

Machine Learning: The Basics

!! ROUGH DRAFT !!

Alexander Jung

December 22, 2020



Figure 1: Machine learning methods implement the scientific principle of iteratively validating and refining a hypothesis (or model) about some “real-world” phenomenon. This principle involves some data, some model (or hypothesis space) and some loss function.

Preface

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

This book portrays machine learning methods as the combination of three basic components: data, model and loss. ML methods combine these three components as computationally efficient implementations of the basic scientific principle “trial and error”. Trial and error amounts to continuously adapting a model about some data generating phenomenon by minimizing some notion of loss. A plethora of different ML methods is obtained by combining different design choices for the data representation, model and loss.

Machine learning methods differ vastly in their actual implementations which obscures their unifying basic principles. Deep learning methods use cloud computing frameworks to train large models on huge online datasets. Operating on a much finer granularity for data and computation, linear least squares regression can be implemented on small embedded systems. However, deep learning methods and linear regression use the same principle of iteratively updating a model based on the discrepancy between model predictions and actual observed data.

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

To make good use of widely available ML tools it is instrumental to understand its underlying principles at different levels of detail. On a lower-level, this tutorial helps machine learning engineers to choose suitable methods for the application at hand. We also provide

leaders a higher-level view on the development of ML which is required to manage a machine learning team. We believe that thinking about ML using this three components helps to navigate the steadily growing offer for ready-to-use ML methods.

Acknowledgement

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

This text benefited from the numerous feedback of the students within the courses that have been (co-)taught by the author. The author is indebted to Shamsiat Abdurakhmanova, Tomi Janhunen, 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, Oleg Vlasovetc, Georgios Karakasidis, Joni Pääkkö, Harri Wallenius and Satu Korhonen.

Contents

1	Introduction	8
1.1	Linear Algebra	12
1.2	Optimization	13
1.3	Theoretical Computer Science	13
1.4	Communication	14
1.5	Statistics	14
1.6	Artificial Intelligence	15
1.7	Organization of this Book	18
2	Three Components of ML: Data, Model and Loss	21
2.1	The Data	22
2.1.1	Features	23
2.1.2	Labels	26
2.1.3	Scatterplot	27
2.1.4	Probabilistic Models for Data	28
2.2	The Model	29
2.3	The Loss	35
2.4	Putting Together the Pieces	42
2.5	Exercises	45
2.5.1	How Many Features?	45
2.5.2	Multilabel Prediction	45
2.5.3	Average Squared Error Loss as Quadratic Form	45
2.5.4	Find Labeled Data for Given Empirical Risk	45
2.5.5	Dummy Feature Instead of Intercept	46
2.5.6	Approximate Non-Linear Maps Using Indicator Functions for Feature Maps	46

2.5.7	Python Hypothesis Space	46
2.5.8	A Lot of Features	46
2.5.9	Over parametrization	47
2.5.10	Squared Error Loss	47
2.5.11	Classification Loss	47
2.5.12	Intercept Term	47
2.5.13	Picture Classification	48
2.5.14	Maximum Hypothesis Space	48
2.5.15	A Large but Finite Hypothesis Space	48
2.5.16	Size of Linear Hypothesis Space	48
3	Some Examples	49
3.1	(Least Squares) Linear Regression	49
3.2	Polynomial Regression	50
3.3	Least Absolute Deviation Regression	52
3.4	Gaussian Basis Regression	52
3.5	Logistic Regression	54
3.6	Support Vector Machines	56
3.7	Bayes' Classifier	58
3.8	Kernel Methods	58
3.9	Decision Trees	60
3.10	Artificial Neural Networks – Deep Learning	62
3.11	Maximum Likelihood Methods	65
3.12	k -Nearest Neighbours	66
3.13	Dimensionality Reduction	67
3.14	Clustering Methods	67
3.15	Reinforcement Learning Methods	67
3.16	Network Lasso	67
3.17	Exercises	71
3.17.1	How Many Neurons?	71
3.17.2	Linear Classifiers	71
3.17.3	Data Dependent Hypothesis Space	71
4	Empirical Risk Minimization	72
4.1	The Basic Idea of ERM	73

4.2	ERM for Linear Regression	76
4.3	ERM for Decision Trees	78
4.4	ERM for Bayes' Classifiers	80
4.5	Online Learning	83
4.6	Training and Inference Periods	84
4.7	Exercise	84
4.7.1	Linear Regression	84
5	Gradient Based Learning	85
5.1	The Basic GD Step	86
5.2	Choosing Step Size	88
5.3	When To Stop	89
5.4	GD for Linear Regression	89
5.5	GD for Logistic Regression	91
5.6	Data Normalization	93
5.7	Stochastic GD	94
5.8	Exercises	95
5.8.1	Use Knowledge About Problem Class	95
6	Model Validation and Selection	97
6.1	Overfitting	98
6.2	Validation	101
6.3	Model Selection	103
6.4	Bias, Variance and Generalization within Linear Regression	104
6.5	Diagnosis	109
6.6	Exercises	110
6.6.1	Validation Set Size	110
7	Regularization	111
7.1	Regularized ERM	112
7.2	Robustness	112
7.3	Data Augmentation	113
7.4	Regularized Linear Regression	113
7.5	Semi-Supervised Learning	118
7.6	Multitask Learning	119

7.7	Exercises	119
7.7.1	Ridge Regression as Quadratic Form	119
8	Clustering	121
8.1	Hard Clustering	124
8.2	Soft Clustering	128
9	Feature Learning	134
9.1	Dimensionality Reduction	134
9.2	Principal Component Analysis	136
9.2.1	Combining PCA with Linear Regression	138
9.2.2	How To Choose Number of PC?	139
9.2.3	Data Visualisation	139
9.2.4	Extensions of PCA	140
9.3	Linear Discriminant Analysis	141
9.4	Random Projections	141
9.5	Information Bottleneck	141
9.6	Dimensionality Increase	141
10	Privacy-Preserving ML	143
10.1	Privacy-Preserving Feature Learning (Operating on level of individual data points)	144
10.1.1	Privacy-Preserving Information Bottleneck	144
10.1.2	Privacy-Preserving Feature Selection	144
10.1.3	Privacy-Preserving Random Projections	144
10.2	Federated Learning (Operates on level of local datasets)	145
11	Explainable ML	146
11.1	A Model Agnostic Method	146
11.2	Explainable Empirical Risk Minimization	147
12	Lists of Symbols	148
13	Glossary	149

Chapter 1

Introduction

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



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

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

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

This principle contains three components: data, a model and a loss function. Any machine learning method, including linear regression and deep reinforcement learning, combines

these three components. We will discuss in Chapter 2 that each of these components incur design choices such as which data representation or model class to use. One such choice for data and model representation is to use vectors and matrices. The seemingly overwhelming offer for different ML methods is based on different design choices for data representation, models and loss function.

Let us now illustrate the (rather abstract) concepts behind the main components of ML with the above problem of predicting the maximum daytime temperature. The data are weather observations collected by the Finnish metrological institute. We depict such data in Figure 1.2, with each dot representing some previous day. Each day is characterized by the minimum daytime temperature x and the maximum daytime temperature y .

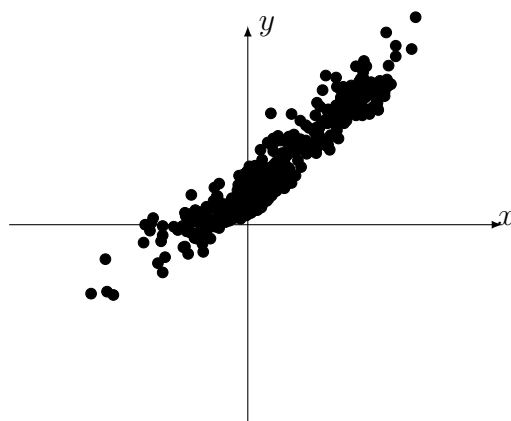


Figure 1.2: A scatterplot where each dot represents some day that is characterized by its minimum daytime temperature x and its maximum daytime temperature y .

ML methods allow to learn a predictor $\hat{y} = h(x)$, reading in the minimum temperature x and delivering a prediction/estimate/approximation $\hat{y} = h(x)$ for the actual maximum daytime temperature y . This prediction is based on a simple model for how the minimum and maximum daytime temperature during some day are related. We assume that they are related approximately by a linear model: $y \approx w_1x + w_0$. Based on this model, it seems reasonable to restrict the ML method to only consider linear predictor maps $h(x) = w_1x + w_0$.

??? Work out this principle for linear regression using FMI data points ???

Figure 1 illustrates the typical workflow of a ML method. Any ML method must start with some initial guess or choice for a good hypothesis. This initial guess might not be made

explicit but each ML method must use such an initial guess. This initial guess is typically based on some prior knowledge or domain expertise [?].

Based on a current hypothesis, ML methods make predictions or forecasts about future observations. These observations are in the form of data collected by some hard or software. The discrepancy between the predictions and the actual observations is used to improve the hypothesis. Learning happens during improving the current hypothesis based on the discrepancy between its predictions and the actual observations.

A prominent example of a phenomenon is the weather. Forecasting the weather is based on learning or fitting models to massive amounts of weather observations. The Finnish Meteorological Institute collects weather measurements at different observation stations distributed all over Finland. We can download the history of minimum and maximum daytime temperature recorded by the station nearest to our home location. Let us denote the resulting dataset by

$$\mathcal{X} = \{z^{(1)}, \dots, z^{(m)}\}. \quad (1.1)$$

Each data point $\mathbf{z}^{(i)}$ represents some previous day for which the minimum and maximum daytime temperature $x^{(i)}$ and $y^{(i)}$ has been recorded. Thus, we can represent such a day by these measurements only. We will use the minimum daytime temperature $x^{(i)}$ as the feature and the maximum daytime temperature $y^{(i)}$ as the label of the datapoint $\mathbf{z}^{(i)}$.

Consider waking up on some day i . We can easily determine the minimum temperature since this is what we can measure in the morning. However, we cannot easily determine the maximum day temperature as this will be attained only a few hours later in early afternoon. However, for this application we can get labeled datapoint, i.e., for which we know the true label by using historic recordings. One main source for labeled data is by looking into the past.

Machine learning (ML) uses data to learn models that allow, in turn, to make predictions of forecasts. The models used in ML come in the form of assumptions or hypothesis for the properties of observed data. Consider the data point \mathbf{z} representing some day with minimum temperature x and maximum temperature y . Then, it seems reasonable to assume that these two properties of the same day are correlated.

For a larger minimum temperature $x^{(i)}$, we typically expect a larger maximum daytime temperature y . We can formalize this assumption by modelling the maximum daytime temperature as $y \approx h(x)$ with some hypothesis map h which is monotonically increasing.

One candidate for such a map is a linear function of the form

$$h(x) = w_1x + w_0 \text{ with some weights } w_1, w_0 \in \mathbb{R}. \quad (1.2)$$

The map (1.2) is monotonically increasing whenever $w_1 > 0$. Note that (1.2) defines a whole ensemble of hypothesis maps, each individual map corresponding to a particular choice for $w_1 \geq 0$ and w_0 . We say that the map (1.2) is parametrized by the weight vector $\mathbf{w} = (w_1, w_0)$ and indicate this by writing $h^{(\mathbf{w})}$. For a given weight vector $\mathbf{w} = (w_1, w_0)$, we obtain the map $h^{(\mathbf{w})}(x) = w_1x + w_0$. Figure 1.3 depicts three particular maps $h^{(\mathbf{w})}$ obtained for three different choices for the weights \mathbf{w} .

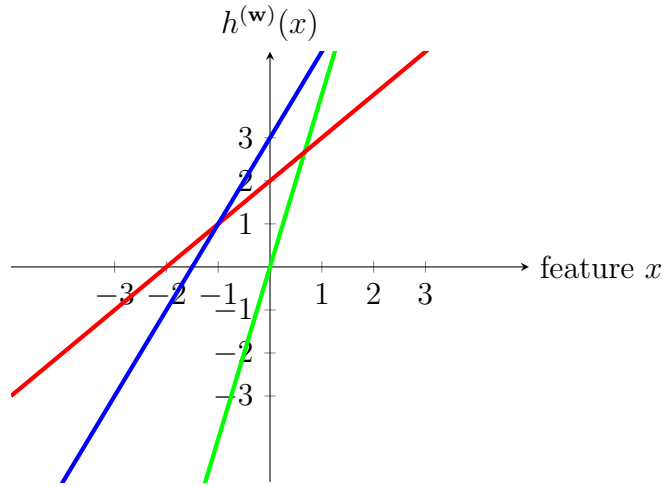


Figure 1.3: Three particular hypothesis maps of the form (1.2).

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 (machine) learning we need to allow for more than one hypothesis. ML allows to choose (learn) a good hypothesis out of (typically very large) hypothesis spaces. The hypothesis space constituted by the maps (1.2) for different weights is uncountably infinite.

To find, or **learn**, a good hypothesis we need to somehow assess the quality of a given hypothesis map. ML methods use data for this purpose. These methods assess the quality of a hypothesis by comparing its predictions with actual observed data.

Consider the hypothesis map (1.2) for predicting the maximum daytime temperature based on the minimum (morning) daytime temperature x . We can evaluate the quality of this map by comparing its predictions with actual observed minimum and maximum temperature for previous days. Figure 1.2 depicts a set of data points represented earlier

days characterized by minimum and maximum temperature. These data points can be downloaded from the Finnish meteorological institute (FMI).

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 important mathematical structure that provides such a representation is the Euclidean space \mathbb{R}^n with some dimension n . The rich algebraic and geometric structure of \mathbb{R}^n allows for the design of ML algorithms that can process vast amounts of data in order to quickly update a model (parameters).

The scatter plot in Figure 1.2 is based on representing data points (individual days) using vectors $\mathbf{z} \in \mathbb{R}$. For a given data point, its vector representation $\mathbf{z} = (x, y)^T$ is obtained by stacking the minimum daytime temperature x and the maximum daytime temperature y into a vector of length two.

We can use the Euclidean space \mathbb{R}^n not only to represent data points but also 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 smallest (see Figure 1.4).

The properties of linear structures, such as straight lines, is studied within linear algebra [?]. 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).

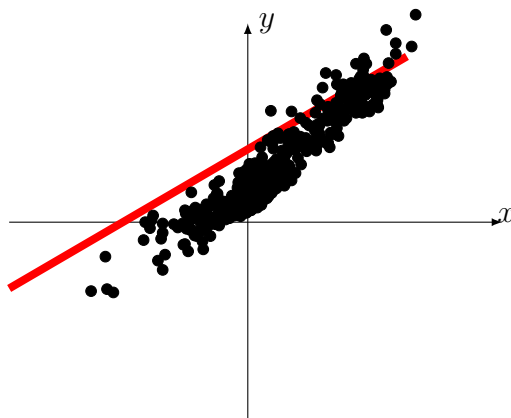


Figure 1.4: Dots represent earlier days characterized by its minimum daytime temperature x and its maximum daytime temperature y . We also depict a straight line as a simple model for the inherent structure in the data.

1.2 Optimization

A main design principle for ML methods is to formulate learning tasks as optimization problems [?]. 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 data points (the training set).

1.3 Theoretical Computer Science

On a high level, ML methods take data as input and produce some prediction as the 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 [? ?]. One of the key factors for the recent success of ML methods is that they use vectors and matrices to represent data and models. Using this representation allows to implement the resulting ML methods using highly efficient hard- and software implementations for numerical linear algebra.

1.4 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. Loosely speaking, the learning progress of ML method will be slowed down if we cannot feed it with data at high speed. Given limited memory or storage capacity, being too slow to process data at their rate of arrival (in real-time) means that we need to “throw away” data. The lost data might have carried relevant information for the ML task at hand.

1.5 Statistics

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

Probability theory offers a great selection on methods for estimating the probability distribution from observed data (see Section 3.11). Given (an estimate of) the probability distribution $p(\mathbf{z})$, we can compute predictions or estimates for the label y of any data point

for which we know its feature value x' . What is more, having a probability distribution $p(\mathbf{z})$ for a randomly drawn data point $\mathbf{z} = (x, y)$, 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 data point. 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 data point $\mathbf{z} = (x, y) \sim p(\mathbf{z})$.

Having an (estimate of) probability distribution $p(\mathbf{z})$ for the observed data points not only allows us to compute predictions but also to generate new data points. Indeed, we can artificially augment the observed data points by random drawing new data point from the probability distribution $p(\mathbf{z})$ (see Section 7.3). A recently popularized class of ML methods that use a probabilistic models to generating synthetic data points is known as **generative adversarial networks** [?].

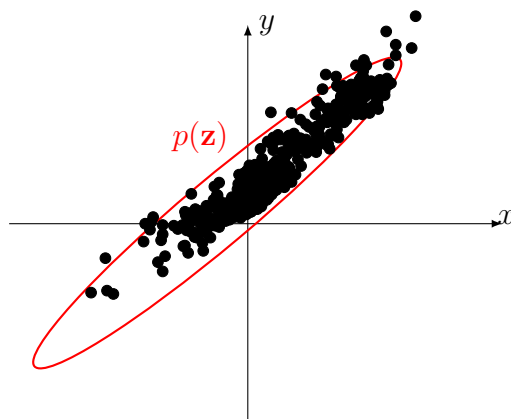


Figure 1.5: A scatterplot where each dot represents some day that is characterized by its minimum daytime temperature x and its maximum daytime temperature y .

1.6 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 reduction of number of forest fires.
- an **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 yummy 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 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, organisation 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).

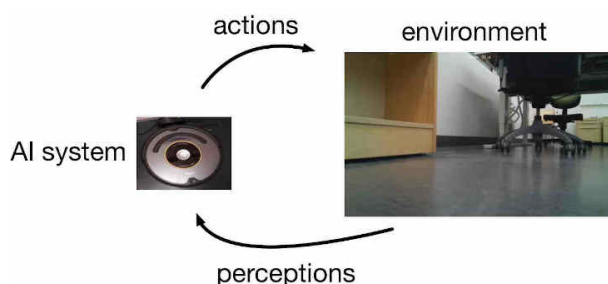


Figure 1.6: A cleaning robot chooses actions (moving directions) to maximize a long-term reward measured by the amount of cleaned floor area per day.

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. As an example we could use an ML method 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 requires an AI system or agent to learn a good hypothesis about how her behaviour affects its environment. We can think of optimal behaviour as the consequent choice of actions that are predicted as optimal according to some hypothesis which could be obtained by ML methods.

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

Consider data points to represent the different states of a toy car. For such data points 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.

1.7 Organization of this Book

We can formalize a particular ML application or problem by defining the **features** and **labels** of data points and choosing the **loss function** for measuring how good a particular ML method performs (see Chapter 2). Besides features, labels and loss function we also highlight the importance of restricting the class of potential predictor maps from features to labels. We will refer to the restricted class of **computationally feasible** maps used by a ML method as its **hypothesis space** or **model**.

We detail in Chapter 3 how some widely used ML methods are obtained as combinations of particular choices for feature and label space, loss function and hypothesis space.

In Chapter 4 we introduce and discuss the principle of **empirical risk minimization (ERM)** which amounts to choosing a predictor based on minimizing the average loss (or empirical risk) incurred over **labeled training data**. The ERM principle amounts to a precise mathematical formulation of the “learning by trial and error” paradigm.

In Chapter 5 we discuss a particular optimization method, i.e., **gradient descent (GD)** which is an iterative method for solving the ERM problem in order to find good predictors. Variations of GD are currently the de-facto standard method for solving ERM arising in large-scale ML problems [?].

We then discuss in Chapter 6 the basic idea of validating a predictor by trying it out on labeled data (so called “validation data”) which is different from the training data used to learn a predictor. As detailed in Chapter 7, a main reason for doing validation is to detect and avoid **overfitting** which causes poor performance of ML methods.

The main focus of this tutorial is on **supervised ML methods** which assign labels to each data point and require the availability of some data points (the training set) whose labels are known. The label of a data point is some quantity of interest such as the fact if an image shows a cat or not. Another example is weather prediction where a data point represents a user query containing a location and time. The label associated with this data point could be the local temperature at the queried location and time.

Supervised ML methods aim at finding a (predictor or classifier) map that reads in

features of a data point and outputs a prediction. The prediction should be an accurate approximation to the true label (see Chapter 2). To find such a map, supervised ML methods use historic data in order to try out different choices for the map and picking the best one.

The basic idea of supervised ML methods, as illustrated in Figure 1.7, is to fit a curve (representing the predictor map) to data points obtained from historic data (see Chapter 4). While this sounds like a trivial task, the challenge of modern ML applications is the amount of data points. ML methods must process billions of data points with each single data point characterized by a potentially vast number of features. Consider data points representing social network users, whose features include all media that has been posted (videos, images, text).

Beside the size of datasets, another computational 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 [?].

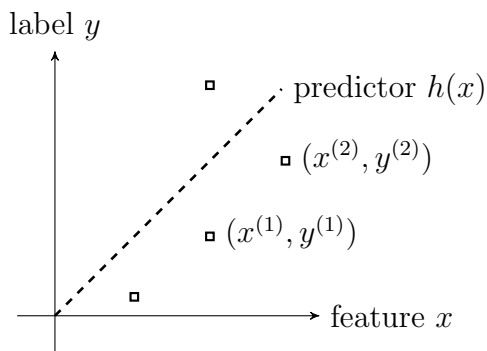


Figure 1.7: Supervised ML methods fit a curve to (a huge number of) data points.

In some ML applications there is no reason for defining labels associated with data points. The resulting ML methods are then called **unsupervised ML methods**. In Chapter 8, we will study an important family of unsupervised ML methods which amount to **clustering** data points into coherent groups (clusters).

Chapter 9 discusses another important unsupervised ML problem referred to as **dimensionality reduction**. Dimensionality reduction methods exploit intrinsic correlations and redundancy in the raw data to construct few relevant features for data points. These relevant features can be subsets of raw features (feature selection) or combinations of all raw features (feature transformation).

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

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

Notation. We mainly follow the notational conventions used in [?]. Boldface upper case letters (such as $\mathbf{A}, \mathbf{X}, \dots$) denote matrices while boldface lower case letters (such as $\mathbf{y}, \mathbf{x}, \dots$) 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 $\mathbf{x} = (x_1, \dots, x_n)^T$ is denoted $\|\mathbf{x}\| = \sqrt{\sum_{r=1}^n x_r^2}$.

Chapter 2

Three Components of ML: Data, Model and Loss

This book portrays ML as combinations of three components:

- **data** as collections of data points 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, how to model the relation between features and label of a data point and in what loss function to measure the quality of a ML method. Once the ML problem is formally defined, we can apply ML methods to solve them.

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

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

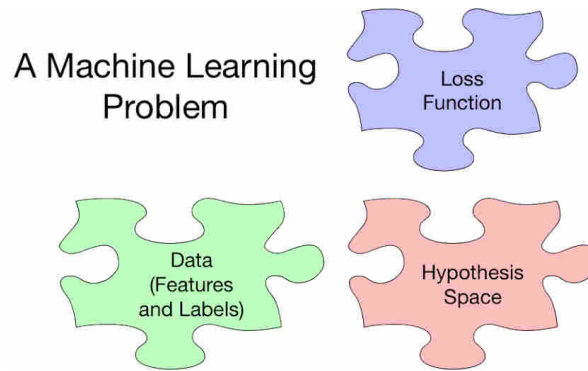


Figure 2.1: A formal ML problem consists of (i) a particular choice of features and label to characterize data points, (ii) a hypothesis space which consists of (computationally) feasible predictor maps from features to labels and (iii) a loss function which is used to assess the quality of a particular predictor.

2.1 The Data

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

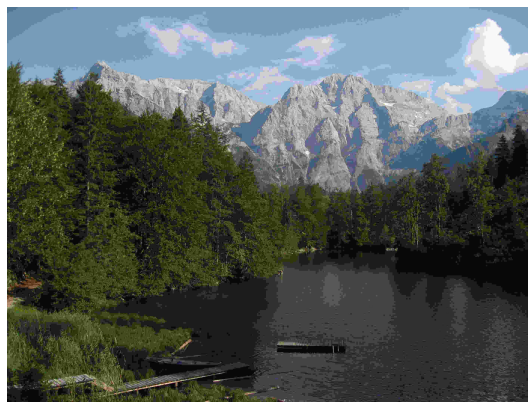


Figure 2.2: Photo taken at the beginning of a mountain hike.

We use the concept of datapoints in a highly abstract and therefore very flexible manner. Datapoints can represent very different types of objects. For an image processing application it might be useful to define datapoints as images. Within a recommendation system it might make sense to use datapoints representing human users or consumers. The meaning or definition of datapoints is nothing but a design choice.

The only conceptual requirement we have for this design choice is that datapoints represent objects of a similar “type” (e.g., every datapoint of a given dataset represents a mountain hike somewhere in Austria). Another, more practical, requirement for a good choice for data points is that we should have access to many of them (e.g., via hiker forums). Loosely speaking, most ML methods rely on statistical averages to obtain predictions and **the more datapoints used for the averaging, the better**. A key parameter of a dataset is therefore the number m of individual datapoints it contains. Statistically, the larger the sample size m the better. However, there might be restrictions on computational resources that limit the maximum sample size m that can be processed. ??? nice figure illustrating a dataset with m data points????

In most cases it is impossible to have full access to every single microscopic property of a data point. Consider a data point that represents a particular mountain hike. A full characterization of such a datapoint would require to know the precise weather condition at every point in time during the hike. Moreover, a mountain hike also involves the physical conditions of the hikers themselves which, in turn, depend on many other properties such as body height, weight and fitness level.

It will be useful to distinguish between two different types of properties of a datapoint. The first type of properties is referred to as “features” and the second type of property is referred to as “label”, or “target” or “output”. We emphasize that this distinction is somewhat blurry. The same property of a data point might be used as a feature in one application, while it might be used as a label in other applications.

2.1.1 Features

Similar to the definition of data points, also the choice of what properties to be used as features is a design choice. We typically use as features any quantity that can be computed or measured easily. Note that this is a highly informal characterization since we do not have an objective criterion to measure the difficulty in measuring a specific property.

If we develop a ML method that can use snapshots taken by a digital camera, then these snapshots might be a useful choice for the features. However, if we only have a thermometer

at our disposal then we might only use the measured temperature as the feature. In what follows we will denote the total number of features used to describe a data point by the letter n .

The ability of ML methods has been boosted by modern information-technology which allows to measure a huge number of properties about datapoints in many application domains. Consider a data point representing the book author “Alex Jung”. Alex uses a smartphone to take snapshots.

Let us assume that Alex takes five snapshots per day on average (sometimes more, e.g., during a mountain hike). This results in more than 1000 snapshots per year. Each snapshot contains around 10^6 pixels. If we only use the greyscale levels of the pixels in all those snapshots, we would obtain more than 10^9 new features per year! Modern ML applications face extremely high-dimensional feature vectors which calls for methods from high-dimensional statistics [? ?].

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

Using too many irrelevant features might overwhelm or jam ML algorithms which should invest their computational resources mainly in the processing of the most relevant features. It is difficult to give a precise characterization on the maximum number of features that should be used. Some guidance is offered by the checking the condition $n/m \gg 1$, i.e., the number of features is much larger than the number of data points available for an ML algorithm. In this high-dimensional regime, there is a high risk of overwhelming ML algorithms by having too many irrelevant features. To avoid this we could apply some feature selection or model regularization techniques (see Chapter 9 and Chapter 7).

Choosing “good” features of the datapoints arising within a given ML application is far from trivial and might be the most difficult task within the overall ML application. The family of ML methods known as **kernel methods** [?] is based on constructing efficient features by applying high-dimensional **feature maps**.

A recent breakthrough achieved by modern ML methods, which are known as **deep learning methods** (see Section 3.10), is their ability to automatically learn good features without requiring too much manual engineering (“tuning”) [?]. We will discuss the very basic ideas behind such **feature learning** methods in Chapter 9 but for now assume the task of selecting good features is already solved.

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

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

Each datapoint is then characterized by such a feature vector \mathbf{x} . The set of all possible values that the feature vector can take on is sometimes referred to as **feature space**, which we denote as \mathcal{X} . Note that we allow the feature space to be finite. This can be useful for **network-structured datasets** where the data points can be compared with each other by some application-specific notion of similarity [? ? ? ?]. These approaches use as a feature space the node set of an “empirical graph” whose nodes represent individual datapoints. The edges in the empirical graph encode similarities between individual datapoints.

The feature space \mathcal{X} is a design choice for the ML engineer facing a particular ML application and computational infrastructure. If the computational infrastructure allows for efficient numerical linear algebra, then using $\mathcal{X} = \mathbb{R}^n$ might be a good choice. In general, to obtain computationally efficient ML methods one typically uses feature spaces \mathcal{X} with a rich mathematical structure.

The Euclidean space \mathbb{R}^n is a prime example of a feature space with a rich geometric and algebraic structure [?]. The algebraic structure of \mathbb{R}^n is defined by linear algebra of vector addition and multiplication with scalars. A geometric structure is obtained by defining distances between two elements of \mathbb{R}^n via the Euclidean norm. The interplay between these two structures allows us then to efficiently search over subsets of \mathbb{R}^n to find an element that is closest to some other given element of \mathbb{R}^n .

Throughout this book we will mainly use feature spaces $\mathcal{X} \subseteq \mathbb{R}^n$ which are subsets of the Euclidean space \mathbb{R}^n with some fixed dimension n . Using RGB intensities (modelled as real numbers) of the pixels within a (rather small) 512×512 pixel bitmap, we end up with a feature space $\mathcal{X} = \mathbb{R}^n$ of (rather large) dimension $n = 3 \cdot 512^2$. Indeed, for each of the 512×512 pixels we obtain 3 numbers which encode the red, green and blue colour intensity of the respective pixel (see Figure 2.3).

Consider data points representing images. A natural construction for the feature vector of such data points is to stack the red, green and blue intensities for all image pixels (see Figure 2.3). For other types of data points it is less obvious how to represent the datapoints by a numeric feature vector in \mathbb{R}^n . Feature learning methods are ML methods that aim at automatically determining useful feature vectors. For natural language processing, some successful feature learning methods have been proposed recently [?].

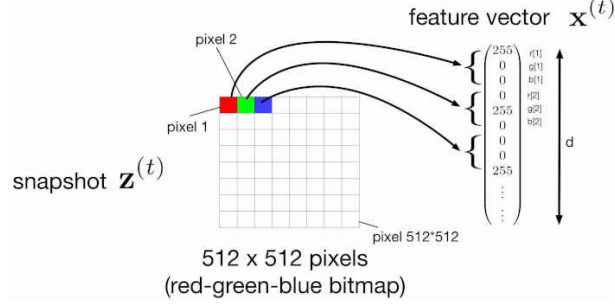


Figure 2.3: If the snapshot $\mathbf{z}^{(i)}$ is stored as a 512×512 RGB bitmap, we could use as features $\mathbf{x}^{(i)} \in \mathbb{R}^n$ the red-, green- and blue component of each pixel in the snapshot. The length of the feature vector would then be $n = 3 \cdot 512 \cdot 512 \approx 786000$.

2.1.2 Labels

Besides the features of a data point, there are other properties of a data point that represent some higher-level information or “quantity of interest” associated with the data point. We refer to the higher level information, or quantity of interest, associated with a data point 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 data point given its features.

As already mentioned above, the distinction of data point properties into labels and those that are features is blurry. Roughly speaking, labels are properties of data points that might only be determined with the help of human experts. For data points representing humans, the label y indicating if the person has flu ($y = 1$) or not ($y = 0$) 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 data point that represents some hike, at the start of which the snapshot in Figure 2.2 has been taken. The features of this data point could be the red, green and blue intensities of each pixel in the snapshot in Figure 2.2. We can stack these values into a vector $\mathbf{x} \in \mathbb{R}^n$ whose length n is given by three times the number of pixels in the image. The label y associated with this data point 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} contains all possible label values of data points arising in some ML problem. For the choice $\mathcal{Y} = \mathbb{R}$, it is customary to refer to such a problem as a **regression**

problem. It is 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 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.

A data point is called *labeled* if, besides its features \mathbf{x} , the associated 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 [?], running a particle physics experiment at the European organization for nuclear research (CERN) [?], running animal testing in pharmacology [?]. There are also online marketplaces to rent human labelling workforce [?]. In these labelling services, you can upload your data points, such as images, and then some human labels them, such as marking those who show a cat.

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

In the extreme case, we do not know the label of any single data point. Even in the absence of any labeled data, ML methods can be useful for extracting relevant information out of the features only. We refer to ML methods which do not require any labeled data points 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 data point 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 . As an example consider datapoints representing individual days whose features and labels are the minimum and maximum daytime temperature, respectively, at some place in Finland.

