \big(z_1, \ldots, z_d\big)^T \in \mathbb{R}^d$ and $\mathbf{x} = \big(x_1, \ldots, x_n\big)^T \in \mathbb{R}^n$, respectively.

Many ML application domains generate datapoints for which can access a huge number of raw features. Consider data points being snapshots generated by a smartphone. It seems natural to use the pixel colour intensities as the raw features of the snapshot. Since modern smartphone have Megapixel cameras, the pixel intensities would provide us with millions of raw features. It might seem a good idea to use as many (raw) features of a data point as possible since more features should offer more information about a datapoint and 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**.

Computationally, using very large feature vectors $\mathbf{x} \in \mathbb{R}^n$ (with n being billions), might result in excessive resource requirements (bandwidth, storage, time) of the resulting ML method. Statistically, using a large number of features makes the resulting ML methods more prone to overfitting. For example, linear regression will typically 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).

Both from a computational and a statistical perspective, it is beneficial to use only the maximum necessary amount of features. The challenge is to select those features which carry most of the relevant information required for the prediction of the label y. Finding the most relevant features out of a huge number of (raw) features is the goal of dimensionality reduction methods. Dimensionality reduction methods form an important sub-class of feature learning methods. Formally, dimensionality reduction methods learn a hypothesis $h(\mathbf{z})$ that map a long raw feature vector $\mathbf{z} \in \mathbb{R}^d$ to a short new feature vector $\mathbf{x} \in \mathbb{R}^n$ with $d \gg n$.

Beside avoiding overfitting and coping with 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 data points in the two-dimensional plane.

We will discuss the basic idea underlying dimensionality reduction methods in Section 9.1. Section 9.2 presents one particular example of a dimensionality reduction method that computes relevant features by a linear transformation of the raw feature vector. Section 9.4 discusses a method for dimensionality reduction that exploits the availability of labelled datapoints. Section 9.6 shows how randomness can be used to obtain computationally cheap dimensionality reduction .

Most of this chapter discusses dimensionality reduction methods that determine a small number of relevant features from a large set of raw features. However, sometimes it might be useful to go the opposite direction. There are applications where it might be beneficial to construct a large (even infinite) number of new features from a small set of raw features. Section 9.7 will showcase how computing additional features can help to improve the prediction accuracy of ML methods.

9.1 Basic Principle of Dimensionality Reduction

Figure 9.1 illustrates the basic idea of dimensionality reduction methods. Their purpose is to learn (or find) a "compression" map $h(\cdot): \mathbb{R}^d \rightarrow \mathbb{R}^n$ that transforms a (long) raw feature vector $\mathbf{z} \in \mathbb{R}^d$ to a (short) feature vector $\mathbf{x} = (x_1, \ldots, x_n)^T := h(\mathbf{z})$ (typically $n \ll d$).

The new feature vector $\mathbf{x} = h(\mathbf{z})$ serves as a compressed representation (or code) for the original feature vector $\mathbf{z}$. We can reconstruct the raw feature vector using a reconstruction map $g(\cdot): \mathbb{R}^n \rightarrow \mathbb{R}^d$. The reconstructed vector $\widehat{\mathbf{z}} := g(\mathbf{x}) = g(h(\mathbf{z}))$ will typically by different from the original raw feature vector $\mathbf{z}$. Thus, we will obtain a non-zero **reconstruction error**

$$\underbrace{\widehat{\mathbf{z}}}_{=g(h(\mathbf{z})))} - \mathbf{z}. \tag{9.1}$$

Dimensionality reduction methods learn a compression map $h(\cdot)$ such that the reconstruction error (9.1) is minimized. In particular, for a dataset $\mathcal{D} = \big\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\big\}$, we measure the quality of a pair of compression map h and reconstruction map g by the average reconstruction error

$$\mathcal{E}\big(h, g|\mathcal{D}\big) := (1/m) \sum_{i=1}^{m} \mathcal{L}(\mathbf{z}^{(i)}, g\big(h\big(\mathbf{z}^{(i)}\big)\big)). \tag{9.2}$$

Here, $\mathcal{L}(\mathbf{z}, g\big(h\big(\mathbf{z}^{(i)}\big)\big)$ denotes a loss function that is used to measure the reconstruction error $\underbrace{g\big(h\big(\mathbf{z}^{(i)}\big)\big)}_{\widehat{\mathbf{z}}} - \mathbf{z}$. Different choices for the loss function in (9.2) result in different dimensionality reduction methods. One popular choice for the loss is the squared Euclidean norm

$$\mathcal{L}(\mathbf{z}, g\big(h\big(\mathbf{z}\big)\big)) := \big\|\mathbf{z} - g\big(h\big(\mathbf{z}\big)\big)\big\|_2^2. \tag{9.3}$$

Practical dimensionality reduction methods have only finite computational resources. Any practical method must therefore restrict the set of possible compression and reconstruction maps to small subsets $\mathcal{H}$ and $\mathcal{H}^*$, respectively. These subsets are the hypothesis spaces for

the compression map $h \in \mathcal{H}$ and the reconstruction map $g \in \mathcal{H}^*$. Feature learning methods differ in their choice for these hypothesis spaces.

Dimensionality reduction methods learn a compression map by solving

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} \min_{g \in \mathcal{H}^*} \mathcal{E}\big(h, g|\mathcal{D}\big)$$
$$\stackrel{(9.2)}{=} \operatorname*{argmin}_{h \in \mathcal{H}} \min_{g \in \mathcal{H}^*} (1/m) \sum_{i=1}^{m} \mathcal{L}\big(\mathbf{z}^{(i)}, g\big(h\big(\mathbf{z}^{(i)}\big)\big)\big). \tag{9.4}$$

We can interpret (9.4) as a (typically non-linear) approximation problem. The optimal compression map $\hat{h}$ is such that the reconstruction$g(\hat{h}(\mathbf{z}))$, with a suitably chosen reconstruction map $g(\cdot)$, approximates the original raw feature vector $\mathbf{z}$ as good as possible. Note that we use a single compression map $h(\cdot)$ and a single reconstruction map $g(\cdot)$ for all data points in the dataset $\mathcal{D}$.

We obtain different dimensionality methods from different choices for the hypothesis spaces $\mathcal{H}, \mathcal{H}^*$ and loss function in (9.4). Section 9.2 discusses a method that solves (9.4) for $\mathcal{H}, \mathcal{H}^*$ constituted by linear maps and the loss (9.3). **Deep autoencoders** are another family of dimensionality reduction methods that solve (9.4) for $\mathcal{H}, \mathcal{H}^*$ constituted by non-linear maps that are represented by deep neural networks [33, Ch. 14].

