YOUNG MAN, IN MATHEMATICS YOU DON'T UNDERSTAND THINGS.

YOU JUST GET USED TO THEM.

JOHN VON NEUMANN

FELIX BIESSMANN,

JAN SAPUTRA MÜLLER,

PAUL VON BÜNAU

MACHINE LEARNING COOKBOOK

First printing, April 2020

Layout http://www.latextemplates.com/template/tufte-style-book

Contents

Nomenclature

Table 1: Symbols and notations used throughout the book

Symbol	Meaning
$\mathbb{R}$	Space of real numbers
$\mathbb{R}_+$	Space of positive real numbers
$x \in \mathbb{R}^1$	Scalar (one-dimensional variable)
$\mathbf{x} \in \mathbb{R}^D$	Column vector with D real-valued entries
$\mathbf{X} \in \mathbb{R}^{D \times N}$	Real-valued matrix with D rows and N columns
$\mathbf{x}^n \in \mathbb{R}^D$	nth column vector of a matrix $\mathbf{X}$ with N columns
$\mathbf{x}_d^n \in \mathbb{R}^D$	dth row-element of $\mathbf{x}^n$
$\mathbf{x}^\top$	Transposed vector $\mathbf{x}$
$\mathbf{Ax}$	Multiplication of vector $\mathbf{x}$ with matrix $\mathbf{A}$
$\lambda\mathbf{x}$	Multiplication of vector $\mathbf{x}$ with scalar λ
$\frac{\partial \mathcal{E}(\mathbf{w})}{\partial \mathbf{w}}$	Partial derivative of $\mathcal{E}(\mathbf{w})$ w.r.t $\mathbf{w}$

I

Introduction

1

The purpose of this book

MACHINE LEARNING ALGORITHMS can be understood from different angles. One can analyse them from a purely theoretical perspective, set up model equations, come up with an optimisation technique for it and analyse its convergence properties and generalisation performance without ever touching a single data point. Others prefer to first download an implementation of an algorithm and just try it out on different data sets in order to see when it fails and when it works. Both approaches have their own weaknesses. Theory can lead astray if one relies on wrong model assumptions about the data. And only playing with algorithms without ever looking inside of them one can get stuck quickly or be forced to make mediocre compromises in terms of model performance. This cookbook tries to address both approaches by providing a quick reference highlighting the mathematical idea, a representative toy data example and a pseudocode implementation for some often used algorithms. The material was collected during several lecture series, but this collection is by no means exhaustive: Many important algorithms are not yet included, but we think of this as work in progress. We present algorithms for supervised learning, algorithms that can be used if one has labeled data of some kind, and unsupervised learning, algorithms that can help to find structure in data and exploit this structure for visualisation or compression.

What this book is not: A machine learning textbook. We do not include full derivations of algorithms, nor mathematical proofs of correctness or performance guarantees. Although there are some derivations in the appendix our theoretical treatment stops at stating the underlying objective function for each algorithm. For a good introduction into machine learning, we recommend the standard textbooks [Duda et al., 2001, Hastie et al., 2003, Bishop, 2007, Murphy, 2012]. What this book also does not cover in detail is the topic of artificial neural networks, but there are excellent deep learning textbooks for the theoretically inclined reader [Goodfellow et al., 2016] as well as the practitioner [Chollet, 2017].

2

What is machine learning?

What is machine learning? The science of algorithms that make sense of data, is one way to approach an answer. But this is clearly not satisfactory: what sense of what data, and which machine? Today, the academic discipline has so many facets and flavours, from abstract mathematical learning theory to the design of bioinformatics algorithms that it is hard to pinpoint the essence of machine learning. To understand why this is so, and explain where some of the terminology comes from, it is worth looking at the history of the field.

In the 1960s, machine learning set out to engineer systems, that can learn from observation and experience in the same way that humans do. Right from the beginning, the community focused on designing the software for such an intelligent machine, and left the hardware to the field of robotics. This essentially means developing learning algorithms. In the spirit of the day, abstract algorithms were called machine, following Alan Turing's famous concept of the Turing machine. The objective was emphatically enthusiastic: to construct intelligent, probably useful, machines to understand the nature of learning itself by constructive imitation. However, since the heyday of artificial intelligence, the ambitions of the machine learning community have become a lot humbler and in effect narrower. Unlike the field of artificial intelligence, machine learning did not experience a full on crises like the famous AI winter. Even so, the insights machine learning did deliver into human learning have been limited, if not disappointing. At the same time, machine learning methods achieved remarkable success in industrial applications, notably optical character recognition (OCR), which nudged the field further away from basic research towards engineering and algorithmic theory. Soon enough, the grand ambitions of machine learning became victim of their own preliminary successes in a few specialized tasks. As a result, the academic community focused on a small set of highly specialized but practically useful problems under the headlines of supervised and unsupervised learning.

Machine Learning is about designing algorithms that learn rules from data; it is not about engineering hardware nor software solutions for a specific application.

AI WINTER was the dramatic decrease in funding for artificial intelligence (AI) research. A critical role played the report by James Lighthill:

James Lighthill. Artificial intelligence: A general survey. In *Artificial Intelligence: a paper symposium, Science Research Council*, 1973

The report that gave a critical account of the progress made by this scientific field.

Supervised Learning In *supervised learning*, the goal is to learn how to predict a particular target variable given some observed data. For example, in optical character recognition (OCR), the target variable is the actual character shown, and the data is a digital image of a hand-written character. In a so-called learning phase, the machine (algorithm) is trained on a set of training data which consists of pairs of data (here: images) and target variable (here: letters $a, \dots, z$). Training, in essence, amounts to determining the free parameters of the particular type of learning algorithms. After the training has been completed, the machine can produce predictions for new, hitherto unseen, data.

Supervised learning: humans act as teacher for the algorithm by providing a label for each data point.

Unsupervised Learning The aim of *unsupervised learning* is a lot less specific, and the space of available methods more difficult to navigate. While supervised learning focuses on a specific prediction target, unsupervised learning methods find patterns of a particular type in data which often serve visualization, efficient data-descriptions or preprocessing. Prominent examples include clustering (chapter 12) and principal component analysis (chapter 7).

Unsupervised learning: the algorithm gets only data without any label information and is supposed to find meaningful structure in the data, e.g. for visualization.

Other Machine Learning Paradigms There are other important branches of ML research that do not fall under these two categories, supervised and unsupervised learning. For instance when the labels in a supervised task are not available for all data points, one might be interested in variants of active learning in order to identify those data points for which labels would improve the ML model most in terms of its uncertainty [Settles, 2010]. Or if a system is learning continuously and can only obtain labels for those data points that are shown to users of the ML system, as is the case for recommendation algorithms for instance, then one could try *bandit algorithms* to optimally exploit the current learnings of a model while also exploring new behaviour that the model has not gotten feedback for [Scott, 2010]. And even worse in some settings the labels needed to train the ML model are only obtained after a potentially long series of actions of the ML model, such as in chess or Go, when the ML model needs to make a series of decisions and only in the end receives feedback on its success. In this most complicated learning setting one can resort to reinforcement learning algorithms [Sutton and Barto, 1998]. While active learning, bandit learning or reinforcement learning do not fall under the categories of unsupervised or supervised learning, all of these research directions make use of unsupervised or supervised ML models – they just use them in different ways. The most important difference to the simpler cases of supervised ML discussed here is that the signal used for learning is available later.

Why not calling it Artificial Intelligence? The term Artificial Intelligence (AI) has been enjoyed a lot of interest in recent years. In discussions with ML researchers however you will barely hear the term being used. It almost seems as if researchers in the community from the outside referred to as AI researchers actively avoid the term and rather use the slightly better defined notion of machine learning. The two terms mean of course different things, for instance most ML research does not claim intelligence, and vice versa there are many great AI systems that do not rely on learning from data. A great example of such a rule based system is the ELIZA chatbot by Joseph Weizenbaum [Weizenbaum, 1966]. The learning here was done by humans who understood the structure of language and decomposed it into rules that could be used to mimic a human conversational agent.

The main reason why ML researchers avoid the term AI is that intelligence, be it artificial or biological, is very difficult to define. It is not exactly clear what intelligence tests measure, and how that relates to what we perceive as intelligent. The cognitive tasks in intelligence tests can be trained and hence one could imagine a scenario where one can improve his IQ score but not improving any other cognitive capability. Intelligence is not only difficult to measure it is also difficult to find a definition that would fit only human intelligence. Many animals for instance can perform tasks that humans like to believe can only be solved with human intelligence [Tomasello, 2003]. In a similar spirit Alan Turing proposed one the most famous tests for (artificial) intelligence [Turing, 1950].

Figure 2.1: Alan Turing (source Wikipedia) was upset about the debate on whether machines could become intelligent, as intelligence is difficult to quantify. He proposed a test for intelligence which he called the *Imitation Game*, now referred to as the *Turing Test* and institutionalized as the Loebner Prize. Nowadays many programs can pass the Turing Test, but critical voices claim that to win the Turing Test, machines need to be made as funny and dumb – not as intelligent – as humans.

Alan Turing proposed a test for intelligence which he called the *Imitation Game*. This test is commonly referred to as the *Turing Test* and consists of a human that interrogates two partners via a text terminal, one of the partners being a machine and the other partner being human. If the interrogator can not tell apart which partner is human, then the machine won the imitation game or in other words, it passed the Turing Test. The main motivation for Alan Turing for this research was that already in the 1940 researchers were using the term intelligent machines and there were heated debates over whether machines can ever reach human intelligence. Turing was upset by the fact that people were arguing about whether of not machines could be intelligent when nobody could define a measurable quantity of what humans meant by intelligent. So he tried to make it measurable by establishing the test for how well a machine can imitate humans. If machines are indistinguishable from humans in that test, then they probably are as intelligent as we are.

Many algorithms nowadays can pass the Turing Test[1] but when asking the programmers of the winning algorithms, they would usually not claim intelligence for their algorithms. In fact one of the winners

[1] See `https://en.wikipedia.org/wiki/Loebner_Prize`.

stated that one needs to make the algorithms artificially dumber then they could be to pass the test. For instance if you would ask a computer *How high is Mount Everest* the machine could give you the exact height – and fail the Turing Test, because most humans probably could not give you the exact height but most often something like *Hm, probably pretty high?*

This functionalist perspective on intelligence and the human mind in general was heavily criticised by philosophers. A famous thought experiment that challenges the view put forward by the Turing Test is the *Chinese Room Argument* by Jon Searle [Searle, 1980]: Imagine a program can answer questions posed in Chinese correctly e.g. by using a dictionary of all possible questions – does that program understand Chinese or does is merely simulate the understanding of Chinese? In summary the term AI is often considered a bit of an overstatement of what ML methods do. Consequently most researchers prefer the term ML over AI.

Does a machine understand anything if it always gives the correct answer to a question?

What do Machines Learn? In contrast to rule based systems, Machine Learning algorithms learn from data. Rather than requiring humans to extract rules, ML algorithms just need a data set that is put together by some humans. The ML algorithm then uses that data to extract rules from it. So an important characteristic of machine learning algorithms is: the data used to train an algorithm determines the quality of an algorithm's prediction. If a computer vision algorithm should learn to predict whether a given picture contains a tank or not and all pictures in the training data with tanks were shot on a sunny day, then blue sky might serve as a good predictor for tanks. A similar and more recent example is given in [Lapuschkin et al., 2019] where the authors show that an often used benchmark data set for computer vision has some artefacts that lead to models that learn to use the watermark symbol on images in the data set to determine a certain object class – rather than the object itself. Providing a well balanced representative training data set is thus essential for any machine learning algorithm.

But the question of what a ML model has learned is important beyond the idea of debugging data sets. Whether these rules learned by ML models are similar to what humans learn is a difficult question yet it is probably important for humans' trust in ML models. If a machine has learned something that humans cannot intuitively understand humans might loose trust in ML technology. This insight gave rise to the field of explainable AI or XAI [Samek et al., 2019].