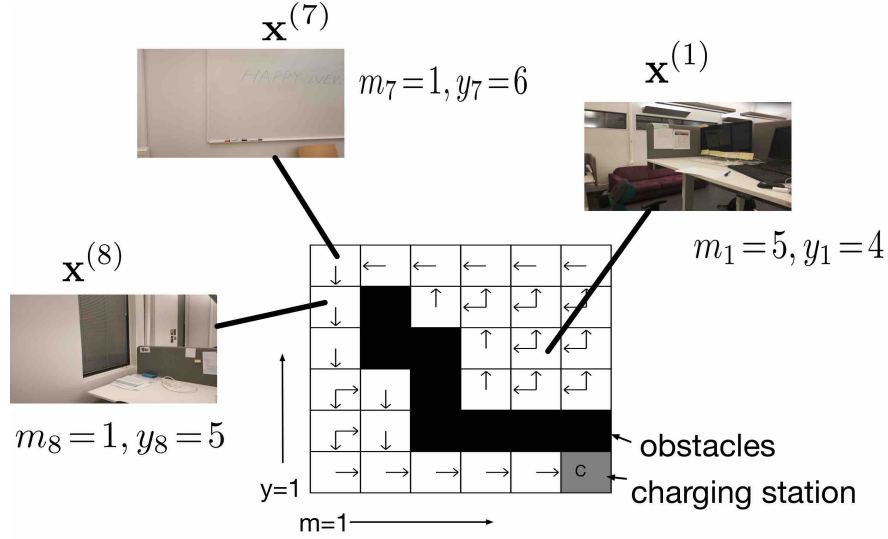


Figure 2.4: The cleaning robot Rumba is collecting snapshots $\mathbf{z}^{(i)}$, each of which is represented by the feature vector $\mathbf{x}^{(i)}$ and labeled with the y -coordinate $y^{(i)}$ of Rumba's location at time i .

We can access these datapoints via the archives of the Finnish Meteorological Institute (FMI). To get more insight into the relation between feature $x^{(i)}$ and label $y^{(i)}$, it can be helpful to generate a scatter plot as shown in Figure 1.2. A scatter plot depicts the data points $\mathbf{z}^{(i)} = (x^{(i)}, y^{(i)})$ 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.

2.1.4 Probabilistic Models for Data

Consider data points representing previous days in Finland. Each data point is characterized by its (average) daytime temperature $x \in \mathbb{R}$. Assume we have downloaded m data points $(x^{(1)}, \dots, x^{(m)})$.

Many successful ML methods are based on a simple but crucial idea: Interpret data points as realizations of random variables. The most basic and widely used form of a probabilistic model for the data points in ML is the **“independent and identically distributed”**

(i.i.d.) assumption. Here, the data points are interpreted as statistically independent realizations of one single random variable x .

Interpreting data points as realizations of a random variable x , allows to use the properties of the probability distribution $p(x)$ to characterize the statistical properties of the data. The probability distribution $p(x)$ is either assumed known or estimated from data (see Section 3.11). It is often enough to not estimate the distribution $p(x)$ entirely but only some of its parameters.

Some of the most basic and widely used parameters of a probability distribution $p(x)$ is the expected value or mean $\mu_x = \mathbb{E}\{x\} := \int_x xp(x)dx$ and variance $\sigma_x^2 := \mathbb{E}\{(x - \mathbb{E}\{x\})^2\}$. These parameters can be estimated using the sample mean (average) and sample variance,

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

A widely used estimator for the square root of the variance is the (sample) standard deviation $\hat{s}_x := \sqrt{(1/(m-1)) \sum_{i=1}^m (x^{(i)} - \hat{\mu}_x)^2}$.

2.2 The Model

Consider the problem of classifying hand-drawings into two categories. The first category consists of all hand-drawings that show an apple. The second category consists of all hand-drawings that do not show an apple. We consider each hand-drawing as one particular data point. Each data point is associated with a numeric label $y \in \mathbb{R}$ which is a percentage measuring how much the underlying hand-drawing resembles an apple. Beside its label y , each data point (hand-drawing) is characterized by a feature vector $\mathbf{x} = (x_1, \dots, x_n)^T$ obtained by stacking the greyscale intensities of a 128×128 pixel image of the hand-drawing.

Given a set of m labeled data points (the “training data”)

$$\mathcal{X} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})\} \quad (2.2)$$

ML methods learn a predictor $h(\mathbf{x})$ such that

$$y \approx \underbrace{h(\mathbf{x})}_{\hat{y}} \quad (2.3)$$

for any data point with features \mathbf{x} . The approximation (2.3) should be accurate for any data

point, also those not contained in the training set (2.2).

To have any chance to solve this task, we must assume (or hypothesize) an underlying relation between the features $\mathbf{x}^{(i)}$ and the label $y^{(i)}$. We represent this relation between features $\mathbf{x}^{(i)}$ and label $y^{(i)}$ using a **hypothesis map** $h : \mathcal{X} \rightarrow \mathcal{Y}$, which maps the feature vector $\mathbf{x}^{(i)}$ of the snapshot to a predicted label $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$.

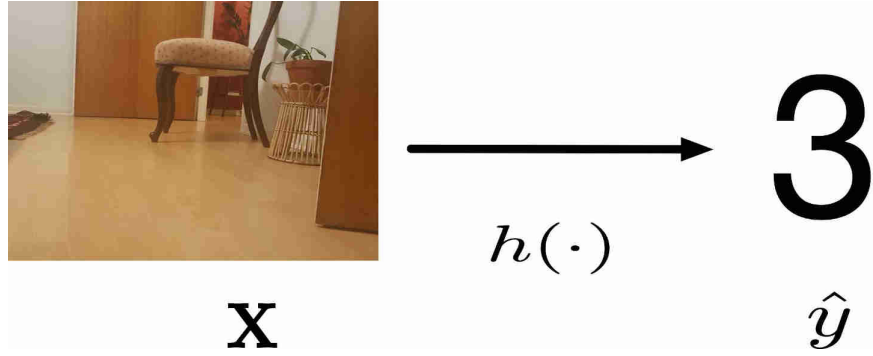


Figure 2.5: A predictor (hypothesis) h maps features $\mathbf{x} \in \mathcal{X}$, of an on-board camera snapshot, to the prediction $\hat{y} = h(\mathbf{x}) \in \mathcal{Y}$ for the coordinate of the current location of a cleaning robot. ML methods use data to learn predictors h such that $\hat{y} \approx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

In principle, we could use any possible map $h : \mathcal{X} \rightarrow \mathcal{Y}$ for predicting the label $y \in \mathcal{Y}$ based on the features $\mathbf{x} \in \mathcal{X}$ via computing $\hat{y} = h(\mathbf{x})$. However, any practical ML methods 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”) predictors is referred to as the **hypothesis space** or **concept class**.

The choice of which hypothesis space to use depends crucially on the available computational infrastructure. Different types of computational infrastructure call for different choices for the hypothesis space. For ML methods implemented in a small embedded system, we might prefer a linear hypothesis space which can be processed using numerical linear algebra. Deep learning methods implemented using cloud computing can make use of much larger hypothesis spaces obtained from deep neural networks.

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

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

The hypothesis space $\mathcal{H}^{(n)}$ in (2.4) is constituted by the linear maps (functions) $h^{(\mathbf{w})} : \mathbb{R}^n \rightarrow \mathbb{R}$ which map the feature vector $\mathbf{x} \in \mathbb{R}^n$ to the predicted label (or output) $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{x}^T \mathbf{w} \in \mathbb{R}$.

\mathbb{R} . For $n=1$ the feature vector reduces to one single feature x and the hypothesis space (2.4) consists of all maps $h^{(w)}(x) = wx$ with some weight $w \in \mathbb{R}$ (see Figure 2.7).

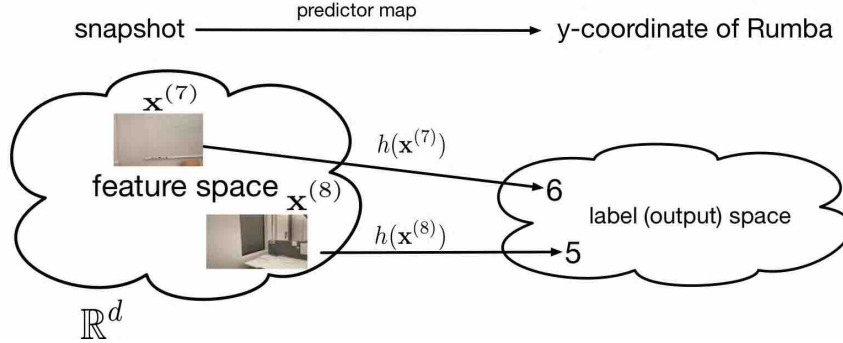


Figure 2.6: A predictor (hypothesis) $h : \mathcal{X} \rightarrow \mathcal{Y}$ takes the feature vector $\mathbf{x}^{(t)} \in \mathcal{X}$ (e.g., representing the snapshot taken by Rumba at time t) as input and outputs a predicted label $\hat{y}_t = h(\mathbf{x}^{(t)})$ (e.g., the predicted y -coordinate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

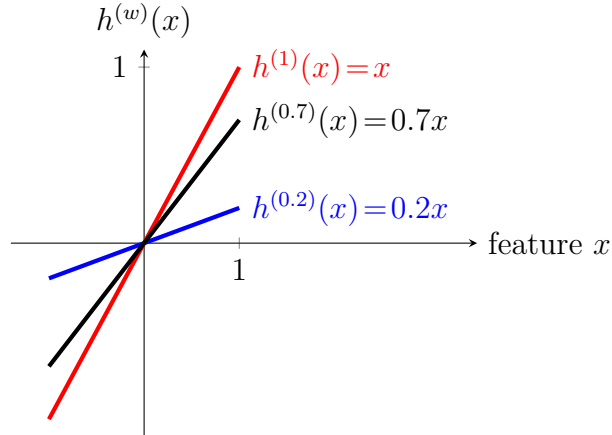


Figure 2.7: Three particular members of the hypothesis space $\mathcal{H} = \{h^{(w)} : \mathbb{R} \rightarrow \mathbb{R}, h^{(w)}(x) = w \cdot x\}$ which consists of all linear functions of the scalar feature x . We can parametrize this hypothesis space conveniently using the weight $w \in \mathbb{R}$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

The elements of the hypothesis space \mathcal{H} in (2.4) are parametrized 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} . Thus, instead of searching a good hypothesis directly in the function space \mathcal{H} (its elements are functions!), we can equivalently search over all possible weight vectors $\mathbf{w} \in \mathbb{R}^n$. The search space \mathbb{R}^n is still (unaccountably) infinite but it has lots of geometric and algebraic structure which allows to efficiently search over this space.

ML methods based on the linear hypothesis space (2.4) can be implemented by combinations of few basic operations from linear algebra such as vector-matrix multiplications. One reason for the popularity of methods using (2.4) is the broad availability of computing hardware (such as graphic processing units) and programming frameworks (such as numerical linear algebra libraries) allow efficient computations of the elementary operations.

The hypothesis space (2.4) can only be used for ML problems within which the features of data points are represented by vectors $\mathbf{x} = (x_1, \dots, x_n)^T \in \mathbb{R}^n$. Since ML methods based on the hypothesis space (2.4) are quite well developed (using numerical linear algebra), it might be useful to re-formulate the ML problem at hand using vectors as features. This suggests the “country wisdom” that within ML any data has to be turned into vectors (or matrices). For text data, there has been significant progress recently on methods that map a human-generated text into sequences of vectors (see [?, Chap. 12] for more details).

The hypothesis space, i.e., the subset of predictor maps $h : \mathcal{X} \rightarrow \mathcal{Y}$ used in a ML method, is a **design choice**. Some particular choices have proven useful for a wide range of applications (see Chapter 3). As with the choice of what properties to use as features of a data point, finding a good choice of the hypothesis space typically requires a good understanding (“domain expertise”) of statistical properties of the data and the limitations of the available computational infrastructure.

One important aspect guiding the choice for the hypothesis space is the available computational infrastructure. If we only have a spreadsheet program at our disposal then it might be reasonable to use a hypothesis space which is constituted by maps h which can be represented by look-up tables (see Figure 2.1). If our computational infrastructure allows for efficient numerical linear algebra, which is the case for most desktop computers and scientific programming languages), the linear hypothesis space (2.4) might be useful.

Remember that a hypothesis (or predictor) h is a map which takes the features $\mathbf{x} \in \mathcal{X}$ as its input and outputs a predicted (or estimated) label $\hat{y} = h(\mathbf{x}) \in \mathcal{Y}$. If the label space \mathcal{Y} is finite (such as $\mathcal{Y} = \{-1, 1\}$) we use the term **classifier** instead of hypothesis. For regression problems involving a continuous label space (such as $\mathcal{Y} = \mathbb{R}$) it is common to refer to a hypothesis as a **predictor**.

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 all border points between the different regions $\mathcal{R}_{\hat{y}} := \{\mathbf{x} : h(\mathbf{x}) = \hat{y}\} \subseteq \mathcal{X}$ of the feature space \mathcal{X} . A particular region $\mathcal{R}_{\hat{y}}$ contains all feature values $\mathbf{x} \in \mathcal{X}$ which are mapped to the same prescribed label value $\hat{y} \in \mathcal{Y}$.

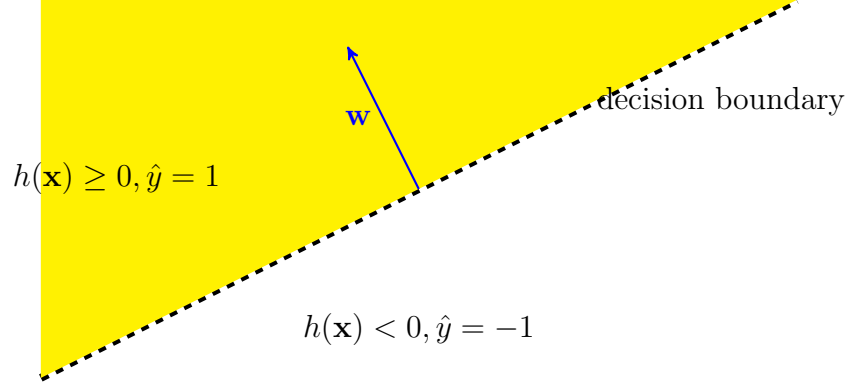


Figure 2.8: A hypothesis $h : \mathcal{X} \rightarrow \mathcal{Y}$ for a binary classification problem, with label space $\mathcal{Y} = \{-1, 1\}$ and feature space $\mathcal{X} = \mathbb{R}^2$, can be represented conveniently via the decision boundary (dashed line) which separates all feature vectors \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**.

ML methods known as **linear classifiers**, which includes logistic regression (see Section 3.5), the SVM (see Section 3.6) and naive Bayes' classifiers (see Section 3.7), use classifier maps whose decision boundary is a hyperplane $\{\mathbf{x} : \mathbf{w}^T \mathbf{x} = b\}$ (see Figure 2.8).

One important aspect guiding the choice for the hypothesis space is the available computational framework. If we can only use a **spreadsheet program**, we should use a hypothesis space constituted by maps $h : \mathcal{X} \rightarrow \mathcal{Y}$ 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.

The largest possible choice for the hypothesis space \mathcal{H} of an ML problem using feature space \mathcal{X} and label space \mathcal{Y} is the set $\mathcal{Y}^{\mathcal{X}}$ constituted by all maps from the feature space \mathcal{X} to the label space \mathcal{Y} .¹ The choice $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ is rarely used in practice since this space is simply too large to work within a reasonable amount of computational resources. Instead, the hypothesis space \mathcal{H} is typically a (very small) subset of $\mathcal{Y}^{\mathcal{X}}$ (see Figure 2.9). However, choosing the hypothesis space too small might result in poor performance of the overall ML method, as we might not find any suitable predictor (or classifier) map $h \in \mathcal{H}$ which allows to capture the true underlying relationship between features \mathbf{w} and label y of data points.

¹The elements of $\mathcal{Y}^{\mathcal{X}}$ are maps $h : \mathcal{X} \rightarrow \mathcal{Y}$.

The design choice for 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 ML problem of classifying an images into “cat image” and “no cat image” using the pixel colour intensities as features. 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 [?]. 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 well the training dataset. Such a predictor will 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. For a finite hypothesis spaces \mathcal{H} , we might simply use the cardinality $|\mathcal{H}|$ as a measure of its size. Consider data points represented by $10 \times 10 = 100$ black and white pixels (see Figure 2.3) and characterized by a binary label. This amounts to a feature space $\mathcal{X} = \{0, 1\}^{1000}$ and label space $\mathcal{Y} = \{0, 1\}$. The largest possible hypothesis space consisting of all maps is finite with size $|\mathcal{H}| = 2^{1000}$, which is larger than the estimated number of atoms in the visible universe (which is around 10^{80}).

Many ML methods use a hypothesis space which contains infinitely many different predictors (see, e.g., (2.4)). For infinite hypothesis spaces, we cannot easily use its cardinality as a measure for its size. Different concepts have been studied for measuring the size of infinite hypothesis spaces with the **Vapnik–Chervonenkis (VC) dimension** being maybe the most famous one [?].

We will use a simpler variant of the VC dimension concept and define the size of a

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

Table 2.1: A spreadsheet representing of 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 .

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

Let illustrate the our notion of hypothesis space size with two examples: linear regression and polynomial regression. First, we discuss the size of the hypothesis space $\mathcal{H}^{(n)}$ used in linear regression. Consider m data points characterized by feature vectors $\mathbf{x} \in \mathbb{R}^{(n)}$ and a numeric label $y \in \mathbb{R}$. We assume that data points are i.i.d. realizations of some continuous random vector (probability of the vector belonging to a zero measure set is zero). This implies that the matrix obtained by columnwise stacking the feature vectors is full rank with probability one. Basic linear algebra allows then to show that such a set of data points can be perfectly fit by a linear map $h \in \mathcal{H}^{(n)}$ as long as $m \leq n$. Thus, the size of the linear hypothesis space $\mathcal{H}^{(n)}$ is $D = n$.

As a second example, consider data points characterized by a single numeric feature x and numeric label y . Polynomial regression aims at learning a predictor out of the hypothesis space $\mathcal{H}_{\text{poly}}^{(n)}$ of polynomials with maximum degree n . The fundamental theorem of algebra tells us that any set of m data points with different features can be perfectly fit by a polynomial of degree n as long as $n \geq m$. Therefore, the size of the hypothesis space $\mathcal{H}_{\text{poly}}^{(n)}$ is $D = n$. We will discuss polynomial regression in more detail in Section 3.2.

2.3 The Loss

Every practical ML method uses some hypothesis space \mathcal{H} which consists of all **computationally feasible** predictor maps h . Which predictor map h out of all the maps in the hypothesis space \mathcal{H} is the best for the ML problem at hand? To this end, we need to define a measure

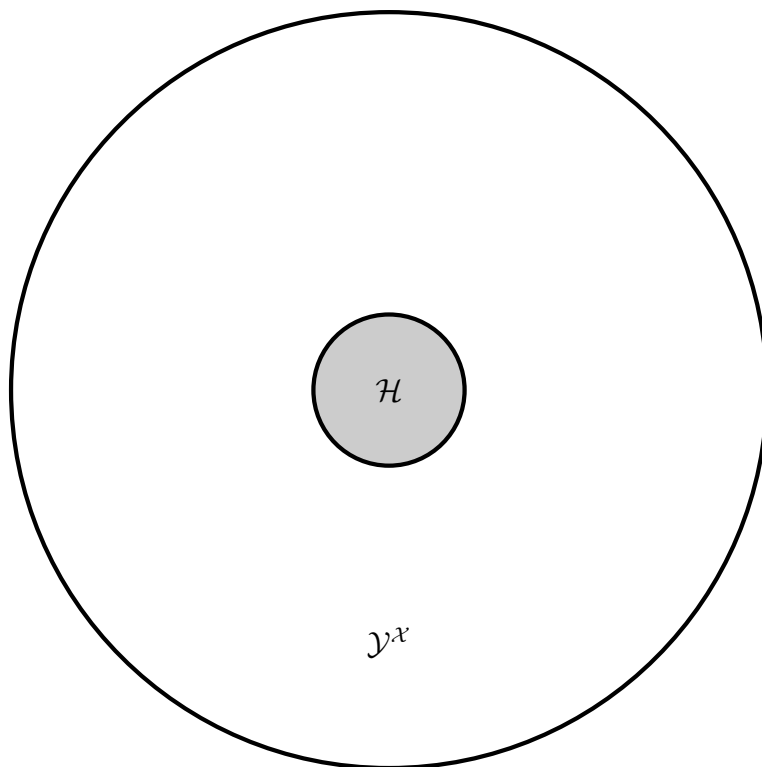


Figure 2.9: The hypothesis space \mathcal{H} is a (typically very small) subset of the (typically very large) set $\mathcal{Y}^{\mathcal{X}}$ of all possible maps from feature space \mathcal{X} into the label space \mathcal{Y} .

of the **loss** (or error) incurred by using the particular predictor $h(\mathbf{x})$ when the true label is y . More formally, we define a loss function $\mathcal{L} : \mathcal{X} \times \mathcal{Y} \times \mathcal{H} \rightarrow \mathbb{R}$ which measures the loss $\mathcal{L}((\mathbf{x}, y), h)$ incurred by predicting the label y of a data point using the prediction $h(\mathbf{x})(=: \hat{y})$. The concept of loss functions is best understood by considering some particular examples.

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

$$\mathcal{L}((\mathbf{x}, y), h) := (y - \underbrace{h(\mathbf{x})}_{=\hat{y}})^2. \quad (2.5)$$

Note that the squared error loss (2.5) 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 . Beside the prediction $h(\mathbf{x})$, no other properties of the data point's features \mathbf{x} are required to determine the squared error loss.

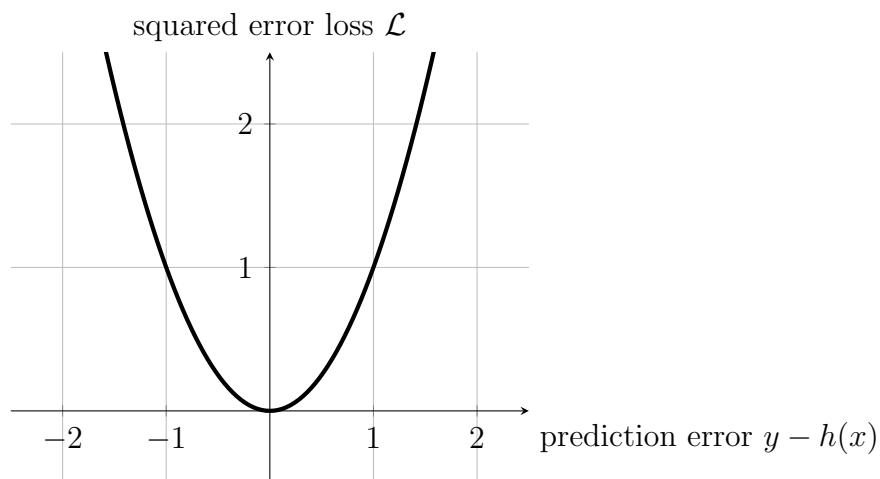


Figure 2.10: A widely used choice for the loss function in regression problems (with label space $\mathcal{Y} = \mathbb{R}$) is the squared error loss $\mathcal{L}((\mathbf{x}, y), h) := (y - h(\mathbf{x}))^2$. Note that in order to evaluate the loss function for a given hypothesis h , so that we can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

The squared error loss (2.5) 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. In particular, minimizing the squared error loss is equivalent to

maximum likelihood estimation within a linear Gaussian model [?, Sec. 2.6.3].

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

Classification Loss. In classification problems with a discrete label space \mathcal{Y} , such as in binary classification where $\mathcal{Y} = \{-1, 1\}$, the squared error $(y - h(\mathbf{x}))^2$ is not a useful measure for the quality of a classifier $h(\mathbf{x})$. We would like to punish wrong classifications, e.g., when the true label is $y = -1$ but the classifier produces a large positive number, e.g., $h(\mathbf{x}) = 1000$. On the other hand, for a true label $y = -1$, we do not want to punish a classifier h which yields a large negative number, e.g., $h(\mathbf{x}) = -1000$. But exactly this unwanted result would happen for the squared error loss.

Figure 2.11 depicts a dataset consisting of 5 labeled data points with binary labels represented by circles (for $y = 1$) and squares (for $y = -1$). The squared error loss incurred by the classifier h_1 , which does not separate the two classes perfectly, is smaller than the squared error loss incurred by classifier h_2 which perfectly separates the two classes. The squared error loss is a bad choice for classification problems with a discrete label space \mathcal{Y} .

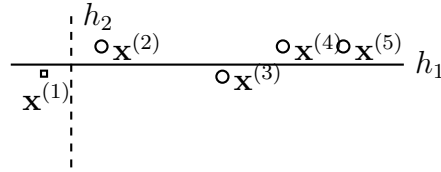


Figure 2.11: Minimizing the squared error loss would prefer the (poor) classifier h_1 over the (reasonable) classifier h_2 .

We now present some popular choices for the loss function suitable for ML problems with binary labels. While the representation of the label values is completely irrelevant, it will be convenient to encode the two label values by the real numbers -1 and 1 . As label space we use $\mathcal{Y} = \mathbb{R}$. The formulas for the loss functions we present only apply to this encoding. The modification of these formulas to a different encoding, such as label values 0 and 1 , is not very difficult.

Consider the problem of detecting forest fires as early as possible using webcam snapshots such as the one depicted in Figure 2.12. A particular snapshot is characterized by the features \mathbf{x} and the label $y \in \mathcal{Y} = \{-1, 1\}$ with $y = 1$ if the snapshot shows a forest fire and $y = -1$ if there is no forest fire. We would like to find or learn a classifier $h(\mathbf{x})$ which takes the



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

features \mathbf{x} as input and provides a classification according to $\hat{y} = 1$ if $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ if $h(\mathbf{x}) \leq 0$. Ideally we would like to have $\hat{y} = y$, which suggests to use the 0/1 **loss** (see Figure 2.13)

$$\mathcal{L}((\mathbf{x}, y), h) := \begin{cases} 1 & \text{if } yh(\mathbf{x}) < 0 \\ 0 & \text{else.} \end{cases} \quad (2.6)$$

The 0/1 loss is statistically appealing as it approximates the misclassification (error) probability $P(y \neq \hat{y})$ with $\hat{y} = \text{sign}\{h(\mathbf{x})\}$. If we interpret the data points $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ as i.i.d. realizations of a random feature vector $\mathbf{x} \in \mathcal{X}$ and random label $y \in \{-1, 1\}$,

$$(1/m) \sum_{i=1}^m \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) \approx P(y \neq \hat{y}) \quad (2.7)$$

with high probability for sufficiently large sample size m . The approximation (2.7) is based on the fact that the average of a large number of independent realizations of some random variable can be well approximated by its mean or expectation (“law of large numbers”) (see [?]). ?? Indeed, the values $\mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h)$ are i.i.d. realizations of the random variable $\mathcal{L}((\mathbf{x}, y), h)$.??

In view of (2.7), the 0/1 loss seems the most 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 data point (\mathbf{x}, y) , the 0/1 loss (2.6) 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 3.7).

In order to “cure” the non-convexity of the 0/1 loss we approximate it by a convex loss function. This convex approximation results in the **hinge loss** (see Figure 2.13)

$$\mathcal{L}((\mathbf{x}, y), h) := \max\{0, 1 - y \cdot h(\mathbf{x})\}. \quad (2.8)$$

While the hinge loss avoids the non-convexity of the 0/1 loss it still is a non-differentiable function of the classifier h . The next example of a loss function for classification problems cures also the non-differentiability issue.

As a third example for a loss function which is useful for binary classification problems we mention the **logistic loss** (used within logistic regression, see Section 3.5)

$$\mathcal{L}((\mathbf{x}, y), h) := \log(1 + \exp(-yh(\mathbf{x}))). \quad (2.9)$$

For a fixed feature vector \mathbf{x} and label y , both, the hinge and the logistic loss function are **convex functions** of the hypothesis h . However, while the logistic loss (2.9) depends **smoothly** on h (such that we could define a derivative of the loss with respect to h), the hinge loss (2.8) is **non-smooth** which makes it more difficult to minimize.

Thus, while ML methods based on the logistic loss function (such as logistic regression in Section 3.5), can make use of simple **gradient descent methods** (see Chapter 5), ML methods based on the hinge loss (such as support vector machines [?]) have to make use of more advanced tools for solving the resulting learning problem (see Chapter 4).

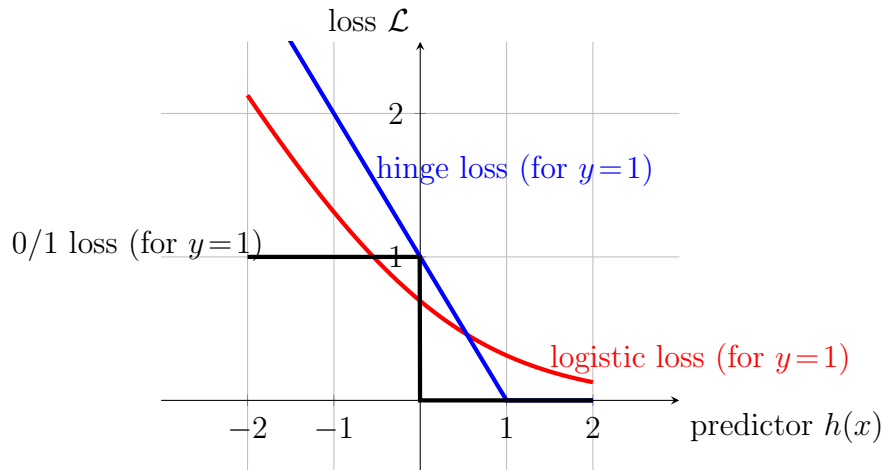


Figure 2.13: Some popular loss functions for binary classification problems with label space $\mathcal{Y} = \{-1, 1\}$. Note that the more correct a decision, i.e., the more positive $h(x)$ is (when $y = 1$), the smaller is the loss. In particular, all depicted loss functions tend to 0 monotonically with increasing $h(x)$.

Let us emphasize that, very much like the choice of features and hypothesis space, the question of which particular loss function to use within an ML method is a **design choice**, which has to be tailored to the application at hand. What qualifies as a useful loss function depends heavily on the overall goal we wish to achieve using ML methods. Thus, there is no useful concept of “optimal loss function”. In general, different loss functions are preferable for different ML applications.

An important aspect guiding the choice of loss function is the computational complexity of the resulting ML method. Indeed, the basic idea behind ML methods is quite simple: learn (find) the particular hypothesis out of a given hypothesis space which yields the smallest loss (on average). The difficulty of the resulting optimization problem (see Chapter 4) depends crucially on the properties of the chosen loss function. Some loss functions allow to use very simple but efficient iterative methods for solving the optimization problem underlying an ML method (see Chapter 5).

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

We define the empirical risk of some predictor when applied to labeled data points $\mathbb{X} = (\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})$ as

$$\mathcal{E}(h|\mathbb{X}) = (1/m) \sum_{i=1}^m \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h). \quad (2.10)$$

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

Regret. In some application we might have not access to true labels but only to the predictions obtained from some other methods (“experts”). The quality of a hypothesis can then be measured by evaluating the loss between predictions $h(\mathbf{x})$ and the predictions of experts [?]. The concept of regret minimization is useful when we do not make any assumptions (such as i.i.d. samples) of the data points. If we do not use such a probabilistic model we cannot use the Bayes risk (of Bayes optimal estimator) as benchmark. Regret minimization techniques can be designed and analyzed without any such probabilistic model for the data [?]. This approach replaces the Bayes risk benchmark with few given reference predictors (experts) to serve as benchmark.

Partial Feedback, “Reward”. In some application it is not possible to determine the true label of any data point. Without any labeled data we cannot use the concept of a loss function to measure the quality of a prediction (by comparing with the true label). Instead we must use some other form of indirect feedback or “reward” that can be attributed to a particular prediction [? ?].

Consider the ML problem of predicting the optimal direction for moving next a toy car given the current state. The state is sensed via a feature vector \mathbf{x} whose entries are pixel intensities of an on-board camera. A hypothesis maps the feature vector \mathbf{x} to a guess $\hat{y} = h(\mathbf{x})$ for the optimal steering direction. i.e., on-board camera snapshots annotated (labeled) with the optimal steering direction y (true label). Instead, we might have only some indirect signal about the loss incurred by the prediction $\hat{y} = h(\mathbf{x})$. Such a feedback signal, or reward, could be obtained by a distance sensor who measures the change of the distance between the car and its goal such as the charging station.