9.2 Principal Component Analysis

We now consider the special case of dimensionality reduction where the compression and reconstruction map are required to be linear maps. Consider a datapoint which is characterized by a (typically very long) (raw) feature vector $\mathbf{z} \in \mathbb{R}^d$ 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 raw feature vector,

$$\mathbf{x} = \mathbf{W}\mathbf{z}. \tag{9.5}$$

Here, the "compression" matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$ maps (in a linear fashion) the (long) raw feature vector $\mathbf{z} \in \mathbb{R}^d$ to the (shorter) 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 (9.5) 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

$$\mathbf{z} \approx \mathbf{R}\mathbf{x} \overset{(9.5)}{=} \mathbf{R}\mathbf{W}\mathbf{z}. \tag{9.6}$$

The approximation error $\mathcal{E}\big(\mathbf{W}, \mathbf{R} \mid \mathcal{D}\big)$ resulting when (9.6) is applied to each datapoint in a dataset $\mathcal{D} = \{\mathbf{z}^{(i)}\}_{i=1}^{m}$ is then

$$\mathcal{E}\big(\mathbf{W}, \mathbf{R} \mid \mathcal{D}\big) = (1/m) \sum_{i=1}^{m} \|\mathbf{z}^{(i)} - \mathbf{R}\mathbf{W}\mathbf{z}^{(i)}\|^2. \tag{9.7}$$

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

$$\mathbf{W} = \mathbf{W}_{\rm PCA} := \big(\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}\big)^T \in \mathbb{R}^{n \times d}, \tag{9.8}$$

with n orthonormal vectors $\mathbf{u}^{(j)}$, for $j = 1, \ldots, n$. The vectors $\mathbf{u}^{(j)}$ are the eigenvectors corresponding to the n largest eigenvalues of the **sample covariance matrix**

$$\mathbf{Q} := (1/m)\mathbf{Z}^T\mathbf{Z} \in \mathbb{R}^{d \times d}. \tag{9.9}$$

Here we used the data matrix $\mathbf{Z} = \big(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\big)^T \in \mathbb{R}^{m \times d}$. [1] It can be verified easily, using the definition (9.9), that the matrix $\mathbf{Q}$ is **psd**. As a **psd** matrix, $\mathbf{Q}$ has an eigenvalue decomposition (EVD) of the form [79]

$$\mathbf{Q} = \big(\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(d)}\big) \begin{pmatrix} \lambda^{(1)} & \ldots & 0 \\ 0 & \ddots & 0 \\ 0 & \ldots & \lambda^{(d)} \end{pmatrix} \big(\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(d)}\big)^T$$

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 feature vectors $\mathbf{x}^{(i)}$ are obtained by applying the compression matrix $\mathbf{W}_{\rm PCA}$ (9.8) to the raw datapoints $\mathbf{z}^{(i)}$. We refer to the entries of the vector $\mathbf{x}^{(i)}$, obtained via the eigenvectors of $\mathbf{Q}$ (see (9.10)), as the **principal components (PC)** of the raw feature vectors $\mathbf{z}^{(i)}$. Algorithm 14 summarizes the overall procedure of determining the compression

[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 $\widehat{\mathbf{m}} = (1/m) \sum_{i=1}^{m} \mathbf{z}^{(i)}$.

matrix (9.8) and computing the vectors $\mathbf{x}^{(i)}$ whose entries are the PC of the raw feature vectors. This procedure is known as **principal component analysis (PCA)**. Note that

Algorithm 14 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 (9.10) 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 (9.10)).

Output: $\mathbf{x}^{(i)}$, for $i = 1, \ldots, m$, and the approximation error $\mathcal{E}^{(\rm PCA)}$.

the length $n(\leq d)$ of the mew feature vector $\mathbf{x}$, which is also the number of PCs used, is an input (or hyper) parameter of Algorithm 14. The number n can be chosen between the two extreme cases $n = 0$ (maximum compression) and $n = d$ (no compression). We finally note that the choice for the orthonormal eigenvectors in (9.8) might not be unique. Depending on the sample covariance matrix $\mathbf{Q}$, there might different sets of orthonormal vectors that correspond to the same eigenvalue of $\mathbf{Q}$. Thus, for a given length n of the new feature vectors, there might be several different matrices $\mathbf{W}$ that achieve the same (optimal) reconstruction error $\mathcal{E}^{(\rm PCA)}$.

From a computational perspective, Algorithm 14 essentially amounts to performing an EVD of the sample covariance matrix $\mathbf{Q}$ (see (9.9)). 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 $\mathbf{x}^{(i)} \in \mathbb{R}^n$. In particular, this information loss is measured by the approximation error (obtained for the optimal reconstruction matrix $\mathbf{R}_{\rm opt} = \mathbf{W}_{\rm PCA}^T$)

$$\mathcal{E}^{(\rm PCA)} := \mathcal{E}\big(\mathbf{W}_{\rm PCA}, \underbrace{\mathbf{R}_{\rm opt}}_{=\mathbf{W}_{\rm PCA}^T} \mid \mathcal{D}\big) = \sum_{r=n+1}^{d} \lambda^{(r)}. \tag{9.10}$$

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 (9.5). For the extreme case $n=0$, where we completely ignore the raw feature vectors $\mathbf{z}^{(i)}$, the optimal reconstruction error is $\mathcal{E}^{(\rm PCA)} = (1/m)\sum_{i=1}^m \|\mathbf{z}^{(i)}\|^2$. The other extreme case $n=d$ allows to use the raw features directly as the new features $\mathbf{x}^{(i)} = \mathbf{z}^{(i)}$, which amounts to no compression at all, and trivially results in a zero reconstruction

error $\mathcal{E}^{(\mathrm{PCA})} = 0$.

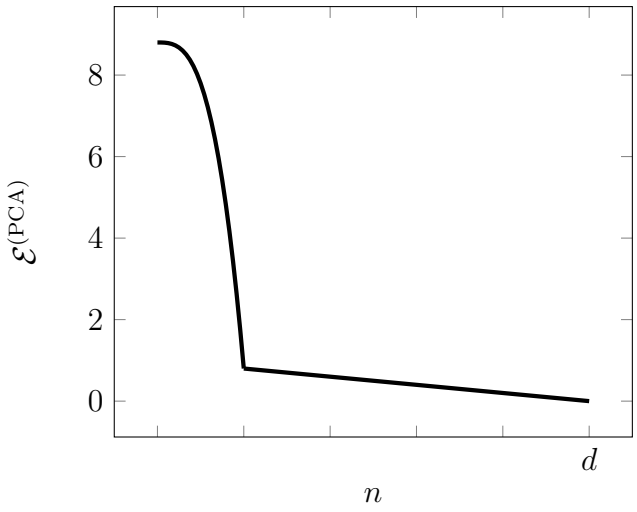


Figure 9.2: Reconstruction error $\mathcal{E}^{(\mathrm{PCA})}$ (see (9.10)) of PCA for varying number n of PCs.