Explaining vs. predicting can be regarded the quintessential difference between statistics, physics or other fields and machine learning. Statistics or physics aim at building models that capture causal relationships in experimental data, with a view to understanding the fundamental laws that govern the world. Machine learning, on the other hand, focuses entirely on making accurate predictions. To do this, it is free to use any correlation that has predictive value: one's postcode, for instance, does not cause one's creditworthiness, but it is correlated to it via an unobservable factor of solvency. Statisticians need to be a lot more careful in the choice of their models, in order not to confuse correlations with explanations, as in the tank example mentioned above. The abdiction of full explanatory power might seem naive: why would one prefer a dull algorithm over a model that transparently explains the origin of our data? Would a generative model of the data not be better? Yes, but these are hard to get, while algorithms that merely aim to predict are often cheaper to train and as good at making predictions as the "true" explanation of the data. This is nicely illustrated by an anecdote due to Vladimir Vapnik[2]: suppose it takes c data points to estimate one parameter of a generative model. In an OCR setting for handwritten digits then training a full generative model of a 20×20 pixel matrix that generates digits from 0 to 9 requires around $10(20^2)c$ examples; however one of the first machine learning algorithms, the perceptron[3] needed only 512 examples of training data to accurately predict the digits read from a 20×20 matrix. This surprised theoreticans in statistics and highlights a certain pragmatism often encountered among machine learning researchers: why solve a problem that is more complicated than what we are actually interested in? This must not be misunderstood as a nonchalant attitude towards generative models – understanding the data generating process is key to accurate predictions and learning generative models efficiently is one of the most important aspects of machine learning.

[2] Vladimir Vapnik. *Estimation of dependencies based on empirical data*. Springer Series in Statistics. Springer-Verlag, New York, 1982

[3] F Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958

II
Machine Learning Systems

3 Machine Learning Pipelines

Machine learning (ML) terminology can be confusing, because it is a mixed blend, stemming from computer science, mathematics, statistics and more application-oriented fields such as data mining. It can be helpful to structure the world of machine learning by taking a practitioner's perspective who implement machine learning systems for applications. A concept often used by ML practitioners is that of a *Machine Learning Pipeline*.

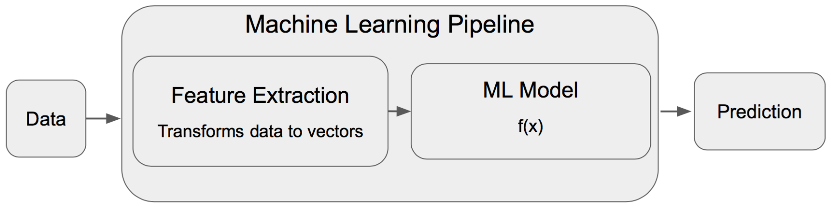


Figure 3.1: A Machine Learning (ML) Pipeline can be divided in a *feature extraction* component and the actual *ML model* or algorithm. The feature extraction step transforms the data into mathematical vectors $\mathbf{x}$. The ML model is a mathematical function $f(\mathbf{x})$ that takes a feature vector $\mathbf{x}$ and returns a prediction.

A Machine Learning Pipeline can be divided in a *feature extraction* component and the actual *ML model* or algorithm. The feature extraction step transforms the data (numbers, categorical variables, images, text, ...) into mathematical vectors $\mathbf{x} \in \mathbb{R}^d$, where d denotes the number of features extracted (for mathematical nomenclature see also Table 1). The ML model in its simplest form can be formalized as a mathematical function $f(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}$ that takes a feature vector $\mathbf{x}$ and returns a prediction, often just a number. At the end of the pipeline that number is then transformed into a prediction, often in text form.

Implementing and maintaining ML pipelines is challenging, but there are a number of very popular software packages that offer easy to use APIs for constructing ML pipelines, training them and serializing them, for instance general purpose ML libraries such as spark[1] or scikit-learn[2], but also specialized deep learning libraries like tensorflow[3] or PyTorch[4].

[1] https://spark.apache.org/

[2] https://scikit-learn.org/

[3] https://www.tensorflow.org/

[4] https://www.pytorch.org/

Using pipeline APIs simplifies many software engineering challenges, but when it comes to turning a ML pipeline into a fully functional ML system for industrial applications, there are many more challenges, many of which are not directly related to ML algorithms, as shown in Figure 3.2 and outlined in [Sculley et al., 2015]. In standard software systems testing is a well established practice and it is tedious but feasible to test the execution paths of deterministic code. ML systems in contrast learn from data, and depending on the data ingested, the ML system behaves differently. So data quality is essential for well functioning ML systems, and should also be tested in order to ensure robustness in ML systems [Schelter et al., 2018b].

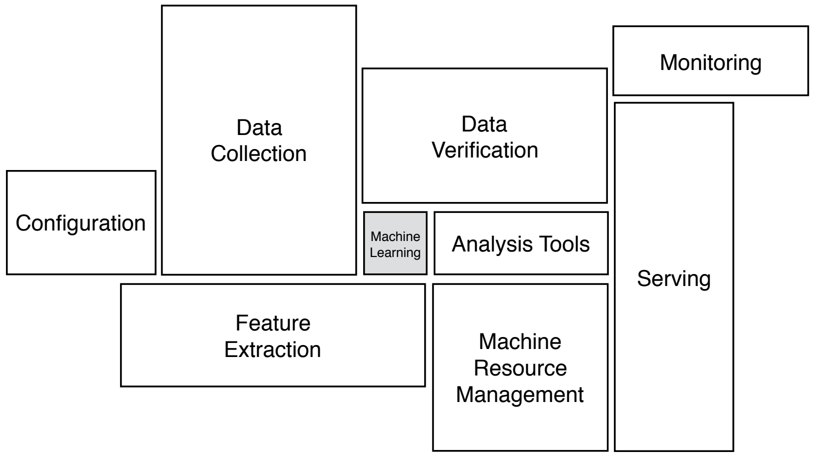


Figure 3.2: A Machine Learning System as used in industrial applications is a complex software system that consists of various components. Often the ML component requires less effort to implement and maintain than the other non-ML components. Figure adapted from [Sculley et al., 2015].

Most of the ML systems aspects are ignored in the remainder of this manuscript, but it is important to keep in mind that most of the work in a real world ML application often goes not into the ML part, but in the wiring of different components, data management and various software architecture challenges [Schelter et al., 2018a].

4

Feature Extraction

Most machine learning algorithms expect vector data as input. This assumption is crucial as it allows machine learning algorithms to abstract away from the concrete data type in order to focus on learning *patterns* or *structure* independent of the type of data. The following paragraphs summarise some simple approaches to *vectorise* different kinds of data. We focus on the most common types of data encountered, structured data, as stored in tables or relational data basesd, comprising real valued data, ordinal data or categorical data, and unstructured data, such as text data and images. Many data sets contain heterogeneous types of features, maybe text and numbers or text, categorical data and real valued data. Using the feature extractors below one can transform each data type of a data set individually and then simply concatenate them into one bigger vector, maybe after normalizing the single data types. Maintaining such ML pipelines with complex feature aggregations can be drastically simplified using the above mentioned ML pipeline APIs.

4.1 Real Valued data

The most basic data type that we will deal with in this book are real valued data, that is variables that have real numbers as their value. The space of real numbers is usually denoted by $\mathbb{R}$. We will denote scalar variables (variables with just one real number) as

$$x \in \mathbb{R}^1 \tag{4.1}$$

and D-dimensional vectors of real valued variables as

$$\mathbf{x} \in \mathbb{R}^D = \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}. \tag{4.2}$$

4.2 Ordinal Data

If the data is not numerical but the attributes have some order and can be ranked they are called *ordinal data*. One can use their (normalized) ranking in order to vectorize the data. Consider the example where the *i*th data point has D ordinal features which can take on values *low, medium* and *high*, so $\mathbf{x}_i \in \{\text{low}, \text{medium}, \text{high}\}^D$. We can transform the $\mathbf{x}_{id}$th entry by

$$\mathbf{x}_{id} = \begin{cases} 1/3 & \text{if } \mathbf{x}_{id} = \text{low} \\ 2/3 & \text{if } \mathbf{x}_{id} = \text{medium} \\ 3/3 & \text{if } \mathbf{x}_{id} = \text{high} \end{cases} \tag{4.3}$$

4.3 *Categorical Data*

If there is no inherent order to the non-numerical attributes the data is called *categorical data*. Examples are for instance colour values (of course colors can be described as real numbers in standard SI-units, but that's not how they occur in most data sets) or gender attributes. If the *i*th data point has D categorical features which can take on $K = 3$ values *red*, *green* and *blue* we can transform the $\mathbf{x}_{id}$th entry to a K-dimensional binary vector $\mathbf{x}_{id} \in \{0, 1\}^K$

$$\mathbf{x}_{id} = \begin{cases} [1\ 0\ 0] & \text{if } \mathbf{x}_{id} = \text{red} \\ [0\ 1\ 0] & \text{if } \mathbf{x}_{id} = \text{green} \\ [0\ 0\ 1] & \text{if } \mathbf{x}_{id} = \text{blue} \end{cases} \tag{4.4}$$

Note that while in the above cases of numerical and ordinal data we get for each data point a vector, when transforming categorical data like in Equation 4.4 we obtain for each feature $\mathbf{x}_{id}$ a vector, not a single number. Each categorical feature that can take one out ouf K values is transformed into a K-dimensional binary vector and thus the *i*th data point becomes a matrix $\mathbf{X} \in \mathbb{R}^{D \times K}$ (see Table 4.1). In many application settings this matrix can be "flattened" into a DK-dimensional vector. An overview of the different vectorisation strategies can be found in Table 4.1.

	Numerical	Ordinal	Categorical
Raw Data	$\begin{pmatrix} 0.4 \\ 1.8 \\ -0.35 \\ -0.7 \end{pmatrix}$	$\begin{pmatrix} \text{low} \\ \text{medium} \\ \text{low} \\ \text{high} \end{pmatrix}$	$\begin{pmatrix} \text{red} \\ \text{blue} \\ \text{green} \\ \text{green} \end{pmatrix}$
Vectorized	$\mathbf{x}_i = \begin{pmatrix} 0.4 \\ 1.8 \\ -0.35 \\ -0.7 \end{pmatrix}$	$\mathbf{x}_i = \begin{pmatrix} 1/3 \\ 2/3 \\ 1/3 \\ 3/3 \end{pmatrix}$	$\mathbf{X}_i = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Table 4.1: Common types of raw data and their transformed counterparts that can be used directly in all algorithms discussed in later chapters.

4.4 Image data

Numerical or categorical data, stored in tabular form or relational data bases, is often referred to as *structured data*. These types of data can be straightforwardly analyzed with the models presented in this book, with minimal preprocessing using simple feature extractors for normalization or one-hot-encoding. But methods from classical statistics can deal with such simple data types, too. One of the main strengths of ML is that it allows to extract useful information from *unstructured data* as well. Such unstructured data types are for example text and images. Following the idea of ML we just need to apply the right feature extractor for these unstructured data sources and then apply a generic ML model to those features. The better the feature extractor, the simpler the ML model can be.

For images, until 2012 researchers used handcrafted feature extractors that essentially decomposed the images into its spatial frequencies, often in patches of the entire image. This technique still works well in some application scenarios, mainly in scenarios where images are relatively standardized. For instance in a product catalog of an online retailer, the objects are always depicted on a white background in the image center. Here simple oldschool image features work often well.

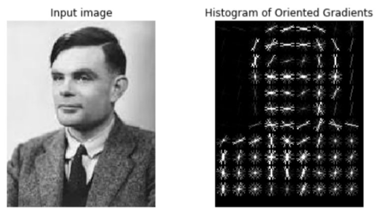


Figure 4.1: Example of old-school (before neural networks) computer vision features, taken from the `https://scikit-image.org` documentation.