2.4 Putting Together the Pieces

To illustrate how ML methods combine particular design choices for data, model and loss, we consider data points 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 data points

$$(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)}) \quad (2.11)$$

for which we know the true label values $y^{(i)}$. Note that the assumption of knowing the exact true label values $y^{(i)}$ for any data point is an idealization. We might often face labelling or measurement errors such that the observed labels are noisy versions of the true label. However, there are efficient methods that can cope with noisy labels (see Chapter 7).

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

$$h^{(w_0, w_1)}(x) = w_1 x + w_0. \quad (2.12)$$

The predictor (2.12) is parametrized 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 data points.

In general, the predictions $\hat{y}^{(i)} = h^{(w_0, w_1)}(x^{(i)})$ 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 (2.5))

$$\begin{aligned} f(w_0, w_1) &:= (1/m) \sum_{i=1}^m (y^{(i)} - h^{(w_0, w_1)}(x^{(i)}))^2 \\ &\stackrel{(2.12)}{=} (1/m) \sum_{i=1}^m (y^{(i)} - (w_1 x^{(i)} + w_0))^2. \end{aligned} \quad (2.13)$$

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 data points (2.11).

It seems natural to learn a good predictor (2.12) by choosing the weights w_0, w_1 to minimize the training error

$$\min_{w_1, w_0 \in \mathbb{R}} f(w_0, w_1) \stackrel{(2.13)}{=} (1/m) \sum_{i=1}^m (y^{(i)} - (w_1 x^{(i)} + w_0))^2. \quad (2.14)$$

The optimal weights w'_0, w'_1 are characterized by the “zero gradient” condition [],

$$\frac{\partial f(w'_0, w'_1)}{\partial w_0} = 0, \text{ and } \frac{\partial f(w'_0, w'_1)}{\partial w_1} = 0. \quad (2.15)$$

Inserting (2.13) into (2.15), using basic rules for calculating derivatives, we obtain the following optimality conditions

$$(1/m) \sum_{i=1}^m (y^{(i)} - (w'_1 x^{(i)} + w'_0)) = 0, \text{ and } (1/m) \sum_{i=1}^m x^{(i)} (y^{(i)} - (w'_1 x^{(i)} + w'_0)) = 0. \quad (2.16)$$

Any weights w'_0, w'_1 that satisfy (2.16) 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 (2.16) using matrices and vectors. To this end, we first rewrite the predictor (2.12) as

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

Let us stack the feature vectors $\mathbf{x}^{(i)} = (1, x^{(i)})^T$ and labels $y^{(i)}$ of training data points (2.11) into the feature matrix and label vector,

$$\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)})^T \in \mathbb{R}^{m \times 2}, \mathbf{y} = (y^{(1)}, \dots, y^{(m)})^T \in \mathbb{R}^m. \quad (2.17)$$

We can then reformulate (2.16) as

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\mathbf{w}') = \mathbf{0}. \quad (2.18)$$

The entries of any weight vector $\mathbf{w}' = (w'_0, w'_1)$ that satisfies (2.18) are solutions to (2.16).

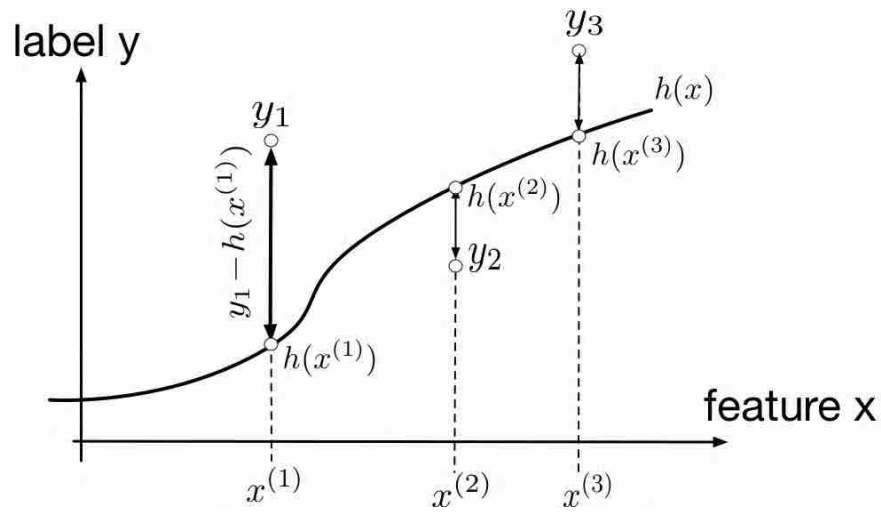


Figure 2.14: We can evaluate the quality of a particular predictor $h \in \mathcal{H}$ by measuring the prediction error $y - h(x)$ obtained for a labeled data point (x, y) .

2.5 Exercises

2.5.1 How Many Features?

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

2.5.2 Multilabel Prediction

Consider data points, 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 data point using a linear predictor map

$$\mathbf{h}(\mathbf{x}) = \mathbf{W}\mathbf{x} \text{ with some matrix } \mathbf{W} \in \mathbb{R}^{30 \times 10}. \quad (2.19)$$

How many different linear predictors (2.19) are there ? 10, 30, 40, infinite.

2.5.3 Average Squared Error Loss as Quadratic Form

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

2.5.4 Find Labeled Data for Given Empirical Risk

Consider linear hypothesis space consisting of linear maps parameterized by weights \mathbf{w} . We try to find the best linear map by minimizing the average squared error loss (empirical risk) incurred on some labeled training data points. Assume we know the shape of the empirical risk as a function of the weight. Can you reconstruct the labeled training data that resulted

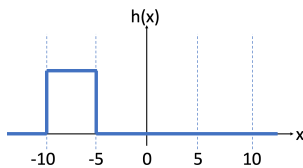
in that empirical risk function? Is the resulting labeled training data unique or are there different training sets that could have resulted in the same empirical risk function?

2.5.5 Dummy Feature Instead of Intercept

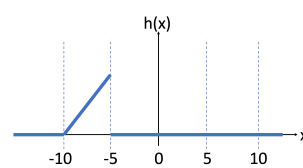
Show that any predictor of the form $h(x) = w_1x + w_0$ can be emulated by combining a feature map $x \mapsto \mathbf{z}$ with a predictor of the form $\mathbf{w}^T \mathbf{z}$.

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

Consider an ML application generating data points 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 $\phi_1(x), \dots, \phi_4(x)$ are obtained by indicator functions of intervals $[-10, -5), [-5, 0), [0, 5), [5, 10]$. In particular, $\phi_1(x) = 1$ for $x \in [-10, -5)$ and $\phi_1(x) = 0$ otherwise. We construct a hypothesis space \mathcal{H}_1 by all maps of the form $\mathbf{w}^T \mathbf{z}$. Note that the map is a function of the feature x since the feature vector \mathbf{z} is a function of x . Which of the following predictor maps belong to \mathcal{H}_1 ?



(a)



(b)

2.5.7 Python Hypothesis Space

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

2.5.8 A Lot of Features

In many application domains, we have access too a large number of features for each individual data point. Consider healthcare, where data points 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 data points, is it in general a good idea to use as much features as possible for data points ?

2.5.9 Over parametrization

Consider data points 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

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

Here, we used the matrix $\mathbf{A} = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$ and the set $\mathcal{S} = \{(1, 1)^T, (2, 2)^T, (-1, 3)^T, (0, 4)^T\} \subseteq \mathbb{R}^2$. What is the cardinality of \mathcal{H} , i.e., how many different predictor maps does \mathcal{H} contain?

2.5.10 Squared Error Loss

Consider a hypothesis space \mathcal{H} constituted by three predictors $h_1(\cdot), h_2(\cdot), h_3(\cdot)$. Each predictor $h_j(x)$ is a real-valued function of a real-valued argument x . Moreover, for each $j \in \{1, 2, 4\}$, $h_j(x) = 0$ 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) data points $(x = 1/10, y = 3)$, $(0, 0)$ and $(1, -1)$.

2.5.11 Classification Loss

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

2.5.12 Intercept Term

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

2.5.13 Picture Classification

Assume you want to sort a huge number of outdoor pictures you have taken during your last adventure trip to into three categories (or classes) *dog*, *bird* and *fish*. How could we formalize this sorting problem as a ML problem?

2.5.14 Maximum Hypothesis Space

Consider data points 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 data point and deliver a real-valued prediction $\hat{y} = h(x)$?

2.5.15 A Large but Finite Hypothesis Space

Consider data points whose features are 10×10 black and white pixel images. Each data point is also characterized by a binary label $y \in \{0, 1\}$. Consider the hypothesis space constitute by all maps that take the bw image as input and deliver a prediction for the label. How large is this hypothesis space?

2.5.16 Size of Linear Hypothesis Space

Consider a training set of m data points with feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and numeric labels $y^{(1)}, \dots, y^{(m)}$. The feature vectors and label values of the training set are arbitrary except that we assume the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \dots)$ 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., $y^{(1)} = h(\mathbf{x}^{(1)}), \dots, y^{(m)} = h(\mathbf{x}^{(m)})$.

Chapter 3

Some Examples

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

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

Each of these three components involves design choices such as the type of data representation. This chapter details the design choices used by some of the most popular ML methods.

3.1 (Least Squares) Linear Regression

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

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

The quality of a particular predictor $h^{(\mathbf{w})}$ is measured by the squared error loss (2.5). Based on labeled training data $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, linear regression aims at learning a predictor

\hat{h} which minimizes the average squared error loss (mean squared error) (see (2.5))

$$\begin{aligned}\hat{h} &= \operatorname{argmin}_{h \in \mathcal{H}^{(n)}} \mathcal{E}(h|\mathbb{X}) \\ &\stackrel{(2.10)}{=} \operatorname{argmin}_{h \in \mathcal{H}^{(n)}} (1/m) \sum_{i=1}^m (y^{(i)} - h(\mathbf{x}^{(i)}))^2.\end{aligned}\tag{3.2}$$

Since the hypothesis space $\mathcal{H}^{(n)}$ is parametrized 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} :

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

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

3.2 Polynomial Regression

Consider an ML problem involving data points 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 data points which are depicted in Figure 3.1.

Figure 3.1 suggests that the relation $x \mapsto y$ between feature x and label y is highly non-linear. For such non-linear relations between features and labels it is useful to consider a hypothesis space which is constituted by polynomial functions

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

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

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

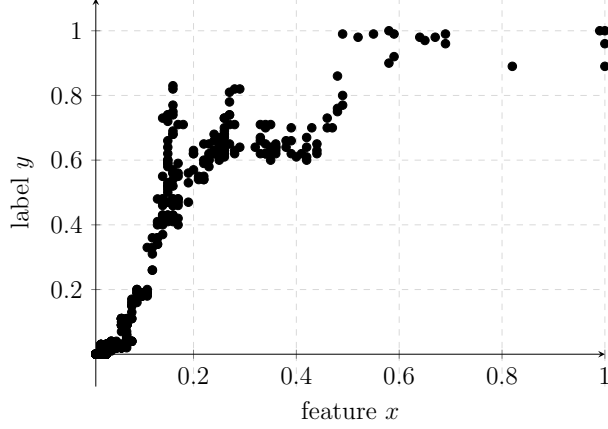


Figure 3.1: A scatterplot of some data points $(x^{(i)}, y^{(i)})$.

As for linear regression (see Section 3.1), we measure the quality of a predictor by the squared error loss (2.5). Based on labeled training data $\mathbb{X} = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$, with scalar features $x^{(i)}$ and labels $y^{(i)}$, polynomial regression amounts to minimizing the average squared error loss (mean squared error) (see (2.5)):

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

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

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

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

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

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

$$\mathbf{x}^{(i)} = \phi(x^{(i)}) = (1, x^{(i)}, \dots, (x^{(i)})^n)^T \in \mathbb{R}^{n+1}, \quad (3.8)$$

and then applying linear regression (see Section 3.1) to these new feature vectors. In

particular, 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 $h^{(\mathbf{w})} \in \mathcal{H}_{\text{poly}}^{(n)}$ 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 data point. However, it might be useful to tolerate a large prediction error $\hat{y} - y$ for few data points 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 known as "Huber loss" [?]]

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

Note that 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 data points with large 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 (the precise value depending on the value range of the features and labels), such that the condition $|y - \hat{y}| \leq \varepsilon$ is always satisfied. In this case, the Huber loss is equivalent to the squared error loss $(y - \hat{y})^2$ (with an additional factor 1/2).

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

3.4 Gaussian Basis Regression

As we have 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 $\phi : \mathbb{R} \rightarrow \mathbb{R}^n$ 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 . However, we can use also other functions (different from polynomials) to construct the feature map ϕ .

In principle, we can extend linear regression using an arbitrary feature map

$$\phi(x) = (\phi_1(x), \dots, \phi_n(x))^T \quad (3.10)$$

with the scalar maps $\phi_j : \mathbb{R} \rightarrow \mathbb{R}$ 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 data points. The basis functions underlying polynomial regression are $\phi_j(x) = x^j$. Another popular choice for the basis functions are “Gaussians”

$$\phi_{\sigma, \mu}(x) = \exp(-(1/(2\sigma^2))(x - \mu)^2) \quad (3.11)$$

which are parametrized by the variance σ^2 and the mean (shift) μ .

Thus, we obtain Gaussian basis linear regression by combining the feature map

$$\phi(x) = (\phi_{\sigma_1, \mu_1}(x), \dots, \phi_{\sigma_n, \mu_n}(x))^T \quad (3.12)$$

with linear regression (see Figure 3.2). The resulting hypothesis space is then

$$\begin{aligned} \mathcal{H}_{\text{Gauss}}^{(n)} &= \{h^{(\mathbf{w})} : \mathbb{R} \rightarrow \mathbb{R} : h^{(\mathbf{w})}(x) = \sum_{j=1}^n w_j \phi_{\sigma_j, \mu_j}(x) \\ &\text{with weights } \mathbf{w} = (w_1, \dots, w_n)^T \in \mathbb{R}^n\}. \end{aligned} \quad (3.13)$$

Note that we obtain a different hypothesis space $\mathcal{H}_{\text{Gauss}}$ for different choices of the variance σ^2 and shifts μ_j used for the Gaussian function in (3.11). These parameters have to be chosen suitably for the ML application at hand (e.g., using model selection techniques discussed in Section 6.3). The hypothesis space (3.13) is parameterized by the weight vector $\mathbf{w} \in \mathbb{R}^n$ since each element of $\mathcal{H}_{\text{Gauss}}$ corresponds to a particular choice for the weight vector \mathbf{w} .

Exercise. Try to approximate the hypothesis map depicted in Figure 3.11 by an element of $\mathcal{H}_{\text{Gauss}}$ (see (3.13)) using $\sigma = 1/10$, $n = 10$ and $\mu_j = -1 + (2j/10)$.



Figure 3.2: 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 $h^{(\mathbf{w})} \in \mathcal{H}_{\text{Gauss}}^{(2)}$.

3.5 Logistic Regression

Logistic regression is a method for classifying data points 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, using predictor maps $h \in \mathcal{H}^{(n)}$ might seem wasteful. Indeed, a linear map $h(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$, with some weight vector $\mathbf{w} \in \mathbb{R}^n$, can take on any real number, while the label $y \in \{-1, 1\}$ takes on only one of the two real numbers 1 and -1 . It turns out to be quite useful to use classifier maps h which can take on arbitrary real numbers.

We can always threshold the value $h(\mathbf{x})$ to obtain a predicted label $\hat{y} \in \{-1, 1\}$. In what follows we implicitly assume that the predicted label is obtained by thresholding the predictor map at 0, \hat{y} is 1 if $h(\mathbf{x}) \geq 0$ and $\hat{y} = -1$ otherwise. Thus, we use the sign of the predictor map $h(\mathbf{x})$ to determine the final prediction for the label. The absolute value $|h(\mathbf{x})|$ is then used to quantify the reliability of (or confidence in) the classification \hat{y} .

Consider two data points 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 both yields the same classifications $\hat{y}^{(1)} = \hat{y}^{(2)} = 1$, the classification of the data point with feature vector $\mathbf{x}^{(2)}$ seems to be much more reliable. In general it is beneficial to complement a particular prediction (or classification) result by some reliability information.

Within logistic regression, we assess the quality of a particular classifier $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}$ using the logistic loss (2.9) Given some labeled training data $\mathbb{X} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^m$, logistic regression

²It 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 $\mathcal{Y} = \{0, 1\}$.

amounts to minimizing the empirical risk (average logistic loss)

$$\begin{aligned}\mathcal{E}(\mathbf{w}|\mathbb{X}) &= (1/m) \sum_{i=1}^m \log(1 + \exp(-y^{(i)} h^{(\mathbf{w})}(\mathbf{x}^{(i)}))) \\ &\stackrel{h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}}{=} (1/m) \sum_{i=1}^m \log(1 + \exp(-y^{(i)} \mathbf{w}^T \mathbf{x}^{(i)})).\end{aligned}\quad (3.14)$$

Once we have found the optimal weight vector $\hat{\mathbf{w}}$ which minimizes (3.14), we can classify a data point based on its features \mathbf{x} according to

$$\hat{y} = \begin{cases} 1 & \text{if } h^{(\hat{\mathbf{w}})}(\mathbf{x}) \geq 0 \\ -1 & \text{otherwise.} \end{cases}\quad (3.15)$$

Since $h^{(\hat{\mathbf{w}})}(\mathbf{x}) = (\hat{\mathbf{w}})^T \mathbf{x}$ (see (3.1)), the classifier (3.15) amounts to testing whether $(\hat{\mathbf{w}})^T \mathbf{x} \geq 0$ or not. Thus, the classifier (3.15) partitions the feature space $\mathcal{X} = \mathbb{R}^n$ into two half-spaces $\mathcal{R}_1 = \{\mathbf{x} : (\hat{\mathbf{w}})^T \mathbf{x} \geq 0\}$ and $\mathcal{R}_{-1} = \{\mathbf{x} : (\hat{\mathbf{w}})^T \mathbf{x} < 0\}$ which are separated by the hyperplane $(\hat{\mathbf{w}})^T \mathbf{x} = 0$ (see Figure 2.8). Any data point 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 particular probabilistic inference method. This interpretation is based on modelling the labels $y \in \{-1, 1\}$ as i.i.d. random variables with some probability $P(y=1)$ which is parameterized by a linear predictor $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$ via

$$\log P(y=1)/(1 - P(y=1)) = \mathbf{w}^T \mathbf{x},\quad (3.16)$$

or, equivalently,

$$P(y=1) = 1/(1 + \exp(-\mathbf{w}^T \mathbf{x})).\quad (3.17)$$

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

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

Given the probabilistic model (3.17), a principled approach to choosing the weight vector \mathbf{w} is based on maximizing the probability (or likelihood) of the observed dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ under the probabilistic model (3.17). This yields the maximum likelihood

estimator

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

The maximizer of a positive function $f(\mathbf{w}) > 0$ is not affected by replacing $f(\mathbf{w})$ with $\log f(x)$, i.e., $\operatorname{argmax}_{\mathbf{w} \in \mathbb{R}^n} h(\mathbf{w}) = \operatorname{argmax}_{\mathbf{w} \in \mathbb{R}^n} \log h(\mathbf{w})$. Therefore, (3.19) can be further developed as

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

Comparing (3.20) with (3.14) reveals that logistic regression is nothing but maximum likelihood estimation of the weight vector \mathbf{w} in the probabilistic model (3.17).

3.6 Support Vector Machines

Support vector machines (SVM) are classification methods which use the hinge loss (2.8) to evaluate the quality of a given classifier $h \in \mathcal{H}$. 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), which is also underlying linear regression (see Section 3.1) and logistic regression (see Section 3.5).

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

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

with a tuning parameter $\lambda > 0$. According to [?, Chapter 2], a classifier $h^{(\mathbf{w}_{\text{SVM}})}$ minimizing the loss (3.21), averaged over some labeled data points $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, is equivalent to maximizing the distance (margin) ξ between the decision boundary, given by the set

of points \mathbf{x} satisfying $\mathbf{w}_{\text{SVM}}^T \mathbf{x} = 0$, and each of the two classes $\mathcal{C}_1 = \{\mathbf{x}^{(i)} : y^{(i)} = 1\}$ and $\mathcal{C}_2 = \{\mathbf{x}^{(i)} : y^{(i)} = -1\}$. Maximizing this margin is sensible 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 3.3, the margin between the decision boundary and the classes \mathcal{C}_1 and \mathcal{C}_2 is typically determined by few data points (such as $\mathbf{x}^{(6)}$ in Figure 3.3) which are closest to the decision boundary. Such data points are referred to as **support vectors** and entirely determine the resulting classifier $h^{(\mathbf{w}_{\text{SVM}})}$. In other words, once the support vectors are identified the remaining data points become irrelevant for learning the classifier $h^{(\mathbf{w}_{\text{SVM}})}$.

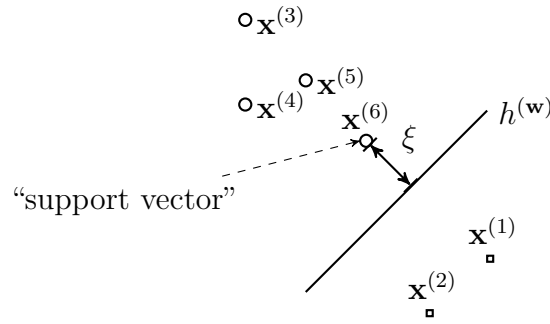


Figure 3.3: The SVM aims at a classifier $h^{(\mathbf{w})}$ with small hinge loss. Minimizing hinge loss of a classifier is the same as maximizing the margin ξ between the decision boundary (of the classifier) and each class of the training set.

We highlight that both, the SVM and logistic regression amount to linear classifiers $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}$ (see (3.1)) whose decision boundary is a hyperplane in the feature space $\mathcal{X} = \mathbb{R}^n$ (see Figure 2.8). The difference between SVM and logistic regression is the loss function used for evaluating the quality of a particular classifier $h^{(\mathbf{w})} \in \mathcal{H}^{(n)}$. The SVM uses the hinge loss (2.8) which is the best convex approximation to the 0/1 loss (2.6). 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 $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ otherwise) compared to logistic regression which uses the logistic loss (2.9).

The statistical superiority of the SVM comes at the cost of increased computational complexity. In particular, the hinge loss (2.8) is non-differentiable which prevents the use of simple gradient-based methods (see Chapter 5) and requires more advanced optimization methods. In contrast, the logistic loss (2.9) is convex and differentiable which allows to apply simple iterative methods for minimization of the loss (see Chapter 5).

3.7 Bayes' Classifier

Consider a classification problem involving data points characterized by features $\mathbf{x} \in \mathcal{X}$ and associated with labels $y \in \mathcal{Y}$ out of a discrete label space \mathcal{Y} . The family of Bayes' classifier methods is based on ML problems using the 0/1 loss (2.6) for assessing the quality of classifiers h .

The goal of ML is to find (or learn) a classifier $h : \mathcal{X} \rightarrow \mathcal{Y}$ 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 (2.6). If we model the data points, with its features \mathbf{x} and label y , as i.i.d. random variables, the 0/1 loss approximates the misclassification (error) probability $P_{\text{err}} = P(y \neq h(\mathbf{x}))$.

An important subclass of Bayes' classifiers uses the hypothesis space (3.1) which is also underlying logistic regression (see Section 3.5) and the SVM (see Section 3.6). Thus, logistic regression, the SVM and Bayes' classifiers yield linear classifiers (see Figure 2.8) which 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 4.4.

3.8 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 data point based on its features $\mathbf{x} \in \mathcal{X}$, we apply a predictor h selected out of some hypothesis space \mathcal{H} . Let us assume that the available computational infrastructure only allows to use a linear hypothesis space $\mathcal{H}^{(n)}$ (see (3.1)).

For some applications using only linear predictor maps in $\mathcal{H}^{(n)}$ is not sufficient to model the relation between features and labels (see Figure 3.1 for a data set which suggests a non-linear relation between features and labels). In such cases it is beneficial to add a pre-processing step before applying a predictor 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}' [?]. 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 3.4).

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 $\phi : \mathcal{X} \rightarrow \mathcal{X}'$ which maps the original feature vector \mathbf{x} to a new feature vector $\hat{\mathbf{x}} = \phi(\mathbf{x})$.

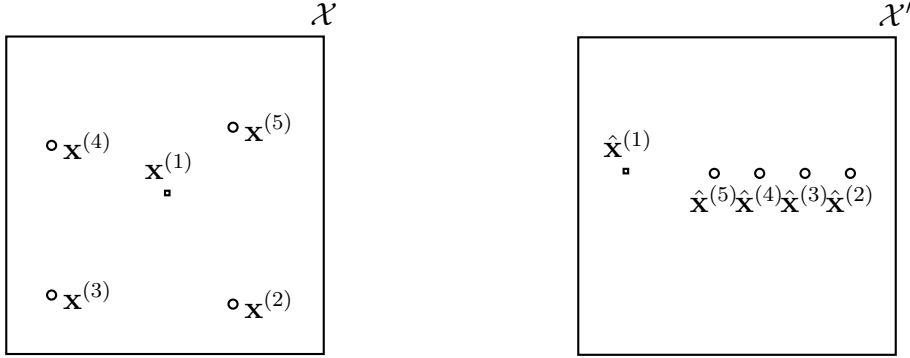


Figure 3.4: Consider a data set $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^5$ constituted by data points with features $\mathbf{x}^{(i)}$ and binary labels $y^{(i)}$. Left: In the original feature space \mathcal{X} , the data points cannot be separated perfectly by any linear classifier. Right: The feature map $\phi : \mathcal{X} \rightarrow \mathcal{X}'$ transforms the features $\mathbf{x}^{(i)}$ to the new features $\hat{\mathbf{x}}^{(i)} = \phi(\mathbf{x}^{(i)})$ in the new feature space \mathcal{X}' . In the new feature space \mathcal{X}' the data points can be separated perfectly by a linear classifier.

3.9 Decision Trees

A decision tree is a flowchart-like description of a map $h : \mathcal{X} \rightarrow \mathcal{Y}$ which maps the features $\mathbf{x} \in \mathcal{X}$ of a data point to a predicted label $h(\mathbf{x}) \in \mathcal{Y}$ [?]. While decision trees can be used for arbitrary feature space \mathcal{X} and label space \mathcal{Y} , we will discuss them for the particular feature space $\mathcal{X} = \mathbb{R}^2$ and label space $\mathcal{Y} = \mathbb{R}$.

We have depicted an example of a decision tree in Figure 3.5. The decision tree consists of nodes which are connected by directed edges. It is useful to think of a decision tree as a step-by-step instruction (a “recipe”) for how to compute the predictor value $h(\mathbf{x})$ given the input feature $\mathbf{x} \in \mathcal{X}$. This computation starts at the **root node** and ends at one of the **leaf nodes**. A leaf node m , which does not have any outgoing edges, corresponds to a certain subset or “region” $\mathcal{R}_m \subseteq \mathcal{X}$ of 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 3.5 consists of two decision nodes (including the root node) and three leaf nodes.

Given limited computational resources, we need to restrict ourselves to decision trees which are not too large. We can define a particular hypothesis space by collecting all decision trees which uses the tests “ $\|\mathbf{x} - \mathbf{u}\| \leq r$ ” and “ $\|\mathbf{x} - \mathbf{v}\| \leq r$ ” (for fixed vectors \mathbf{u} and \mathbf{v} and fixed radius $r > 0$) and depth not larger than 2.³ To assess the quality of different decision trees we need to use some loss function. Examples of loss functions used to measure the quality of a decision tree are the squared error loss (for numeric labels) or the impurity of individual decision regressions (for categorical labels).

In general, we are not interested in one particular decision tree only but in a large set of different decision trees from which we choose the most suitable given some data (see Section 4.3). We can define a hypothesis space by collecting predictor maps h represented by a set of decision trees (such as depicted in Figure 3.6).

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 test

³The 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 3.5 has depth 2.

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 implement elementary tests.

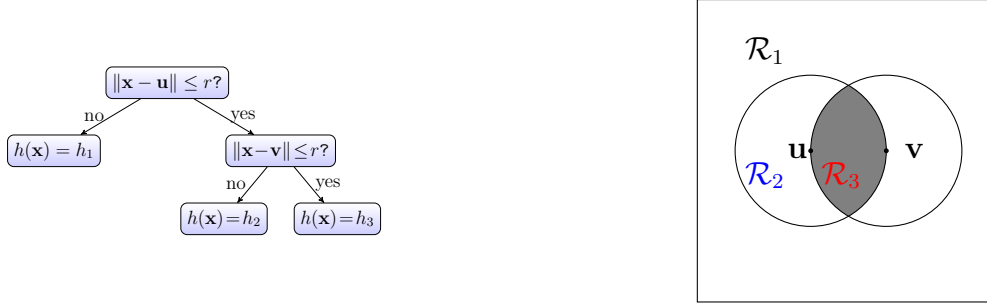


Figure 3.5: A decision tree represents a hypothesis h which is constant on subsets \mathcal{R}_m , i.e., $h(\mathbf{x}) = h_m$ for all $\mathbf{x} \in \mathcal{R}_m$. Each subset $\mathcal{R}_m \subseteq \mathcal{X}$ corresponds to a leaf node in the decision tree.

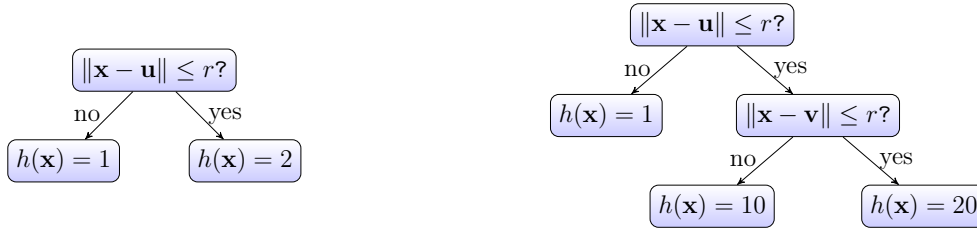


Figure 3.6: A hypothesis space \mathcal{H} consisting of two decision trees with depth at most 2 and using the tests $\|\mathbf{x} - \mathbf{u}\| \leq r$ and $\|\mathbf{x} - \mathbf{v}\| \leq r$ with a fixed radius r and points \mathbf{u} and \mathbf{v} .

A decision tree represents a map $h : \mathcal{X} \rightarrow \mathcal{Y}$, which is piecewise-constant over regions of the feature space \mathcal{X} . These non-overlapping regions form a partitioning of the feature space. Each leaf node of a decision tree corresponds to one particular region. Using large decision trees, which involve many different test nodes, we can represent very complicated partitions that resemble any given labeled dataset (see Figure 3.7).

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, since a linear map is constant along hyperplanes. In particular, linear classifiers partition the feature-space into two half-spaces (see Figure 2.8). In contrast, the geometry of the map represented by a decision tree maps can be arbitrary complicated if the decision tree is sufficiently large (deep).

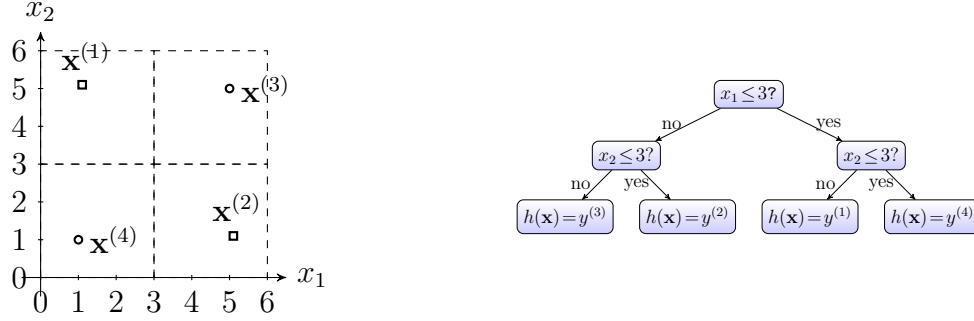


Figure 3.7: Using a sufficiently large (deep) decision tree, we can construct a map h that perfectly fits any given labeled dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ such that $h(\mathbf{x}^{(i)}) = y^{(i)}$ for $i = 1, \dots, m$.