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 $n = 2$, we obtain feature vectors $\mathbf{x}^{(i)} = \mathbf{W}_{\mathrm{PCA}}\mathbf{z}^{(i)}$ (see (9.5)) 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 $\mathbf{z}^{(i)} \in \mathbb{R}^d$ is a vector of length $d = 6$. It is difficult to visualise points in an Euclidean space $\mathbb{R}^d$ of dimension $d > 2$. Therefore, we apply PCA with $n = 2$ which results in feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^2$. These new feature vectors (of length 2) can be depicted conveniently as a scatter plot (see Figure 9.3).

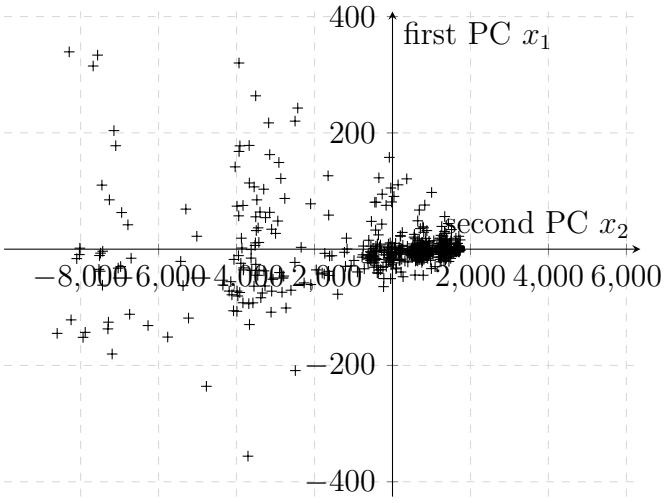


Figure 9.3: A scatter plot of feature vectors $\mathbf{x}^{(i)} = \left(x_1^{(i)}, x_2^{(i)}\right)^T$ whose entries are the first two PCs of the Bitcoin statistics $\mathbf{z}^{(i)}$ of the i-th day.

9.2.4 Extensions of PCA

There have been proposed several extensions of the basic PCA method:

- **Kernel PCA [37, Ch.14.5.4]:** The PCA method is most effective if the raw feature vectors of data points are nearby a low-dimensional linear subspace of $\mathbb{R}^d$. Kernel PCA extends PCA to handle data points that are located near a low-dimensional manifold which might be highly non-linear. This is achieved by applying PCA to transformed feature vectors instead of the original feature vectors. Kernel PCA first transforms (in a non-linear fashion) the original feature vectors $\mathbf{x}^{(i)}$ to new feature vectors $\mathbf{z}^{(i)}$ (see Section 3.9) and then applies PCA to $\mathbf{z}^{(i)}$, for $i = 1, \ldots, m$.

- **Robust PCA [90]:** In its basic form, PCA is sensitive to outliers which are a small number of data points with fundamentally different statistical properties than the bulk of data points. This sensitivity might be attributed to the properties of the squared Euclidean norm (9.3) which is used in PCA to measure the reconstruction error (9.1). We have seen in Chapter 3 that linear regression (see Section 3.1 and 3.3) can be made robust against outliers by replacing the squared error loss with another loss function. In a similar spirit, robust PCA replaces the squared Euclidean norm with another norm that is less sensitive to having very large reconstruction errors (9.1) for a small number of data points (which are outliers).

- **Sparse PCA [37, Ch.14.5.5]:** The basic PCA method transforms the raw feature vector $\mathbf{z}^{(i)}$ of a datapoint to a new (shorter) feature vector $\mathbf{x}^{(i)}$. In general each entry $x_j^{(i)}$ of the new feature vectors will depend on every raw feature. Since PCA uses the linear transformation (9.5), the new feature $x_j^{(i)}$ depends on all raw features $z_{j'}^{(i)}$ for which the corresponding entry $W_{j,j'}$ of the matrix $\mathbf{W} = \mathbf{W}_{\rm PCA}$ (9.8) is non-zero. For an arbitrary data set, all entries of the matrix $\mathbf{W}_{\rm PCA}$ will typically be non-zero.

  In some applications of linear dimensionality reduction we would like to construct new features that depend only on a small subset of raw features. Equivalently we would like to learn a linear compression map $\mathbf{W}$ (9.5) such that each row of $\mathbf{W}$ contains only few non-zero entries. To this end, sparse PCA extends the basic PCA by enforcing the rows of the compression matrix $\mathbf{W}$ to contain only a small number of non-zero entries. This enforcement can be implement either using additional constraints on $\mathbf{W}$ or a penalty term to the reconstruction error (9.7) that is minimized by PCA.

- **Probabilistic PCA [70, 82]:** We have motivated PCA as a method for learning an optimal linear compression map (matrix) (9.5) such that the compressed feature vectors allows to linearly reconstruct the original raw feature vector with minimum

reconstruction error (9.7). Another interpretation of PCA is that of a method that learns a subspace of $\mathbb{R}^d$ that best fits the set of raw feature vectors $\mathbf{z}^{(i)}$, for $i = 1, \ldots, m$. This optimal subspace is precisely the subspace spanned by the rows of $\mathbf{W}_{\rm PCA}$ (9.8).

Probabilistic PCA (PPCA) extends the basic PCA method by using a **probabilistic (generative) model** for the data points. In particular, PPCA interprets the raw feature vectors $\mathbf{z}^{(i)}$ as a realization of i.i.d. RVs. Moreover, these realizations are given by a generative model

$$\mathbf{z}^{(i)} = \mathbf{W}^T \mathbf{x}^{(i)} + \boldsymbol{\varepsilon}^{(i)}, \text{ for } i = 1, \ldots, m. \tag{9.11}$$

Here, $\mathbf{W} \in \mathbb{R}^{n \times d}$ is some unknown matrix with orthonormal rows. The rows of $\mathbf{W}$ span the subspace around which the raw features are concentrated. The vectors $\mathbf{x}^{(i)}$ in (9.11) are realizations of i.i.d. RVs whose common probability distribution is $\mathcal{N}(\mathbf{0}, \mathbf{I})$. The vectors $\boldsymbol{\varepsilon}^{(i)}$ are realizations of i.i.d. RVs whose common probability distribution is $\mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$ with some fixed but unknown variance σ^2. Note that $\mathbf{W}$ and σ^2 parametrize the joint probability distribution of the feature vectors $\mathbf{z}^{(i)}$ via (9.11). PPCA amounts to maximum likelihood estimation (see Section 3.12) of the parameters $\mathbf{W}$ and σ^2. This maximum likelihood estimation problem can be solved using computationally efficient estimation techniques such as EM [82, Appendix B]. The implementation of PPCA via EM also offers a principled approach to handle missing data. Roughly speaking, the EM method allows to use the probabilistic model (9.11) to estimate missing raw features [82, Sec. 4.1].