Histograms of Oriented Gradients One of these classical techniques is the histograms-of-oriented-gradients feature extractor [?]. It first splits the image in subpatches and in each patch extracts the strength of a number of spatial frequencies corresponding to edges centered in the image patch. An illustration, adopted from `https://scikit-image.org`, is shown in Figure 4.1, on the right the orginal image is shown, on the right the histograms of oriented edges in subpatches. These edge histograms can then be used in downstream ML models to detect shape patterns in an image.

Ironically the hardware improvements that gave rise to the renaissance of neural networks were primarily driven by the need of gamers for faster graphics processing units (GPUs), which coincidentally happen to accelerate not only rendering of games but also the matrix vector algebra operations needed for fast learning in artificial neural networks.

Convolutional Neural Networks After 2012 and the seminal *AlexNet* paper presented at NeurIPS [?] revolutionized the way images were analyzed by improving the state of the art in image recognition substantially using deep convolutional neural networks. The idea of convolutional neural networks was proposed much earlier by Yann LeCun [?]. But the success of these types of neural networks took more than the biologically inspired architecture of convolutional neural networks. Only with a combination of deeper architectures, faster hardware, a number of software innovations, such as automatic differentiation and modular neural network toolkits for training neural networks on GPUs, and last but not least well curated and large data sets, neural networks finally took over the lead in all image recognition high score charts. And that has not changed since then. Because the data set used in the AlexNet paper was ImageNet [**?**] the starting point of the neural network renaissance is often called the *ImageNet moment*.

Training such deep neural networks is computationally challenging, and as many researchers train their networks on the same data set anyways, ImageNet, it has become standard practice to not retrain a deep convolutional network, but to just take an off the shelf model trained by researchers that have a lot of compute at their disposal and make their trained models available. These models can then be used as omnipurpose feature extractors for other tasks than object recognition on ImageNet data. One usually just extracts the network activations in a layer close to the output layer of the network as features. These features are often good enough for other tasks and while extracting them can take some time without GPUs, it is still probably the best option as retraining a deep neural network on your new data set would be infeasible in that case; and besides oldschool computer vision features would probably take similarly long to compute and result in worse features. While the details of training deep convolutional networks are beyond the scope of the current version of this book, the extraction of these features is a standard procedure well covered in the tutorials of all major deep learning libraries[1].

[1] See for instance https://keras.io/applications/

4.5 Text data

Probably one of the most often encountered data types that data scientists have to deal with is text data. While many younger researchers tend to focus on neural network based text processing, especially after some have claimed that the *imagenet moment for natural language processing (NLP) has arrived*[2], there is an extremely powerful feature extractor for text data which often achieves comparable results with neural networks at a fraction of the computational cost: Bag-of-Words (BoW) feature extractors. The idea is very simple, BoW feature extractors merely count the occurrence of tokens in a text. In its simplest form BoW feature extractors assume tokens are words. This only requires to split each text into separate words and count how often each word occurs in each document. In some languages, like Japanese, this tokenization is difficult, but in many languages, including European languages, splitting at punctuation and blanks often results in decent tokens. Consider for instance the following corpus of two documents (here containing just a single sentence each):

[2] https://ruder.io/nlp-imagenet/

Word histograms, or in general Bag-of-Words (BoW), are simple but competitive feature extractors for text data.

1. *document one is great*

2. *document two is mediocre*

The dictionary of all words in this corpus would be *document, one, two, is, great, mediocre*, so the BoW feature space would be of dimensionality six and for each document it would contain the count for the respective word: Usually the number of tokens is very large, but each

Table 4.2: Bag-of-Words feature vector for the two documents in section 4.5

	document	one	two	is	great	mediocre
document 1	1	1	0	1	1	0
document 2	0	1	1	1	0	1

individual sentence contains only a very small subset of those tokens. Some words, like *the, and, he* occur very often, but do not convey a lot of meaning. Other words that occur infrequently convey more meaning. So most of the entries in real world natural language BoW features are very high dimensional vectors with very few entries being non zero, in other words, the vectors are *sparse*. For this type of data, it is ideal to account for that sparsity with dedicated data structures. This will dramatically speed up all computations.

One of the challenges when using BoW features is that most of the language structure is lost. When only counting single words, it is for instance difficult to resolve things like negation, not to speak of more complex linguistic features, like sarcasm. There is a simple method to alleviate some of these problems and this is commonly referred to as *ngram Bag-of-Word features*. In the above example we only counted single words, our tokens were words. But we can define our tokens to be not single words but rather sequences of two words. For sequences of length two, meaning the n of the *ngram* would be two, we would then extract bigrams, for sequences of length three trigrams and so forth. The largest collection of ngrams went up to $n = 5$ and was released by Google[3].

[3] `https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html`

So in the example above when using bigrams (n=2) the dictionary of all token sequences up to length two would be the dictionary of the unigrams *document, one, one, two, is, great, mediocre* plus the dictionary of all length two sequences *document one, document two, one is, two is, is great, is mediocre* and the count vector would have 12 dimensions.

Space and Time Complexity of BoW Features The dimensionality of an ngram BoW feature space in the worst case is V^n. But usually not all syntactically possible ngrams actually occur or are semantically/grammatically correct. And while the feature space can become quite large, each single document only has a very small fraction of non zero entries, which renders computations with ngram BoW feature vectors very efficient when using appropriate data structures. Another important aspect is the time it takes to extract BoW features. The simplest implementation uses two passes through the entire data set: One pass to find all tokens or ngrams to build up the dictionary, and another pass to count the occurrences. This can be implemented more efficiently, using parallelization and distributed infrastructure. But for all datasets below a couple ten thousand documents, any mobile phone can easily run this two pass implementation fairly quickly. There is still the requirement for large dictionaries, but often one only accounts for the top 1,000,000 words or so. Another alternative that works very well and does not require maintaining a dictionary is to use a hash function to map strings/tokens to feature space indices[4]. This allows to set the amount of memory required to store the feature vectors upfront, at the expense of allowing for hash collisions. This approach is not only a lot faster than standard BoW computations, it also does not have a state (the dictionary) and hence does not require that first pass through the data to compute the dictionary. But the disadvantage of allowing for hash collisions also means that it is difficult to map the features extracted back to the actual meaning of the feature dimensions.

[4] See for instance `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html`

5

Evaluation Metrics

In order to train ML models one needs an evaluation metric that can be optimised. Often the evaluation metric is directly optimised in the objective function of the ML algorithm. But in many metrics of interest are difficult to optimise directly. In the following some evaluation metrics are highlighted that can be useful to look at different aspects of ML model performance.

False positives and true positives In order to define some standard performance measures, it is helpful to first introduce the notions of *True Positives* (TP), *False Positives* (FP), *True Negatives* (TN) and *False Negatives* (FN). These definitions are explained in Table 5.1 and reflect how many of the true positive (respectively negative) labels are correctly detected or missed. For multi-class problems these definitions can be useful, too, by remembering that every multiclass problem can be formulated as a one-class-vs-rest binary classification problem.

Table 5.1: Types of misclassifications and correct classifications.

		True Class	
		Positive	**Negative**
Predicted Class	**Positive**	True Positive (TP)	False Positive (FP)
	Negative	False Positive (FP)	True Negative (TN)

Accuracy The accuracy of a classifier is the simplest measure of performance. It is computed as the percentage of correctly classified data points

$$\text{accuracy} = \frac{TP + TN}{N}, \tag{5.1}$$

where N is the number of all data points.

Sensitivity The *sensitivity* of a classifier is defined as the ratio of the number of correctly identified positive labels divided by the total number of true positive labels. The total number of true positive labels is equal to the number of predicted positive labels plus the number of not detected positive cases, i.e. the false negatives

$$\text{sensitivity} = \frac{TP}{TP + FN} \tag{5.2}$$

Sensitivity can be interpreted as the probability that a classifier will return a positive label, given that the true label is positive. In information retrieval sensitivity is often called *recall*. Classifiers with high sensitivity have a low *type II error*, meaning this classifier rarely misses true positives.

Specificity The *specificity* of a binary classifier is defined as the ratio between the number of correctly identified negative labels divided by the total number of negative labels. The total number of negative labels is equal to the number of predicted negative labels plus the number of not detected negative cases, i.e. the false positives

$$\text{specificity} = \frac{TN}{TN + FP} \tag{5.3}$$

Specificity can be interpreted as the probability that a classifier will return a negative label, given that the true label is negative. Classifiers with high specificity have a low *type I error*, meaning that they can be considered reliable when they predict a positive label.

Precision The *precision* of a classifier is defined as the ratio of the number of correctly identified positive labels divided by the total number of positive predictions made by the classifier, i.e. including the false positive ones

$$\text{precision} = \frac{TP}{TP + FP} \tag{5.4}$$

Often the *precision* is also called the *positive predictive value*. It can be interpreted as the probability of a true positive prediction given that a classifier returns a positive prediction.

F-Score One way to integrate the above measures is the *F-Score* (sometimes called *F1-Score* or *F-Measure*) of a classifier. It is defined as the harmonic mean of precision (Equation 5.4) and recall (or sensitivity, Equation 5.2)

$$\text{F-Score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{5.5}$$

The F-Score is between 1 and 0, where a perfect classifier reaches a score of 1.

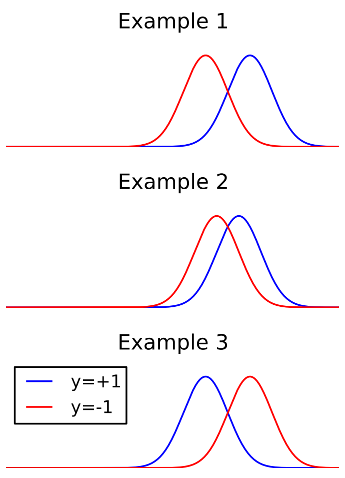


Figure 5.1: Toy data histograms of classifier outputs. Depending on the distribution of classifier outputs, the ROC curves in Figure 5.2 have different shapes.

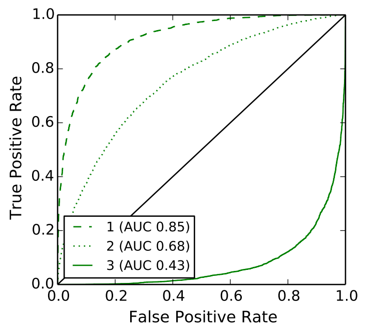


Figure 5.2: The receiver-operator characteristic curve shows the relationship between false positive rate (x-axis) and true positive rate (y-axis) of a binary classifier. Three examples of classifier outputs illustrated in Figure 5.1 correspond to the different ROC curves.

Receiver-Operator Characteristics (ROC)

Another useful way of illustrating the relationship between false positive rate and true positive rate is the *Receiver-Operator Characteristic* (ROC) curve. It simply shows the false positive rate on the x-axis and the true positive rate on the y-axis, see Figure 5.2.

Thresholding the outputs of a binary classifier (or a univariate signal in general), one can trade off false positives against true positives. Depending on the distribution of a classifier's output, the shape of the ROC curve differs. If the curve rises steeply, the classifier is always right. An example distribution of classifier outputs is shown in Figure 5.1, example 1. If the curve rises only for very large false positive rates the classifier is just as good – it consistently estimates negative labels for positive examples, see example 3 in Figure 5.1. If the curve is getting closer to a straight line, the performance of the classifier becomes worse, as illustrated in example 2 in Figure 5.1.