3.10 Artificial Neural Networks – Deep Learning

Another example of a hypothesis space, which has proven useful in a wide range of applications, e.g., image captioning or automated translation, is based on a **network representation** of a predictor $h : \mathbb{R}^n \rightarrow \mathbb{R}$. We can define a predictor $h^{(\mathbf{w})} : \mathbb{R}^n \rightarrow \mathbb{R}$ using an **artificial neural network** (ANN) structure as depicted in Figure 3.8. A feature vector $\mathbf{x} \in \mathbb{R}^n$ is

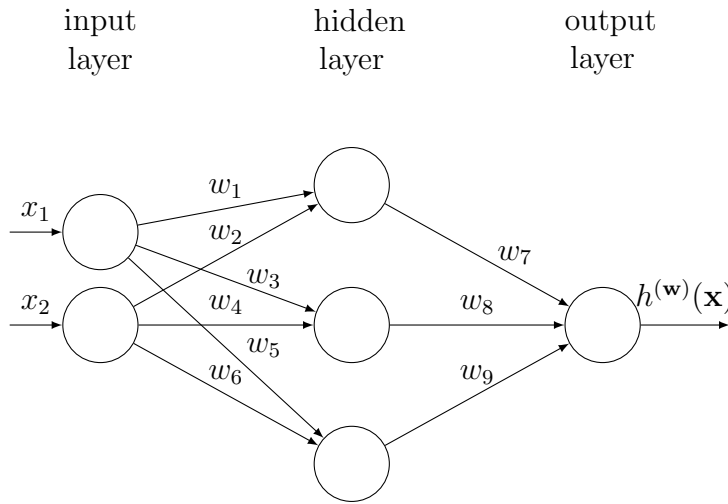


Figure 3.8: ANN representation of a predictor $h^{(\mathbf{w})}(\mathbf{x})$ which maps the input (feature) vector $\mathbf{x} = (x_1, x_2)^T$ to a predicted label (output) $h^{(\mathbf{w})}(\mathbf{x})$.

fed into the input units, each of which reads in one single feature $x_i \in \mathbb{R}$. The features x_i are then multiplied with the weights $w_{j,i}$ associated with the link between the i -th input node (“neuron”) with the j -th node in the middle (hidden) layer. The output of the j -th

node in the hidden layer is given by $s_j = g(\sum_{i=1}^n w_{j,i}x_i)$ with some (typically non-linear) **activation function** $g(z)$. The input (or activation) z for the activation (or output) $g(z)$ of a neuron is a weighted (linear) combination $\sum_{i=1}^n w_{j,i}s_i$ of the outputs s_i of the nodes in the previous layer. For the ANN depicted in Figure 3.8, 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 [?] for an in-depth introduction to deep learning methods).

Remarkably, using some simple non-linear activation function $g(z)$ as the building block for ANNs allows to represent an extremely large class of predictor maps $h^{(\mathbf{w})} : \mathbb{R}^n \rightarrow \mathbb{R}$. 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 [?, Ch. 6.4.1.]. It can be shown that an ANN with only one single hidden layer can approximate any given map $h : \mathcal{X} \rightarrow \mathcal{Y} = \mathbb{R}$ to any desired accuracy [?]. 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 [?].

Exercise. Consider the simple ANN structure in Figure 3.9 using the “ReLU” activation function $g(z) = \max\{z, 0\}$ (see Figure 3.10). Show that there is a particular choice for the weights $\mathbf{w} = (w_1, \dots, w_9)^T$ such that the resulting hypothesis map $h^{(\mathbf{w})}(x)$ is a triangle as depicted in Figure 3.11. Can you also find a choice for the weights $\mathbf{w} = (w_1, \dots, w_9)^T$ that produce the same triangle shape if we replace the ReLU activation function with the linear function $g(z) = 10 \cdot z$?

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 ANN representation also allows to efficiently how the loss function changes with small modifications of the weights \mathbf{w} . The gradient of the overall loss or empirical risk (see Chapter 5) can be obtained via a message passing procedure known as **back-propagation** [?].

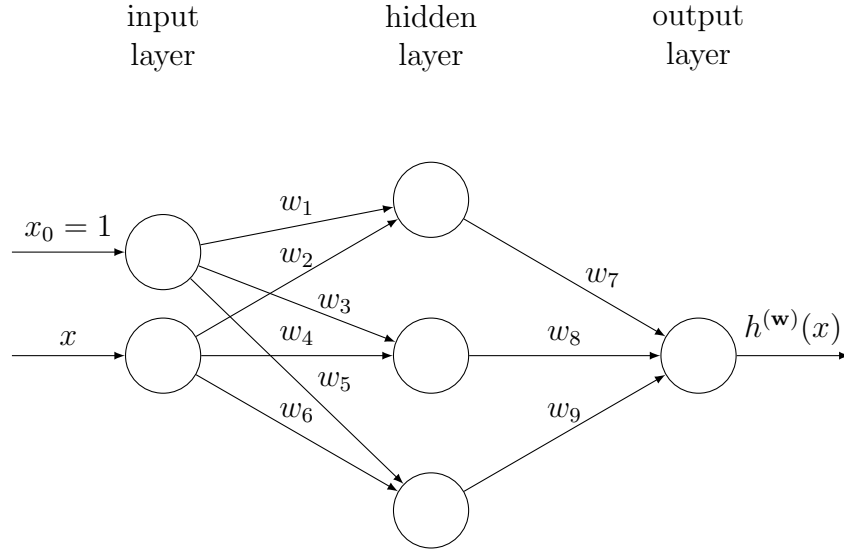


Figure 3.9: This ANN with one hidden layer defines a hypothesis space consisting of all maps $h^{(\mathbf{w})}(x)$ obtained by implementing the ANN with different weight vectors $\mathbf{w} = (w_1, \dots, w_9)^T$.

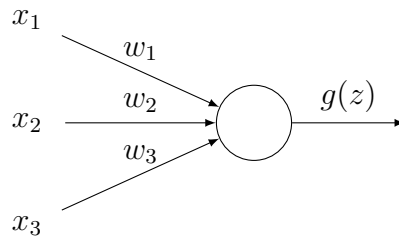


Figure 3.10: Each single neuron of the ANN depicted in Figure 3.9 implements a weighted summation $z = \sum_i w_i x_i$ of its inputs x_i followed by applying a non-linear activation function $g(z)$.

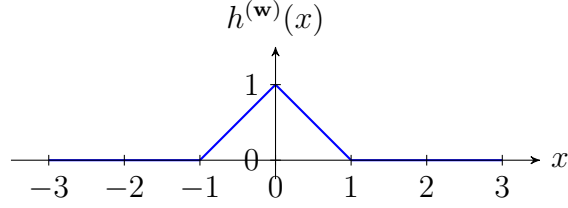


Figure 3.11: A hypothesis map with the shape of a triangle.

3.11 Maximum Likelihood Methods

For many applications it is useful to model the observed data points $\mathbf{z}^{(i)}$ as realizations of a random variable \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 $\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)} \sim P(\mathbf{z}; \mathbf{w})$ is **maximum likelihood estimation** [?].

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

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

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 $\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)} \sim P(\mathbf{z}; \mathbf{w})$, the maximum likelihood estimates $\hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}}$ of the mean vector and the covariance matrix are obtained via

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

The optimization in (3.23) is over all pairs given by some mean vector $\boldsymbol{\mu} \in \mathbb{R}^n$ and some covariance matrix $\boldsymbol{\Sigma} \in \mathbb{S}_+^n$. Here, \mathbb{S}_+^n denotes the set of all psd Hermitian $n \times n$ matrices. Note that this maximum likelihood problem (3.23) can be interpreted as an instance of ERM (4.1) using the particular loss function (3.22). The resulting estimates are given explicitly as

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

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

3.12 k -Nearest Neighbours

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 [?]). A prime example of a metric space is \mathbb{R}^n with the Euclidean metric induced by the distance measure $\|\mathbf{x} - \mathbf{y}\|$ between two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

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

In contrast to the ML problems discussed above in Section 3.1 - Section 3.10, the hypothesis space of k -NN depends on the training data \mathbb{X} .

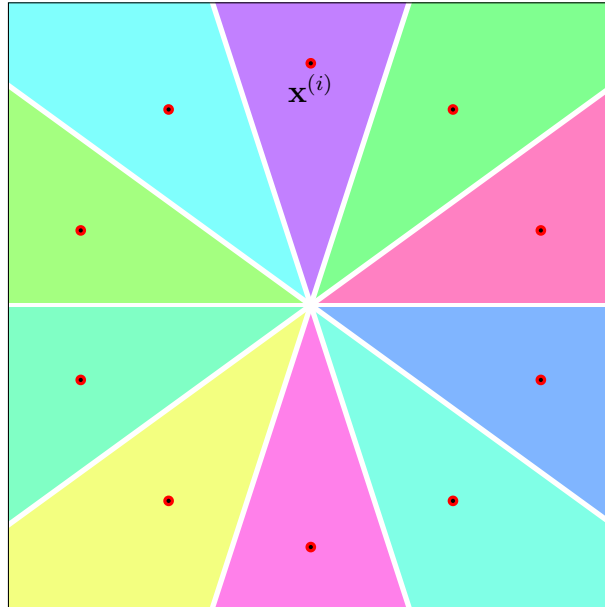


Figure 3.12: A hypothesis map h for k -NN with $k = 1$ and feature space $\mathcal{X} = \mathbb{R}^2$. The hypothesis map is constant over regions (indicated by the coloured areas) located around feature vectors $\mathbf{x}^{(i)}$ (indicated by a dot) of a dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}$.

3.13 Dimensionality Reduction

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

3.14 Clustering Methods

data points are whose datasets; labels are correct partitioning/clustering; loss function is some notion of purity;

3.15 Reinforcement Learning Methods

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

3.16 Network Lasso

Maybe the most widely used choice for the feature space \mathcal{X} in ML methods is the Euclidean space \mathbb{R}^n . If the features of the data points are available in numeric form it is quite natural to stack them into feature vectors. But even for non-numerical data such as text it is often preferable to transform it to numeric features (word-embedding). The feature space \mathbb{R}^n is attractive since it has a rich algebraic and geometric structure which allows to navigate (search) it efficiently.

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

An particular class of ML problems involve partially labeled network-structured data arising in many important application domains including signal processing [? ?], image processing [? ?], social networks, internet and bioinformatics [? ? ?]. Such network-structured data (see Figure 3.13) can be described by an “empirical graph” $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$, whose nodes \mathcal{V} represent individual data points which are connected by edges \mathcal{E} if they are considered “similar” in an application-specific sense. The extend of similarity between connected nodes $i, j \in \mathcal{V}$ is encoded in the edge weights $W_{i,j} > 0$ which are collected into the weight matrix $\mathbf{W} \in \mathbb{R}_+^{|\mathcal{V}| \times |\mathcal{V}|}$.

The notion of similarity between data points can be based on physical proximity (in time or space), communication networks or probabilistic graphical models [? ? ?]. Besides the graph structure, datasets carry additional information in the form of labels associated with individual data points. In a social network, we might define the personal preference for some product as the label associated with a data point (which represents a user profile). Acquiring labels is often costly and requires manual labor or experiment design. Therefore, we assume to have access to the labels of only few data points which belong to a small “training set”.

The availability of accurate network models for datasets provides computational and statistical benefits. Computationally, network models lend naturally to highly scalable ML methods which can be implemented as message passing over the empirical graph [?]. Network models enable to borrow statistical strength between connected data points, which allows semi-supervised learning (SSL) methods to capitalize on massive amounts of unlabeled data [?].

The key idea behind many SSL methods is the assumption that labels of close-by data points are similar, which allows to combine partially labeled data with its network structure in order to obtain predictors which generalize well [? ?]. While SSL methods on graphs have been applied to many application domains, the precise understanding of which type of data allow for accurate SSL is still in its infancy [? ? ?].

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

Most methods for processing graph signals rely on a signal model which are inspired by a cluster assumption [?]. The cluster assumption requires similar signal values $h[i] \approx h[j]$ at nodes $i, j \in \mathcal{V}$, which belong to the same well-connected subset of nodes (“cluster”) of the empirical graph. The clusteredness of a graph signal $h[\cdot]$ can be measured by the total

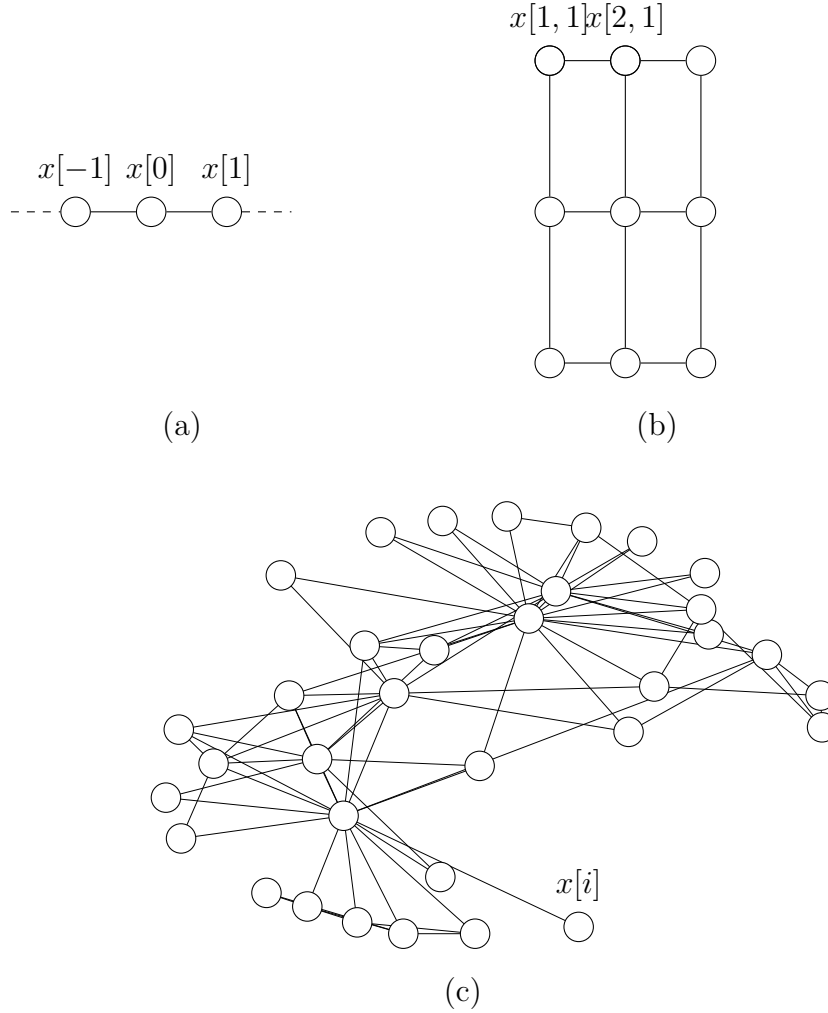


Figure 3.13: Examples for the empirical graph of networked data. (a) Chain graph representing signal amplitudes of discrete time signals. (b) Grid graph representing pixels of 2D-images. (c) Empirical graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ for a dataset obtained from the social relations between members of a Karate club [?]. The empirical graph contains m nodes $i \in \mathcal{V} = \{1, \dots, m\}$ which represent m individual club members. Two nodes $i, j \in \mathcal{V}$ are connected by an edge $\{i, j\} \in \mathcal{E}$ if the corresponding club members have interacted outside the club.

variation (TV):

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

Clustered graph signals arise in digital signal processing which studies graph signals defined over the chain graph representing sampling time instants. Signal samples at adjacent time instants are strongly correlated for sufficiently high sampling rate. Image processing methods rely on close-by pixels tending to be coloured likely which amounts to a clustered graph signal over a grid graph representing pixels of a 2D image.

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

$$\mathcal{H} = \{h : \mathcal{V} \rightarrow \mathcal{Y}\}. \quad (3.26)$$

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

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

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

Logistic Network Lasso. The logistic network Lasso [? ?] is a modification of the network Lasso (see Section 3.16) for classification problems involving partially labeled networked data represented by an empirical graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$.

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

$$\mathcal{H} = \{h : \mathcal{V} \rightarrow \mathcal{Y}\} \quad (3.28)$$

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

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

3.17 Exercises

3.17.1 How Many Neurons?

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

3.17.2 Linear Classifiers

Consider data points 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?

3.17.3 Data Dependent Hypothesis Space

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

- logistic regression
- linear regression
- k-NN

Chapter 4

Empirical Risk Minimization

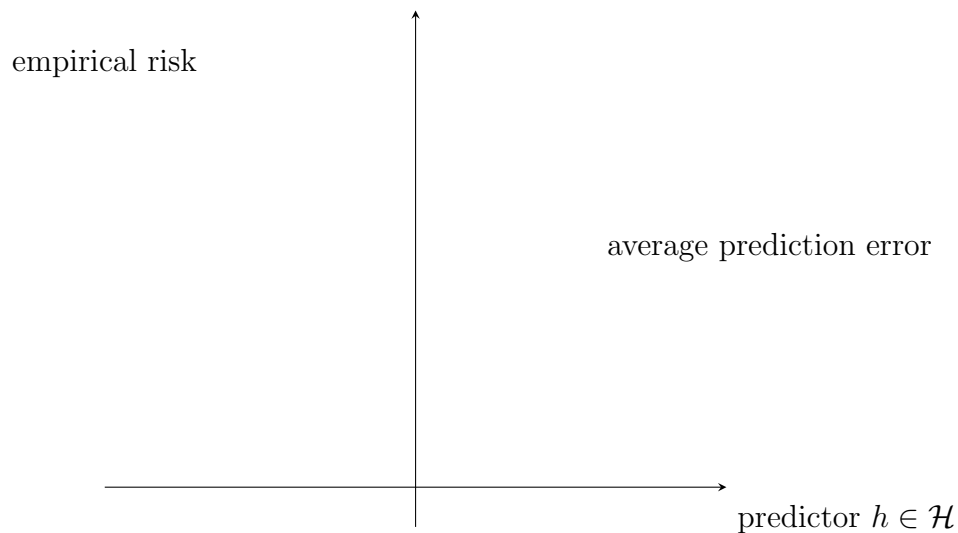


Figure 4.1: ML methods aim at learning (finding) the predictor $h \in \mathcal{H}$ with minimal average prediction error. However, the average prediction error (dashed curve) cannot be determined in general. Empirical risk minimization methods are based on replacing (or approximating) the average prediction error with the empirical risk (solid curve) incurred by a predictor when applied to a finite set of labeled data points (the training set).

As detailed in Chapter 2, ML problems (and methods) consist of the following components (see Figure 2.1):

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

Given such a ML problem specification, ML methods aim at finding (learning) an accurate predictor map h out of \mathcal{H} such that $h(\mathbf{x}) \approx y$ for a random data point (\mathbf{x}, y) . We make this informal statement precise by using a simple probabilistic model for the data. In particular, we assume data points are i.i.d. realizations drawn from some fixed (unknown) probability distribution $p(\mathbf{x}, y)$.

The average prediction error or average risk of a given predictor is then defined as the expected loss $\mathbb{E}\{\mathcal{L}((\mathbf{x}, y), h)\}$. ML methods can be considered as optimization methods that choose (optimize or learn) a predictor out of \mathcal{H} such that it has minimum average risk. 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 a Bayes predictor and depends on the loss function. For the squared error loss, the Bayes predictor is the posterior mean of y given the features \mathbf{x} .

Note that in practice we cannot directly construct Bayes' optimal predictors using probability calculus since we do not know probability distribution underlying the data. However, Bayes' optimal predictors can provide a very useful benchmark against we can compare practical ML methods. Let us now see how the construction of Bayes' optimal predictors can be adapted to obtain practical ML methods.

4.1 The Basic Idea of ERM

In practice, we typically do not know exactly the probability distribution and therefore cannot determine precisely the average error uncured by a particular hypothesis. However, we can estimate this average error using an empirical average on some data points ("training data"). Many ML methods are optimization methods that use the empirical risk on the training data as an estimate for the average (expected) error. Thus, in contrast to standard optimization methods, these ML methods optimize an estimate (approximation) of the true objective function (the average error of a predictor map).

A key idea underlying many ML methods is to find (learn) a predictor map with small empirical risk, which serves as a proxy for the average prediction error. The resulting ML methods are optimization methods which solve an **empirical risk minimization (ERM)**

problem

$$\begin{aligned}\hat{h} &= \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{E}(h|\mathbb{X}) \\ &\stackrel{(2.10)}{=} \operatorname{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^m \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h).\end{aligned}\tag{4.1}$$

The ERM (4.1) amounts to learning (finding) a good predictor \hat{h} (out of the hypothesis space \mathcal{H}) by “training” it on the dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, which is referred to as the **training set**.

Solving the optimization problem (4.1) provides two things. First, the minimizer \hat{h} is a predictor which performs optimal on the training set \mathbb{X} . Second, the corresponding objective value $\mathcal{E}(\hat{h}|\mathbb{X})$ (the “training error”) indicates how accurate the predictions of \hat{h} will be. However, as we will discuss in Chapter 7, in certain ML problems the training error $\mathcal{E}(\hat{h}|\mathbb{X})$ obtained for the dataset \mathbb{X} can be very different from the average prediction error of \hat{h} when applied to new data points which are not contained in \mathbb{X} .

Given a parameterization $h^{(\mathbf{w})}(\mathbf{x})$ of the predictor maps in the hypothesis space \mathcal{H} , such as within linear regression (2.4) or for ANNs (see Figure 3.8), we can reformulate the optimization problem (4.1) (with optimization domain being a function space!) as an optimization directly over the weight vector:

$$\mathbf{w}_{\text{opt}} = \operatorname{argmin}_{\mathbf{w} \in \mathbb{R}^n} f(\mathbf{w}) \text{ with } f(\mathbf{w}) := \mathcal{E}(h^{(\mathbf{w})}|\mathbb{X}).\tag{4.2}$$

The objective function $f(\mathbf{w})$ in (4.2) is the empirical risk $\mathcal{E}(h^{(\mathbf{w})}|\mathbb{X})$ achieved by $h^{(\mathbf{w})}$ when applied to the data points in the dataset \mathbb{X} . Note that the two formulations (4.2) and (4.1) are fully equivalent. In particular, given the optimal weight vector \mathbf{w}_{opt} solving (4.2), the predictor $h^{(\mathbf{w}_{\text{opt}})}$ is an optimal predictor solving (4.1).

Learning a good predictor map via ERM (4.1) can be interpreted as 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 $y^{(i)}$. We (as the learner) then try out some predictor map h in order to tell the label $y^{(i)}$ only from the snapshot features $\mathbf{x}^{(i)}$ and determine the (training) error $\mathcal{E}(h|\mathbb{X})$ incurred. If the error $\mathcal{E}(h|\mathbb{X})$ is too large we try out another predictor map h' instead of h with the hope of achieving a smaller training error $\mathcal{E}(h'|\mathbb{X})$.

This principle of learning by supervision, i.e., using labeled data points (“training examples”),

could also be used to model the development of language in human brains (“concept learning”). Consider a child, who should learn the concept “tractor” (see Figure 4.1). We could try to show many different pictures to the child and for each picture say “tractor” or “no tractor”. Based on this “labeled data”, the child tries to learn the relation between features of an image and the presence (or absence) of a tractor in the image.



Figure 4.2: A bunch of labeled images. The label y of an image indicates if a tractor is shown ($y = 1$) or not ($y = -1$).

We highlight that the precise shape of the objective function $f(\mathbf{w})$ in (4.2) 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} . Moreover, 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 4.3, the different combinations of predictor parametrisation and loss functions can result in objective functions with fundamentally different properties such that their optimization is more or less difficult.

The objective function $f(\mathbf{w})$ for the ERM obtained for linear regression (see Section 3.1) is differentiable and convex and can therefore be minimized using simple iterative gradient descent methods (see Chapter 5). In contrast, the objective function $f(\mathbf{w})$ of ERM obtained for the SVM (see Section 3.6) 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 [?].

The objective function $f(\mathbf{w})$ obtained for ANN tends to be **highly non-convex** but still differentiable for particular activation functions. 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 [?]. Even more challenging is the ERM obtained for decision trees or Bayes' classifiers. These ML problems involve non-differentiable and non-convex objective functions.

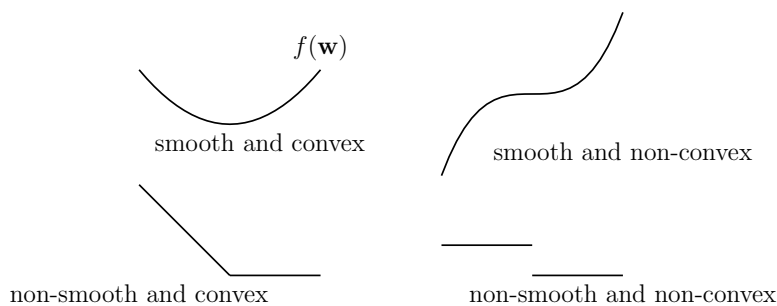


Figure 4.3: Different types of objective functions obtained for ERM in different settings.

In what follows, we discuss the structure of the ERM obtained for three particular ML problems: In Section 4.2, we discuss the ERM obtained for linear regression (see Section 3.1). The resulting ERM has appealing properties as it amounts to minimizing a differentiable (smooth) and convex function which can be done efficiently using efficient gradient based methods (see Chapter 5).

We then discuss in Section 4.3 the ERM obtained for decision trees which yields a discrete optimization problem and is therefore fundamentally different from the smooth ERM obtained for linear regression. In particular, we cannot apply gradient based methods (see Chapter 5) to solve them but have to rely on discrete search methods.

Finally, in Section 4.4, we discuss how Bayes' methods can be used to solve the non-convex and non-differentiable ERM problem obtained for classification problems with the 0/1 loss (2.6).

4.2 ERM for Linear Regression

Let us now focus on linear regression problem (see Section 3.1) which arises from using the squared error loss (2.5) and linear predictor maps $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{x}^T \mathbf{w}$. Here, we can rewrite the

ERM problem (4.2) as

$$\begin{aligned} \mathbf{w}_{\text{opt}} &= \underset{\mathbf{w} \in \mathbb{R}^n}{\operatorname{argmin}} f(\mathbf{w}) \\ \text{with } f(\mathbf{w}) &:= (1/|\mathbb{X}|) \sum_{(\mathbf{x}, y) \in \mathbb{X}} (y - \mathbf{x}^T \mathbf{w})^2. \end{aligned} \quad (4.3)$$

Here, $|\mathbb{X}|$ denotes the cardinality (number of elements) of the set \mathbb{X} . The objective function $f(\mathbf{w})$ in (4.3) has some computationally appealing properties, since it is convex and smooth (see Chapter 5).

It will be useful to rewrite the ERM problem (4.3) using matrix and vector representations of the feature vectors $\mathbf{x}^{(i)}$ and labels $y^{(i)}$ contained in the dataset \mathbb{X} . To this end, we stack the labels $y^{(i)}$ and the feature vectors $\mathbf{x}^{(i)}$, for $i = 1, \dots, m$, into a “label vector” \mathbf{y} and “feature matrix” \mathbf{X} as follows

$$\begin{aligned} \mathbf{y} &= (y^{(1)}, \dots, y^{(m)})^T \in \mathbb{R}^m, \text{ and} \\ \mathbf{X} &= \left(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)} \right)^T \in \mathbb{R}^{m \times n}. \end{aligned} \quad (4.4)$$

This allows to rewrite the objective function in (4.3) as

$$f(\mathbf{w}) = (1/m) \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2. \quad (4.5)$$

Inserting (4.5) into (4.3) we obtain the quadratic problem

$$\begin{aligned} \min_{\mathbf{w} \in \mathbb{R}^n} \underbrace{(1/2) \mathbf{w}^T \mathbf{Q} \mathbf{w} - \mathbf{q}^T \mathbf{w}}_{=f(\mathbf{w})} \\ \text{with } \mathbf{Q} = (1/m) \mathbf{X}^T \mathbf{X}, \mathbf{q} = (1/m) \mathbf{X}^T \mathbf{y}. \end{aligned} \quad (4.6)$$

Since $f(\mathbf{w})$ is a differentiable and convex function, a necessary and sufficient condition for some \mathbf{w}_0 to satisfy $f(\mathbf{w}_0) = \min_{\mathbf{w} \in \mathbb{R}^n} f(\mathbf{w})$ is the **zero-gradient condition** [?, Sec. 4.2.3]

$$\nabla f(\mathbf{w}_{\text{opt}}) = \mathbf{0}. \quad (4.7)$$

Combining (4.6) with (4.7), yields the following sufficient and necessary condition for a weight vector \mathbf{w}_{opt} to solve the ERM (4.3):

$$(1/m) \mathbf{X}^T \mathbf{X} \mathbf{w}_{\text{opt}} = (1/m) \mathbf{X}^T \mathbf{y}. \quad (4.8)$$

It can be shown that, for any given feature matrix \mathbf{X} and label vector \mathbf{y} , there always exists at least one optimal weight vector \mathbf{w}_{opt} which solves (4.8). The optimal weight vector might not be unique, such that there are several different vectors which achieve the minimum in (4.3). However, any optimal solution \mathbf{w}_{opt} , which solves (4.8), achieves the same minimum empirical risk

$$\mathcal{E}(h^{(\mathbf{w}_{\text{opt}})} \mid \mathbb{X}) = \min_{\mathbf{w} \in \mathbb{R}^n} \mathcal{E}(h^{(\mathbf{w})} \mid \mathbb{X}) = \|(\mathbf{I} - \mathbf{P})\mathbf{y}\|^2. \quad (4.9)$$

Here, we used the orthogonal projection matrix $\mathbf{P} \in \mathbb{R}^{m \times m}$ on the linear span of the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)})^T \in \mathbb{R}^{m \times n}$ (see (4.4)).¹

If the feature matrix \mathbf{X} (see (4.4)) has full column rank, implying invertibility of the matrix $\mathbf{X}^T \mathbf{X}$, the projection matrix \mathbf{P} is given explicitly as

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

Moreover, the solution of (4.8) is then unique and given by

$$\mathbf{w}_{\text{opt}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \quad (4.10)$$

The closed-form solution (4.10) requires the inversion of the $n \times n$ matrix $\mathbf{X}^T \mathbf{X}$. Computing the inverse can be computationally challenging for large feature length n (see Figure 2.3 for a simple ML problem where the feature length is almost a million). Moreover, inverting a matrix which is close to singular typically introduces numerical errors.

In Section 5.4, we will discuss an alternative method for computing (approximately) the optimal weight vector \mathbf{w}_{opt} which does not require any matrix inversion. This alternative method, referred to as “gradient descent”, is based on constructing a sequence $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \dots$ of increasingly accurate approximations of \mathbf{w}_{opt} . Two particular benefits of using this iterative method, instead of using the formula (4.10), for computing \mathbf{w}_{opt} are (i) it is computationally cheaper and (ii) it also works when the matrix \mathbf{X} is not full rank in which case the formula (4.10) becomes invalid.

4.3 ERM for Decision Trees

Let us now consider the ERM problem (4.1) 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

¹The linear span $\text{span} \mathbf{A}$ of a matrix $\mathbf{A} = (\mathbf{a}^{(1)}, \dots, \mathbf{a}^{(m)}) \in \mathbb{R}^{n \times m}$ 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} .

3.9). 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 3.6. 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 3.6, the ERM problem is easy since we just have to evaluate the empirical risk for each of the elements in \mathcal{H} and pick the one yielding the smallest. However, in ML applications we typically use significantly larger hypothesis spaces and then discrete optimization tends to be more complicated compared to smooth optimization which can be solved by (variants of) gradient descent (see Chapter 5).

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 [?, 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 \mathbb{X} 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 computational resources, i.e., we can only afford to use decision trees up to certain maximum depth. Beside the computational limitations, we also face statistical limitations for the maximum size of decision trees. Indeed, if we allow very large decision trees, which represent highly complicated maps, we might end up overfitting the training data (see Figure 3.7 and Chapter 7) which is detrimental for the prediction performance of decision trees obtained for new data (which has not been used for training or growing the decision tree).