9.3 Feature Learning for Non-Numeric Data

We have motivated dimensionality reduction methods as transformations of (very long) raw feature vectors to a new (shorter) feature vector $\mathbf{x}$ such that it allows to reconstruct $\mathbf{z}$ with minimum reconstruction error (9.1). To make this requirement precise we need to define a measure for the size of the reconstruction error and specify the class of possible reconstruction maps. PCA uses the squared Euclidean norm (9.7) to measure the reconstruction error and only allows for linear reconstruction maps (9.6).

Alternatively, we can view dimensionality reduction as the generation of new feature vectors $\mathbf{x}^{(i)}$ that maintain the intrinsic geometry of the data points with their raw feature vectors $\mathbf{z}^{(i)}$. Different dimensionality reduction methods using different concepts for characterizing

the "intrinsic geometry" of data points. PCA defines the intrinsic geometry of data points using the squared Euclidean distances between feature vectors. Indeed, PCA produces feature vectors $\mathbf{x}^{(i)}$ such that for data points whose raw feature vectors have small squared Euclidean distance, also the new feature vectors $\mathbf{x}^{(i)}$ will have small squared Euclidean distance.

Some application domains generate data points for which the Euclidean distances between raw feature vectors does not reflect the intrinsic geometry of data points. As a point in case, consider data points representing scientific articles which can be characterized by the relative frequencies of words from some given set of relevant words (dictionary). A small Euclidean distance between the resulting raw feature vectors typically does not imply that the corresponding text documents are similar. Instead, the similarity between two articles might depend on the number of authors that are contained in author lists of both papers. We can represent the similarities between all articles using a similarity graph whose nodes represent data points which are connected by an edge (link) if they are similar (see Figure 8.8).

Consider a dataset $\mathcal{D} = \big(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\big)$ whose intrinsic geometry is characterized by an unweighted similarity graph $\mathcal{G} = \big(\mathcal{V} := \{1, \ldots, m\} \mathcal{E}\big)$. The node $i \in \mathcal{V}$ represents the i-th data point, with raw feature vector $\mathbf{z}^{(i)}$. Two nodes are connected by an undirected edge if the corresponding data points are similar. We would like to find short feature vectors $\mathbf{x}^{(i)}$, for $i = 1, \ldots, m$, such that two data points i, i', whose feature vectors $\mathbf{x}^{(i)}, \mathbf{x}^{(i')}$ have small Euclidean distance, are well-connected to each other. To make this requirement precise we need to define a measure for how well two nodes of an undirected graph are connected. We refer the reader to literature on network theory for an overview and details of various connectivity measures [62].

Let us discuss a simple but powerful technique to map the nodes $i \in \mathcal{V}$ of an undirected graph $\mathcal{G}$ to (short) feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$. This map is such that the Euclidean distances between the feature vectors of two nodes reflect their connectivity within $\mathcal{G}$. This technique uses the Laplacian matrix $\mathbf{L} in \mathbb{R}^{(i)}$ which is defined for an undirected graph $\mathcal{G}$ (with node set $\mathcal{V} = \{1, \ldots, m\}$) element-wise

$$L_{i,j} := \begin{cases} -1 & \text{, if } \{i,j\} \in \mathcal{E} \\ d^{(i)} & \text{, if } i = j \\ 0 & \text{otherwise.} \end{cases} . \tag{9.12}$$

Here, $d^{(i)} := \big|\{j : \{i,j\} \in \mathcal{E}\}\big|$ denotes the degree, or the number of neighbours, of node $i \in \mathcal{V}$. It can be shown that the Laplacian matrix $\mathbf{L}$ is **psd** [85, Proposition 1]. Therefore we can find an orthonormal set of eigenvectors

$$\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(m)} \in \mathbb{R}^m \tag{9.13}$$

with corresponding (ordered in a non-decreasing fashion) eigenvalues $\lambda_1 \leq \ldots \leq \lambda_m$ of $\mathbf{L}$.

It turns out that, for a prescribed number n of numeric features, the entries $u_i^{(1)}, \ldots, u_i^{(n)}$ of the first n eigenvectors (9.13) result in feature vectors whose Euclidean distances reflect the connectivities of data points in the similarity graph $\mathcal{G}$. For a more precise statement of this informal claim we refer to the excellent tutorial [85]. Thus, we obtain a feature learning method for (non-numeric)data points via using the eigenvectors of the graph Laplacian associated with the similarity graph of the data points. Algorithm 15 summarizes this feature learning method which requires the similarity graph of the dataset and the desired number of new features as input. Note that Algorithm 15 does not make any use of the Euclidean distances between raw feature vectors and uses solely the similarity graph $\mathcal{G}$ for determining the intrinsic geometry of $\mathcal{D}$.

Algorithm 15 Feature Learning for Non-Numeric Data

Input: dataset $\mathcal{D} = \{\mathbf{z}^{(i)} \in \mathbb{R}^d\}_{i=1}^m$; similarity graph $\mathcal{G}$; number n of features to be constructed for each data point.

1: construct Laplacian matrix $\mathbf{L}$ of similarity graph (see (9.12))

2: compute EVD of $\mathbf{L}$ to obtain n orthonormal eigenvectors (9.13) corresponding to the smallest eigenvalues of $\mathbf{L}$

3: for each data point i, construct feature vector

$$\mathbf{x}^{(i)} := \big(u_i^{(1)}, \ldots, u_i^{(n)}\big)^T \in \mathbb{R}^n \tag{9.14}$$

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

9.4 Feature Learning for Labeled Data

We have discussed PCA as a linear dimensionality reduction method. PCA learns a compression matrix that maps raw features $\mathbf{z}^{(i)}$ of data points to new (much shorter) feature vectors $\mathbf{x}^{(i)}$.

The feature vectors $\mathbf{x}^{(i)}$ determined by PCA depend solely on the raw feature vectors $\mathbf{z}^{(i)}$ of the data points in a given dataset $\mathcal{D}$. In particlar, PCA determines the compression matrix such that the new features allow for a linear reconstruction (9.6) with minimum reconstruction error (9.7).

For some application domains we might not only have access to raw feature vectors but also to the label values $y^{(i)}$ of the data points in $\mathcal{D}$. Indeed, dimensionality reduction methods might be used as pre-processing step within a regression or classification problem that involves a labeled training set. However, in its basic form, PCA (see Algorithm 14) does not allow to exploit the information provided by available labels $y^{(i)}$ of data points $\mathbf{z}^{(i)}$. For some datasets, PCA might deliver feature vectors that are not very relevant for the overall task of predicting the label of a data point.