6

Model selection

For an optimal ML system, there are many choices one has to make. First, choosing the right ML algorithm can be difficult. While there are some guidelines based on the data at hand, see also section 6.3, there are many situations in which it is hard to decide which ML algorithm to use. Selecting the type of ML algorithm is often not enough. Many ML algorithms require the user to specify what is often referred to as *hyper parameters*. Those are parameters that are usually not learned during the training phase of an ML algorithm, they are more like specifications of the ML model. Some for instance constrain the complexity of the ML model, so called *regularizers*, more on those below. And besides the type of ML model and the optimal hyper parameters one also has to choose some parameters for the feature extractors, see chapter 4. For text data an important parameter can be how many words should be considered in a bag-of-words feature extractor. All these choices, ML algorithm, its hyper parameters and the parameters of the feature extractors, are important for a successful ML system. Sometimes all these choices together, model type, model hyper parameters and feature extractor parameters are referred to as the *hyper parameters* of a ML system. And they can be difficult to set. The process of automatically learning the right hyper parameters of an ML system is often called *hyper parameter optimization* (HPO), model selection or *AutoML*. Before highlighting some of these methods, let us first look at why model selection is important.

6.1 Overfitting

Whether or not the accuracy in eq. 5.1 or any other performance measure is a meaningful quantity depends critically on the data set that it has been computed on. In a classification task, if the data set for instance contains features that are not related to a certain class but just happen to be correlated with a class because of the small size of the data set, or the way it was sampled, this could lead astray the ML model. More concretely imagine in an object classification task that in the training data set all images that have the label *contains horse* have a watermark of a horse photographer, then the ML model might learn that if an image contains that watermark, then there is a horse on depicted [Lapuschkin et al., 2019]. This is not too bad a guess, given only that training data. And all metrics computed with that model on that very same data will tell us that it is a very good model. But the watermark obviously has nothing to do with horses. And if we would apply such a ML model to new data, which do not contain that watermark, then the model would not be capable of detecting horses. The ML model has just *overfitted* to the training data. Or in other words the model does not *generalize* well beyond the training data. Overfitting is one of the biggest challenges in ML. And it can happen in many ways, for all types of models.

So in essence we want to obtain ML models that generalize well beyond training data and do not overfit to peculiarities of our training data, which will always limited in its representativeness of the true mechanisms of the underlying data generation process. For an intuitive understanding of overfitting it can be helpful to remember how we learned in school. Some teachers just asked literally the same questions in exams as in class. It was easy for us to memorize the answers and just repeat them in the exam. Other teachers asked questions in the exams that were not directly covered in class, but which required some transfer of knowledge to new situations. Memorizing answers did not help in exams of those teachers, we needed to understand the problem to give the right answer. We needed to *generalize* from the training data in class.

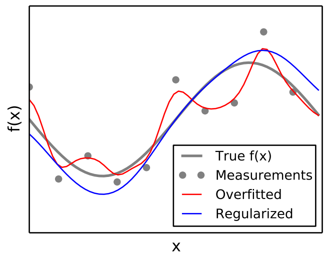


Figure 6.1: Overfitting example with chapter 23. The grey line shows the true function $f(x) = sin(x)$, grey dots are some noisy measurements of this function. Training an overly complex model on these few noisy measurements will learn an *overfitted* approximation of the true function that captures too much noise (red line). The blue line corresponds to a model that is less complex and captures the true function better.

Regularization If we want our ML models to generalize well to new unseen data, it is useful to constrain the complexity of those models. The intuition behind that is that many ML models can in principle memorize any data set, which will result in poor generalization. Controlling the complexity will help to better calibrate the models and overfit less to the training data.

There are many different forms of regularization. Some of the models presented in this book come with the option of regularization. But for most models tuning the regularization is difficult. Regularization parameters are amongst the most important hyper parameters that need to be optimized for a well functioning ML system.

6.2 *Hyper Parameter Optimization*

All hyper parameters of a ML system, which in the broader sense comprises the choice of the ML model and its regularizers as well as the feature extractors and their parameters, should be tuned. The field of

AutoML [Feurer et al., 2015] is an active area of research and its vision, fully automated ML systems, is very appealing, even if it probably implies that many data scientists would loose their job. There are a variety of methods to automatically tune hyper parameters which can be broadly categorized as

- *Grid Search*: All combinations of hyper parameters (the grid) are tested in a trial and error manner
- *Random Search*: A number of random samples from the hyper parameter space are tested [Bergstra and Bengio, 2012]
- *(Bayesian) Global Optimization*: A (bayesian/probabilistic) ML model is used to predict the best hyper parameters [Snoek et al., 2012].

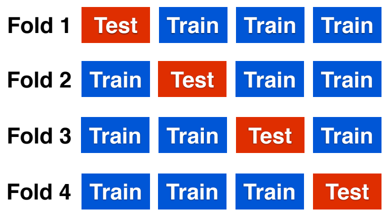


Figure 6.2: *K*-Fold Cross-Validation for model evaluation. First, split your data set into four equally sized random subsets. In each of the k iterations the ML model is trained on the training data set (blue) and tested on the test data set (red). Training and testing the data on non-overlapping data splits we can obtain a good estimate of the true performance of an algorithm, assuming that training and test data are statistically independent.

Cross-Validation Independent of the type of model selection or hyper parameter optimization procedure, there is one concept that is common to all of them: *cross-validation*. It refers to the simple idea to simulate the real world testing scenario of an ML model, when it is applied to data it has not seen during training. When used in real world applications one is mostly interested in a model that predicts new, unseen data well. The idea of cross-validation is to split the data in two or more parts, also called *cross-validation folds* and keep one part of the data out of the training process. That data can then be used to estimate how well the trained ML model would have done on new data that it has not seen during training.

Cross-Validation for Model Evaluation In its simplest form cross-validation is used only for evaluating a ML system by splitting of a small part of the entire data set, the *test set*, for evaluation of the ML system. That test set is often around 10% to 20% of the entire data. Because that one test set might have some peculiarities that are not representative for the entire data, one can use different parts of the data to test the model. If one would like to test the model k times one simply splits the data into k equally sized random subsets and tests the machine learning model on one of these k test data sets when it was trained on the rest of the data, all other $k - 1$ parts of the data. This is called $k - foldcross - validation$. If one sets k to be the size of the data set then each test set contains only one data point, this is called *leave-one-out cross-validation*. While it can be computationally challenging to evaluate models with k-fold cross-validation, the procedure can be easily parallelized and helps to assess the true generalization performance of a ML system much better than doing just a single split and considering just a single test data set.

Cross-Validation for Model Selection and Evaluation The above procedure can be used to train a ML system for which all model choices were

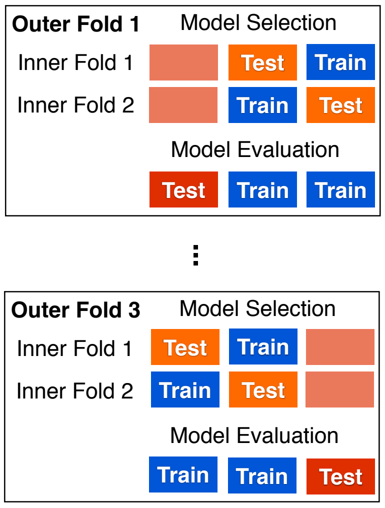


Figure 6.3: Nested Cross-Validation for model selection and model evaluation. For model evaluation an outer cross-validation loop tests the performance of the ML pipeline. The inner cross-validation loop selects the optimal hyper parameters.

made and all hyper parameters are fixed. If one needs to tune the hyper parameters with any of the aforementioned techniques, then a slightly different version of cross-validation is used. One still has a test data set that is not touched neither for training each individual ML pipeline, nor for optimizing hyper parameters. But one splits off another part from the training data, which is used to do model selection. That third split is often referred to as *validation data set*. Often one chooses around the same size for the validation data set as for the test data set.

For the simplest case of model selection with grid search, one can use then the training data to train each ML pipeline for each hyper parameter candidate. A hyper parameter candidate is for instance a certain type of ML model with a certain regularization parameter and a certain feature extractor with some parameters. Each of those ML models is then evaluated on the validation data. After all hyper parameter candidates were evaluated, one chooses the ML pipeline that performed best on the validation set and trains it on the training data and the validation data. The resulting model is then evaluated on the test data set.

Again, due to the random split for the test data, the estimate of how well a ML pipeline with optimized hyper parameters generalizes could be over- or underestimated. If one would like to do k-fold cross-validation to obtain a more robust estimate of the generalization performance, one can resort to *nested cross-validation*. One simply performs a standard k-fold cross-validation for evaluating each ML pipeline (the *outer cross-validation loop*), but for each training data set the hyper parameters are optimized by doing another $(k-1)$-fold cross-validation, the *inner cross-validation loop*.

6.3 Which method to choose

Not every choice for an ML system needs to be made with hyper parameter optimization. Some choices are determined by the type of data, for instance if there are no labels available then supervised methods cannot be used, or by constraints of the application scenario, for instance if the model is run on a small device with little memory complex ML models can often not be deployed. The diagram in Table 6.1 lists some of the classification models presented in this book and their properties that could be relevant in applications.

Table 6.1: Table summarizing some aspects of classification methods.

Method	Multiclass	Interpretable		Time			Memory Demands	Model Complexity
		Output	Model	Training	Retraining	Prediction		
Nearest Centroid	●	●	●	·	·	·	·	·
Naive Bayes	●	●	●	·	·	·	·	·
Linear Discriminant	●	●	●	•	•	·	·	•
Logistic Regression	●	●	·	·	·	·	·	•
Linear SVM	●	·	·	·	·	·	·	•
Non-Linear SVM	·	·	·	●	●	●	●	●
k-NN	●	·	·			●	●	●
Neural Networks	●	·	·	●	●	•	●	●

In order to chose the right method or model for a given data set there are a few questions one needs to answer. The decision tree in Figure 6.4 illustrates which questions we need to answer in order to decide which is the right algorithm for analysing a given data set. The most important question is: Do we have labels for our data or not? If there are labels, we have a **supervised learning** scenario.

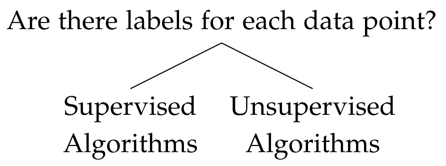


Figure 6.4: Decision tree showing how to choose the appropriate algorithm depending on data set properties.

Labels can be the category of a data point. An example of binary categories would be *spam-mail* or *non-spam mail*. For categorical variables we can use *classification methods*. Labels can also be continuous rather than categorical. For continuous labels we have a *regression setting*. If there are no labels, we have an **unsupervised learning** scenario.

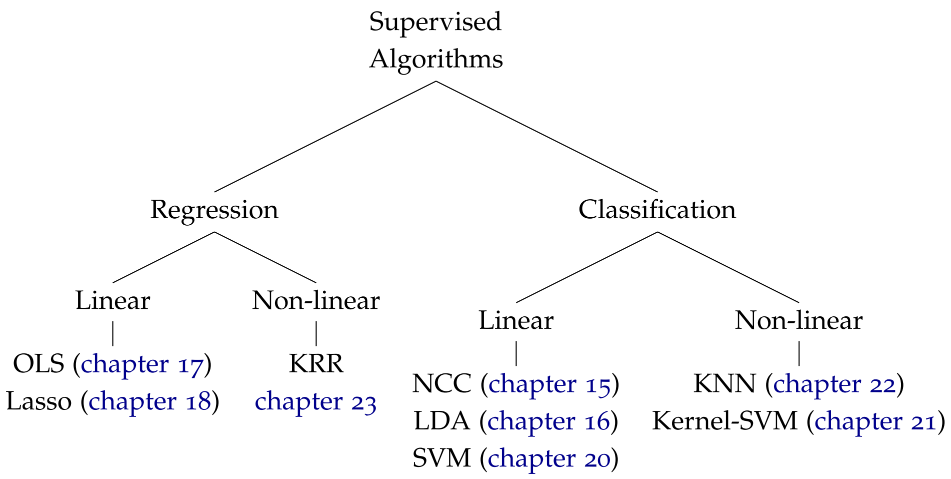


Figure 6.5: Decision tree showing how to choose the appropriate algorithm depending on data set properties.