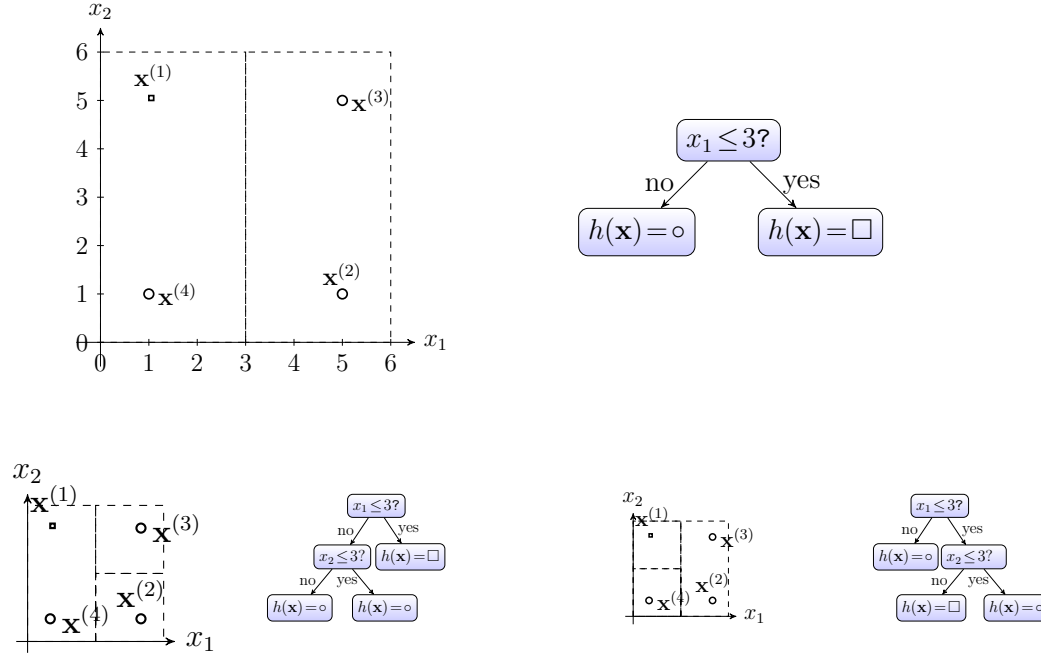


Figure 4.4: Given the labeled dataset and a decision tree in the top row, we would like to grow the decision tree by expanding it at one of its two leaf nodes. The resulting new decision trees obtained by expanding different leaf node is shown in the bottom row.

4.4 ERM for Bayes' Classifiers

The family of Bayes' classifiers is based on using the 0/1 loss (2.6) for measuring the quality of a classifier h . The resulting ERM is

$$\begin{aligned} \hat{h} &= \operatorname{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^m \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) \\ &\stackrel{(2.6)}{=} \operatorname{argmin}_{h \in \mathcal{H}} (1/m) \sum_{i=1}^m \mathcal{I}(h(\mathbf{x}^{(i)}) \neq y^{(i)}). \end{aligned} \quad (4.11)$$

Note that the objective function of this optimization problem is non-smooth (non differentiable) and non-convex (see Figure 4.3). This prevents us from using standard gradient based optimization methods (see Chapter 5) to solve (4.11).

We will now approach the ERM (4.11) via a different route by interpreting the data points $(\mathbf{x}^{(i)}, y^{(i)})$ as realizations of i.i.d. random variables which are distributed according to some probability distribution $p(\mathbf{x}, y)$. As discussed in Section 2.3, the empirical risk obtained using 0/1 loss approximates the error probability $P(\hat{y} \neq y)$ with the predicted label $\hat{y} = 1$

for $h(\mathbf{x}) > 0$ and $\hat{y} = -1$ otherwise (see (2.7)). Thus, we can approximate the ERM (4.11) as

$$\hat{h} \stackrel{(2.7)}{\approx} \operatorname{argmin}_{h \in \mathcal{H}} P(\hat{y} \neq y). \quad (4.12)$$

Note that the hypothesis h , which is the optimization variable in (4.12), enters into the objective function of (4.12) 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 (4.12) can be found easily via elementary Bayesian decision theory [?]. 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 data points $(\mathbf{x}^{(i)}, y^{(i)})$ which are modelled as i.i.d. random variables distributed according to $p(\mathbf{x}, y)$.

The estimation of $p(\mathbf{x}, y)$ can be based on a particular probabilistic model for the features and labels which depends on certain parameters and then determining the parameters using maximum likelihood (see Section 3.11). A widely used probabilistic model is based on Gaussian random vectors. In particular, conditioned on the label y , we model the feature vector \mathbf{x} as a Gaussian vector with mean $\boldsymbol{\mu}_y$ and covariance $\boldsymbol{\Sigma}$, i.e.,

$$p(\mathbf{x}|y) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_y, \boldsymbol{\Sigma}).^2 \quad (4.13)$$

Note that the mean vector of \mathbf{x} depends on the label such that for $y = 1$ the mean of \mathbf{x} is $\boldsymbol{\mu}_1$, while for data points with label $y = -1$ the mean of \mathbf{x} is $\boldsymbol{\mu}_{-1}$. In contrast, the covariance matrix $\boldsymbol{\Sigma} = \mathbb{E}\{(\mathbf{x} - \boldsymbol{\mu}_y)(\mathbf{x} - \boldsymbol{\mu}_y)^T | y\}$ of \mathbf{x} is the same for both values of the label $y \in \{-1, 1\}$. Note that, while conditioned on y the random vector \mathbf{x} is Gaussian, the marginal distribution of \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

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

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

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

$h(\mathbf{x}) > 0$ and $\hat{y} = -1$ for $h(\mathbf{x}) \leq 0$ using the classifier map

$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \text{ with } \mathbf{w} = \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_{-1}). \quad (4.14)$$

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

We cannot implement the classifier (4.14) 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 $\hat{\mathbf{\Sigma}}$, like the maximum likelihood estimates which are given by (see (3.24))

$$\begin{aligned} \hat{\boldsymbol{\mu}}_1 &= (1/m_1) \sum_{i=1}^m \mathcal{I}(y^{(i)} = 1) \mathbf{x}^{(i)}, \\ \hat{\boldsymbol{\mu}}_{-1} &= (1/m_{-1}) \sum_{i=1}^m \mathcal{I}(y^{(i)} = -1) \mathbf{x}^{(i)}, \\ \hat{\boldsymbol{\mu}} &= (1/m) \sum_{i=1}^m \mathbf{x}^{(i)}, \\ \text{and } \hat{\mathbf{\Sigma}} &= (1/m) \sum_{i=1}^m (\mathbf{z}^{(i)} - \hat{\boldsymbol{\mu}})(\mathbf{z}^{(i)} - \hat{\boldsymbol{\mu}})^T, \end{aligned} \quad (4.15)$$

with $m_1 = \sum_{i=1}^m \mathcal{I}(y^{(i)} = 1)$ denoting the number of data points with label $y = 1$ (m_{-1} is defined similarly). Inserting the estimates (4.15) into (4.14) yields the implementable classifier

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

We highlight that the classifier (4.16) is only well-defined if the estimated covariance matrix $\hat{\mathbf{\Sigma}}$ (4.15) is invertible. This requires to use a sufficiently large number of training data points such that $m \geq n$.

Using the route via maximum likelihood estimation, we arrived at (4.16) as an approximate solution to the ERM (4.11). The final classifier (4.16) turns out to be a linear classifier very much like logistic regression and SVM. In particular, the classifier (4.16) partitions the feature space \mathbb{R}^n into two halfspaces: one for $\hat{y} = 1$ and one for $\hat{y} = -1$ (see Figure 2.8). Thus, the Bayes' classifier (4.16) belongs to the same family (of linear classifiers) as logistic regression and the SVM. These three classification methods differ only in the way of choosing the decision boundary (see Figure 2.8) separating the two half-spaces in the feature space.

For the estimator $\hat{\mathbf{\Sigma}}$ (3.24) to be accurate (close to the unknown covariance matrix) we

need a number of data points (sample size) which is at least on the order of n^2 . This sample size requirement might be infeasible in applications with only few data points. Moreover, the maximum likelihood estimate $\hat{\Sigma}$ (4.15) is not invertible whenever $m < n$ such that the expression (4.16) becomes useless in this case. In order to cope with small sample size $m < n$ we can simplify the model (4.13) by requiring the covariance to be diagonal $\Sigma = \text{diag}(\sigma_1^2, \dots, \sigma_n^2)$. This is equivalent to modelling the individual features x_1, \dots, x_n of a particular data point as conditionally independent, given the label y the data point. The resulting special case of a Bayes' classifier is often referred to as a **naïve Bayes** classifier.

We finally highlight that the classifier (4.16) is obtained using the generative model (4.13) 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 data points but aim directly at finding the relation between features \mathbf{x} and label y of a data point. 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 (4.13) allows to synthetically generate more labeled data by generating random features and labels according to the probability distribution (4.13). We refer to [?] for a more detailed comparison between generative and discriminative methods.

4.5 Online Learning

So far we considered the training set to be an unordered set of data points whose labels are known. Many applications generate data in a sequential fashion, data points 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 data points using a single numeric feature x . We learn the predictor based on some training data (??). The weight vector for the optimal linear predictor is characterized by (2.18).

Let us assume that the training data is built up sequentially, we start with $m = 1$ data points in the first time step, then in the next time step collect another data point to get $m = 2$ data points, \dots . We denote the feature matrix and label vector at time m by $\mathbf{X}^{(m)}$ and $\mathbf{y}^{(m)}$:

$$m = 1 : \quad \mathbf{X}^{(1)} = (\mathbf{x}^{(1)})^T, \quad \mathbf{y}^{(1)} = (y^{(1)})^T, \quad (4.17)$$

$$m = 2 : \quad \mathbf{X}^{(2)} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)})^T, \quad \mathbf{y}^{(2)} = (y^{(1)}, y^{(2)})^T, \quad (4.18)$$

$$m = 3 : \quad \mathbf{X}^{(3)} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)})^T, \quad \mathbf{y}^{(3)} = (y^{(1)}, y^{(2)}, y^{(3)})^T. \quad (4.19)$$

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 (2.18) for each time step m . However, this approach does not reuse computations already invested in solving (2.18) at previous time steps $m' < m$.

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 have only a very short training period to learn a good hypothesis and then only use the hypothesis for inference.

4.7 Exercise

4.7.1 Linear Regression

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

Chapter 5

Gradient Based Learning

This chapter is devoted to a workhorse of many ML algorithms. This workhorse, known as gradient descent, is an optimization method that can be applied to vastly different ML applications. Variants of gradient descent are used to tune the weights of artificial neural networks within deep learning methods [?]. Gradient descent 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 parametrized 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} . This requirement on the loss function is not too restrictive as many important ML problems, including linear regression (see Section 3.1) and logistic regression (see Section 3.5), result in a smooth loss function.¹

For a smooth loss function, the resulting ERM (see (4.2))

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

¹A smooth function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 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} .

is a **smooth optimization problem** of the form

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

with a smooth function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ of the vector argument $\mathbf{w} \in \mathbb{R}^n$. It turns out that a smooth function $f(\mathbf{w})$ can be approximated locally around a point \mathbf{w}_0 using a hyperplane, which passes through the point $(\mathbf{w}_0, f(\mathbf{w}_0))$ and having normal vector $\mathbf{n} = (\nabla f(\mathbf{w}_0), -1)$ (see Figure 5.1). Indeed, elementary analysis yields the following linear approximation (around a point \mathbf{w}_0) [?]

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

As detailed in the next section, the approximation (5.3) lends naturally to a simple but powerful 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 (such as deep learning methods).

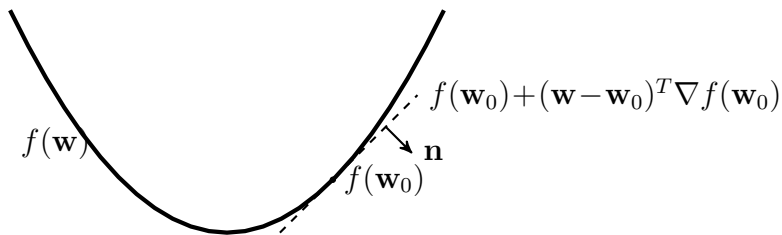


Figure 5.1: A smooth function $f(\mathbf{w})$ can be approximated locally around a point \mathbf{w}_0 using a hyperplane whose normal vector $\mathbf{n} = (\nabla f(\mathbf{w}_0), -1)$ is determined by the gradient $\nabla f(\mathbf{w}_0)$.

5.1 The Basic GD Step

We now discuss a very simple, yet quite powerful, algorithm for finding the weight vector \mathbf{w}_{opt} which solves continuous optimization problems like (5.1). Let us assume we have already some guess (or approximation) $\mathbf{w}^{(k)}$ for the optimal weight vector \mathbf{w}_{opt} and would like to improve it to a new guess $\mathbf{w}^{(k+1)}$ which yields a smaller value of the objective function $f(\mathbf{w}^{(k+1)}) < f(\mathbf{w}^{(k)})$. For a differentiable objective function $f(\mathbf{w})$, we can use the approximation $f(\mathbf{w}^{(k+1)}) \approx f(\mathbf{w}^{(k)}) + (\mathbf{w}^{(k+1)} - \mathbf{w}^{(k)})^T \nabla f(\mathbf{w}^{(k)})$ (cf. (5.3)) for $\mathbf{w}^{(k+1)}$ not



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

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

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

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**. It turns out that 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., $\lim_{k \rightarrow \infty} f(\mathbf{w}^{(k)}) = f(\mathbf{w}_{\text{opt}})$ (see Figure 5.2). When the GD step is used within an ML method (see Section 5.4 and Section 3.5), 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 3.10), 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 [?].

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

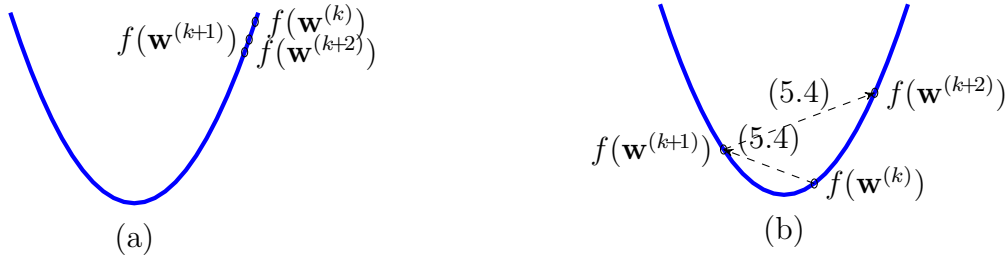


Figure 5.3: Effect of choosing learning rate α in GD step (5.4) too small (a) or too large (b). If the steps size α in the GD step (5.4) is chosen too small, the iterations make only very little progress towards the optimum. If the learning rate α is chosen too large, the iterates $\mathbf{w}^{(k)}$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

The choice of the step size α in the GD step (5.4) has a significant influence 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 in delivering an approximation of the optimal weight vector \mathbf{w}_{opt} (see (5.7)). If we choose the step size α too small (see Figure 5.3-(a)), the updates (5.4) make only very little progress towards approximating the optimal weight vector \mathbf{w}_{opt} . In many applications it is possible to repeat the GD steps only for a moderate number (e.g., when processing streaming data). Thus, if the GD step size is chosen too small, Algorithm 1 will fail to deliver a good approximation of \mathbf{w}_{opt} within an acceptable amount of 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 [?, 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 $k = 1, 2, \dots$. If the objective function $f(\mathbf{w})$ is convex and smooth, the GD steps (5.4) converge to an optimum \mathbf{w}_{opt} if the step size α satisfies [?]

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

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 $\frac{\partial^2 f(\mathbf{w})}{\partial w_i \partial w_j}$ 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 $\lambda_{\max}(\nabla^2 f(\mathbf{w}))$ 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 $U \geq \lambda_{\max}(\nabla^2 f(\mathbf{w}))$ (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; use gradient as indicator for distance to optimum;

5.4 GD for Linear Regression

We can now formulate a full-fledged ML algorithm for solving a linear regression problem (see Section 3.1). This algorithm amounts to finding the optimal weight vector \mathbf{w}_{opt} for a linear predictor (see (3.1)) of the form

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

The optimal weight vector \mathbf{w}_{opt} for (5.6) should minimize the empirical risk (under squared error loss (2.5))

$$\mathcal{E}(h^{(\mathbf{w})}|\mathbb{X}) \stackrel{(4.2)}{=} (1/m) \sum_{i=1}^m (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)})^2, \quad (5.7)$$

incurred by the predictor $h^{(\mathbf{w})}(\mathbf{x})$ when applied to the labeled dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$. Thus, \mathbf{w}_{opt} is obtained as the solution of a particular smooth optimization problem (5.2), i.e.,

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

In order to apply GD (5.4) to solve (5.8), and to find the optima weight vector \mathbf{w}_{opt} , we need to compute the gradient $\nabla f(\mathbf{w})$. The gradient of the objective function in (5.8) is given by

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

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

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

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

Algorithm 1 “Linear Regression via GD”

Input: labeled dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ 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: $\mathbf{w}^{(k)} := \mathbf{w}^{(k-1)} + \alpha(2/m) \sum_{i=1}^m (y^{(i)} - (\mathbf{w}^{(k-1)})^T \mathbf{x}^{(i)}) \mathbf{x}^{(i)}$ (do a GD step (5.4))

4: **until** convergence

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

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

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

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 $e^{(i)} = (y^{(i)} - (\mathbf{w}^{(k-1)})^T \mathbf{x}^{(i)})$, which is the prediction (approximation) error obtained by the linear predictor $h^{(\mathbf{w}^{(k-1)})}(\mathbf{x}^{(i)}) = (\mathbf{w}^{(k-1)})^T \mathbf{x}^{(i)}$ 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 $h(\mathbf{x}^{(i)}) = (\mathbf{w}^{(k-1)})^T \mathbf{x}^{(i)}$ and then correcting the weight vector $\mathbf{w}^{(k-1)}$ according to the error $e^{(i)} = y^{(i)} - (\mathbf{w}^{(k-1)})^T \mathbf{x}^{(i)}$.

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

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

with the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)})^T \in \mathbb{R}^{m \times n}$ (see (4.4)). 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 $\lambda_{\max}((1/m) \mathbf{X}^T \mathbf{X})$ of this product and (iii) set the step size to $\alpha = 1/\lambda_{\max}((1/m) \mathbf{X}^T \mathbf{X})$.

While it might be challenging to compute the maximum eigenvalue $\lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X})$, it might be easier to find an upper bound U for it.² Given such an upper bound $U \geq \lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X})$, 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 $i = 1, \dots, m$. Then, by elementary linear algebra, one can verify the upper bound $U = 1$, i.e., $1 \geq \lambda_{\max}((1/m)\mathbf{X}^T\mathbf{X})$. 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 3.5, the classification method logistic regression amounts to constructing a classifier $h(\mathbf{w}_{\text{opt}})$ by minimizing the empirical risk (3.14) obtained for a labeled dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, with features $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and binary labels $y^{(i)} \in \{-1, 1\}$. Thus, logistic regression amounts to an instance of the smooth optimization problem (5.2), i.e.,

$$\begin{aligned} \mathbf{w}_{\text{opt}} &= \underset{\mathbf{w} \in \mathbb{R}^n}{\operatorname{argmin}} f(\mathbf{w}) \\ \text{with } f(\mathbf{w}) &= (1/m) \sum_{i=1}^m \log(1 + \exp(-y^{(i)} \mathbf{w}^T \mathbf{x}^{(i)})). \end{aligned} \quad (5.13)$$

In order 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

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

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

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

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

The update (5.15) has an appealing form as it amounts to correcting the previous guess (or

²The 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 (4.8), which we want to avoid by using the “cheaper” GD algorithm.

Algorithm 2 “Logistic Regression via GD”

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

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

1: **repeat**

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

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

4: **until** convergence

Output: $\mathbf{w}^{(k)}$ (which approximates the optimal weight vector \mathbf{w}_{opt} defined in (5.13))

approximation) $\mathbf{w}^{(k-1)}$ for the optimal weight vector \mathbf{w}_{opt} by the correction term

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

The correction term (5.16) is a weighted average of the feature vectors $\mathbf{x}^{(i)}$, each of those vectors is weighted by the factor $(\alpha/m) \cdot e^{(i)}$. These weighting factors consist of the global factor (α/m) (that applies equally to all feature vectors $\mathbf{x}^{(i)}$) and a sample-specific factor $e^{(i)} = \frac{y^{(i)}}{1 + \exp(y^{(i)} \mathbf{w}^{(k-1)T} \mathbf{x}^{(i)})}$, which quantifies the error of the classifier $h^{(\mathbf{w}^{(k-1)})}(\mathbf{x}^{(i)}) = (\mathbf{w}^{(k-1)})^T \mathbf{x}^{(i)}$ for a data point having true label $y^{(i)} \in \{-1, 1\}$ and the features $\mathbf{x}^{(i)} \in \mathbb{R}^n$.

We can use the sufficient condition (5.5) (which guarantees convergence of GD) to guide the choice of the step size α in Algorithm 2. In order 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 [? , Ch. 4.4.]

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

Here, we used the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)})^T \in \mathbb{R}^{m \times n}$ (see (4.4)) and the diagonal matrix $\mathbf{D} = \text{diag}\{d_1, \dots, d_m\} \in \mathbb{R}^{m \times m}$ with diagonal elements

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

We highlight that, in contrast to the Hessian (5.12) obtained for the objective function arising in linear regression, the Hessian (5.17) varies with the weight vector \mathbf{w} . This makes the analysis of Algorithm 2 and the optimal choice of step size somewhat more difficult compared

to Algorithm 1. However, since the diagonal entries (5.18) take values in the interval $[0, 1]$, for normalized features (with $\|\mathbf{x}^{(i)}\| = 1$) the step size $\alpha = 1$ ensures convergence of the GD updates (5.15) to the optimal weight vector \mathbf{w}_{opt} solving (5.13).

5.6 Data Normalization

The convergence speed of the GD steps (5.4), i.e., the number of steps required to reach the minimum of the objective function (4.3) within a prescribed accuracy, depends crucially on the condition number $\kappa(\mathbf{X}^T \mathbf{X})$. This condition number is defined as the ratio

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

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 (4.4)), 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 $\kappa(\mathbf{X}^T \mathbf{X})$ [?]. 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 $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$

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

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

- 2: normalise features to have unit variance, i.e.,

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

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

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 $\hat{\mathbf{X}} = (\hat{\mathbf{x}}^{(1)}, \dots, \hat{\mathbf{x}}^{(m)})^T$ tends to be well-conditioned, i.e., $\kappa(\hat{\mathbf{X}}^T \hat{\mathbf{X}}) \approx 1$.

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 $(\hat{\mathbf{X}})^T \hat{\mathbf{X}}$ with the matrix $\hat{\mathbf{X}} = (\hat{\mathbf{x}}^{(1)}, \hat{\mathbf{x}}^{(2)})^T$ 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 parametrized 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).

Note that the gradient $\nabla f(\mathbf{w})$ obtained for the optimization problem (5.1) has a particular structure. Indeed, the gradient is a sum

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

Evaluating the gradient $\nabla f(\mathbf{w})$ (e.g., within a GD step (5.4)) by 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 huge amount of network resources and also put limits on the rate by which the GD steps (5.4) can be executed.

ImageNet. The “ImageNet” database contains more than 10^6 images [?]. 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 quite simple: 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 randomness (stochastic approximations).

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

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

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 data points 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 data points is stored in a distributed fashion. However, this saving in computational complexity comes at the cost of introducing a non-zero gradient noise

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

into the SGD updates. In order avoid the accumulation of the gradient noise (5.22) while running SGD updates (5.21) the step size α needs to be gradually decreased, e.g., using $\alpha = 1/k$ with k being the iteration counter (see [?]).

The SGD iteration (5.21) 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 the data points are collected sequentially, one new data point $\mathbf{x}^{(t)}, y^{(t)}$ at each new time step t , we could a SGD variant for online learning (see Section 4.5). This online SGD algorithm amounts to computing, for each time step t , the iteration

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

5.8 Exercises

5.8.1 Use Knowledge About Problem Class

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

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

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

Chapter 6

Model Validation and Selection

The idea of ERM is to learn a good predictor by searching the hypothesis space \mathcal{H} for a predictor map h with minimum average loss (empirical error) on a training set. For ML methods using high-dimensional hypothesis spaces, such as linear maps with a large number of features or deep neural networks, this approach bears the risk of overfitting. A method overfits if it learns a predictor $h \in \mathcal{H}$ that, merely by luck, fits well the training data but does a poor job on other data. Such a predictor will fail to **generalize** well to new data for which we do not know the label y but only the features \mathbf{x} (if we would know the label, then there is no point in learning predictors which estimate the label).

This chapter discusses few basic techniques to detect and avoid overfitting. To detect overfitting we need to monitor or **to validate** the performance of the predictor h on new data point which are not contained in the training set. We call the set of data points used for validation, as the validation set. The empirical risk incurred by the predictor h on the validation set is referred to as validation error. If a method overfits, it will learn a predictor whose training error is much smaller than the validation error.

Validation is useful not only for verifying if the predictor generalises well to new data (in particular detecting overfitting) but also for guiding model selection. In what follows, we mean by model selection the problem of selecting a particular hypothesis space out of a whole ensemble of potential hypothesis spaces $\mathcal{H}_1, \mathcal{H}_2, \dots$

We first study the phenomenon of overfitting within a simple probabilistic model for the data points in Section 6.1. Then, in Section 6.2, we analyze a simple validation technique that allows to detect overfitting.

6.1 Overfitting

Let us illustrate the phenomenon of overfitting using a simplified model for how a human child learns the concept “tractor”. In particular, this learning task amounts to finding an association (or predictor) between an image and the fact if the image shows a tractor or not. To teach this association to a child, we show it many pictures and tell for each picture if there is a “tractor” or if there is “no tractor” depicted.

Consider that we have taught the child using the image collection $\mathbb{X}^{(\text{train})}$ depicted in Figure 6.1. For some reason, one of the images is labeled erroneously as “tractor” but actually shows an ocean wave. As a consequence, if the child is good in memorizing images, it might predict the presence of tractors whenever looking at a wave (Figure 6.2).



Figure 6.1: A (misleading) training dataset $\mathbb{X}^{(\text{train})} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m_t}$ consisting of $m_t = 9$ images. The i -th image is characterized by the feature vector $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and labeled with $y^{(i)} = 1$ (if image depicts a tractor) or with $y^{(i)} = -1$ (if image does not depict a tractor).

For the sake of the argument, we assume that the child uses a linear predictor $h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{x}^T \mathbf{w}$, using the features \mathbf{x} of the image, and encodes the fact of showing a tractor by $y = 1$ and if it is not showing a tractor by $y = -1$. In order to learn the weight vector, we use ERM with squared error loss over the training dataset, i.e., its learning process amounts to solving the ERM problem (4.3) using the labeled training dataset $\mathbb{X}^{(\text{train})}$.

If we stack the feature vectors $\mathbf{x}^{(i)}$ and labels $y^{(i)}$ into the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m_t)})^T$ and label vector $\mathbf{y} = (y^{(1)}, \dots, y^{(m_t)})^T$, the optimal linear predictor is obtained for the weight



Figure 6.2: The child, who has been taught the concept “tractor” using the image collection $\mathbb{X}^{(\text{train})}$ in Figure 6.1, might “see” a lot of tractors during the next beach holiday.

vector solving (4.8) and the associated training error is given by (4.9), which we repeat here for convenience:

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

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

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

of the feature matrix

$$\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m_t)})^T \in \mathbb{R}^{m_t \times n}. \quad (6.2)$$

Let us now study how overfitting arises whenever the sample size m is smaller or equal to the length n of the feature vectors \mathbf{x} , i.e., whenever

$$m_t \leq n. \quad (6.3)$$

In practice, a set of m_t feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ is typically linearly independent whenever (6.3) is satisfied. In this case, the span of the transposed feature matrix (6.2) coincides with \mathbb{R}^m which implies, in turn, $\mathbf{P} = \mathbf{I}$. Inserting $\mathbf{P} = \mathbf{I}$ into (4.9) yields

$$\mathcal{E}(h^{(\mathbf{w}_{\text{opt}})} \mid \mathbb{X}^{(\text{train})}) = 0. \quad (6.4)$$

To sum up: as soon as the number of training examples $m_t = |\mathbb{X}_{\text{train}}|$ is smaller than the size n of the feature vector \mathbf{x} , there is a linear predictor $h^{(\mathbf{w}_{\text{opt}})}$ achieving **zero empirical risk** (see (6.4)) on the training data.

While this “optimal” predictor $h^{(\mathbf{w}_{\text{opt}})}$ is perfectly accurate on the training data (the training error is zero!), it will typically incur a non-zero average prediction error $y - h^{(\mathbf{w}_{\text{opt}})}(\mathbf{x})$ on new data points (\mathbf{x}, y) (which are different from the training data). Indeed, using a simple toy model for the data generation, we obtained the expression (6.26) for the average prediction error. This average prediction error is lower bounded by the noise variance σ^2 which might be very large even the training error is zero. Thus, in case of overfitting, a small training error can be highly misleading regarding the average prediction error of a predictor.

A simple, yet quite useful, strategy to detect if a predictor \hat{h} overfits the training dataset $\mathbb{X}^{(\text{train})}$, is to compare the resulting training error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{train})})$ (see (6.6)) with the validation error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{val})})$ (see (6.7)). The validation error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{val})})$ is the empirical risk of the predictor \hat{h} on the validation dataset $\mathbb{X}^{(\text{val})}$. If overfitting occurs, the validation error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{val})})$ is significantly larger than the training error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{train})})$. The occurrence of overfitting for polynomial regression with degree n (see Section 3.2) chosen too large is depicted in Figure 7.1.

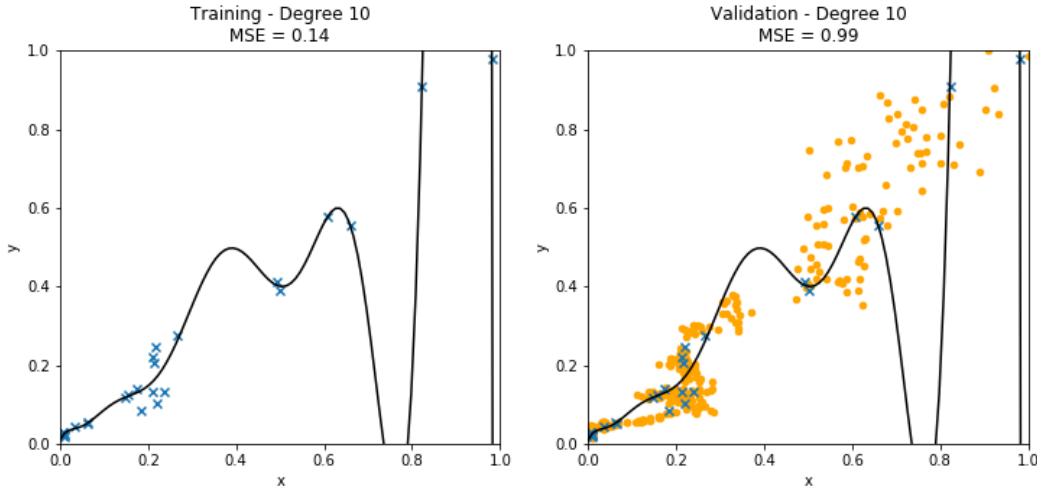


Figure 6.3: The training dataset consists of the blue crosses and can be almost perfectly fit by a high-degree polynomial. This high-degree polynomial gives only poor results for a different (validation) dataset indicated by the orange dots.

6.2 Validation

Consider an ML problem using a hypothesis space \mathcal{H} . We have chosen a particular predictor $\hat{h} \in \mathcal{H}$ by ERM (4.1) using a labeled dataset (the training set). The basic idea of validating the predictor \hat{h} is simple: compute the empirical risk of \hat{h} over another set of data points (\mathbf{x}, y) which have not been already used for training. It is very important to validate the predictor \hat{h} using labeled data points which do not belong to the dataset which has been used to learn \hat{h} (e.g., via ERM (4.1)). This is reasonable since the predictor \hat{h} tends to “look better” on the training set than for other data points, since it is optimized precisely for the data points in the training set.

A golden rule of ML practice: try always to use different data points for the training (see (4.1)) and the validation of a predictor \hat{h} !

A very simple recipe for implementing learning and validation of a predictor based on one single labeled dataset $\mathbb{X} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ is as follows (see Figure 6.4):

1. randomly divide (“split”) the entire dataset \mathbb{X} of labeled snapshots into two disjoint subsets $\mathbb{X}^{(\text{train})}$ (the “training set”) and $\mathbb{X}^{(\text{val})}$ (the “validation set”): $\mathbb{X} = \mathbb{X}^{(\text{train})} \cup \mathbb{X}^{(\text{val})}$ (see Figure 6.4).
2. learn a predictor \hat{h} via ERM using the training data $\mathbb{X}^{(\text{train})}$, i.e., compute (cf. (4.1))

$$\begin{aligned} \hat{h} &= \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{E}(h | \mathbb{X}^{(\text{train})}) \\ &= \operatorname{argmin}_{h \in \mathcal{H}} (1/m_t) \sum_{(\mathbf{x}, y) \in \mathbb{X}^{(\text{train})}} \mathcal{L}((\mathbf{x}, y), h) \end{aligned} \quad (6.5)$$

with corresponding **training error**

$$\mathcal{E}(\hat{h} | \mathbb{X}^{(\text{train})}) = (1/m_t) \sum_{i=1}^{m_t} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}) \quad (6.6)$$

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

$$\mathcal{E}(\hat{h} | \mathbb{X}^{(\text{val})}) = (1/m_v) \sum_{(\mathbf{x}, y) \in \mathbb{X}^{(\text{val})}} \mathcal{L}((\mathbf{x}, y), \hat{h}) \quad (6.7)$$

obtained when applying the predictor \hat{h} to the **validation dataset** $\mathbb{X}^{(\text{val})}$. We might refer to $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{val})})$ as the **validation error**.