Let us now discuss a modification of PCA that exploits the information provided by available labels of the data points. The idea is to learn a linear construction map (matrix) $\mathbf{W}$ such that the new feature vectors $\mathbf{x}^{(i)} = \mathbf{W}\mathbf{z}^{(i)}$ allow to predict the label $y^{(i)}$ as good as possible. We restrict the prediction to be linear,

$$\hat{y}^{(i)} := \mathbf{r}^T \mathbf{x}^{(i)} = \mathbf{r}^T \mathbf{W} \mathbf{z}^{(i)}, \tag{9.15}$$

with some weight vector $\mathbf{r} \in \mathbb{R}^n$.

While PCA is motivated by minimizing the reconstruction error (9.1), we now aim at minimizing the prediction error $\hat{y}^{(i)} - y^{(i)}$. In particular, we assess the usefulness of a given pair of construction map $\mathbf{W}$ and predictor $\mathbf{r}$ (see (9.15)), using the empirical risk

$$\begin{aligned} \mathcal{E}\big(\mathbf{W}, \mathbf{r} \mid \mathcal{D}\big) &:= (1/m) \sum_{i=1}^{m} \big(y^{(i)} - \hat{y}^{(i)}\big)^2 \\ &\overset{(9.15)}{=} (1/m) \sum_{i=1}^{m} \big(y^{(i)} - \mathbf{r}^T \mathbf{W} \mathbf{z}^{(i)}\big)^2. \end{aligned} \tag{9.16}$$

to guide the learning of a compressing matrix $\mathbf{W}$ and corresponding linear predictor weights $\mathbf{r}$ ((9.15)).

To characterize the optimal matrix $\mathbf{W}$, minimizing the empirical risk (9.16), we use the EVD (9.10) of the sample covariance matrix $\mathbf{Q}$ (9.9) which we already used for PCA (see (9.8)). Remember that PCA uses the n eigenvectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}$ of $\mathbf{Q}$ corresponding to the n largest eigenvalues of $\mathbf{Q}$. In contrast, to minimize (9.16), we need to use a different set of eigenvectors in the rows of $\mathbf{W}$ in general. To find the right set of n eigenvectors, we need

the sample cross-correlation vector

$$\mathbf{q} := (1/m) \sum_{i=1}^{m} y^{(i)} \mathbf{z}^{(i)}. \tag{9.17}$$

The entry q_j of the vector $\mathbf{q}$ estimates the correlation between the raw feature $z_j^{(i)}$ and the label $y^{(i)}$. We then define the index set

$$\mathcal{S} := \{j_1, \ldots, j_n\} \text{ such that } \big(q_j\big)^2/\lambda_j \geq \big(q_{j'}\big)^2/\lambda_{j'} \text{ for any } j \in \mathcal{S}, j' \in \{1, \ldots, d\} \notin \mathcal{S}. \tag{9.18}$$

It can then be shown that the rows of the optimal compression matrix $\mathbf{W}$ are the eigenvectors $\mathbf{u}^{(j)}$ with $j \in \mathcal{S}$. We summarize the overall feature learning method in Algorithm 16.

Algorithm 16 Linear Feature Learning for Labeled Data

Input: dataset $\big(\mathbf{z}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{z}^{(m)}, y^{(m)}\big)$ with raw features $\mathbf{z}^{(i)} \in \mathbb{R}^d$ and numeric labels $y^{(i)} \in \mathbb{R}$; number n of new features.

1: compute EVD (9.10) of the sample-covariance matrix (9.9) 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: compute the sample cross-correlation vector (9.17) and, in turn, the sequence

$$\big(q_1\big)^2/\lambda_1, \ldots, \big(q_d\big)^2/\lambda_d \tag{9.19}$$

3: determine indices $i_1, \ldots, i_n$ of n largest elements in (9.19)

4: construct compression matrix $\mathbf{W} := \big(\mathbf{u}^{(i_1)}, \ldots, \mathbf{u}^{(i_n)}\big)^T \in \mathbb{R}^{n \times d}$

5: compute feature vector $\mathbf{x}^{(i)} = \mathbf{W}\mathbf{z}^{(i)}$

Output: $\mathbf{x}^{(i)}$, for $i = 1, \ldots, m$, and compression matrix $\mathbf{W}$.

The main focus of this section was on regression problems involving data points with numeric labels. Given the raw features and labels of the data point in the dataset $\mathcal{D}$, Algorithm 16 determines new feature vectors $\mathbf{x}^{(i)}$ that allow to linearly predict a numeric label with minimum squared error. A similar approach can be used for classification problems involving data points with discrete labels. The resulting linear feature learning methods are known as **linear discriminant analysis** or **Fisher discriminant analysis** [37].

9.5 Privacy-Preserving Feature Learning

Many important application domains of ML involve sensitive data that is subject to data protection law [86]. Consider a health-care provider (such as a hospital) holding a large database of patient records. From a ML perspective this databases is nothing but a (typically large) set of data points representing individual patients. The data points are characterized by many features including personal identifiers (name, social security number), bio-physical parameters as well as examination results . We could apply ML to learn a predictor for the risk of particular disease given the features of a data point.

Given large patient databases, the ML methods might not be implemented locally at the hospital but using cloud computing. However, data protection requirements might prohibit the transfer of raw patient records that allow to match individuals with bio-physical properties. In this case we might apply feature learning methods to construct new features for each patient such that they allow to learn an accurate hypothesis for predicting a disease but do not allow to identify sensitive properties of the patient such as its name or a social security number.

Let us formalize the above application by characterizing each data point (patient in the hospital database) using raw feature vector $\mathbf{z}^{(i)} \in \mathbb{R}^d$ and a sensitive numeric property $\pi^{(i)}$. We would like to find a compression map $\mathbf{W}$ such that the resulting features $\mathbf{x}^{(i)} = \mathbf{W}\mathbf{z}^{(i)}$ do not allow to accurately predict the sensitive property $\pi^{(i)}$. The prediction of the sensitive property is restricted to be a linear $\hat{\pi}^{(i)} := \mathbf{r}^T\mathbf{x}^{(i)}$ with some weight vector $\mathbf{r}$.

Similar to Section 9.4 we want to find a compression matrix $\mathbf{W}$ that transforms, in a linear fashion, the raw feature vector $\mathbf{z} \in \mathbb{R}^d$ to a new feature vector $\mathbf{x} \in \mathbb{R}^n$. However the design criterion for the optimal compression matrix $\mathbf{W}$ was different in Section 9.4 where the new feature vectors should allow for an accurate linear prediction of the label. In contrast, here we want to construct feature vectors such that there is no linear predictor of the sensitive property $\pi^{(i)}$.