The next question we need to answer for both unsupervised and supervised learning problems is whether the dependencies in the data are linear or non-linear. Examples for linear dependencies are shown in Figure 7.1 and examples for non-linear dependencies are shown in Figure 8.2. While it is clear in these toy data examples which kind of dependencies the data set exhibits, in general this is not easy to see. An easy solution is to do model selection and simply try out both kind of methods, linear and non-linear ones, on a subset of your data and check which one describes the data best. Non-linear methods can do anything linear models can do. So if linear models perform as good as non-linear ones, then it is unlikely that there is useful non-linear

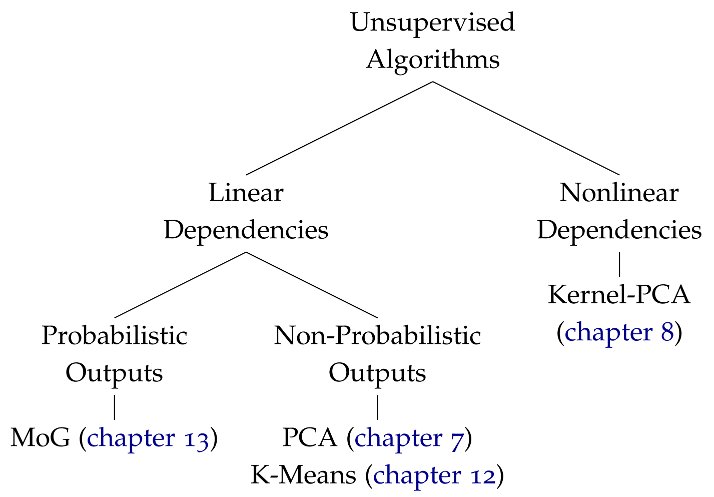


Figure 6.6: Decision tree showing how to choose the appropriate algorithm depending on data set properties.

structure in the data.

Another potentially relevant question is whether the output of the model should be probabilistic or not. What this usually refers to that probabilistic models can return an estimate of their own uncertainty about their prediction. Probabilistic models are often more challenging to design and train, but their ability to not only make predictions but also return confidence intervals for their predictions can be tremendously useful.

III
Unsupervised Methods

7

Principal Component Analysis (PCA)

APPLICATIONS Dimension reduction, Denoising, Visualization

OBJECTIVE Principal component analysis[1] (PCA) tries to find a new representation of the data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$ (i.e. a *subspace* spanned by $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_K] \in \mathbb{R}^{D \times K}$ of the original data space) in which the variance of the data is maximized. This can be formulated as

$$\operatorname*{argmax}_{\mathbf{W}} \operatorname{Tr}\left(\mathbf{W}^\top \mathbf{X}\mathbf{X}^\top \mathbf{W}\right), \qquad \text{subject to } \mathbf{W}^\top \mathbf{W} = \mathbf{I} \tag{7.1}$$

[1] K Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2:559–572, 1901

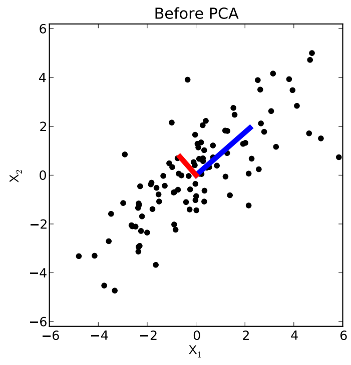


Figure 7.1: Data before PCA: Two-dimensional data drawn from a standard normal distribution; first principal direction is plotted in blue, second principal direction is plotted in red.

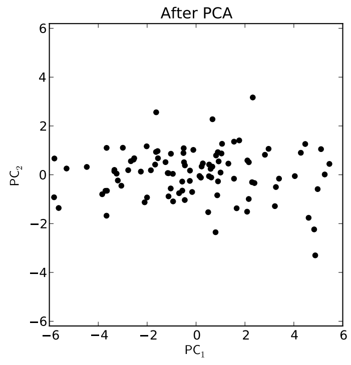


Figure 7.2: Data after PCA Same data projected onto principal axes; directions of maximal variance are now aligned with standard euclidean basis.

The PCA solution $\mathbf{W}$ are the eigenvectors of the covariance matrix $\mathbf{X}\mathbf{X}^\top$. Once this new representation of the data is found, projecting the data into the PCA subspace allows to discard or retain certain aspects of the data. For instance, discarding low variance components can *denoise* a data set, assuming that noise has lower variance than the signal of interest. In different settings the noise can have a much higher variance than the signal of interest and the high variance components can be discarded (*artifact removal*). In either case PCA is used to *reduce the dimensionality* of a data set. This is done in three steps.

1. Compute principal directions of variance $\mathbf{W}$

2. Project the data on a selection of columns of $\mathbf{W}$

3. Reconstruct data in original data space

PCA is the workhorse of data analysists and probably the most often used tool (next to heuristics-based feature selection) to reduce the dimensionality of a data set.

PSEUDOCODE Algorithm 1 shows how to compute the principal directions of variance $\mathbf{W}$. The principal directions $\mathbf{w}_i$ are typically sorted according to the magnitude of their corresponding eigenvalues in line 4 of algorithm 1. So the first principal direction is that direction along which the data has maximal variance. The directions are orthogonal; that is, $\mathbf{w}_i^\top \mathbf{w}_j = 0$ if $i \neq j$ and thus $\mathbf{W}^\top \mathbf{W} = I$, which will come in handy for a number of computations. Using a selection of the principal directions (columns of $\mathbf{W}$), we can compute the low-dimensional representations of the data points $\hat{\mathbf{X}}$ as shown in Algorithm 1, line 5. In the case of dimensionality reduction we would select the first M columns of $\mathbf{W}$ to retain only the directions with maximal variance.

Algorithm 1 Compute Principal Directions of Variance

```
Input: data matrix $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{D\times N}$
Output: principal directions $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_K] \in \mathbb{R}^{D\times K}$,
    principal components $\hat{\mathbf{X}}$, variance in components $\Lambda$
1: Compute mean $\bar{\mathbf{x}} \leftarrow \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i$.
2: Subtract mean $\forall i : \mathbf{x}_i \leftarrow \mathbf{x}_i - \bar{\mathbf{x}}$.
3: Compute covariance matrix $\mathbf{C} = 1/N\ \mathbf{X}\mathbf{X}^\top$.
4: Solve eigenvalue equation $\mathbf{C}\mathbf{W} = \Lambda\mathbf{W}$.
5: Compute principal components $\hat{\mathbf{X}} = \mathbf{W}^\top\mathbf{X}$.
```

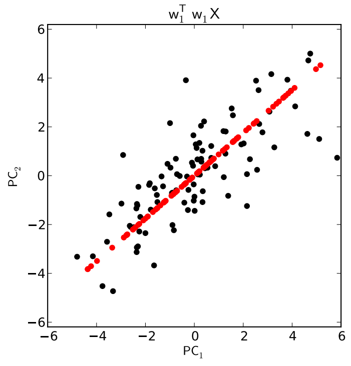


Figure 7.3: The original data (black dots) projected onto the first principal component ($\mathbf{w}_1^\top \mathbf{x}$) falls on a line. The approximation of the original data using only the first principal component can be visualised by computing $\mathbf{w}_1\mathbf{w}_1^\top\mathbf{X}$ (red dots)

Finally, one may want to reconstruct the de-noised data points in the original space (for example with images). This is accomplished by Algorithm 2.

Algorithm 2 Reconstructing projected data in original space

```
Input: transformed data points $\hat{\mathbf{X}} \in \mathbb{R}^M$, principal directions $\mathbf{W}$
Output: reconstructed data points $\mathbf{X} = [\mathbf{x}_i, \dots, \mathbf{x}_N] \in \mathbb{R}^D$.
1: $\mathbf{X} \leftarrow \mathbf{W}\hat{\mathbf{X}}$
```

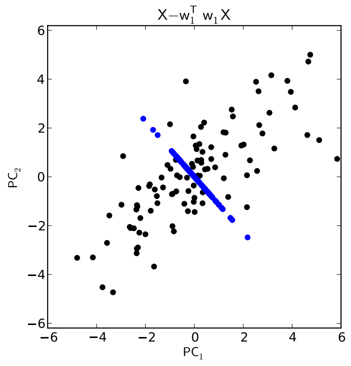


Figure 7.4: Approximation of the original data using only the second strongest principal component $\mathbf{w}_2$, using the same procedure as in Figure 7.3. Note that the line described by $\mathbf{w}_2$ is orthogonal to that of the first principal component.

DISCUSSION The number of principal components m used in PCA has to be supplied by the user. Fewer components means larger reconstruction error. Often m is determined by choosing the percentage of variance (of the data $\mathbf{X}$) explained by the PCA approximation. The variance explained by the ith component is $\mathbf{w}_i^\top\mathbf{X}\mathbf{X}^\top\mathbf{w}_i$, or $\Lambda_{i,i}$ the ith eigenvalue of 4. The total variance is the sum of all $\Lambda_{i,i}$. Hence $(1 - \sum_{i=1}^{m}\Lambda_{i,i}/\sum_{j=1}^{d}\Lambda_{i,i})$ is the variance explained by the strongest m components. The reconstruction error is the sum of the remaining eigenvalues $\Lambda_{m+1,m+1}, \dots, \Lambda_{d,d}$. As PCA is optimizing a quadratic objective function, the algorithm is not very robust against outliers.

EXAMPLE Figures 7.1 shows a two dimensional data set and the corresponding first and second principal directions. Figure 7.2 shows the same data after projecting in onto the first two principal directions. As a real world application example one can apply PCA to images of faces in order to extract main directions of variance in pixel space[2]. These so called *eigenfaces* can be used for face recognition.

[2] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991

SEE ALSO Kernel PCA (chapter 8) (a non-linear extension of PCA)

8

Kernel PCA

APPLICATIONS Nonlinear Dimension reduction, Denoising

OBJECTIVE There are cases when computing PCA as in chapter 7 does not yield good solutions. Two examples are:

1. *The data has more dimensions than samples.* In this case the covariance matrix in Equation 7.1 is difficult to estimate.

2. *The data is not well described by linear components.* This is illustrated in Figure 8.1. The relevant structure is captured by circles which are best described by their radius, or the distance of the data from the center, $\sqrt{\mathbf{x}^\top \mathbf{x}}$.