The choice of the split ratio $|\mathbb{X}^{(\text{val})}|/|\mathbb{X}^{(\text{train})}|$, i.e., how large should the training set be relative to the validation set is often based on experimental tuning. It seems difficult to make a precise statement on how to choose the split ratio which applies broadly to different ML problems [?].

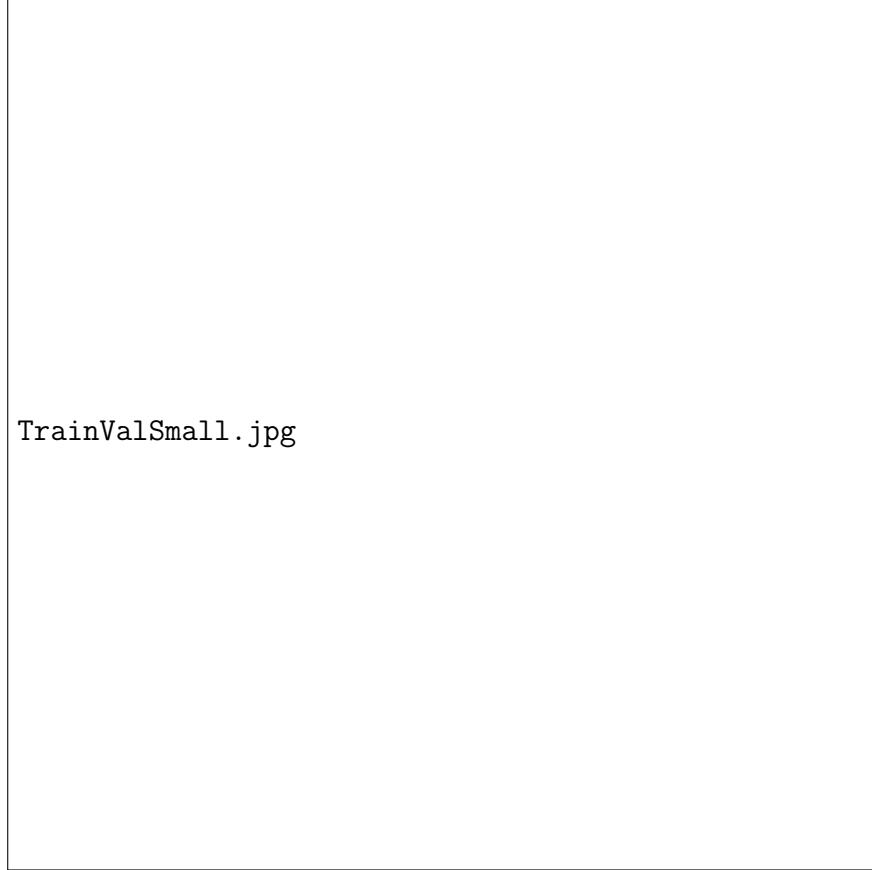


Figure 6.4: If we have only one single labeled dataset \mathbb{X} , we split it into a **training set** $\mathbb{X}^{(\text{train})}$ and a **validation set** $\mathbb{X}^{(\text{val})}$. We use the training set in order to learn (find) a good predictor $\hat{h}(\mathbf{x})$ by minimizing the empirical risk $\mathcal{E}(h|\mathbb{X}^{(\text{train})})$ (see (4.1)). In order to validate the performance of the predictor \hat{h} on new data, we compute the empirical risk $\mathcal{E}(h|\mathbb{X}^{(\text{val})})$ incurred by $\hat{h}(\mathbf{x})$ for the validation set $\mathbb{X}^{(\text{val})}$. We refer to the empirical risk $\mathcal{E}(h|\mathbb{X}^{(\text{val})})$ obtained for the validation set as the **validation error**.

The basic idea of randomly splitting the available labeled data into training and validation sets is underlying many validation techniques. A popular extension of the above approach, which is known as k -fold cross-validation, is based on repeating the splitting into training

and validation sets k times. During each repetition, this method uses different subsets for training and validation. We refer to [?, Sec. 7.10] for a detailed discussion of k -fold cross-validation.

6.3 Model Selection

We will now discuss how to use the validation principle of Section 6.2 to perform model selection. As discussed in Chapter 2, the choice of the hypothesis space from which we select a predictor map (e.g., via solving the ERM (4.1)) is a design choice. However, it is often not obvious what a good first choice for the hypothesis space is. Rather, we often try out different choices $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_M$ for the hypothesis space.

Consider a prediction problem where the relation between feature x and y is non-linear (see, e.g., Figure ??). We might then use polynomial regression (see Section 3.2) using the hypothesis space $\mathcal{H}_{\text{poly}}^{(n)}$ with some maximum degree n . For each value of the maximum degree n we get a different hypothesis space: $\mathcal{H}_1 = \mathcal{H}_{\text{poly}}^{(0)}, \mathcal{H}_2 = \mathcal{H}_{\text{poly}}^{(1)}, \dots, \mathcal{H}_M = \mathcal{H}_{\text{poly}}^{(M-1)}$. Or, instead of polynomial regression, we might use Gaussian basis regression (see Section 3.4), with different choices for the variance σ and shifts μ of the Gaussian basis function (3.11), e.g., $\mathcal{H}_1 = \mathcal{H}_{\text{Gauss}}^{(2)}$ with $\sigma = 1$ and $\mu_1 = 1$ and $\mu_2 = 2$, $\mathcal{H}_2 = \mathcal{H}_{\text{Gauss}}^{(2)}$ with $\sigma = 1/10$, $\mu_1 = 10$, $\mu_2 = 20$.

A principled approach for choosing a hypothesis space out of a given a set of candidates $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_M$ is as follows:

- randomly divide (split) the entire dataset \mathbb{X} of labeled snapshots into two disjoint subsets $\mathbb{X}^{(\text{train})}$ (the “training set”) and $\mathbb{X}^{(\text{val})}$ (the ”validation set”): $\mathbb{X} = \mathbb{X}^{(\text{train})} \cup \mathbb{X}^{(\text{val})}$ (see Figure 6.4).
- for each hypothesis space \mathcal{H}_l learn predictor $\hat{h}_l \in \mathcal{H}_l$ via ERM (4.1) using training data $\mathbb{X}^{(\text{train})}$.

$$\begin{aligned} \hat{h}_l &= \underset{h \in \mathcal{H}_l}{\operatorname{argmin}} \mathcal{E}(h | \mathbb{X}^{(\text{train})}) \\ &= \underset{h \in \mathcal{H}_l}{\operatorname{argmin}} (1/m_t) \sum_{i=1}^{m_t} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), h) \end{aligned} \tag{6.8}$$

- compute the validation error of \hat{h}_l

$$\mathcal{E}(\hat{h}_l | \mathbb{X}^{(\text{val})}) = (1/m_v) \sum_{i=1}^{m_v} \mathcal{L}((\mathbf{x}^{(i)}, y^{(i)}), \hat{h}_l) \quad (6.9)$$

obtained when applying the predictor \hat{h}_l to the **validation dataset** $\mathbb{X}^{(\text{val})}$.

- pick the hypothesis space \mathcal{H}_l resulting in the smallest validation error $\mathcal{E}(\hat{h}_l | \mathbb{X}^{(\text{val})})$

6.4 Bias, Variance and Generalization within Linear Regression

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

A core problem or challenge within ML is the verification (or validation) if a predictor or classifier which works well on a labeled training dataset will also work well (generalize) to new data points. In practice we can only validate by using different data points than for training an ML method via ERM. However, if we can find some generative probabilistic model which well explains the observed data points $\mathbf{z}^{(i)}$ we can study the generalization ability via probability theory.

In order to study generalization within a linear regression problem (see Section 3.1), we will invoke a **probabilistic toy model** for the data arising in an ML application. In particular, we assume that any observed data point $\mathbf{z} = (\mathbf{x}, y)$ with features $\mathbf{x} \in \mathbb{R}^n$ and label $y \in \mathbb{R}$ can be considered as an i.i.d. realization of a Gaussian random vector. The feature vector \mathbf{x} is assumed to have zero mean and covariance being the identity matrix, i.e., $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The label y of a data point is related to its features \mathbf{x} via a **linear Gaussian model**

$$y = \mathbf{w}_{\text{true}}^T \mathbf{x} + \varepsilon, \text{ with noise } \varepsilon \sim \mathcal{N}(0, \sigma^2). \quad (6.10)$$

The noise variance σ^2 is assumed fixed (non-random) and known. Note that the error component ε in (6.10) is intrinsic to the data (within our toy model) and cannot be overcome by any ML method. We highlight that this model for the observed data points might not be accurate for a particular ML application. However, this toy model will allow us to study some fundamental behaviour of ML methods.

In order to predict the label y from the features \mathbf{x} we will use predictors h that are linear

maps of the first r features x_1, \dots, x_r . This results in the hypothesis space

$$\mathcal{H}^{(r)} = \{h^{(\mathbf{w})}(\mathbf{x}) = (\mathbf{w}^T, \mathbf{0})\mathbf{x} \text{ with } \mathbf{w} \in \mathbb{R}^r\}. \quad (6.11)$$

The design parameter r determines the size of the hypothesis space $\mathcal{H}^{(r)}$ and allows to control the computational complexity of the resulting ML method which is based on the hypothesis space $\mathcal{H}^{(r)}$. 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 $\mathcal{E}(h^{(\mathbf{w})} \mid \mathbb{X}^{(\text{train})})$ incurred over a labeled training set $\mathbb{X}^{(\text{train})} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m_t}$. Within our toy model (see (6.10), (6.12) and (6.13)), the training data points $(\mathbf{x}^{(i)}, y^{(i)})$ are i.i.d. copies of the data point $\mathbf{z} = (\mathbf{x}, y)$. In particular, each of the data points in the training dataset is statistically independent from any other data point (\mathbf{x}, y) (which has not been used for training). However, the training data points $(\mathbf{x}^{(i)}, y^{(i)})$ and any other (new) data point (\mathbf{x}, y) share the same probability distribution (a multivariate normal distribution):

$$\mathbf{x}, \mathbf{x}^{(i)} \text{ i.i.d. with } \mathbf{x}, \mathbf{x}^{(i)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (6.12)$$

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

$$y^{(i)} = \mathbf{w}_{\text{true}}^T \mathbf{x}^{(i)} + \varepsilon^{(i)}, \text{ and } y = \mathbf{w}_{\text{true}}^T \mathbf{x} + \varepsilon \quad (6.13)$$

with i.i.d. noise $\varepsilon, \varepsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$.

As discussed in Chapter 4, the training error $\mathcal{E}(h^{(\mathbf{w})} \mid \mathbb{X}^{(\text{train})})$ is minimized by the predictor $h^{(\hat{\mathbf{w}})}(\mathbf{x}) = \hat{\mathbf{w}}^T \mathbf{I}_{r \times n} \mathbf{x}$, with weight vector

$$\hat{\mathbf{w}} = (\mathbf{X}_r^T \mathbf{X}_r)^{-1} \mathbf{X}_r^T \mathbf{y} \quad (6.14)$$

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

$$\begin{aligned} \mathbf{X}_r &= (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m_t)})^T \mathbf{I}_{n \times r} \in \mathbb{R}^{m_t \times r}, \text{ and} \\ \mathbf{y} &= (y^{(1)}, \dots, y^{(m_t)})^T \in \mathbb{R}^{m_t}. \end{aligned} \quad (6.15)$$

It will be convenient to tolerate a slight abuse of notation and denote by $\hat{\mathbf{w}}$ both, the length- r

vector (6.14) as well as the zero padded length- n vector $(\hat{\mathbf{w}}^T, \mathbf{0})^T$. This allows us to write

$$h^{(\hat{\mathbf{w}})}(\mathbf{x}) = \hat{\mathbf{w}}^T \mathbf{x}. \quad (6.16)$$

We highlight that the formula (6.14) for the optimal weight vector $\hat{\mathbf{w}}$ is only valid if the matrix $\mathbf{X}_r^T \mathbf{X}_r$ is invertible. However, it can be shown that within our toy model (see (6.12)), this is true with probability one whenever $m_t \geq r$. In what follows, we will consider the case of having more training samples than the dimension of the hypothesis space, i.e., $m_t > r$ such that the formula (6.14) is valid (with probability one). The case $m_t \leq r$ will be studied in Chapter 7.

The optimal weight vector $\hat{\mathbf{w}}$ (see (6.14)) depends on the training data $\mathbb{X}^{(\text{train})}$ via the feature matrix \mathbf{X}_r and label vector \mathbf{y} (see (6.15)). Therefore, since we model the training data as random, the weight vector $\hat{\mathbf{w}}$ (6.14) is a random quantity. For each different realization of the training dataset, we obtain a different realization of the optimal weight $\hat{\mathbf{w}}$.

Within our toy model, which relates the features \mathbf{x} of a data point to its label y via (6.10), the best case would be if $\hat{\mathbf{w}} = \mathbf{w}_{\text{true}}$. However, in general this will not happen since we have to compute $\hat{\mathbf{w}}$ based on the features $\mathbf{x}^{(i)}$ and noisy labels $y^{(i)}$ of the data points in the training dataset \mathbb{X} . Thus, we typically have to face a non-zero **estimation error**

$$\Delta \mathbf{w} := \hat{\mathbf{w}} - \mathbf{w}_{\text{true}}. \quad (6.17)$$

Note that this estimation error is a random quantity since the learnt weight vector $\hat{\mathbf{w}}$ (see (6.14)) is random.

Bias and Variance. As we will see below, the prediction quality achieved by $h^{(\hat{\mathbf{w}})}$ depends crucially on the **mean squared estimation error (MSE)**

$$\mathcal{E}_{\text{est}} := \mathbb{E}\{\|\Delta \mathbf{w}\|_2^2\} = \mathbb{E}\{\|\hat{\mathbf{w}} - \mathbf{w}_{\text{true}}\|_2^2\}. \quad (6.18)$$

It is useful to characterize the MSE \mathcal{E}_{est} by decomposing it into two components, one component (the “bias”) which depends on the choice r for the hypothesis space and another component (the “variance”) which only depends on the distribution of the observed feature vectors $\mathbf{x}^{(i)}$ and labels $y^{(i)}$. It is then not too hard to show that

$$\mathcal{E}_{\text{est}} = \underbrace{\|\mathbf{w}_{\text{true}} - \mathbb{E}\{\hat{\mathbf{w}}\}\|_2^2}_{\text{“bias” } B^2} + \underbrace{\mathbb{E}\|\hat{\mathbf{w}} - \mathbb{E}\{\hat{\mathbf{w}}\}\|_2^2}_{\text{“variance” } V} \quad (6.19)$$

The bias term in (6.19), which can be computed as

$$B^2 = \|\mathbf{w}_{\text{true}} - \mathbb{E}\{\hat{\mathbf{w}}\}\|_2^2 = \sum_{l=r+1}^n w_{\text{true},l}^2, \quad (6.20)$$

measures the distance between the “true predictor” $h^{(\mathbf{w}_{\text{true}})}(\mathbf{x}) = \mathbf{w}_{\text{true}}^T \mathbf{x}$ and the hypothesis space $\mathcal{H}^{(r)}$ (see (6.11)) of the linear regression problem. The bias is zero if $w_{\text{true},l} = 0$ for any index $l = r + 1, \dots, n$, or equivalently if $h^{(\mathbf{w}_{\text{true}})} \in \mathcal{H}^{(r)}$. We can guarantee $h^{(\mathbf{w}_{\text{true}})} \in \mathcal{H}^{(r)}$ only if we use the largest possible hypothesis space $\mathcal{H}^{(r)}$ with $r = n$. For $r < n$, we cannot guarantee a zero bias term since we have no access to the true underlying weight vector \mathbf{w}_{true} in (6.10). In general, the bias term decreases for increasing model size r (see Figure 6.5). We also highlight that the bias term does not depend on the variance σ^2 of the noise ε in our toy model (6.10).

Let us now consider the variance term in (6.19). Using the properties of our toy model (see (6.10), (6.12) and (6.13))

$$V = \mathbb{E}\{\|\hat{\mathbf{w}} - \mathbb{E}\{\hat{\mathbf{w}}\}\|_2^2\} = \sigma^2 \text{trace}\{\mathbb{E}\{(\mathbf{X}_r^T \mathbf{X}_r)^{-1}\}\}. \quad (6.21)$$

By (6.12), the matrix $(\mathbf{X}_r^T \mathbf{X}_r)^{-1}$ is random and distributed according to an **inverse Wishart distribution** [?]. In particular, for $m_t > r + 1$, its expectation is obtained as

$$\mathbb{E}\{(\mathbf{X}_r^T \mathbf{X}_r)^{-1}\} = 1/(m_t - r - 1) \mathbf{I}_{r \times r}. \quad (6.22)$$

By inserting (6.22) and $\text{trace}\{\mathbf{I}_{r \times r}\} = r$ into (6.21),

$$V = \mathbb{E}\{\|\hat{\mathbf{w}} - \mathbb{E}\{\hat{\mathbf{w}}\}\|_2^2\} = \sigma^2 r / (m_t - r - 1). \quad (6.23)$$

As indicated by (6.23), the variance term increases with increasing model complexity r (see Figure 6.5). 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 tradeoff**. Thus, the choice of model complexity r (see (6.11)) has to balance between small variance and small bias term.

Generalization. In most ML applications, we are primarily interested in how well a predictor $h^{(\hat{\mathbf{w}})}$, which has been learnt from some training data \mathbb{X} (see (4.1)), predicts the label y of a new datapoint (which is not contained in the training data \mathbb{X}) with features \mathbf{x} . Within our linear regression model, the prediction (approximation guess or estimate) \hat{y} of

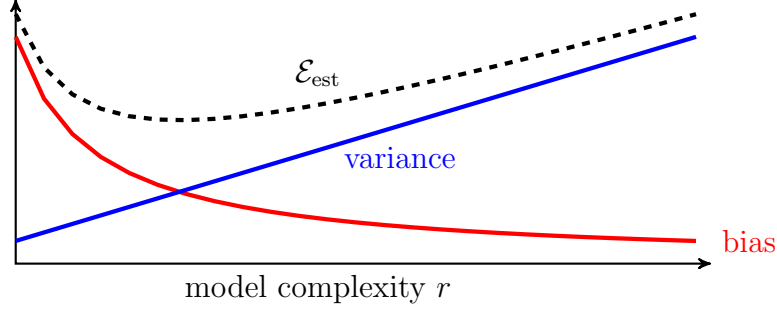


Figure 6.5: The estimation error \mathcal{E}_{est} incurred by linear regression can be decomposed into a bias term B^2 and a variance term V (see (6.19)). These two components depend on the model complexity r in an opposite manner resulting in a bias-variance tradeoff.

the label y is obtained using the learnt predictor $h^{(\hat{\mathbf{w}})}$ via

$$\hat{y} = \hat{\mathbf{w}}^T \mathbf{x}. \quad (6.24)$$

Note that the prediction \hat{y} is a random variable since (i) the feature vector \mathbf{x} is modelled as a random vector (see (6.12)) and (ii) the optimal weight vector $\hat{\mathbf{w}}$ (see (6.14)) is random. In general, we cannot hope for a perfect prediction but have to face a non-zero prediction error

$$\begin{aligned} e_{\text{pred}} &:= \hat{y} - y \\ &\stackrel{(6.24)}{=} \hat{\mathbf{w}}^T \mathbf{x} - y \\ &\stackrel{(6.10)}{=} \hat{\mathbf{w}}^T \mathbf{x} - (\mathbf{w}_{\text{true}}^T \mathbf{x} + \varepsilon) \\ &= \Delta \mathbf{w}^T \mathbf{x} - \varepsilon. \end{aligned} \quad (6.25)$$

Note that, within our toy model (see (6.10), (6.12) and (6.13)), the prediction error e_{pred} is a random variable since (i) the label y is modelled as a random variable (see (6.10)) and (ii) the prediction \hat{y} is random.

Since, within our toy model (6.13), ε is zero-mean and independent of \mathbf{x} and $\hat{\mathbf{w}} - \mathbf{w}_{\text{true}}$,

we obtain the **average predictor error** as

$$\begin{aligned}
\mathcal{E}_{\text{pred}} &= \mathbb{E}\{e_{\text{pred}}^2\} \\
&\stackrel{(6.25),(6.10)}{=} \mathbb{E}\{\Delta \mathbf{w}^T \mathbf{x} \mathbf{x}^T \Delta \mathbf{w}\} + \sigma^2 \\
&\stackrel{(a)}{=} \mathbb{E}\{\mathbb{E}\{\Delta \mathbf{w}^T \mathbf{x} \mathbf{x}^T \Delta \mathbf{w} \mid \mathbb{X}\}\} + \sigma^2 \\
&\stackrel{(b)}{=} \mathbb{E}\{\Delta \mathbf{w}^T \Delta \mathbf{w}\} + \sigma^2 \\
&\stackrel{(6.17),(6.18)}{=} \mathcal{E}_{\text{est}} + \sigma^2 \\
&\stackrel{(6.19)}{=} B^2 + V + \sigma^2.
\end{aligned} \tag{6.26}$$

Here, step (a) is due to the law of total expectation [?] and step (b) uses that, conditioned on the dataset \mathbb{X} , the feature vector \mathbf{x} of a new data point (not belonging to \mathbb{X}) has zero mean and covariance matrix \mathbf{I} (see (6.12)).

Thus, as indicated by (6.26), the average (expected) prediction error $\mathcal{E}_{\text{pred}}$ is the sum of three contributions: (i) the bias B^2 , (ii) the variance V and (iii) the noise variance σ^2 . The bias and variance, whose sum is the estimation error \mathcal{E}_{est} , can be influenced by varying the model complexity r (see Figure 6.5) which is a design parameter. The noise variance σ^2 is the intrinsic accuracy limit of our toy model (6.10) and is not under the control of the ML engineer. It is impossible for any ML method (no matter how clever it is engineered) to achieve, on average, a small prediction error than the noise variance σ^2 .

We finally highlight that our analysis of bias (6.20), variance (6.23) and the average prediction error (6.26) achieved by linear regression only applies if the observed data points are well modelled as realizations of random vectors according to (6.10), (6.12) and (6.13). The usefulness of this model for the data arising in a particular application has to be verified in practice by some validation techniques [? ?].

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 data points by some random number generator [?]. Using these i.i.d. copies, we can replace exact computations (expectations) by empirical approximations (sample averages).

6.5 Diagnosis

Consider a predictor \hat{h} obtained from ERM (4.1) with training error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{train})})$ and validation error $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{val})})$. By comparing the two numbers $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{train})})$ and $\mathcal{E}(\hat{h}|\mathbb{X}^{(\text{val})})$ with some

desired or tolerated error E_0 , we can get some idea of how to adapt the current ERM approach (see (4.1)) to improve performance:

- $\mathcal{E}(h|\mathbb{X}^{(\text{train})}) \approx \mathcal{E}(h|\mathbb{X}^{(\text{val})}) \approx E_0$: There is not much to improve regarding prediction accuracy since we achieve the desired error on both training and validation set.
- $\mathcal{E}(h|\mathbb{X}^{(\text{val})}) \gg \mathcal{E}(h|\mathbb{X}^{(\text{train})}) \approx E_0$: The ERM (4.1) results in a hypothesis \hat{h} with sufficiently small training error but when applied to new (validation) data the performance of \hat{h} is significantly worse. This is an indicator of overfitting which can be addressed by regularization techniques (see Section 7.4).
- $\mathcal{E}(h|\mathbb{X}^{(\text{train})}) \gg \mathcal{E}(h|\mathbb{X}^{(\text{val})})$: This indicates that the method for solving the ERM (4.1) is not working properly. The training error obtained by solving the ERM (4.1) should always be smaller than the validation error. When using GD for solving ERM, one particular reason for $\mathcal{E}(h|\mathbb{X}^{(\text{train})}) \gg \mathcal{E}(h|\mathbb{X}^{(\text{val})})$ could be that the step size α in the GD step (5.4) is chosen too large (see Figure 5.3-(b)).

6.6 Exercises

6.6.1 Validation Set Size

Consider a linear regression problem with data points characterized by a scalar feature and numeric label. Assume data points are i.i.d. Gaussian with zero-mean and covariance \mathbf{C} . How many data points do we need for a validation set such that the probability that the MSE incurred on the validation does not deviate by more than 20 percent from the average MSE is larger than 0.8.

Chapter 7

Regularization

A main reason for validating predictors is to detect **overfitting**. The phenomenon of overfitting is one of the key obstacles for the successful application of ML methods. In case of overfitting, the ERM approach can be highly misleading.

The ERM principle only makes sense if the empirical risk (training error), (see (??)) incurred by a predictor when applied to some labeled data points (training data) $\mathbb{X} = \{\mathbf{z}^{(i)} = (\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, is a good indicator for the average prediction error (see (6.26)) incurred by that predictor on new data points which are different from the training data.

One main pitfall for ERM is the phenomenon of overfitting. A predictor $h : \mathbb{R}^n \rightarrow \mathbb{R}$ obtained by the ERM is said to **overfit** the training set if it has a **small training error** but a **large average prediction error** on other data points outside the training set.

A main cause for overfitting is that the hypothesis space \mathcal{H} is chosen too large. If the hypothesis space is too large, ML methods based on solving the ERM (4.1) can choose from so many different maps $h \in \mathcal{H}$ (from features \mathbf{x} to label y) that just “by luck” it will find a good one for a given training dataset. However, the resulting small empirical risk on the training dataset is highly misleading since if a predictor was good for the training dataset just “by accident”, we can not expect hat it will be any good for other data points.

Section 7.2 discusses the relation between the tendency of a method to overfit training data and its robustness. A ML method is robust if it tolerates small perturbations (errors) in the training data. Intuitively, forcing a method to tolerate small perturbations in the training error should counteract the tendency of the method to overfit the training data.

7.1 Regularized ERM

It seems reasonable to avoid overfitting by pruning the hypothesis space \mathcal{H} , i.e., removing some of its elements. In particular, instead of solving (4.1) we solve the restricted ERM

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

Another approach to avoid overfitting is to regularize the ERM (4.1) by adding a penalty term $\mathcal{R}(h)$ which somehow measures the complexity or non-regularity of a predictor map h using a non-negative number $\mathcal{R}(h) \in \mathbb{R}_+$. We then obtain the regularized ERM

$$\hat{h} = \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{E}(h|\mathbb{X}) + \mathcal{R}(h). \quad (7.2)$$

The additional term $\mathcal{R}(h)$ aims at approximating (or anticipating) the increase in the empirical risk of a predictor \hat{h} when it is applied to new data points, which are different from the dataset \mathbb{X} used to learn the predictor \hat{h} by (7.2).

The two approaches (7.1) and (7.2), for making ERM (4.1) robust against overfitting are closely related. In particular, these two approaches are, in a certain sense, **dual** to each other: for a given restriction $\mathcal{H}' \subset \mathcal{H}$ we can find a penalty $\mathcal{R}(h)$ term such that the solutions of (7.1) and (7.2) coincide. Similarly for a many popular types of penalty terms $\mathcal{R}(h)$, we can find a restriction $\mathcal{H}' \subset \mathcal{H}$ such that the solutions of (7.1) and (7.2) coincide. This statements can be made precise using the theory of duality for optimization problems (see [?]).

In what follows we will analyze the occurrence of overfitting in Section ?? and then discuss in Section 7.4 how to avoid overfitting using regularization.

7.2 Robustness

Overfitting is one of main challenges in applying modern ML methods. Modern ML methods use large hypothesis spaces that allow to represent highly non-linear predictor maps. Just by pure luck we can find one such predictor map that perfectly fits the training set resulting in zero training error and, in turn, solving ERM (4.1).

Overfitting is closely related to another property of ML methods: robustness. If a method overfits it will typically be not robust to small perturbations in the training data. The robustness to small perturbations in the data is almost a mandatory requirement for ML

methods to be useful in important application domains.

The ML methods discussed in Chapter 4 rest on the idealizing assumption that we have access to the true label values and feature values of a set of data points (the training set). However, the means by which the label and feature values are determined are prone to errors. These errors might stem from the measurement device itself (hardware failures) or might be due to modelling errors. We need ML methods that do not “break” if we feed it slightly perturbed label values for the training data.

7.3 Data Augmentation

implement robustness principle by augmenting dataset with random perturbations of original training data.

7.4 Regularized Linear Regression

As mentioned above, the overfitting of the training data $\mathbb{X}^{(\text{train})} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m_t}$ might be caused by choosing the hypothesis space too large. Therefore, we can avoid overfitting by making (pruning) the hypothesis space \mathcal{H} smaller to obtain a new hypothesis space $\mathcal{H}_{\text{small}}$. This smaller hypothesis space $\mathcal{H}_{\text{small}}$ can be obtained by pruning, i.e., removing certain maps h , from \mathcal{H} .

A more general strategy is **regularization**, which amounts to modifying the loss function of an ML problem in order to favour a subset of predictor maps. Pruning the hypothesis space can be interpreted as an extreme case of regularization, where the loss functions become infinite for predictors which do not belong to the smaller hypothesis space $\mathcal{H}_{\text{small}}$.

In order to avoid overfitting, we have to augment our basic ERM approach (cf. (4.1)) by **regularization techniques**. According to [?], regularization aims at “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.” By generalization error, we mean the average prediction error (see (6.26)) incurred by a predictor when applied to new data points (different from the training set).

A simple but effective method to regularize the ERM learning principle, is to augment the empirical risk (5.7) of linear regression by the penalty term $\mathcal{R}(h^{(\mathbf{w})}) := \lambda \|\mathbf{w}\|_2^2$, which


The image area is mostly blank, with the text 'RobustnessOverfitting.jpg' located in the lower-left quadrant.

Figure 7.1: Modern ML methods allow to find a predictor map that perfectly fits training data. Such a predictor might perform poorly on a new data point outside the training set. To prevent learning such a predictor map we could require it to be robust against small perturbations in the features of the training data points or the predictor map itself.

penalizes overly large weight vectors \mathbf{w} . Thus, we arrive at **regularized ERM**

$$\begin{aligned}\hat{\mathbf{w}}^{(\lambda)} &= \operatorname{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}} [\mathcal{E}(h^{(\mathbf{w})} | \mathbb{X}^{(\text{train})}) + \lambda \|\mathbf{w}\|^2] \\ &= \operatorname{argmin}_{h^{(\mathbf{w})} \in \mathcal{H}} \left[(1/m_t) \sum_{i=1}^{m_t} \mathcal{L}(\mathbf{x}^{(i)}, y^{(i)}, h^{(\mathbf{w})}) + \lambda \|\mathbf{w}\|^2 \right],\end{aligned}\tag{7.3}$$

with the regularization parameter $\lambda > 0$. The parameter λ trades a small training error $\mathcal{E}(h^{(\mathbf{w})} | \mathbb{X})$ against a small norm $\|\mathbf{w}\|$ of the weight vector. In particular, if we choose a large value for λ , then weight vectors \mathbf{w} with a large norm $\|\mathbf{w}\|$ are “penalized” by having a larger objective function and are therefore unlikely to be a solution (minimizer) of the optimization problem (7.3).

Specialising (7.3) to the squared error loss and linear predictors yields **regularized linear regression** (see (4.3)):

$$\hat{\mathbf{w}}^{(\lambda)} = \operatorname{argmin}_{\mathbf{w} \in \mathbb{R}^n} \left[(1/m_t) \sum_{i=1}^{m_t} (y^{(i)} - \mathbf{w}^T \mathbf{x}^{(i)})^2 + \lambda \|\mathbf{w}\|_2^2 \right],\tag{7.4}$$

The optimization problem (7.4) is also known under the name **ridge regression** [?].

Using the feature matrix $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m_t)})^T$ and label vector $\mathbf{y} = (y^{(1)}, \dots, y^{(m_t)})^T$, we can rewrite (7.4) more compactly as

$$\hat{\mathbf{w}}^{(\lambda)} = \operatorname{argmin}_{\mathbf{w} \in \mathbb{R}^n} \left[(1/m_t) \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 \right].\tag{7.5}$$

The solution of (7.5) is given by

$$\hat{\mathbf{w}}^{(\lambda)} = (1/m_t) ((1/m_t) \mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}.\tag{7.6}$$

This reduces to the closed-form expression (6.14) when $\lambda = 0$ in which case regularized linear regression reduces to ordinary linear regression (see (7.4) and (4.3)). It is important to note that for $\lambda > 0$, the formula (7.6) is always valid, even when $\mathbf{X}^T \mathbf{X}$ is singular (not invertible). This implies, in turn, that for $\lambda > 0$ the optimization problem (7.5) (and (7.4)) have a unique solution (which is given by (7.6)).