As in Section 9.4, the optimal compression matrix $\mathbf{W}$ is given row-wise by the eigenvectors of the sample covariance matrix (9.9). However, the choice of which eigenvectors to use is different and based on the entries of the sample cross-correlation vector

$$\mathbf{c} := (1/m)\sum_{i=1}^{m} \pi^{(i)}\mathbf{z}^{(i)}. \tag{9.20}$$

We summarize the construction of the optimal privacy-preserving compression matrix and

corresponding new feature vectors in Algorithm 17.

Algorithm 17 Privacy Preserving Feature Learning

Input: dataset $\big(\mathbf{z}^{(1)}, y^{(1)}\big), \ldots, \big(\mathbf{z}^{(m)}, y^{(m)}\big)$ with raw features $\mathbf{z}^{(i)} \in \mathbb{R}^d$ and (numeric) sensitive property $\pi^{(i)} \in \mathbb{R}$; number n of new features.

1: compute EVD (9.10) of the sample-covariance matrix (9.9) 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: compute the sample cross-correlation vector (9.20) and, in turn, the sequence

$$\big(c_1\big)^2/\lambda_1, \ldots, \big(c_d\big)^2/\lambda_d \tag{9.21}$$

3: determine indices $i_1, \ldots, i_n$ of n smallest elements in (9.21)

4: construct compression matrix $\mathbf{W} := \big(\mathbf{u}^{(i_1)}, \ldots, \mathbf{u}^{(i_n)}\big)^T \in \mathbb{R}^{n \times d}$

5: compute feature vector $\mathbf{x}^{(i)} = \mathbf{W}\mathbf{z}^{(i)}$

Output: $\mathbf{x}^{(i)}$, for $i = 1, \ldots, m$, and compression matrix $\mathbf{W}$.

9.6 Random Projections

Note that PCA amounts to computing an EVD of the sample covariance matrix $\mathbf{Q}$ (9.9). The computational complexity (e.g., measured by number of multiplications and additions) for computing this EVD is lower bounded by $\min\{D^2, m^2\}$ [22, 74]. 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 computationally cheap alternative to PCA (Algorithm 14) for finding a useful compression matrix $\mathbf{W}$ in (9.5). This alternative is to construct the compression matrix $\mathbf{W}$ entry-wise

$$W_{i,j} := a^{(i,j)} \text{ with i.i.d. } a_{i,j} \sim p(a). \tag{9.22}$$

The entries of the matrix (9.22) are realizations of i.i.d. RVs $a_{i,j}$ with some common probability distribution $p(a)$. Different choices for the probability distribution $p(a)$ have been studied in the literature [29]. The Bernoulli distribution is used to obtain a compression matrix with binary entries. Another popular choice for $p(a)$ is the multivariate normal (Gaussian)

distribution.

Consider data points whose raw feature vectors $\mathbf{z}$ are located near a s-dimensional subspace of $\mathbb{R}^d$. The feature vectors $\mathbf{x}$ obtained via (9.5) using a random matrix (9.22) allows to reconstruct the raw feature vectors $\mathbf{z}$ with high probability whenever

$$n \geq Cs \log d. \tag{9.23}$$

The constant C depends on the maximum tolerated reconstruction error η (such that $\|\widehat{\mathbf{z}} - \mathbf{z}\|_2^2 \leq \eta$ for any data point) and the probability that the features $\mathbf{x}$ (see)(9.22)) allow for a maximum reconstruction error η [29, Theorem 9.27.].

9.7 Dimensionality Increase

The focus of this chapter is on dimensionality reduction methods that learn feature maps delivering new feature vectors which are (significantly) shorter than the raw feature vectors. However, it might be beneficial to learn feature maps that deliver new feature vectors which are longer than the raw feature vectors. We have already discussed two examples for such feature learing methods in Sections 3.2 and 3.9. Polynomial regression maps a single raw feature z to a feature vector containing the powers of the raw feature z. This allows to use apply linear predictor maps to the new feature vectors to obtain predictions that depend non-linearly on the raw feature z. Kernel methods use feature maps that deliver feature vectors belonging to a possibly infinite-dimensional **Hilbert space** [72].

Mapping raw feature vectors into higher-dimensional (or even infinite-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 (see Figure 3.5). Loosely speaking, mapping into higher-dimensional feature space might "flatten-out" a non-linear decision boundary between data points. We can then apply linear classifiers to the higher-dimensional features to achieve accurate predictions.

9.8 Exercises

9.8.1 Computational Burden of Many Features

Discuss the computational complexity of linear regression. How much computation do we need to compute the linear predictor that minimizes the average squared error on a training set?

9.8.2 Linear Classifiers with High-Dimensional Features

Consider a training set $\mathcal{D}$ consisting of $m = 10^{10}$ labeled data points $\left(\mathbf{z}^{(1)}, y^{(1)}\right), \ldots, \left(\mathbf{z}^{(m)}, y^{(m)}\right)$ with raw feature vectors $\mathbf{z}^{(i)} \in \mathbb{R}^{4000}$ and binary labels $y^{(i)} \in \{-1, 1\}$. Assume we have used a feature learning method to obtain the new features $\mathbf{x}^{(i)} \in \{0, 1\}^n$ with $n = m$ and such that the only non-zero entry of $\mathbf{x}^{(i)}$ is $x_i^{(i)} = 1$, for $i = 1, \ldots, m$. Can you find a linear classifier that perfectly classifies the training set?

Chapter 10

Lists of Symbols

10.1 Sets

$\mathbb{R}$	The set of real numbers x.
$\mathbb{R}_+$	The set of non-negative real numbers $x \geq 0$.
$[0, 1]$	Closed interval of real numbers x with $0 \leq x \leq 1$.

10.2 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.
$\mathbf{x} = (x_1, \ldots, x_n)^T$	A vector of length n. The jth entry of the vector is denoted x_j.
$\|\mathbf{x}\|_2$	The Euclidean norm of the vector $\mathbf{x}$, $\|\mathbf{x}\|_2 := \sqrt{\sum_{j=1}^n x_j^2}$.
y	The label of some datapoint.
$\|\mathbf{x}\|$	Some norm of the vector $\mathbf{x}$. Unless specified otherwise, we mean the Euclidean norm.