The first case is discussed in section A.3. The more interesting case, non-linear dimension reduction is solved in kernel PCA[1]with the *kernel trick* (section A.1). In Figure 8.1 it is easy to see which non-linear structure is capturing the two components in the data well. In high-dimensional data sets this is not that easy. Looking at the data after projecting them through the kernel PCA map $\phi^(\mathbf{x})$ can help to see the underlying structure in the data. Just like in standard PCA kernel PCA maximizes the variance of the data, but after the mapping $\phi^(\mathbf{x})$, so we need to optimize

[1] B Schölkopf, A J Smola, and KR Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(6):1299–1319, 1998

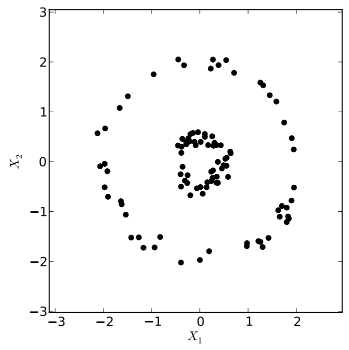


Figure 8.1: When linear PCA fails: The data is not well described by lines but the two components fall on a manifold that is described by two equations $x^\top x = a$ and $x^\top x = b$

$$\underset{\phi}{\operatorname{argmax}} \operatorname{Tr}\left(\mathbf{A}^\top \phi(\mathbf{X})\phi(\mathbf{X})^\top \mathbf{A}\right), \qquad s.t.\ \mathbf{A}^\top\mathbf{A} = \mathbf{I} \tag{8.1}$$

Using the kernel trick one does not need to specify nor evaluate ϕ. All we need is a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ that computes $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$, the inner product of $\mathbf{x}_i$, $\mathbf{x}_j$ in kernel feature space. For some examples of kernel functions $k(.,.)$ see section A.2. The optimal mapping $\phi^*(\mathbf{x}_i)$ onto the curves of maximal variance is then obtained by

$$\phi^*(\mathbf{x}_i) = \sum_{n \neq i} k(\mathbf{x}_i, \mathbf{x}_n)\boldsymbol{\alpha}_n = \mathbf{K}_{[i,n\neq i]}\mathbf{A} \tag{8.2}$$

where $\mathbf{K}$ is the kernel matrix with $\mathbf{K}_{ij} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j)$ and $\mathbf{K}_{[i,n\neq i]}$ is the ith row of $\mathbf{K}$. The matrix $\mathbf{A} = [\mathbf{a}_1, \dots, \mathbf{a}_k]$ contains the k

eigenvectors of the eigenvalue equation

$$\mathbf{KA} = \Lambda\mathbf{A} \tag{8.3}$$

PSEUDOCODE The algorithm for estimating the kernel PCA model is described in algorithm 3, the algorithm for projecting new data onto the curves of maximal variance is described in algorithm 4. For kernel functions see section A.2.

Algorithm 3 Kernel Principal Component Analysis – Training

```
Input: Data $\mathbf{X} \in \mathbb{R}^{D\times N}$, kernel function $k(.,.)$
Output: Dual coefficients for kernel expansion $\mathbf{A}$
1: # Compute kernel Matrix
2: $\mathbf{K} = k(\mathbf{X},\mathbf{X})$
3: # Compute eigenvectors
4: $\mathbf{A} = \mathrm{eig}(\mathbf{K})$
```

Algorithm 4 Kernel Principal Component Analysis – Prediction

```
Input: $\mathbf{X}_{\mathrm{Test}}$, $\mathbf{X}_{\mathrm{Train}}$, $\mathbf{A}$, kernel function $k(.,.)$
Output: Data projected on non-linear principal components $\hat{\mathbf{X}}$
1: $\hat{\mathbf{X}} = k(\mathbf{X}_{\mathrm{Test}},\mathbf{X}_{\mathrm{Train}})\mathbf{A}$
```

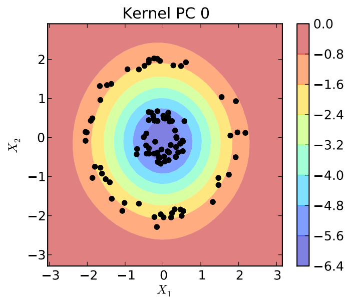


Figure 8.2: Visualisation of first kernel principal component $\phi^*(\mathbf{x})$ learned on the data in Figure 8.1. Background color indicates the output of $\phi^*(\mathbf{x})$. Note that KPCA learned to project the data onto circles around the origin.

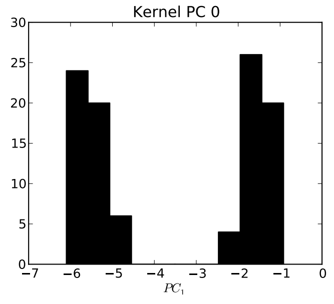


Figure 8.3: Same data as in Figure 8.1 mapped onto the first principal curve $\phi^*(\mathbf{x})$ illustrated in Figure 8.2. The two circles are nicely separated in this representation.

DISCUSSION Kernel PCA is just one of many non-linear dimension reduction or manifold learning techniques. Some alternatives are Local Linear Embedding [Roweis and Saul, 2000], Laplacian Eigenmaps (also called Spectral Clustering, for a review see e.g. [von Luxburg, 2007]) or Multidimensional Scaling (used e.g. in ISOMAP [Tenenbaum et al., 2000]). One advantage of KPCA is that it is easy to implement. A major drawback of KPCA with nonlinear kernels is: It can be hard to interpret. This is easier in the case of linear KPCA. In cases where the dimensionality of the data is much larger than the number of samples, we can use a linear kernel and compute the same solution as standard PCA, but faster. For a discussion of the relationship between eigenvectors of the kernel matrix and the covariance matrix in the linear case see section A.3. A real world application example of non-linear kernel PCA is denoising of images [Mika et al., 1999].

EXAMPLE Figure 8.2 shows a two dimensional data set. We extracted only the first principal curve using a Gaussian kernel with kernel width $\sigma = 1$ and plotted the output of $\phi^*(\mathbf{x})$ for each data point as the background color. Note that KPCA learned that the relevant structure here are circles. Figure 7.2 shows the same data after mapping it through $\phi^*(\mathbf{x})$ in onto the first two principal directions.

SEE ALSO chapter 7, chapter 12, Appendix A

9

Non-negative Matrix Factorization

APPLICATIONS Dimensionality reduction for non-negative data

OBJECTIVE Often data matrices contain only positive values. Examples are images or probabilistic matrices, meaning $\forall i, j: \; 0 < \mathbf{X}_{ij} < 1$, all entries are between zero and one. In these cases a factorization using PCA (see chapter 7) or other matrix factorizations yields results can be difficult to interpret: Some of the factors will always have negative values. In order to obtain more meaningful results Lee and Seung proposed *non-negative matrix factorization* (NMF)[1]. Given non-negative data $\mathbf{X} \in \mathbb{R}_+^{D \times N}$ we want to find $\mathbf{W} \in \mathbb{R}_+^{D \times C}$, $\mathbf{H} \in \mathbb{R}_+^{C \times N}$ such that

[1] D.D. Lee and H.S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–91, 1999

$$\underset{\mathbf{W},\mathbf{H}}{\operatorname{argmin}} \; ||\mathbf{X} - \mathbf{WH}||_{\text{Fro}}^2 \tag{9.1}$$

Gradient descent finds an optimal solution to eq. 9.1 by iterating

$$\begin{aligned} \mathbf{H} &\leftarrow \mathbf{H} - \eta \left(\mathbf{W}^\top \mathbf{WH} - \mathbf{W}^\top \mathbf{X} \right) \\ \mathbf{W} &\leftarrow \mathbf{W} - \eta \left(\mathbf{WHH}^\top - \mathbf{XH}^\top \right) \end{aligned} \tag{9.2}$$

By setting the learning rate $\eta = \mathbf{H} \oslash (\mathbf{W}^\top \mathbf{WH})$ these additive update rules can be transformed into *multiplicative updates*

$$\mathbf{H} = \mathbf{H} \odot \mathbf{W}^\top \mathbf{X} \oslash \mathbf{W}^\top \mathbf{WH} \tag{9.3}$$
$$\mathbf{W} = \mathbf{W} \odot \mathbf{XH}^\top \oslash \mathbf{WHH}^\top \tag{9.4}$$

where $\odot$ is *element-wise* multiplication $\oslash$ is *element-wise* division. The advantage of multiplicative updates is that if the original data matrix $\mathbf{X}$ was positive and the matrices $\mathbf{W}, \mathbf{H}$ are initialized with positive values, then no update can result in negative coefficients. For a detailed derivation of the gradient and the multiplicative update rules see section B.5.

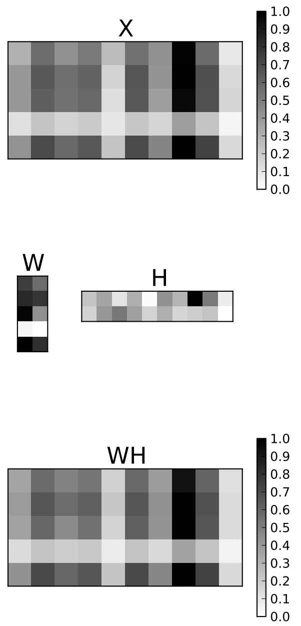


Figure 9.1: NMF example. **Top** Original data matrix **X**. **Middle** Low rank factors of rows **W** and columns **H**. **Bottom** Approximated matrix **WH**

PSEUDOCODE Algorithm 5 shows the pseudocode for computing NMF. $\odot$ denotes elementwise multiplication, $\oslash$ elementwise division.

Algorithm 5 Non-negative Matrix Factorization

```
Input: data $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}_+^{D \times N}$, number of factors $k$
Output: $W, H$
1: # Initialize $\mathbf{W} \in \mathbb{R}_+^{D \times k}$, $\mathbf{H} \in \mathbb{R}_+^{k \times N}$ randomly
2: # Add a small constant $\epsilon = 10^{-19}$ to $X$ to avoid zero-divisions
3: for it $\leq$ Iterations do
4:     $\mathbf{H} = \mathbf{H} \odot \mathbf{W}^\top X \oslash \mathbf{W}^\top \mathbf{W}\mathbf{H}$
5:     $\mathbf{W} = \mathbf{W} \odot \mathbf{X}\mathbf{H}^\top \oslash \mathbf{W}\mathbf{H}\mathbf{H}^\top$
6: end for
```

DISCUSSION NMF has become widely used for a broad spectrum of applications. Among the most prominent applications are factorizations of probabilistic matrices for topic retrieval from documents. In fact NMF is equivalent to probabilistic latent semantic analysis [Ding et al., 2008], a popular topic model. Although NMF has the advantage of easily interpretable factors, it is helpful to keep some of its drawbacks in mind when using it. For one, NMF does not necessarily find the globally optimal solution [Lee and Seung, 2000]. There are guarantees to find local optima, but depending on the initialization the result might change. Another drawback of the original NMF algorithm is that it does not scale well to larger data sets. But there are alternatives to this NMF algorithm. A sensible approach to NMF on large scale data sets is to use online methods, meaning not processing all data points in each iteration but only one data point at a time [Mairal et al., 2010].

EXAMPLE Figure 9.1 shows a toy data example of NMF applied to a small positive matrix $\mathbf{X} \in \mathbb{R}_+^{5 \times 10}$ where each entry was drawn from a uniform distribution. NMF was applied to recover two factors. The column factors $\mathbf{W} \in \mathbb{R}_+^{5 \times 2}$ and the row factors $\mathbf{H} \in \mathbb{R}_+^{2 \times 10}$ capture the true data matrix well as shown by the approximation $\mathbf{WH}$ in the bottom panel.