Let us now study the effect of regularization on the resulting bias, variance and average prediction error incurred by the predictor $h^{(\hat{\mathbf{w}}^{(\lambda)})}(\mathbf{x}) = (\hat{\mathbf{w}}^{(\lambda)})^T \mathbf{x}$. To this end, we will again invoke the simple probabilistic toy model (see (6.10), (6.12) and (6.13)) used already

in Section 6.4. In particular, we interpret the training data $\mathbb{X}^{(\text{train})} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m_t}$ as realizations of i.i.d. random variables according to (6.10), (6.12) and (6.13).

As discussed in Section 6.4, the average prediction error is the sum of three components: the bias, the variance and the noise variance σ^2 (see (6.26)). The bias of regularized linear regression (7.4) is obtained as

$$B^2 = \|(\mathbf{I} - \mathbb{E}\{(\mathbf{X}^T \mathbf{X} + m\lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{X}\}) \mathbf{w}_{\text{true}}\|_2^2. \quad (7.7)$$

For sufficiently large sample size m_t we can use the approximation

$$\mathbf{X}^T \mathbf{X} \approx m_t \mathbf{I} \quad (7.8)$$

such that (7.7) can be approximated as

$$\begin{aligned} B^2 &\approx \|(\mathbf{I} - (\mathbf{I} + \lambda \mathbf{I})^{-1}) \mathbf{w}_{\text{true}}\|_2^2 \\ &= \sum_{l=1}^n \frac{\lambda}{1 + \lambda} w_{\text{true},l}^2. \end{aligned} \quad (7.9)$$

By comparing the (approximate) bias term (7.9) of regularized linear regression with the bias term (6.20) obtained for ordinary linear regression, we see that introducing regularization typically increases the bias. The bias increases with larger values of the regularization parameter λ .

The variance of regularized linear regression (7.4) satisfies

$$\begin{aligned} V &= (\sigma^2/m_t^2) \times \\ &\text{trace} \mathbb{E}\{((1/m_t) \mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{X} ((1/m_t) \mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1}\}. \end{aligned} \quad (7.10)$$

Using the approximation (7.8), which is reasonable for sufficiently large sample size m_t , we can in turn approximate (7.10) as

$$V \approx \sigma^2 (n/m_t) (1/(1 + \lambda)). \quad (7.11)$$

According to (7.11), the variance of regularized linear regression decreases with increasing regularization λ . Thus, as illustrated in Figure 7.2, the choice of λ has to balance between the bias B^2 (7.9) (which increases with increasing λ) and the variance V (7.11) (which decreases with increasing λ). This is another instance of the bias-variance tradeoff (see Figure 6.5).

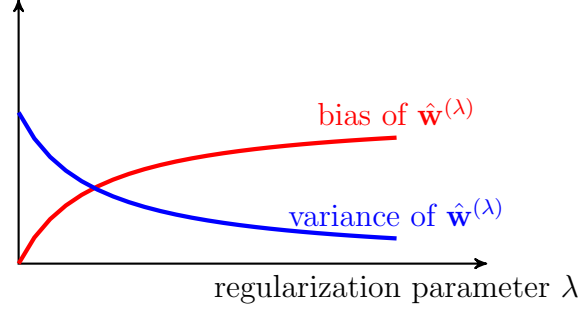


Figure 7.2: The bias and variance of regularized linear regression depend on the regularization parameter λ in an opposite manner resulting in a bias-variance tradeoff.

So far, we have only discussed the statistical effect of regularization on the resulting ML method (how regularization influences bias, variance, average prediction error). However, regularization has also an effect on the computational properties of the resulting ML method. Note that the objective function in (7.5) is a smooth (infinitely often differentiable) convex function. Thus, as for linear regression, we can solve the regularization linear regression problem using GD (2.5) (see Algorithm 4). The effect of adding the regularization term $\lambda \|\mathbf{w}\|_2^2$ to the objective function within linear regression is a **speed up of GD**. Indeed, we can rewrite (7.5) as the quadratic problem

$$\min_{\mathbf{w} \in \mathbb{R}^n} \underbrace{(1/2) \mathbf{w}^T \mathbf{Q} \mathbf{w} - \mathbf{q}^T \mathbf{w}}_{=f(\mathbf{w})}$$

$$\text{with } \mathbf{Q} = (1/m) \mathbf{X}^T \mathbf{X} + \lambda \mathbf{I}, \mathbf{q} = (1/m) \mathbf{X}^T \mathbf{y}. \quad (7.12)$$

This is similar to the quadratic problem (4.6) underlying linear regression but with different matrix \mathbf{Q} . It turns out that the convergence speed of GD (see (5.4)) applied to solving a quadratic problem of the form (7.12) depends crucially on the condition number $\kappa(\mathbf{Q}) \geq 1$ of the psd matrix \mathbf{Q} [?]. In particular, GD methods are fast if the condition number $\kappa(\mathbf{Q})$ is small (close to 1).

This condition number is given by $\frac{\lambda_{\max}((1/m) \mathbf{X}^T \mathbf{X})}{\lambda_{\min}((1/m) \mathbf{X}^T \mathbf{X})}$ for ordinary linear regression (see (4.6)) and given by $\frac{\lambda_{\max}((1/m) \mathbf{X}^T \mathbf{X}) + \lambda}{\lambda_{\min}((1/m) \mathbf{X}^T \mathbf{X}) + \lambda}$ for regularized linear regression (7.12). For increasing regularization parameter λ , the condition number obtained for regularized linear regression (7.12) tends to 1:

$$\lim_{\lambda \rightarrow \infty} \frac{\lambda_{\max}((1/m) \mathbf{X}^T \mathbf{X}) + \lambda}{\lambda_{\min}((1/m) \mathbf{X}^T \mathbf{X}) + \lambda} = 1. \quad (7.13)$$

Thus, according to (7.13), the GD implementation of regularized linear regression (see Algorithm 4) with a large value of the regularization parameter λ in (7.4) will converge faster compared to GD for linear regression (see Algorithm 1).

Algorithm 4 “Regularized Linear Regression via GD”

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

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

1: **repeat**

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

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

4: **until** convergence

Output: $\mathbf{w}^{(k)}$ (which approximates $\hat{\mathbf{w}}^{(\lambda)}$ in (7.5))

Let us finally point out a close relation between regularization (which amounts to adding the term $\lambda \|\mathbf{w}\|^2$ to the objective function in (7.3)) and model selection (see Section 6.3). The regularized ERM (7.3) can be shown (see [?, Ch. 5]) to be equivalent to

$$\hat{\mathbf{w}}^{(\lambda)} = \underset{h^{(\mathbf{w})} \in \mathcal{H}^{(\lambda)}}{\operatorname{argmin}} (1/m_t) \sum_{i=1}^{m_t} (y^{(i)} - h^{(\mathbf{w})}(\mathbf{x}^{(i)}))^2 \quad (7.14)$$

with the restricted hypothesis space

$$\begin{aligned} \mathcal{H}^{(\lambda)} &:= \{h^{(\mathbf{w})} : \mathbb{R}^n \rightarrow \mathbb{R} : h^{(\mathbf{w})}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \\ &\quad , \text{ with some } \mathbf{w} \text{ satisfying } \|\mathbf{w}\|^2 \leq C(\lambda)\} \subset \mathcal{H}^{(n)}. \end{aligned} \quad (7.15)$$

For any given value λ , we can find a bound $C(\lambda)$ such that solutions of (7.3) coincide with the solutions of (7.14). Thus, by solving the regularized ERM (7.3) we are performing implicitly model selection using a continuous ensemble of hypothesis spaces $\mathcal{H}^{(\lambda)}$ given by (7.15). In contrast, the simple model selection strategy considered in Section 6.3 uses a discrete sequence of hypothesis spaces.

7.5 Semi-Supervised Learning

Can we use unlabelled data points to construct better regularizers? We could use unlabeled data to learn some subspace of features that are most relevant ? (relation to feature learning ?)

7.6 Multitask Learning

Remember that a formal ML problem is specified by identifying data points, their features and labels, a model (hypothesis space) and loss function. Note that we can use the very same raw data, model and loss function and still define many different ML problems by using different choices for the label. Multitask learning aims at exploiting relations between similar ML problems or tasks.

Consider the ML problem (task) of predicting the confidence level of a hand-drawing showing an apple. To learn such a predictor we might have a collecting of hand-drawings at our disposal. We might now for each hand-drawing certain higher-level information such as the object it is showing. This allows us to use different choices for the label. We could also use the confidence level of a hand-drawing showing an orange. Clearly this problem is related to the problem of predicting the apple confidence.

The definition (design choice) of the labels corresponds to formulating a particular question we want to have answered by an ML method. Some questions (label choices) are more difficult to answer while others are easier to answer.

Consider the ML problem arising from guiding the operation of a mower robot. For a mowing robot, it is important to determine if it is currently on grassland or not. Let us assume the mower robot is equipped with an on-board camera which allows to take snapshots which are characterized by a feature vector \mathbf{x} (see Figure 2.3). We could then define the label as either $y = 1$ if the snapshot suggests that the mower is on grassland and $y = -1$ if not. However, we might be interested in a finer-grained information about the floor type and define the label as $y = 1$ for grassland, $y = 0$ for soil and $y = -1$ for when the mower is on tiles. The latter problem is more difficult since we have to distinguish between three different types of floor (“grass” vs. “soil” vs. “tiles”) whereas for the former problem we only have to distinguish between two types of floor (“grass” vs. “no grass”).

7.7 Exercises

7.7.1 Ridge Regression as Quadratic Form

Consider linear hypothesis space consisting of linear maps parameterized by weights \mathbf{w} . We try to find the best linear map by minimizing the regularized average squared error loss (empirical risk) incurred on some labeled training data points $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})$.

As regularizer we use $\|\mathbf{w}\|^2$, yielding the following learning problem

$$\min_{\mathbf{w}} f(\mathbf{w}) = \sum_{i=1}^m \dots + \|\mathbf{w}\|_2^2$$

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 ?

Chapter 8

Clustering

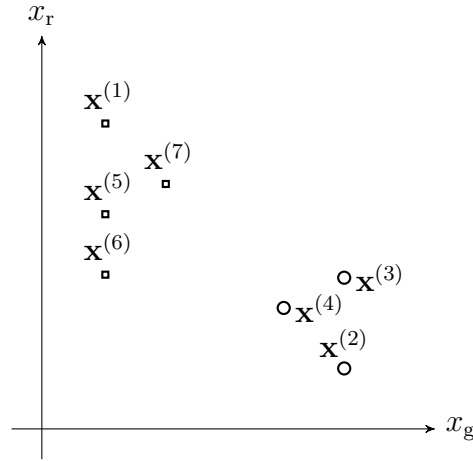


Figure 8.1: A scatterplot obtained from the features $\mathbf{x}^{(i)} = (x_r^{(i)}, x_g^{(i)})^T$, given by the redness $x_r^{(i)}$ and greenness $x_g^{(i)}$, of some snapshots.

Up to now, we mainly considered ML methods which required some labeled training data in order to learn a good predictor or classifier. We will now start to discuss ML methods which do not make use of labels. These methods are often referred to as “unsupervised” since they do not require a supervisor (or teacher) which provides the labels for data points in a training set.

An important class of unsupervised methods, known as clustering methods, aims at grouping data points into few subsets (or **clusters**). While there is no unique formal definition, we understand by cluster a subset of data points which are more similar to each other than to the remaining data points (belonging to different clusters). Different clustering

methods are obtained for different ways to measure the “similarity” between data points.

In what follows we assume that data points $\mathbf{z}^{(i)}$, for $i = 1, \dots, m$, are characterized by feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and measure similarity between data points using the Euclidean distance $\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|$.¹ Thus, we consider two data points $\mathbf{z}^{(i)}$ and $\mathbf{z}^{(j)}$ similar if $\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|$ is small. Moreover, we assume the number k of clusters prescribed.

Consider the scatter plot in Figure 8.1 which is obtained from the redness and greenness of several snapshots, which we would like to order into two groups: those snapshots which depict mostly grass and those which depict mostly something else than grass. We characterize a snapshot using the feature vector $\mathbf{x} = (x_r, x_g)^T \in \mathbb{R}^2$ and the label $y \in \{1, 2\}$ with $y = 1$ if the snapshot shows grass and $y = 2$ otherwise.

If we were able to group the snapshots based solely on their features $\mathbf{x}^{(i)}$ into $k = 2$ groups \mathcal{C}_1 (which correspond to snapshots of grass) and \mathcal{C}_2 (no grass), we would need to acquire the label of one data point only. Indeed, given a perfect clustering, if we know the label of one data point, we immediately know the label of all other data points in the same cluster. Moreover, we would also know the labels of the data points in the other cluster, since there are only two possible label values.

There are two main flavours of clustering methods:

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

Within hard clustering, each data point $\mathbf{x}^{(i)}$ belongs to one and only one cluster. In contrast, soft clustering methods assign a data point $\mathbf{x}^{(i)}$ to several different clusters with varying degree of belonging (confidence).

Clustering methods determine for each data point $\mathbf{z}^{(i)}$ a cluster assignment $y^{(i)}$. The cluster assignment $y^{(i)}$ encodes the cluster to which the data point $\mathbf{x}^{(i)}$ is assigned. For hard clustering with a prescribed number of k clusters, the cluster assignments $y^{(i)} \in \{1, \dots, k\}$ represent the index of the cluster to which $\mathbf{x}^{(i)}$ belongs.

In contrast, soft clustering methods allow each data point to belong to several different clusters. The degree with which data point $\mathbf{x}^{(i)}$ belongs to cluster $c \in \{1, \dots, k\}$ is represented by the degree of belonging $y_c^{(i)} \in [0, 1]$, which we stack into the vector $\mathbf{y}^{(i)} = (y_1^{(i)}, \dots, y_k^{(i)})^T \in [0, 1]^k$. Thus, while hard clustering generates non-overlapping clusters, the clusters produced by soft clustering methods may overlap.

¹With a slight abuse of notation, we will occasionally denote a data point $\mathbf{z}^{(i)}$ using its feature vector $\mathbf{x}^{(i)}$. In general, the feature vector is only a (incomplete) representation of a data point but it is customary in many unsupervised ML methods to identify a data point with its features.

We intentionally used the same symbol $y^{(i)}$ used to denote the cluster assignments of a data point $\mathbf{x}^{(i)}$ as we used to denote an associated label $y^{(i)}$ in earlier sections. There is a strong conceptual link between clustering and classification.

We can interpret clustering as an extreme case of classification without having access to any labeled training data, i.e., we do not know the label of any data point.

Thus, in order to have any chance to find the correct labels (cluster assignments) $y_c^{(i)}$ we have to rely solely on the intrinsic geometry of the data points which is given by the locations of the feature vectors $\mathbf{x}^{(i)}$.

8.1 Hard Clustering

A simple method for hard clustering is the “ k -means” algorithm which requires the number k of clusters to specified before-hand. The idea underlying k -means is quite simple: First, given a current guess for the cluster assignments $y^{(i)}$, determine the cluster means $\mathbf{m}^{(c)} = \frac{1}{|\{i: y^{(i)}=c\}|} \sum_{i: y^{(i)}=c} \mathbf{x}^{(i)}$ for each cluster. Then, in a second step, update the cluster assignments $y^{(i)} \in \{1, \dots, k\}$ for each data point $\mathbf{x}^{(i)}$ based on the nearest cluster mean. By iterating these two steps we obtain Algorithm 5.

Algorithm 5 “ k -means”

Input: dataset $\mathbb{X} = \{\mathbf{x}^{(i)}\}_{i=1}^m$; number k of clusters.

Initialize: choose initial cluster means $\mathbf{m}^{(c)}$ for $c = 1, \dots, k$.

1: **repeat**

2: for each data point $\mathbf{x}^{(i)}$, $i = 1, \dots, m$, do

$$y^{(i)} \in \underset{c' \in \{1, \dots, k\}}{\operatorname{argmin}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\| \text{ (update cluster assignments)} \quad (8.1)$$

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

$$\mathbf{m}^{(c)} = \frac{1}{|\{i : y^{(i)} = c\}|} \sum_{i: y^{(i)}=c} \mathbf{x}^{(i)} \text{ (update cluster means)} \quad (8.2)$$

4: **until** convergence

Output: cluster assignments $y^{(i)} \in \{1, \dots, k\}$

In (8.1) we denote by $\underset{c' \in \{1, \dots, k\}}{\operatorname{argmin}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\|$ the set of all cluster indices $c \in \{1, \dots, k\}$ such that $\|\mathbf{x}^{(i)} - \mathbf{m}^{(c)}\| = \min_{c' \in \{1, \dots, k\}} \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\|$.

The k -means algorithm requires the specification of initial choices for the cluster means $\mathbf{m}^{(c)}$, for $c = 1, \dots, k$. There is no unique optimal strategy for the initialization but several heuristic strategies can be used. One option is to initialize the cluster means with i.i.d. realizations of a random vector \mathbf{m} whose distribution is matched to the dataset $\mathbb{X} = \{\mathbf{x}^{(i)}\}_{i=1}^m$, e.g., $\mathbf{m} \sim \mathcal{N}(\hat{\mathbf{m}}, \hat{\mathbf{C}})$ with sample mean $\hat{\mathbf{m}} = (1/m) \sum_{i=1}^m \mathbf{x}^{(i)}$ and the sample covariance $\hat{\mathbf{C}} = (1/m) \sum_{i=1}^m (\mathbf{x}^{(i)} - \hat{\mathbf{m}})(\mathbf{x}^{(i)} - \hat{\mathbf{m}})^T$. Another option is to choose the cluster means $\mathbf{m}^{(c)}$ by randomly selecting k different data points $\mathbf{x}^{(i)}$. The cluster means might also be chosen by evenly partitioning the principal component of the dataset (see Chapter 9).

It can be shown that k -means implements a variant of empirical risk minimization. To

this end we define the empirical risk (or “clustering error”)

$$\mathcal{E}(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathbb{X}) = (1/m) \sum_{i=1}^m \left\| \mathbf{x}^{(i)} - \mathbf{m}^{(y^{(i)})} \right\|^2. \quad (8.3)$$

Note that the empirical risk (8.3) depends on the current guess for the cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^k$ and cluster assignments $\{y^{(i)}\}_{i=1}^m$.

Finding the global optimum of the function (8.3) over all possible cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^k$ and cluster assignments $\{y^{(i)}\}_{i=1}^m$ is difficult as the function is non-convex. However, minimizing (8.3) only with respect to the cluster assignments $\{y^{(i)}\}_{i=1}^m$ but with the cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^k$ held fixed is easy. Similarly, minimizing (8.3) over the choices of cluster means with the cluster assignments held fixed is also straightforward. This observation is used by Algorithm 5: it is alternatively minimizing \mathcal{E} over all cluster means with the assignments $\{y^{(i)}\}_{i=1}^m$ held fixed and minimizing \mathcal{E} over all cluster assignments with the cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^k$ held fixed.

The interpretation of Algorithm 5 as a method for minimizing the cost function (8.3) is useful for convergence diagnosis. In particular, we might terminate Algorithm 5 if the decrease of the objective function \mathcal{E} is below a prescribed (small) threshold.

A practical implementation of Algorithm 5 needs to fix three issues:

- Issue 1: We need to specify a “tie-breaking strategy” to handle the case when several different cluster indices $c \in \{1, \dots, k\}$ achieve the minimum value in (8.1).
- Issue 2: We need to specify how to handle the situation when after a cluster assignment update (8.1), there is a cluster c with no data points are associated with it, i.e., $|\{i : y^{(i)} = c\}| = 0$. In this case, the cluster means update (8.2) would be not well defined for the cluster c .
- Issue 3: We need to specify a stopping criterion (“checking convergence”).

The following algorithm fixes those three issues in a particular way [?].

The variables $b^{(c)} \in \{0, 1\}$ indicate if cluster c is active ($b^{(c)} = 1$) or cluster c is inactive ($b^{(c)} = 0$), in the sense of having no data points assigned to it during the preceding cluster assignment step (8.4). Using these additional variables allows to ensure the cluster mean update step (8.5) to be well defined, since it is only applied to cluster c associated with at least one data point $\mathbf{x}^{(i)}$.

Algorithm 6 “ k -Means II” (a reformulation of the “Fixed Point Algorithm” presented in [?])

Input: dataset $\mathbb{X} = \{\mathbf{x}^{(i)}\}_{i=1}^m$; number k of clusters; prescribed tolerance $\varepsilon \geq 0$.

Initialize: choose initial cluster means $\{\mathbf{m}^{(c)}\}_{c=1}^k$ and cluster assignments $\{y^{(i)}\}_{i=1}^m$; set iteration counter $r := 0$; compute $E^{(r)} = \mathcal{E}(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathbb{X})$;

1: **repeat**

2: for all data points $i = 1, \dots, m$, update cluster assignment

$$y^{(i)} := \min_{c' \in \{1, \dots, k\}} \{\argmin \|\mathbf{x}^{(i)} - \mathbf{m}^{(c')}\|\} \quad (\text{update cluster assignments}) \quad (8.4)$$

3: for all clusters $c = 1, \dots, k$, update activity indicator $b^{(c)} = \begin{cases} 1 & \text{if } |\{i : y^{(i)} = c\}| > 0 \\ 0 & \text{else.} \end{cases}$

4: for all $c = 1, \dots, k$ with $b^{(c)} = 1$, update cluster means

$$\mathbf{m}^{(c)} = \frac{1}{|\{i : y^{(i)} = c\}|} \sum_{i: y^{(i)} = c} \mathbf{x}^{(i)} \quad (\text{update cluster means}) \quad (8.5)$$

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

6: $E^{(r)} = \mathcal{E}(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathbb{X})$ (see (8.3))

7: **until** $E^{(r-1)} - E^{(r)} > \varepsilon$

Output: cluster assignments $y^{(i)} \in \{1, \dots, k\}$ and cluster means $\mathbf{m}^{(c)}$

It can be shown that Algorithm 6 amounts to a fixed-point iteration

$$\{y^{(i)}\}_{i=1}^m \mapsto \mathcal{P}\{y^{(i)}\}_{i=1}^m \quad (8.6)$$

with a particular operator \mathcal{P} (which depends on the dataset \mathbb{X}). Each iteration of Algorithm 6 updates the cluster assignments $y^{(i)}$ by applying the operator \mathcal{P} . By interpreting Algorithm 6 as a fixed-point iteration (8.6), the authors of [?, Thm. 2] present an elegant proof of the convergence of Algorithm 6 within a finite number of iterations (even for $\varepsilon = 0$). What is more, after running Algorithm 6 for a finite number of iterations the cluster assignments $\{y^{(i)}\}_{i=1}^m$ do not change anymore.

We illustrate the operation of Algorithm 6 in Figure 8.2. Each column corresponds to one iteration of Algorithm 6. The upper picture in each column depicts the update of cluster means while the lower picture shows the update of the cluster assignments during each iteration.

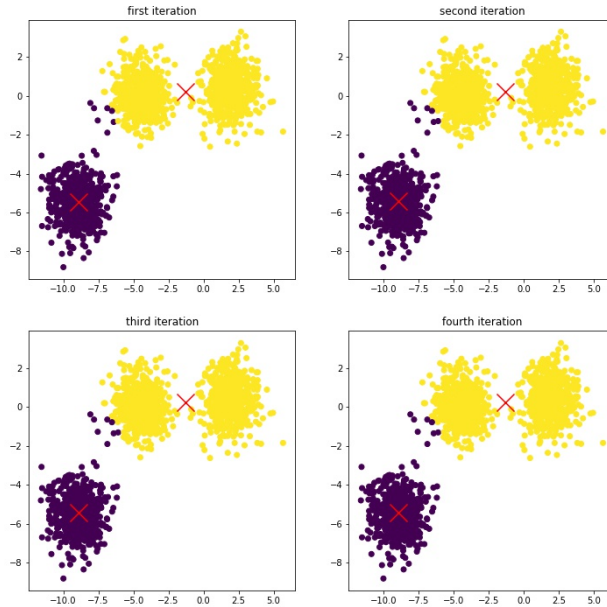


Figure 8.2: Evolution of cluster means and cluster assignments within k -means.

While Algorithm 6 is guaranteed to terminate after a finite number of iterations, the delivered cluster assignments and cluster means might only be (approximations) of local

minima of the clustering error (8.3) (see Figure 8.3). In order to escape local minima, it is useful to run Algorithm 6 several times, using different initializations for the cluster means, and picking the cluster assignments $\{y^{(i)}\}_{i=1}^m$ with smallest clustering error (8.3).

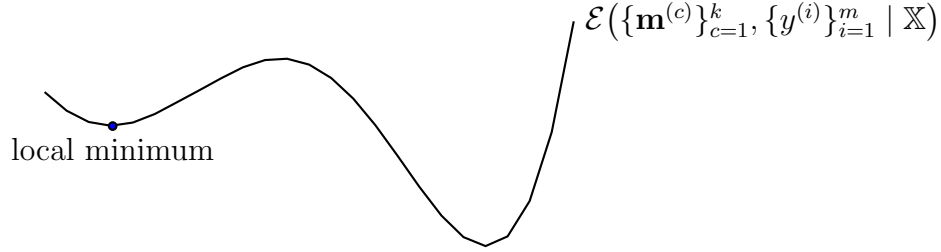


Figure 8.3: The clustering error $\mathcal{E}(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathbb{X})$ (see (8.3)), which is minimized by k -means, is a non-convex function of the cluster means and assignments. It is therefore possible for k -means to get trapped around a local minimum.

Up till now, we have assumed the number k of clusters to be given before hand. However, in some applications it is not clear what a good choice for k can be. One approach to choosing the value of k is if the clustering method acts as a sub-module within an overall supervised ML system, which allows to implement some sort of validation. We could then try out different values of the number k and determine validation errors for each choice. Then, we pick the choice of k which results in the smallest validation error.

Another approach to choosing k is the so-called “elbow-method”. This approach amounts to running k -means Algorithm 6 for different values of k resulting in the (approximate) optimum empirical error $\mathcal{E}^{(k)} = \mathcal{E}(\{\mathbf{m}^{(c)}\}_{c=1}^k, \{y^{(i)}\}_{i=1}^m \mid \mathbb{X})$. We then plot the minimum empirical error $\mathcal{E}^{(k)}$ as a function of the number k of clusters. This plot typically looks like Figure 8.4, i.e., a steep decrease for small values of k and then flattening out for larger values of k . Finally, the choice of k might be guided by some probabilistic model which penalizes larger values of k .

8.2 Soft Clustering

The cluster assignments obtained from hard-clustering methods, such as Algorithm 6, provide rather coarse-grained information. Indeed, even if two data points $\mathbf{x}^{(i)}, \mathbf{x}^{(j)}$ are assigned to the same cluster c , their distances to the cluster mean $\mathbf{m}^{(c)}$ might be very different. For some applications, we would like to have a more fine-grained information about the cluster assignments.

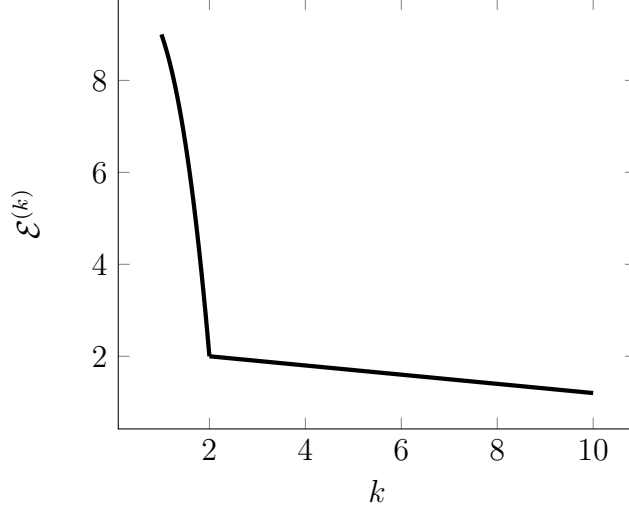


Figure 8.4: The clustering error $\mathcal{E}^{(k)}$ achieved by k -means for increasing number k of clusters.

Soft-clustering methods provide such fine-grained information by explicitly modelling the degree (or confidence) by which a particular data point belongs to a particular cluster. More precisely, soft-clustering methods track for each data point $\mathbf{x}^{(i)}$ the degree of belonging to each of the clusters $c \in \{1, \dots, k\}$.

A principled approach to modelling a degree of belonging to different clusters uses a probabilistic (generative) model for the dataset $\mathbb{X} = \{\mathbf{x}^{(i)}\}_{i=1}^m$. Within this model, we represent each cluster by a probability distribution. One popular choice for this distribution is the multivariate normal distribution

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{\det\{2\pi\boldsymbol{\Sigma}\}}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) \quad (8.7)$$

of a Gaussian random vector with mean $\boldsymbol{\mu}$ and (invertible) covariance matrix $\boldsymbol{\Sigma}$.² We maintain a separate distribution of the form (8.7) for each of the k clusters. Thus, each cluster $c \in \{1, \dots, k\}$ is represented by a distribution of the form (8.7) with a cluster-specific mean $\boldsymbol{\mu}^{(c)} \in \mathbb{R}^n$ and cluster-specific covariance matrix $\boldsymbol{\Sigma}^{(c)} \in \mathbb{R}^{n \times n}$.

Since we do not know the correct cluster $c^{(i)}$ of the data point $\mathbf{x}^{(i)}$, we model the cluster assignment $c^{(i)}$ as a random variable with probability distribution

$$p_c := \mathbb{P}(c^{(i)} = c) \text{ for } c = 1, \dots, k. \quad (8.8)$$

²Note that the distribution (8.7) is only defined for an invertible (non-singular) covariance matrix $\boldsymbol{\Sigma}$.

Note that the (prior) probabilities p_c are unknown and therefore have to be estimated somehow by the soft-clustering method. The random cluster assignment $c^{(i)}$ selects the cluster-specific distribution (8.7) of the random data point $\mathbf{x}^{(i)}$:

$$P(\mathbf{x}^{(i)}|c^{(i)}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}^{(c^{(i)})}, \boldsymbol{\Sigma}^{(c^{(i)})}) \quad (8.9)$$

with mean vector $\boldsymbol{\mu}^{(c)}$ and covariance matrix $\boldsymbol{\Sigma}^{(c)}$.

The modelling of cluster assignments $c^{(i)}$ as (unobserved) random variables lends naturally to a rigorous definition for the notion of the degree $y_c^{(i)}$ by which data point $\mathbf{x}^{(i)}$ belongs to cluster c . In particular, we define the degree $y_c^{(i)}$ of data point $\mathbf{x}^{(i)}$ belonging to cluster c as the “a-posteriori” probability of the cluster assignment $c^{(i)}$ being equal to a particular cluster index $c \in \{1, \dots, k\}$:

$$y_c^{(i)} := P(c^{(i)} = c | \mathbb{X}). \quad (8.10)$$

By their very definition (8.10), the degrees of belonging $y_c^{(i)}$ have to sum to one:

$$\sum_{c=1}^k y_c^{(i)} = 1 \text{ for each } i = 1, \dots, m. \quad (8.11)$$

It is important to note that we use the conditional cluster probability (8.10), conditioned on the dataset, 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 \mathbb{X} .

A probabilistic model for the observed data points $\mathbf{x}^{(i)}$ is obtained by considering each data point $\mathbf{x}^{(i)}$ being the result of a random draw from the distribution $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}^{(c^{(i)})}, \boldsymbol{\Sigma}^{(c^{(i)})})$ with some cluster $c^{(i)}$. Since the cluster indices $c^{(i)}$ are unknown,³ we model them as random variables. In particular, we model the cluster indices $c^{(i)}$ as i.i.d. with probabilities $p_c = P(c^{(i)} = c)$.

The overall probabilistic model (8.9), (8.8) amounts to a **Gaussian mixture model** (GMM). Indeed, using the law of total probability, the marginal distribution $P(\mathbf{x}^{(i)})$ (which is the same for all data points $\mathbf{x}^{(i)}$) is a (additive) mixture of multivariate Gaussian distributions:

$$P(\mathbf{x}^{(i)}) = \sum_{c=1}^k \underbrace{P(c^{(i)} = c)}_{p_c} \underbrace{P(\mathbf{x}^{(i)}|c^{(i)} = c)}_{\mathcal{N}(\mathbf{x}^{(i)}; \boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)})}. \quad (8.12)$$

Within our statistical model for the dataset \mathbb{X} , the cluster assignments $c^{(i)}$ are hidden

³After all, the goal of soft-clustering is to estimate the cluster indices $c^{(i)}$.