10.3 Machine Learning

t	A discrete time index.
i	Generic index used to enumerate datapoints in a list of datapoints.
m	The number of datapoints in a dataset which might be used to learn a good hypothesis.
$h(\cdot)$	A hypothesis map that reads in a feature vector $\mathbf{x}$ of a datapoint and delivers the 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.
$\mathbf{x}$	A feature vector of datapoints whose entries are the features of a datapoint.
$\mathbf{x}^{(i)}$	Feature vector whose entries are the features of the ith datapoint.
$\mathbf{z}$	Beside the symbol $\mathbf{x}$ we use $\mathbf{z}$ as another symbol to denote a vector whose entries are features of a data point. We need two symbols for feature vectors for discussing methods that transform a given set of features to another set of features (see Chapter 9).
n	The number of (real-valued) features of a single datapoint.
x_j	The jth entry of a vector $\mathbf{x} = \big(x_1, \ldots, x_n\big)^T$.
$\mathcal{L}((\mathbf{x}, y), h)$	The loss incurred by predicting the label y of data points using the hypothesis $h(\mathbf{x})$.
E_v	The validation error of a hypothesis, which is the average loss computed on a validation set.
E_t	The training error of a hypothesis, which is the average loss computed on a training set.

Chapter 11

Glossary

- **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 $h : \mathcal{X} \rightarrow \mathcal{Y}$ 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. RVs" means that the joint probability distribution $p(x, y, z)$ of the RVs 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.

Glossary

bagging bagging (or "bootstrap aggregation") is a generic technique to improve or robustify a given ML method. The idea is to use the bootstrap to generate perturbed copy of a given training set and then apply the original ML method to learn a separate hypothesis for each perturbed copy of the training set. The resulting set of hypotheses is then used to predict the label of a data point by combining or aggregating the individual predictions of each hypothesis. For numeric label values (regression) this aggregation could be obtained by the average of individual predictions. 145

classifier A classifier is a hypothesis $h(\mathbf{x})$ that is used to predict a finite-valued label. Strictly speaking a classifier is a hypothesis $h(\mathbf{x})$ that can take only a finite number of different values. However, we are sometimes sloppy and use the term classifier also for a hypothesis that can take on any real number which is then used in a simple thresholding to determine the predicted label value. For example, in a binary classification problem with label values $y \in \{-1, 1\}$, we refer to a linear hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ as classifier if it is used to predict the label value according to $\hat{y} = 1$ when $h(\mathbf{x}) \geq 0$ and $\hat{y} = -1$ otherwise. 35

data augmentation Data augmentation methods add synthetic data points to a given set of data point. The augmented data points might be obtained by applying certain perturbations (adding noise) or transformations (rotations of images) to the original data points. 140

Hilbert space A Hilbert space is a linear vector space that is equipped with an inner product between pairs of vectors. One important example for a Hilbert spaces is the Euclidean spaces $\mathbb{R}^n$, for some dimension n, which consists of Euclidean vectors $\mathbf{u} = \big(u_1, ldots, u_n\big)^T$ along with the inner product $\mathbf{u}^T\mathbf{v}$. 196

law of large numbers The law of large numbers refers to the convergence of the partial sums of i.i.d. RVs to the (common) expectation these RVs. 47, 137, 147, 148

linear classifier A classifier $h(\mathbf{x})$ maps the feature vector $\mathbf{x} \in \mathbb{R}^n$ of a datapoint to a predicted label $\hat{y} \in \mathcal{Y}$ out of a finite set of label values $\mathcal{Y}$. We can characterize such a classifier equivalently by the decision regions $\mathcal{R}^{(a)} := \{\mathbf{x} \in \mathbb{R}^n : \hat{y} = a\}$, for every possible label value $a \in \mathcal{Y}$. Linear classifiers are such that the boundaries between the regions $\mathcal{R}^{(a)}$ are hyperplanes in $\mathbb{R}^n$. 38, 39

maximum Given a set of real numbers, the maximum is the largest of those numbers. 56

mean The expectation of a real-valued random variable. 34

minimum Given a set of real numbers, the minimum is the smallest of those numbers. 56

positive semi-definite A matrix $\mathbf{X} \in \mathbb{C}^{n \times n}$ is referred to as positive semi-definite if $\mathbf{x}^H \mathbf{X} \mathbf{x} \geq 0$ for every vector $\mathbf{x} \in \mathbb{R}^n$. 79, 152, 184, 191, 200

sample A finite sequence (list) of data points $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(i)}$. The length m of the sequence is also known as the **sample size**. The data points might be interpreted as the realizations of i i.i.d. RVs with the common probability distribution $p(\mathbf{z})$. 114

sample size Number of individual data points contained in a dataset. 26

variance The expectation of the squared difference between a real-valued random variable and its expectation. 34

Acronyms

psd positive semi-definite *Glossary:* **positive semi-definite**

Bibliography

[1] M. Abramowitz and I. A. Stegun, editors. *Handbook of Mathematical Functions*. Dover, New York, 1965.

[2] 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.

[3] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding". In *Proc. of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics Philadelphia, 2007.

[4] P. Austin, P. Kaski, and K. Kubjas. Tensor network complexity of multilinear maps. *arXiv*, 2018.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. addwes, 1999.

[6] F. Barata, K. Kipfer, M. Weber, P. Tinschert, E. Fleisch, and T. Kowatsch. Towards device-agnostic mobile cough detection with convolutional neural networks. In *2019 IEEE International Conference on Healthcare Informatics (ICHI)*, pages 1–11, 2019.

[7] M. Belkin, D. Hsu, S. Ma, and S. Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[8] D. Bertsekas and J. Tsitsiklis. *Introduction to Probability*. Athena Scientific, 2 edition, 2008.

[9] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 2nd edition, June 1999.

[10] P. Billingsley. *Probability and Measure*. Wiley, New York, 3 edition, 1995.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] B. Boashash, editor. *Time Frequency Signal Analysis and Processing: A Comprehensive Reference*. Elsevier, Amsterdam, The Netherlands, 2003.

[13] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge Univ. Press, Cambridge, UK, 2004.

[14] S. Bubeck. Convex optimization. algorithms and complexity. In *Foundations and Trends in Machine Learning*, volume 8. Now Publishers, 2015.

[15] P. Bühlmann and S. van de Geer. *Statistics for High-Dimensional Data*. Springer, New York, 2011.

[16] S. Carrazza. Machine learning challenges in theoretical HEP. *arXiv*, 2018.

[17] R. Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

[18] N. Cesa-Bianchi and G. Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, New York, NY, USA, 2006.

[19] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. The MIT Press, Cambridge, Massachusetts, 2006.

[20] I. Cohen and B. Berdugo. Noise estimation by minima controlled recursive averaging for robust speech enhancement. *IEEE Sig. Proc. Lett.*, 9(1):12–15, Jan. 2002.

[21] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*, (4):303–314, 1989.

[22] Q. Du and J. Fowler. Low-complexity principal component analysis for hyperspectral image compression. *Int. J. High Performance Comput. Appl*, pages 438–448, 2008.

[23] O. Dürr, Y. Pauchard, D. Browarnik, R. Axthelm, and M. Loeser. Deep learning on a raspberry pi for real time face recognition. 01 2015.