SEE ALSO section B.5

10

Canonical Correlation Analysis

APPLICATIONS Hidden variable estimation, multimodal data integration, computing correlations between multivariate variables

OBJECTIVE CANONICAL CORRELATION ANALYSIS[1] (CCA) is a versatile and efficient method for integrating sets of variables that share some variability. A simple example would be text with the same content in different languages, such as the suisse constitution. While the original CCA version was formulated for only two variables, one of the extensions to more than two variables proposed in [Kettenring, 1971] is fairly straightforward, so we present it here. Measurements of the variables are stored in data matrices $\mathbf{X}_i \in \mathbb{R}^{D_i \times N}$, where N denotes the number of samples, which has to be the same for all variables. The generative model underlying CCA, depicted in Figure 10.1, assumes that there is a hidden variable $\mathbf{Z} \in \mathbb{R}^{K \times N}$, where $K = \min(D_i)$, that is shared among all variables $\mathbf{X}_i$. While $\mathbf{Z}$ is not measured directly, CCA assumes that it can be approximated by assuming that the best approximation of $\mathbf{Z}$ is that *subspace* $\mathbf{W}_i \in \mathbb{R}^{D_i \times K}$ of each variable $\mathbf{X}_i$ in which the measured variables are maximally correlated. In matrix notation the objective of CCA for M variables is

[1] Harold Hotelling. Relations between two sets of variates. *Biometrika*, 28(3):321–377, 1936

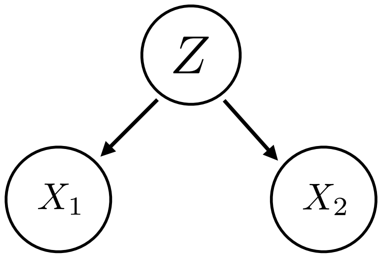


Figure 10.1: Generative model of CCA: A hidden variable Z that cannot be measured directly is reflected in multiple measurements X and Y. CCA approximates Z by finding subspaces $\mathbf{W}_x$, $\mathbf{W}_y$ such that the *principal angle* or *canonical correlation* between $\mathbf{W}_x^\top \mathbf{X}$ and $\mathbf{W}_y^\top \mathbf{Y}$ is maximal.

$$\begin{aligned} &\underset{\mathbf{W}_i,\mathbf{W}_j}{\operatorname{argmax}} \sum_{i=1}^{M} \sum_{j=i+1}^{M} \operatorname{Tr}\left(\mathbf{W}_i^\top \mathbf{X}_i \mathbf{X}_j^\top \mathbf{W}_j\right) \\ &\text{such that } \mathbf{W}_i^\top \mathbf{X}_i \mathbf{X}_i^\top \mathbf{W}_i = \mathbf{I}, \ \forall i, \end{aligned} \tag{10.1}$$

The solution of eq. 10.1 are the top eigenvectors of the generalized eigenvalue equation

$$\mathbf{L}\mathbf{W} = \mathbf{R}\mathbf{W}\Lambda \tag{10.2}$$

where

$$\mathbf{L} = \begin{bmatrix} 0 & \mathbf{C}_{12} & \dots & \mathbf{C}_{1N} \\ \mathbf{C}_{21} & 0 & \dots & \mathbf{C}_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{N1} & \mathbf{C}_{N2} & \dots & 0 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \\ \vdots \\ \mathbf{W}_N \end{bmatrix} \text{ and } \mathbf{R} = \begin{bmatrix} \mathbf{C}_{11} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & \mathbf{C}_{NN} \end{bmatrix}$$

where $\mathbf{C}_{ij} = \mathbf{X}_i^\top \mathbf{X}_j$ denotes the empirical covariance matrix between the ith and jth variable. The eigenvalues are also called *canonical correlations*.

PSEUDOCODE The algorithm 6 shows pseudocode for CCA.

Algorithm 6 Canonical Correlation Analysis

```
Input: Data $\mathbf{X}_1, \dots, \mathbf{X}_M \in \mathbb{R}^{D\times N}$
Output: Canonical subspaces $\mathbf{W}_1, \dots, \mathbf{W}_M \in \mathbb{R}^{D\times K}$, canonical components $\mathbf{W}_1, \dots, \mathbf{U}_M \in \mathbb{R}^{K\times N}$,
1: # Compute block-diagonal matrix on the right hand side of Equation 10.2
2: $\mathbf{R} = \text{blockdiag}(\mathbf{X}_i \mathbf{X}_i^\top),\ \forall i$
3: # Compute block-matrix on left hand side of Equation 10.2
4: $\mathbf{L} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_M \end{bmatrix} \left[\mathbf{X}_1^\top \dots \mathbf{X}_M^\top\right] - \mathbf{R}$
5: # Compute canonical directions (stacked as $[\mathbf{W}_1^\top, \dots, \mathbf{W}_M^\top]^\top$)
6: $\mathbf{W} = \text{eig}(\mathbf{L}, \mathbf{R})]$
7: # Compute canonical components
8: $\mathbf{U}_i = \mathbf{W}_i^\top \mathbf{X}_i$
```

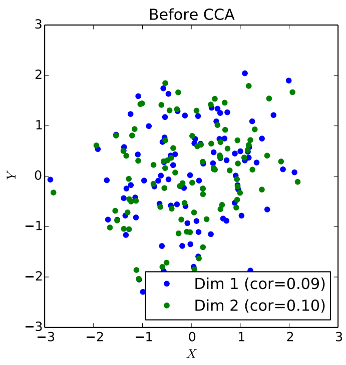


Figure 10.2: CCA toy data, generated according to Equation 10.3; the (common) signal to (view-specific) noise ratio γ was set to 0.1, such that single dimensions show low correlation, despite the common underlying component.

DISCUSSION Whenever there are several multivariate measurements the canonical correlation coefficient is a useful means to quantify what these variable have in common. An important advantage of CCA over computing univariate correlation coefficients between single feature dimensions is: the canonical correlation coefficient is *invariant with respect to all linear transformations of the data* (hence *canonical*); this correlation is very similar to the concept of the *principal angle* [2], which measures the similarity of spaces, in analogy to the angle between two vectors. There are some cases in which there seem not many alternatives to CCA. Consider the case of several measurements that reflect some latent process which can not be monitored. Applying PCA (chapter 7) to each of the variables and correlating the first, second and so forth principal components can lead astray: the variance of the latent process might be stronger or weaker with respect to variable specific processes. In contrast CCA will find the common latent process, irrespective of the strength with which the common process is reflected in each variable.

[2] C. Jordan. Essai sur la Géometrie à n dimensions. *Bull. Soc. Math. France*, 3: 103–174, 1875. Tome III, Gauthiers-Villars, Paris, 1962, 79-149

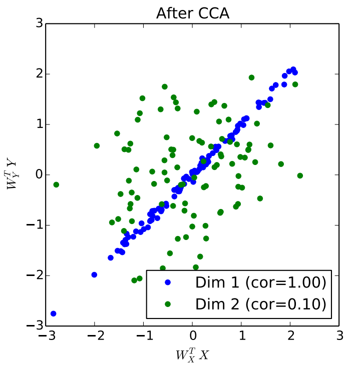


Figure 10.3: Visualisation of first (blue) and second (green) canonical component of $\mathbf{X}$ and of $\mathbf{Y}$, learned on the data in Figure 10.2. Note that while the original data was not correlated in neither of their dimensions, in the CCA subspace the common latent factor is recovered and the residual is uncorrelated, it just reflects the view-specific variation.

EXAMPLE Samples of two variables $\mathbf{X},\ \mathbf{Y} \in \mathbb{R}^2$ were generated according to

$$\mathbf{X} = \mathbf{w}_X \gamma s + (1-\gamma^2)\boldsymbol{\epsilon}_X, \quad \mathbf{Y} = \mathbf{w}_Y \gamma s + (1-\gamma^2)\boldsymbol{\epsilon}_Y \tag{10.3}$$

such that there was one common latent factor $s \sim \mathcal{N}(0,1)$ and some independent variable-specific factor $\boldsymbol{\epsilon}_X \sim \mathcal{N}(0, \mathbf{I})$ and $\boldsymbol{\epsilon}_Y \sim \mathcal{N}(0, \mathbf{I})$.

SEE ALSO section B.2, chapter 11

11

Kernel CCA

APPLICATIONS Hidden Variable Estimation, computing non-linear correlations between multivariate variables

1

OBJECTIVE KERNEL CANONICAL CORRELATION ANALYSIS[1], or kernel CCA, allows to find non-linear dependencies between several multivariate variables $\mathbf{X}_m \in \mathbb{R}^{D_m}$. KCCA solves the same objective as linear CCA, only that the variables are mapped through non-linear function $\phi_m(.) : \mathbb{R}^{D_m} \rightarrow \mathbb{R}_0^D$, from the D_m dimensional input space of each variable to that D_0 dimensional space, in which the correlations between the variables are maximized. Using a compacter notation than in Equation 10.1

$$\underset{\phi_i, \phi_j}{\operatorname{argmax}} \quad \operatorname{Corr}(\phi_i(\mathbf{X}_i), \phi_j(\mathbf{X}_j)), \forall i \neq j. \tag{11.1}$$

As in kernel PCA, kernel CCA, too, uses the kernel trick (see Appendix A) to express the non-linear function ϕ_m. Given N data points the optimal mappings ϕ_m^* are expressed as

$$\phi_m^*(\mathbf{x}_{m0}) = \sum_{n=1}^{N} k(\mathbf{x}_{m0}, \mathbf{x}_{mn}) \mathbf{A}_{mn} \tag{11.2}$$

where $\mathbf{x}_{mn}$ denotes the n-th data point of the m-th variable and, in a slight abuse of notational conventions, $\mathbf{A}_{mn} \in \mathbb{R}^{D_0}$ are the dual coefficients obtained as the solution to the generalized eigenvalue problem

$$\mathbf{LA} = \mathbf{RA}\Lambda \tag{11.3}$$

where

$$\mathbf{R} = \begin{bmatrix} \mathbf{K}_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & \mathbf{K}_M^2 \end{bmatrix}, \mathbf{L} = \begin{bmatrix} \mathbf{K}_1 \\ \vdots \\ \mathbf{K}_M \end{bmatrix} \left[\mathbf{K}_1^\top \dots \mathbf{K}_M^\top\right] - \mathbf{R} \text{ and } \mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_M \end{bmatrix}$$

where $\mathbf{K}_m$ denotes the kernel matrix of the mth variable. Entries of $\mathbf{K}_m$ in the ith row and the jth column are the similarity between the respective data points as evaluated by the kernel function $k_m(\mathbf{x}_i, \mathbf{x}_j)$. As in linear CCA the eigenvalues on the diagonal of the matrix Λ are also called *canonical correlations*.

PSEUDOCODE The algorithm for estimating the kernel CCA solution is described in algorithm 7.

Note that one could simply compute the dual coefficients $\mathbf{A}$ by invoking the linear CCA algorithm with the precomputed kernels $\mathbf{K}_m$ as data matrices.

Algorithm 7 Kernel Canonical Correlation Analysis

```
Input: Data $\mathbf{X}_m \in \mathbb{R}^{D_m \times N}$, kernel functions $k_m(.,.)$
Output: Dual coefficients for kernel expansion $\mathbf{A}_m$
1: # Compute block-diagonal matrix on the right hand side of Equation 11.3
2: $\mathbf{R} = \text{blockdiag}(\mathbf{K}_m \mathbf{X}_m^\top), \forall m$
3: # Compute block-matrix on left hand side of Equation 11.3
4: $\mathbf{L} = \begin{bmatrix} \mathbf{K}_1 \\ \vdots \\ \mathbf{K}_M \end{bmatrix} \begin{bmatrix} \mathbf{K}_1^\top \dots \mathbf{K}_M^\top \end{bmatrix} - \mathbf{R}$
5: # Compute dual coefficients canonical directions (stacked as $[\mathbf{A}_1^\top, \dots, \mathbf{A}_M^\top]^\top$)
6: $\mathbf{A} = \text{eig}(\mathbf{L}, \mathbf{R})$
7: # Compute canonical components
8: $\mathbf{U}_m = \mathbf{A}_m^\top \mathbf{K}_m$
```

DISCUSSION In analogy to linear PCA (see chapter 7), there are cases when the dependencies between several variables are not well described by a line. Figure 11.1 shows such a case: the correlation coefficient between the two variables is zero, but clearly there is a dependency between the two variables. If one does not know anything about the relationship between the two variables, it is very difficult to find the common latent variable, which in this example could be thought of as the position on the circle. Kernel CCA solves this problem, but as all kernel methods this comes at the price of solutions that are sometimes difficult to interpret and a storage and computational cost (importantly: also during prediction time, i.e. when Equation 11.2 is evaluated) that scales quadratically with the number of training examples, which is prohibitive for large scale data sets. Potential solutions to this problem are approximations of the kernel functions using random projections (see chapter 24) [] or neural networks [?].

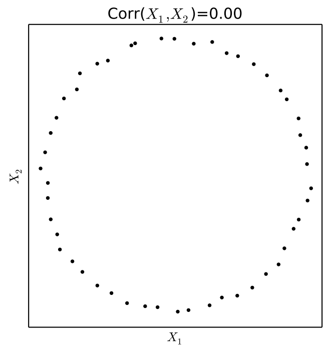


Figure 11.1: When linear (canonical) correlation coefficients fail: The data in $\mathbf{X}_1 = sin(z)$ and $\mathbf{X}_2 = cos(z)$ has a correlation coefficient of 0, despite the obvious dependency between $\mathbf{X}_1$ and $\mathbf{X}_2$.