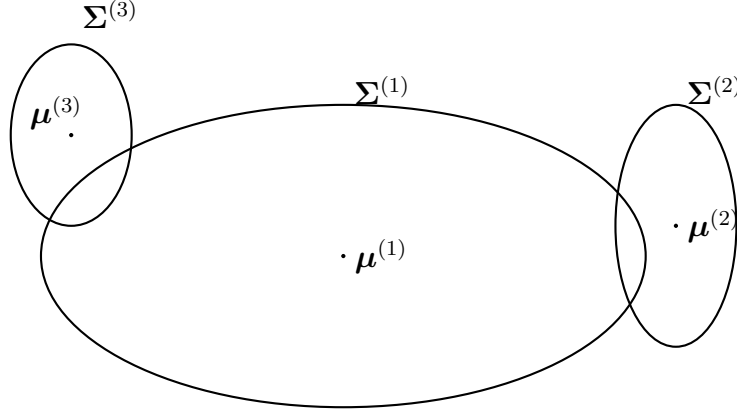


Figure 8.5: The GMM (8.12) yields a probability density function which is a weighted sum of multivariate normal distributions $\mathcal{N}(\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)})$. The weight of the c -th component is the cluster probability $P(c^{(i)} = c)$.

(unobserved) random variables. We thus have to infer or estimate these variables from the observed data points $\mathbf{x}^{(i)}$ which are i.i.d. realizations of the GMM (8.12).

Using the GMM (8.12) for explaining the observed data points $\mathbf{x}^{(i)}$ turns the clustering problem into a **statistical inference** or **parameter estimation problem** [? ?]. The estimation problem is to estimate the true underlying cluster probabilities p_c (see (8.8)), cluster means $\boldsymbol{\mu}^{(c)}$ and cluster covariance matrices $\boldsymbol{\Sigma}^{(c)}$ (see (8.9)) from the observed data points $\mathbb{X} = \{\mathbf{x}^{(i)}\}_{i=1}^m$, which are drawn from the probability distribution (8.12).

We denote the estimates for the GMM parameters by $\hat{p}_c (\approx p_c)$, $\mathbf{m}^{(c)} (\approx \boldsymbol{\mu}^{(c)})$ and $\mathbf{C}^{(c)} (\approx \boldsymbol{\Sigma}^{(c)})$, respectively. Based on these estimates, we can then compute an estimate $\hat{y}_c^{(i)}$ of the (a-posterior) probability

$$y_c^{(i)} = P(c^{(i)} = c \mid \mathbb{X}) \quad (8.13)$$

of the i -th data point $\mathbf{x}^{(i)}$ belonging to cluster c , given the observed dataset \mathbb{X} .

It turns out that this estimation problem becomes significantly easier by operating in an alternating fashion: In each iteration, we first compute a new estimate \hat{p}_c of the cluster probabilities p_c , given the current estimate $\mathbf{m}^{(c)}, \mathbf{C}^{(c)}$ for the cluster means and covariance matrices. Then, using this new estimate \hat{p}_c for the cluster probabilities, we update the estimates $\mathbf{m}^{(c)}, \mathbf{C}^{(c)}$ of the cluster means and covariance matrices. Then, using the new estimates $\mathbf{m}^{(c)}, \mathbf{C}^{(c)}$, we compute a new estimate \hat{p}_c and so on. By repeating these two update steps, we obtain an iterative soft-clustering method which is summarized in Algorithm 7.

As for hard clustering, we can interpret the soft clustering problem as an instance of the

Algorithm 7 “A Soft-Clustering Algorithm” [?]

Input: dataset $\mathbb{X} = \{\mathbf{x}^{(i)}\}_{i=1}^m$; number k of clusters.

Initialize: use initial guess for GMM parameters $\{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^k$

1: **repeat**

2: for each data point $\mathbf{x}^{(i)}$ and cluster $c \in \{1, \dots, k\}$, update degrees of belonging

$$y_c^{(i)} = \frac{\hat{p}_c \mathcal{N}(\mathbf{x}^{(i)}; \mathbf{m}^{(c)}, \mathbf{C}^{(c)})}{\sum_{c'=1}^k \hat{p}_{c'} \mathcal{N}(\mathbf{x}^{(i)}; \mathbf{m}^{(c')}, \mathbf{C}^{(c')})} \quad (8.14)$$

3: for each cluster $c \in \{1, \dots, k\}$, update estimates of GMM parameters:

- cluster probability $\hat{p}_c = m_c/m$, with effective cluster size $m_c = \sum_{i=1}^m y_c^{(i)}$
- cluster mean $\mathbf{m}^{(c)} = (1/m_c) \sum_{i=1}^m y_c^{(i)} \mathbf{x}^{(i)}$
- cluster covariance matrix $\mathbf{C}^{(c)} = (1/m_c) \sum_{i=1}^m y_c^{(i)} (\mathbf{x}^{(i)} - \mathbf{m}^{(c)}) (\mathbf{x}^{(i)} - \mathbf{m}^{(c)})^T$

4: **until** convergence

Output: soft cluster assignments $\mathbf{y}^{(i)} = (y_1^{(i)}, \dots, y_k^{(i)})^T$ for each data point $\mathbf{x}^{(i)}$

ERM principle (Chapter 4). In particular, Algorithm 7 aims at minimizing the empirical risk

$$\mathcal{E}(\{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^k \mid \mathbb{X}) = -\log \text{Prob}\{\mathbb{X}; \{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^k\}. \quad (8.15)$$

The interpretation of Algorithm 7 as a method for minimizing the empirical risk (8.15) suggests to monitor the decrease of the empirical risk $-\log \text{Prob}\{\mathbb{X}; \{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^k\}$ to decide when to stop iterating (“convergence test”).

Similar to k -means Algorithm 5, also the soft clustering Algorithm 7 suffers from the problem of getting stuck in local minima of the empirical risk (8.15). As for k -means, a simple strategy to overcome this issue is to run Algorithm 7 several times, each time with a different initialization for the GMM parameter estimates $\{\mathbf{m}^{(c)}, \mathbf{C}^{(c)}, \hat{p}_c\}_{c=1}^k$ and then picking the result which yields the smallest empirical risk (8.15).

We note that the empirical risk (8.15) underlying the soft-clustering Algorithm 7 is essentially a **log-likelihood function**. Thus, Algorithm 7 can be interpreted as an **approximate maximum likelihood** estimator for the true underlying GMM parameters $\{\boldsymbol{\mu}^{(c)}, \boldsymbol{\Sigma}^{(c)}, p_c\}_{c=1}^k$. In particular, Algorithm 7 is an instance of a generic approximate maximum likelihood technique referred to as **expectation maximization** (EM) (see [?, Chap. 8.5] for more details). The interpretation of Algorithm 7 as a special case of EM allows to characterize

the behaviour of Algorithm 7 using existing convergence results for EM methods [?].

We finally note that k -means hard clustering can be interpreted as an extreme case of soft-clustering Algorithm 7. Indeed, consider fixing the cluster covariance matrices $\Sigma^{(c)}$ within the GMM (8.9) to be the scaled identity:

$$\Sigma^{(c)} = \sigma^2 \mathbf{I} \text{ for all } c \in \{1, \dots, k\}. \quad (8.16)$$

Here, we assume the covariance matrix (8.16), with a particular value for σ^2 , to be the actual “correct” covariance matrix for cluster c . The estimates $\mathbf{C}^{(c)}$ for the covariance matrices are then trivially given by $\mathbf{C}^{(c)} = \Sigma^{(c)}$, i.e., we can omit the covariance matrix updates in Algorithm 7. Moreover, when choosing a very small variance σ^2 in (8.16)), the update (8.14) tends to enforce $y_c^{(i)} \in \{0, 1\}$, i.e., each data point $\mathbf{x}^{(i)}$ is associated to exactly one cluster c , whose cluster mean $\mathbf{m}^{(c)}$ is closest to the data point $\mathbf{x}^{(i)}$. Thus, for $\sigma^2 \rightarrow 0$, the soft-clustering update (8.14) reduces to the hard cluster assignment update (8.1) of the k -means Algorithm 5.

Chapter 9

Feature Learning

“Solving Problems By Changing the Viewpoint.”

Roughly speaking, ML methods exploit the intrinsic geometry of (large) sets of data points to compute predictions. By definition, we represent these data points as elements of the feature space \mathcal{X} . Note that the features are a design choice so we can shape the intrinsic geometry of the data points by using different choices for the features (and feature space). Feature learning methods automate the choice of finding a good feature space for a given data set. A subclass of feature learning methods are dimensionality reduction methods, where the new feature space has a (much) smaller dimension than the original feature space (see Section 9.1). However, sometimes it might be useful to change to a higher-dimensional feature space (see Section 9.6).

??? Develop feature learning as an approximation problem. The raw data is the vector to be approximated. The approximation has to be in a (small) subspace which is spanned by all possible low-dimensional feature vectors???

9.1 Dimensionality Reduction

Consider a ML method that aims at predicting the label y of a data point \mathbf{z} based on some features \mathbf{x} which characterize the data point \mathbf{z} . Intuitively, it should be beneficial to use as many features as possible. Indeed, the more features of a data point we know, the more we should know about its label y .

There are, however, two pitfalls in using an unnecessarily large number of features. The first one is a **computational pitfall** and the second one is a **statistical pitfall**. The larger

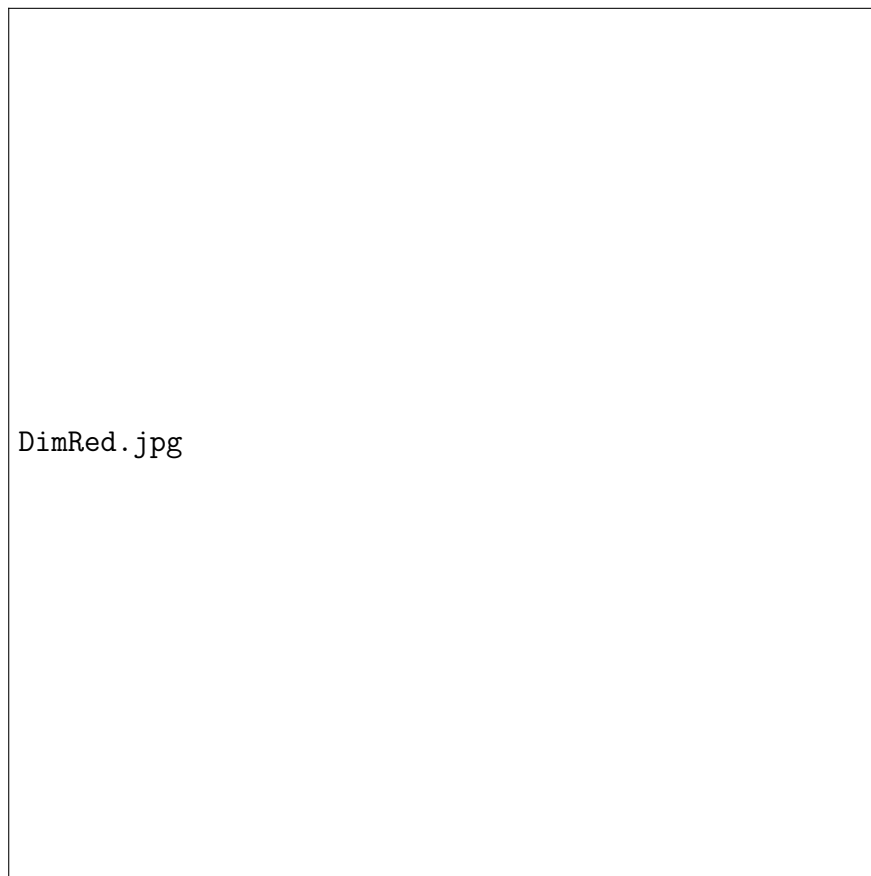


Figure 9.1: Dimensionality reduction methods aim at finding a map h which maximally compresses the raw data while still allowing to accurately reconstruct the original data point from a small number of features x_1, \dots, x_n .

the feature vector $\mathbf{x} \in \mathbb{R}^n$ (with large n), the more computation (and storage) is required for executing the resulting ML method. Moreover, using a large number of features makes the resulting ML methods more prone to overfitting. Indeed, linear regression will overfit when using feature vectors $\mathbf{x} \in \mathbb{R}^n$ whose length n exceeds the number m of labeled data points used for training (see Chapter 7).

Thus, both from a computational and statistical perspective, it is beneficial to use only the maximum necessary amount of relevant features. A key challenge here is to select those features which carry most of the relevant information required for the prediction of the label y . Beside coping with overfitting and limited computational resources, dimensionality reduction can also be useful for data visualization. Indeed, if the resulting feature vector has length $d = 2$, we can use scatter plots to depict datasets.

The basic idea behind most dimensionality reduction methods is quite simple. As illustrated in Figure 9.1, these methods aim at learning (finding) a “compression” map that transforms a raw data point \mathbf{z} to a (short) feature vector $\mathbf{x} = (x_1, \dots, x_n)^T$ in such a way that it is possible to find (learn) a “reconstruction” map which allows to accurately reconstruct the original data point from the features \mathbf{x} . The compression and reconstruction map is typically constrained to belong some set of computationally feasible maps or hypothesis space (see Chapter 3 for different examples of hypothesis spaces). In what follows we restrict ourselves to using only linear maps for compression and reconstruction leading to principal component analysis. The extension to non-linear maps using deep neural networks is known as **deep autoencoders** [?, Ch. 14].

9.2 Principal Component Analysis

Consider a data point $\mathbf{z} \in \mathbb{R}^D$ which is represented by a (typically very long) vector of length D . The length D of the raw feature vector might be easily on 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 data point:

$$\mathbf{x} = \mathbf{W}\mathbf{z}. \tag{9.1}$$

Here, the “compression” matrix $\mathbf{W} \in \mathbb{R}^{n \times D}$ maps (in a linear fashion) the large vector $\mathbf{z} \in \mathbb{R}^D$ to a smaller feature vector $\mathbf{x} \in \mathbb{R}^n$.

It is reasonable to choose the compression matrix $\mathbf{W} \in \mathbb{R}^{n \times D}$ in (9.1) such that the resulting features $\mathbf{x} \in \mathbb{R}^n$ allow to approximate the original data point $\mathbf{z} \in \mathbb{R}^D$ as accurate as possible. We can approximate (or recover) the data point $\mathbf{z} \in \mathbb{R}^D$ back from the features

\mathbf{x} by applying a reconstruction operator $\mathbf{R} \in \mathbb{R}^{D \times n}$, which is chosen such that

$$\mathbf{z} \approx \mathbf{R}\mathbf{x} \stackrel{(9.1)}{=} \mathbf{R}\mathbf{W}\mathbf{z}. \quad (9.2)$$

The approximation error $\mathcal{E}(\mathbf{W}, \mathbf{R} \mid \mathbb{X})$ resulting when (9.2) is applied to each data point in a dataset $\mathbb{X} = \{\mathbf{z}^{(i)}\}_{i=1}^m$ is then

$$\mathcal{E}(\mathbf{W}, \mathbf{R} \mid \mathbb{X}) = (1/m) \sum_{i=1}^m \|\mathbf{z}^{(i)} - \mathbf{R}\mathbf{W}\mathbf{z}^{(i)}\|. \quad (9.3)$$

One can verify that the approximation error $\mathcal{E}(\mathbf{W}, \mathbf{R} \mid \mathbb{X})$ can only be minimal if the compression matrix \mathbf{W} is of the form

$$\mathbf{W} = \mathbf{W}_{\text{PCA}} := (\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(n)})^T \in \mathbb{R}^{n \times D}, \quad (9.4)$$

with n orthonormal vectors $\mathbf{u}^{(l)}$ which correspond to the n largest eigenvalues of the **sample covariance matrix**

$$\mathbf{Q} := (1/m) \mathbf{Z}^T \mathbf{Z} \in \mathbb{R}^{D \times D} \quad (9.5)$$

with data matrix $\mathbf{Z} = (\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)})^T \in \mathbb{R}^{m \times D}$.¹ By its very definition (9.5), the matrix \mathbf{Q} is positive semi-definite so that it allows for an eigenvalue decomposition (EVD) of the form [?]

$$\mathbf{Q} = (\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(D)}) \begin{pmatrix} \lambda^{(1)} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & \lambda^{(D)} \end{pmatrix} (\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(D)})^T$$

with real-valued eigenvalues $\lambda^{(1)} \geq \lambda^{(2)} \geq \dots \geq \lambda^{(D)} \geq 0$ and orthonormal eigenvectors $\{\mathbf{u}_r\}_{r=1}^D$.

The features $\mathbf{x}^{(i)}$, obtained by applying the compression matrix \mathbf{W}_{PCA} (9.4) to the raw data points $\mathbf{z}^{(i)}$, are referred to as **principal components (PC)**. The overall procedure of determining the compression matrix (9.4) and, in turn, computing the PC vectors $\mathbf{x}^{(i)}$ is known as **principal component analysis (PCA)** and summarized in Algorithm 8. Note that the length n of the feature vectors \mathbf{x} , which is also the number of PCs used, is an input parameter of Algorithm 8. The number n can be chosen between $n = 0$ and $n = D$. However, it can be shown that PCA for $n > m$ is not well-defined. In particular, the orthonormal

¹Some authors define the data matrix as $\mathbf{Z} = (\tilde{\mathbf{z}}^{(1)}, \dots, \tilde{\mathbf{z}}^{(m)})^T \in \mathbb{R}^{m \times D}$ using “centered” data points $\tilde{\mathbf{z}}^{(i)} = \mathbf{z}^{(i)} - \hat{\mathbf{m}}$ obtained by subtracting the average $\hat{\mathbf{m}} = (1/m) \sum_{i=1}^m \mathbf{z}^{(i)}$.

Algorithm 8 Principal Component Analysis (PCA)

Input: dataset $\mathbb{X} = \{\mathbf{z}^{(i)} \in \mathbb{R}^D\}_{i=1}^m$; number n of PCs.

- 1: compute EVD (9.6) to obtain orthonormal eigenvectors $(\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(D)})$ corresponding to (decreasingly ordered) eigenvalues $\lambda^{(1)} \geq \lambda^{(2)} \geq \dots \geq \lambda^{(D)} \geq 0$
- 2: construct compression matrix $\mathbf{W}_{\text{PCA}} := (\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(n)})^T \in \mathbb{R}^{n \times D}$
- 3: compute feature vector $\mathbf{x}^{(i)} = \mathbf{W}_{\text{PCA}} \mathbf{z}^{(i)}$ whose entries are PC of $\mathbf{z}^{(i)}$
- 4: compute approximation error $\mathcal{E}^{(\text{PCA})} = \sum_{r=n+1}^D \lambda^{(r)}$ (see (9.6)).

Output: $\mathbf{x}^{(i)}$, for $i = 1, \dots, m$, and the approximation error $\mathcal{E}^{(\text{PCA})}$.

eigenvectors $\mathbf{u}^{(n+1)}, \dots, \mathbf{u}^{(D)}$ are not unique.

From a computational perspective, Algorithm 8 essentially amounts to performing an EVD of the sample covariance matrix \mathbf{Q} (see (9.5)). Indeed, the EVD of \mathbf{Q} provides not only the optimal compression matrix \mathbf{W}_{PCA} but also the measure $\mathcal{E}^{(\text{PCA})}$ for the information loss incurred by replacing the original data points $\mathbf{z}^{(i)} \in \mathbb{R}^D$ 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 $\mathbf{R}_{\text{opt}} = \mathbf{W}_{\text{PCA}}^T$)

$$\mathcal{E}^{(\text{PCA})} := \mathcal{E}(\underbrace{\mathbf{W}_{\text{PCA}}, \mathbf{R}_{\text{opt}}}_{=\mathbf{W}_{\text{PCA}}^T} \mid \mathbb{X}) = \sum_{r=n+1}^D \lambda^{(r)}. \quad (9.6)$$

As depicted in Figure 9.2, the approximation error $\mathcal{E}^{(\text{PCA})}$ decreases with increasing number n of PCs used for the new features (9.1). The maximum error $\mathcal{E}^{(\text{PCA})} = (1/m) \sum_{i=1}^m \|\mathbf{z}^{(i)}\|^2$ is obtained for $n = 0$, which amounts to completely ignoring the data points $\mathbf{z}^{(i)}$. In the other extreme case where $n = D$ and $\mathbf{x}^{(i)} = \mathbf{z}^{(i)}$, which amounts to no compression at all, the approximation error is zero $\mathcal{E}^{(\text{PCA})} = 0$.

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 data points are characterized by feature vectors whose length D exceeds the number m of labeled data points used for training. One simple but powerful strategy to avoid overfitting is to preprocess the original feature vectors (they are considered as the raw data points $\mathbf{z}^{(i)} \in \mathbb{R}^D$) by applying PCA in order to obtain smaller

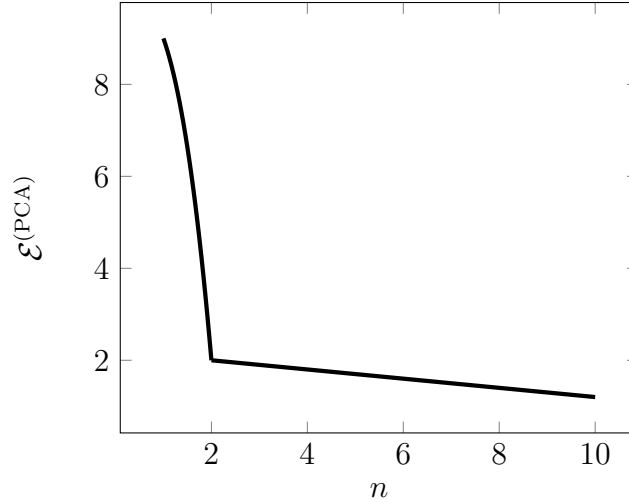


Figure 9.2: Reconstruction error $\mathcal{E}^{(PCA)}$ (see (9.6)) of PCA for varying number n of PCs.

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 computational complexity of overall ML method fits 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 $\mathcal{E}^{(PCA)}$ is reasonably small (see Figure 9.2).

9.2.3 Data Visualisation

If we use PCA with $n = 2$ PC, we obtain feature vectors $\mathbf{x}^{(i)} = \mathbf{W}\mathbf{z}^{(i)}$ (see (9.1)) which can be depicted as points in a scatter plot (see Section 2.1.3). As an example we consider data

points $\mathbf{z}^{(i)}$ obtained from historic recordings of Bitcoin statistics. Each data point $\mathbf{z}^{(i)} \in \mathbb{R}^6$ is a vector of length $D = 6$. It is difficult to visualise points in an Euclidean space \mathbb{R}^D of dimension $D > 2$. It is then helpful to apply PCA with $n = 2$ which results in feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^2$ which can be depicted conveniently as a scatter plot (see Figure 9.3).

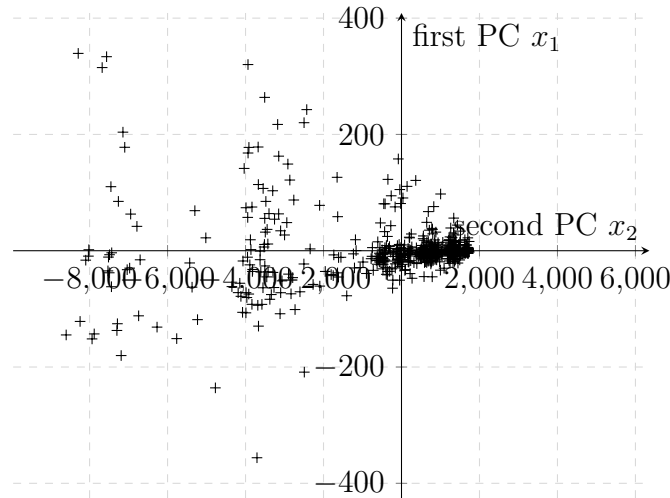


Figure 9.3: A scatter plot of feature vectors $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)})^T$ whose entries are the first two PCs of the Bitcoin statistics $\mathbf{z}^{(i)}$ of the i -th day.

9.2.4 Extensions of PCA

There have been proposed several extensions of the basic PCA method:

- **kernel PCA** [? , Ch.14.5.4]: combines PCA with a non-linear feature map (see Section 3.8).
- **robust PCA** [?]: modifies PCA to better cope with **outliers** in the dataset.
- **sparse PCA** [? , Ch.14.5.5]: requires each PC to depend only on a small number of data attributes z_j .
- **probabilistic PCA** [? ?]: generalizes PCA by using a **probabilistic (generative) model** for the data.

9.3 Linear Discriminant Analysis

Dimensionality reduction is typically used as a preprocessing step within some overall ML problem such as regression or classification. It can then be useful to exploit the availability of labeled data for the design of the compression matrix \mathbf{W} in (9.1). However, plain PCA (see Algorithm 8) does not make use of any label information provided additionally for the raw data points $\mathbf{z}^{(i)} \in \mathbb{R}^D$. Therefore, the compression matrix \mathbf{W}_{PCA} delivered by PCA can be highly suboptimal as a pre-processing step for labeled data points. A principled approach for choosing the compression matrix \mathbf{W} such that data points with different labels are well separated is **linear discriminant analysis** [?].

9.4 Random Projections

Note that PCA amounts to computing an EVD of the sample covariance matrix $\mathbf{Q} = (1/m)\mathbf{Z}\mathbf{Z}^T$ with the data matrix $\mathbf{Z} = (\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)})^T$ containing the data points $\mathbf{z}^{(i)} \in \mathbb{R}^D$ as its columns. The computational complexity (amount of multiplications and additions) for computing this PCA is lower bounded by $\min\{D^2, m^2\}$ [? ?]. This computational complexity can be prohibitive for ML applications with n and m being on the order of millions (which is already the case if the features are pixel values of a 512×512 RGB bitmap, see Section 2.1.1). There is a surprisingly cheap alternative to PCA for finding a good choice for the compression matrix \mathbf{W} in (9.1). Indeed, a randomly chosen matrix \mathbf{W} with entries drawn i.i.d. from a suitable probability distribution (such as Bernoulli or Gaussian) yields a good compression matrix \mathbf{W} (see (9.1)) with high probability [? ?].

9.5 Information Bottleneck

We can use information bottleneck for feature learning. Using Gaussian process model, we even get closed-form solutions of Gaussian Information Bottleneck.

9.6 Dimensionality Increase

Feature learning methods are mainly dimensionality reduction methods. However, it might be beneficial to also consider feature learning methods that produce new feature vectors

which are longer than the raw feature vectors. An extreme example for such a feature map are kernel methods which map finite length vector to infinite dimensional spaces.

Mapping raw feature vectors into higher-dimensional spaces might be useful if the intrinsic geometry of the data points is simpler when looked at in the higher-dimensional space. Consider a binary classification problem where data points are highly inter-winded in the original feature space. By mapping into higher-dimensional feature space we might "even-out" this non-linear geometry such that we can use linear classifiers in the higher-dimensional space.

Chapter 10

Privacy-Preserving ML

Many ML applications involve data points representing individual humans. These data points might include sensitive data, such as medical records, which is subject to privacy protection. This chapter discusses some techniques for preprocessing the raw data to protect privacy of individuals while still allowing to solve the overall ML task. We will illustrate these techniques using a stylized healthcare application.



Figure 10.1: Data points represent humans. We are interested in the fruit preference of humans. Their gender is considered sensitive information and should not be revealed to ML methods.

A key challenge for health-care are pandemics. To optimally manage pandemics it is

important to have accurate information about the dynamics. We can model this as a ML problem with data points representing humans. One key feature of data points is if it represents an infected human or not. This data is sensitive and typically only available to public health-care institutes.

Consider the patient database of a hospital which should provide information about the average number of infected patients. Instead of directly forward the patient files, the hospital must only forward the fraction of infected patients. This is an example of privacy-preserving data processing. For a sufficiently large number of patients at the hospital (say, more than 1000), we cannot infer much about individual patients just from the fraction of infected patients treated in that hospital.

10.1 Privacy-Preserving Feature Learning (Operating on level of individual data points)

Privacy-preserving ML can be implemented using modification of feature learning methods discussed in Chapter 9. Generic feature learning methods aim at learning a compressed representation of the raw data points which contain as much information as possible about the quantity of interest. In contrast, privacy-preserving ML does not aim at compression but rather obscuring the raw data such that it does not reveal sensible information about data points.

10.1.1 Privacy-Preserving Information Bottleneck

10.1.2 Privacy-Preserving Feature Selection

?? ignore features which are sensitive (name, social ID) but not very relevant for actual task (e.g. predicting income). ???

10.1.3 Privacy-Preserving Random Projections

?? cheap form: random projections/compressed sensing. random projections blur features of individual data points but still allow to learn a sparse linear model using e.g. Lasso ???

10.2 Federated Learning (Operates on level of local datasets)

FL method only exchange model parameter updates; no raw local data is revealed;

Chapter 11

Explainable ML

A key challenge for the successful deployment of ML methods to many (critical) application domain is their explainability. Human users of ML seem to have a strong desire to get explanations that resolve the uncertainty about predictions and decisions obtained from ML methods.

Explainable ML is challenging since explanations must be tailored (personalized) to individual users with varying backgrounds. Some users might have received university-level education in ML, while other users might have no formal training in linear algebra. Linear regression with few features might be perfectly interpretable for the first group but might be considered a black-box by the latter.

?????? discuss relation between finding good explanations and active learning. Active learning aims at finding data points (by their features) which provide most information about the true model parameters. XML aims at finding explanations (e.g. data points from training set) which provide most information about the prediction provided by some black-box ML method. ?????????????? discuss relation between XML and feature learning. XML can be obtained from feature learning methods by learning those subset of features which provide most information about the prediction (not about the label itself) ??????????????

11.1 A Model Agnostic Method

We propose a simple probabilistic model for the predictions and user knowledge. This model allows to study explainable ML using information theory. Explaining is here considered as the task of reducing the “surprise” incurred by a prediction. We quantify the effect of an explanation by the conditional mutual information between the explanation and prediction,

given the user background.

11.2 Explainable Empirical Risk Minimization

Chapter 12

Lists of Symbols

t	A discrete time index.
i	Generic index used to enumerate data points in a list of data points.
m	The number of different data points in the training set.
$h(\cdot)$	A predictor that maps a feature vector \mathbf{x} of a data point to a predicted label $\hat{y} = h(\mathbf{x})$.
y	Label of some data point.
$y^{(i)}$	Label of the i th data point.
\mathbf{x}	Feature vector whose entries are the features of some data point.
$\mathbf{x}^{(i)}$	Feature vector whose entries are the features of the i th data point.
n	The number of (real-valued) features of a single data point.

Chapter 13

Glossary

- **classification problem:** an ML problem involving a discrete label space \mathcal{Y} such as $\mathcal{Y} = \{-1, 1\}$ for binary classification, or $\mathcal{Y} = \{1, 2, \dots, K\}$ 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} .
- **data point:** 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.
- **dataset:** a collection of data points.
- **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 $\mathbf{Ax} = \lambda\mathbf{x}$ with some $\lambda \in \mathbb{R}$, which we call an eigenvalue of \mathbf{A} .
- **features:** any measurements (or quantities) used to characterize a data point (e.g., the maximum amplitude of a sound recording 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) $h : \mathcal{X} \rightarrow \mathcal{Y}$ from the feature space \mathcal{X} to the label space \mathcal{Y} . Given a data point 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 $h : \mathcal{X} \rightarrow \mathcal{Y}$.
- **i.i.d.:** independent and identically distributed; e.g., “ x, y, z are i.i.d. random variables” means that the joint probability distribution $p(x, y, z)$ of the random variables x, y, z factors into the product $p(x)p(y)p(z)$ with the marginal probability distribution $p(\cdot)$ which is the same for all three variables x, y, z .
- **label:** some property of a data point 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 a data points that cannot be measured or computed easily in an automated fashion. Instead, acquiring accurate label information often involves human expert labor. Many ML methods aim at learning accurate predictor maps that allow to guess or approximate the label of a data point based on its features.
- **loss function:** a function which associates a given data point (\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 (psd) matrix:** a positive semidefinite matrix \mathbf{Q} , i.e., a symmetric matrix $\mathbf{Q} = \mathbf{Q}^T$ such that $\mathbf{x}^T \mathbf{Q} \mathbf{x} \geq 0$ holds for every vector \mathbf{x} .
- **predictor:** a hypothesis map $h : \mathcal{X} \rightarrow \mathcal{Y}$ with continuous label space (e.g., $\mathcal{Y} = \mathbb{R}$).
- **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.