[24] B. Efron and R. Tibshirani. Improvements on cross-validation: The 632+ bootstrap method. *Journal of the American Statistical Association*, 92(438):548–560, 1997.

[25] R. Eldan and O. Shamir. The power of depth for feedforward neural networks. *CoRR*, abs/1512.03965, 2015.

[26] Y. C. Eldar, P. Kuppinger, and H. Bölcskei. Block-sparse signals: Uncertainty relations and efficient recovery. *IEEE Trans. Signal Processing*, 58(6):3042–3054, June 2010.

[27] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, Portland, Oregon, 1996.

[28] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542, 2017.

[29] S. Foucart and H. Rauhut. *A Mathematical Introduction to Compressive Sensing*. Springer, New York, 2012.

[30] 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.

[31] W. Gautschi and G. Inglese. Lower bounds for the condition number of vandermonde matrices. *Numer. Math.*, 52:241 – 250, 1988.

[32] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[33] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[34] 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.

[35] R. Gray, J. Kieffer, and Y. Linde. Locally optimal block quantizer design. *Information and Control*, 45:178 – 198, 1980.

[36] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, March/April 2009.

[37] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, New York, NY, USA, 2001.

[38] T. Hastie, R. Tibshirani, and M. Wainwright. *Statistical Learning with Sparsity. The Lasso and its Generalizations*. CRC Press, 2015.

[39] E. Hazan. *Introduction to Online Convex Optimization*. Now Publishers Inc., 2016.

[40] J. Howard and S. Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[41] P. Huber. Approximate models. In C. Huber-Carol, N. Balakrishnan, M. Nikulin, and M. Mesbah, editors, *Goodness-of-Fit Tests and Model Validity. Statistics for Industry and Technology.* Birkhäuser, Boston, MA, 2002.

[42] P. J. Huber. *Robust Statistics*. Wiley, New York, 1981.

[43] L. Hyafil and R. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[44] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer, 2013.

[45] A. Jung. A fixed-point of view on gradient methods for big data. *Frontiers in Applied Mathematics and Statistics*, 3, 2017.

[46] S. M. Kay. *Fundamentals of Statistical Signal Processing: Estimation Theory*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[47] P. Koehn. Europarl: A parallel corpus for statistical machine translation. In *The 10th Machine Translation Summit, page 79–86., AAMT,*, Phuket, Thailand, 2005.

[48] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems, NIPS*, 2012.

[49] B. Kulis and M. I. Jordan. Revisiting k-means: New algorithms via bayesian nonparametrics. In *Proc. of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012.

[50] C. Lampert. Kernel methods in computer vision. *Foundations and Trends in Computer Graphics and Vision*, 2009.

[51] 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.

[52] E. L. Lehmann and G. Casella. *Theory of Point Estimation*. Springer, New York, 2nd edition, 1998.

[53] L. Li, W. Chu, J. Langford, and R. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proc. International World Wide Web Conference*, pages 661–670, Raleigh, North Carolina, USA, April 2010.

[54] H. Lütkepohl. *New Introduction to Multiple Time Series Analysis*. Springer, New York, 2005.

[55] S. G. Mallat. *A Wavelet Tour of Signal Processing – The Sparse Way*. Academic Press, San Diego, CA, 3 edition, 2009.

[56] K. V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Academic Press, 1979.

[57] T. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR 5-110,, Rutgers University, New Brunswick, New Jersey, USA, 1980.

[58] 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.

[59] R. Muirhead. *Aspects of Multivariate Statistical Theory*. John Wiley & Sons Inc., 1982.

[60] 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.

[61] Y. Nesterov. *Introductory lectures on convex optimization*, volume 87 of *Applied Optimization*. Kluwer Academic Publishers, Boston, MA, 2004. A basic course.

[62] M. E. J. Newman. *Networks: An Introduction*. Oxford Univ. Press, 2010.

[63] A. Ng. *Shaping and Policy search in Reinforcement Learning*. PhD thesis, University of California, Berkeley, 2003.

[64] 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.

[65] S. Oymak, B. Recht, and M. Soltanolkotabi. Sharp time–data tradeoffs for linear inverse problems. *IEEE Transactions on Information Theory*, 64(6):4129–4158, June 2018.

[66] S. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[67] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):123–231, 2013.

[68] F. Pedregosa. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[69] H. Poor. *An Introduction to Signal Detection and Estimation*. Springer, 2 edition, 1994.

[70] S. Roweis. EM Algorithms for PCA and SPCA. In *Advances in Neural Information Processing Systems*, pages 626–632. MIT Press, 1998.

[71] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, New York, 3 edition, 1976.

[72] B. Schölkopf and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, Dec. 2002.

[73] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning – from Theory to Algorithms*. Cambridge University Press, 2014.

[74] A. Sharma and K. Paliwal. Fast principal component analysis using fixed-point analysis. *Pattern Recognition Letters*, 28:1151 – 1155, 2007.

[75] S.Levine, C. Finn, T. Darrell, and P.Abbeel. End-to-end training of deep visuomotor policies. *J. Mach. Learn. Res.*, 17, 2016.

[76] 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.

[77] A. Sorokin and D. Forsyth. Utility data annotation with amazon mechanical turk. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, 2008.

[78] S. Sra, S. Nowozin, and S. J. Wright, editors. *Optimization for Machine Learning*. MIT Press, 2012.

[79] G. Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, MA, 2007.

[80] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, MA, 5 edition, 2016.

[81] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT press, Cambridge, MA, 2 edition, 2018.

[82] M. E. Tipping and C. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 21/3:611–622, January 1999.

[83] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1999.

[84] O. Vasicek. A test for normality based on sample entropy. *Journal of the Royal Statistical Society. Series B (Methodological)*, 38(1):54–59, 1976.

[85] U. von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, Dec. 2007.

[86] S. Wachter. Data protection in the age of big data. *Nature Electronics*, 2(1):6–7, 2019.

[87] S. Wade and Z. Ghahramani. Bayesian Cluster Analysis: Point Estimation and Credible Balls (with Discussion). *Bayesian Analysis*, 13(2):559 – 626, 2018.

[88] M. Wainwright. *High-Dimensional Statistics: A Non-Asymptotic Viewpoint*. Cambridge: Cambridge University Press, 2019.

[89] A. Wang. An industrial-strength audio search algorithm. In *International Symposium on Music Information Retrieval*, Baltimore, MD, 2003.

[90] 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.

[91] L. Xu and M. Jordan. On convergence properties of the EM algorithm for Gaussian mixtures. *Neural Computation*, 8(1):129–151, 1996.

[92] K. Young. Bayesian diagnostics for checking assumptions of normality. *Journal of Statistical Computation and Simulation*, 47(3–4):167 – 180, 1993.