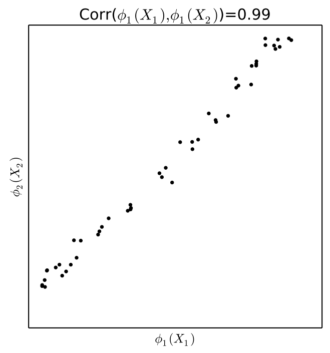


Figure 11.2: Kernel CCA discovers the dependency between x and y: After projecting the data onto their respective kernel canonical projections ϕ_{X_1}, ϕ_{X_2}, the data is linearly correlated.

EXAMPLE Figure 11.1 shows an example of data that is not linearly correlated however in the space found by kernel CCA, the data is strongly correlated.

SEE ALSO chapter 10, Appendix A

12

K-means clustering

APPLICATIONS clustering, quantization

OBJECTIVE The K-means clustering algorithm[1] aims at finding centres of clusters $\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k$ in the data by minimizing the sum of the distances of datapoints to their respective cluster centre. More formally, the objective function can be written as

[1] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982

$$L(\{\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k\}, \mathbf{c}) = \sum_{n=1}^{N} \|\mathbf{x}_n - \boldsymbol{\mu}_{\mathbf{c}_n}\| \tag{12.1}$$

where $\mathbf{c}_i$ is the index of the cluster to which datapoint $\mathbf{x}_i$ belongs to. At each iteration, the algorithm minimizes the loss function in two steps: The **Cluster Assignment Step** in which every data point is assigned to the nearest cluster center and the **Update Step** in which all cluster centers are updated to the mean over their members.

PSEUDOCODE The pseudocode for K-Means Clustering is given in algorithm 8. The Pseudocode for online k-means clustering[2] is listed in algorithm 9.

[2] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967

Algorithm 8 K-means clustering

```
Input: data $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathbb{R}^D$, number of clusters $k$, iterations $m$.
Output: cluster centres $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k \in \mathbb{R}^D$, assignment vector $\mathbf{c}^{\text{old}} \in \mathbb{R}^n$
1:  Choose random data points as initial cluster centres.
2:  $\mathbf{c} \leftarrow \mathbf{0}_N,\ \mathbf{c}^{\text{old}} \leftarrow \mathbf{0}_N,\ i \leftarrow 0$
3:  while $i < m$ do
4:      for $j = 1$ to $N$ do
5:          Find nearest cluster centre $\mathbf{c}_j \leftarrow \text{argmin}_{1 \le l \le k} \|\mathbf{x}_j - \boldsymbol{\mu}_l\|_2$
6:      end for
7:      for $j \leftarrow 1$ to $k$ do
8:          Compute new cluster centre $\boldsymbol{\mu}_j \leftarrow \frac{1}{|\{l : \mathbf{c}_l = j\}|} \sum_{l : \mathbf{c}_l = j} \mathbf{x}_l$
9:      end for
10:     $\mathbf{c}^{\text{old}} \leftarrow \mathbf{c},\ i \leftarrow i + 1$
11: end while
```

Algorithm 9 Online K-means clustering

```
Input: data points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, number of clusters $k$, iterations $m$.
Output: cluster centres $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k \in \mathbb{R}^d$
1: Choose random data points as initial cluster centres
   $\boldsymbol{\mu}_k \leftarrow \mathbf{x}_i, \ldots, \boldsymbol{\mu}_k \leftarrow \mathbf{x}_{i_k}$ where $i_j \neq i_l$ for all $j \neq l$.
2: Initialize cluster assignment counts $n_1, \ldots, n_k \leftarrow 0$
3: for $i = 1, \ldots, m$ do
4:     Draw a new data point randomly $\mathbf{x}_i$
5:     Find nearest cluster centre $k^* \leftarrow \mathrm{argmin}_k \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_2$
6:     Update cluster counts $n_{k^*} \leftarrow n_{k^*} + 1$
7:     Update cluster centers $\boldsymbol{\mu}_{k^*} \leftarrow \boldsymbol{\mu}_{k^*} + \frac{1}{n_{k^*}}(\mathbf{x}_i - \boldsymbol{\mu}_{k^*})$
8: end for
```

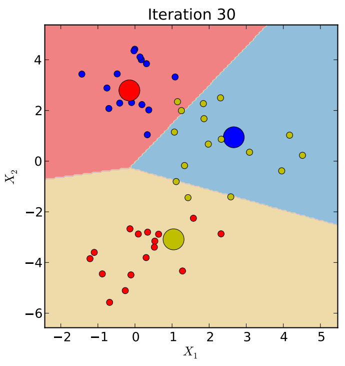


Figure 12.1: Example of Online K-Means (dot color indicates original distribution); cluster assignments after 30 iterations are indicated as background color, and cluster centroids as large dots..

DISCUSSION K-means clustering suffers mainly from two limitations: first, each datapoint belongs to exactly one cluster (so-called hard memberships) and all data points in one cluster have equal weight in the update step. Thus outliers may drag their centre into wrong directions. Secondly, clusters are assumed to be spherical. These two points are improved in the Mixtures-of-Gaussians Model (see chapter 13), which can be seen as a probabilistic extension of the K-Means algorithm. Algorithm 8 operates in *batch mode* and uses all data points in each iteration to update the centroids. This can become slow for large data sets. The online version of k-means picks one data point at a time and thus needs less memory. Like all clustering algorithms also K-means depends on a good choice of the distance function. Often used distance functions for numerical, ordinal and categorical data are listed in section C.4. Normalization of features can also be necessary if the scaling of different features is arbitrary. In some of these cases it can be useful to combine K-means clustering with kernel PCA. Computing clusters in the kernel PCA space can reveal more useful structures than in the input space, illustrated in Figure 12.2.

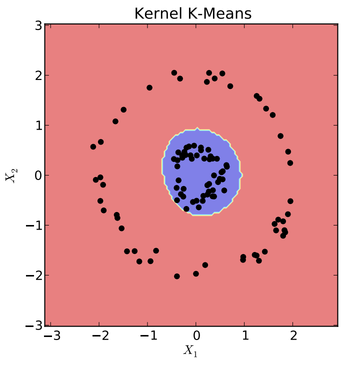


Figure 12.2: K-Means clustering algorithm applied to the data in the kernel PCA representation. The background color indicates the cluster membership.

EXAMPLE Figure 12.1 shows an example of online K-Means applied to a data set with three clusters in two dimensions. Data for each cluster was drawn from a gaussian distribution with different means and isotropic covariance. True cluster memberships are indicated as the dot color. The predicted cluster membership after 30 iterations is indicated as the background color. After 30 iterations the clusters capture (up to label permutations) the true cluster membership.

SEE ALSO K-Means Clustering chapter 12, Kernel PCA (chapter 8)

13

Mixtures of Gaussians

APPLICATIONS clustering, density estimation, quantization

OBJECTIVE Mixture models are a powerful probabilistic modelling tool. *Mixtures-of-Gaussians* (MoG) are a popular variant thereof. The MoG model assumes that a data set is generated from K clusters, each modelled as a Gaussian distribution. An example is shown in Figure 13.1. According to the MoG model the data $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^D$ are independent samples from K Gaussian distributions. Each Gaussian is characterized by its mean $\boldsymbol{\mu}_k$ and covariance matrix Σ_k. These model parameters are often estimated using Expectation-Maximization[1] (EM).

[1] N. M. Dempster, A.P. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society B*, 39:185–197, 1977

For each data point $\mathbf{x}$ we can compute $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \Sigma_k)$, the likelihood that $\mathbf{x}$ originates from a Gaussian distribution with mean $\boldsymbol{\mu}_k$ and covariance Σ_k as

$$\mathcal{N}(\mathbf{x}; \mu_k, \Sigma_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)\right)}{\sqrt{(2\pi)^D \det(\Sigma_k)}}. \tag{13.1}$$

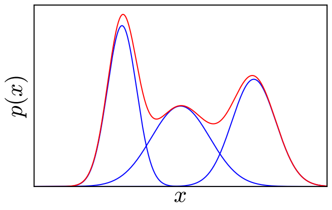


Figure 13.1: Example of a Mixtures-of-Gaussians model (red) fitted to an empirical distribution (blue).

Next to the means and covariances of each cluster we also need the prior probability ϕ_k of each cluster, which have to be positive and sum to one

$$\sum_{k=1}^{K} \phi_k = 1, \quad \phi_k \geq 0. \tag{13.2}$$

Using Equation 13.1 and Equation 13.2 we can compute the likelihood of the data by

$$p(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) = \sum_{n=1}^{N} \sum_{k=1}^{K} \phi_k \mathcal{N}(\mathbf{x}; \mu_k, \Sigma_k). \tag{13.3}$$

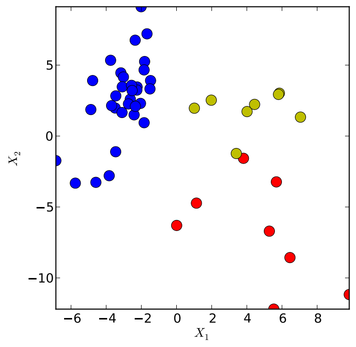


Figure 13.2: Data drawn from three Gaussians with different means and covariances in two dimensions. Color indicates cluster membership.

Unfortunately Equation 13.3 cannot be optimized in closed form. In order to estimate parameters in a maximum likelihood approach the EM procedure can maximize Equation 13.3 iteratively. To that end we introduce another quantity that we will call γ_{nk}; it can be thought of as a probabilistic cluster membership (in contrast to the binary cluster membership vector in the K-Means algorithm in chapter 12) and is computed as

$$\gamma_{nk} = \frac{\hat{\phi}_k \mathcal{N}(\mathbf{x}_n | \hat{\boldsymbol{\mu}}_k, \hat{\Sigma}_k)}{\sum_{j=1}^{K} \hat{\phi}_j \mathcal{N}(\mathbf{x}_n | \hat{\boldsymbol{\mu}}_j, \hat{\Sigma}_j)} \tag{13.4}$$

PSEUDOCODE The EM algorithm for estimating a MoG model is summarized in Algorithm 10. It alternates between estimating a probabilistic cluster membership γ_{nk} (the "E"-step) and the updates of the parameters $\boldsymbol{\mu}_k, \Sigma_k, \phi_k$ (the "M"-step).

Algorithm 10 The EM Algorithm for Mixture of Gaussians

```
Input: Data points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^D$, number of components $K$
Output: Means $\hat{\boldsymbol{\mu}}_k$ and covariance matrices $\hat{\Sigma}_k$, $1 \le k \le K$.
 1: $\hat{\phi}_k \leftarrow 1/K \quad \hat{\Sigma}_k \leftarrow \mathbf{I}_d, \ \hat{\boldsymbol{\mu}}_k \leftarrow$ random points out of $\mathbf{x}_1, \ldots, \mathbf{x}_n$
 2: repeat
 3:   {The "E"-step}
 4:   for $k \leftarrow 1$ to $K$ do
 5:     for $n \leftarrow 1$ to $N$ do
 6:       Update $\gamma_{nk}$      {see Equation 13.4}
 7:     end for
 8:   end for
 9:   { The "M"-step }
10:   for $k \leftarrow 1$ to $K$ do
11:     $N_k \leftarrow \sum_{n=1}^{N} \gamma_{nk}$
12:     $\hat{\phi}_k \leftarrow N_k / N$
13:     $\hat{\boldsymbol{\mu}}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n$
14:     $\hat{\Sigma}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)(\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)^\top$
15:   end for
16: until "convergence"
```

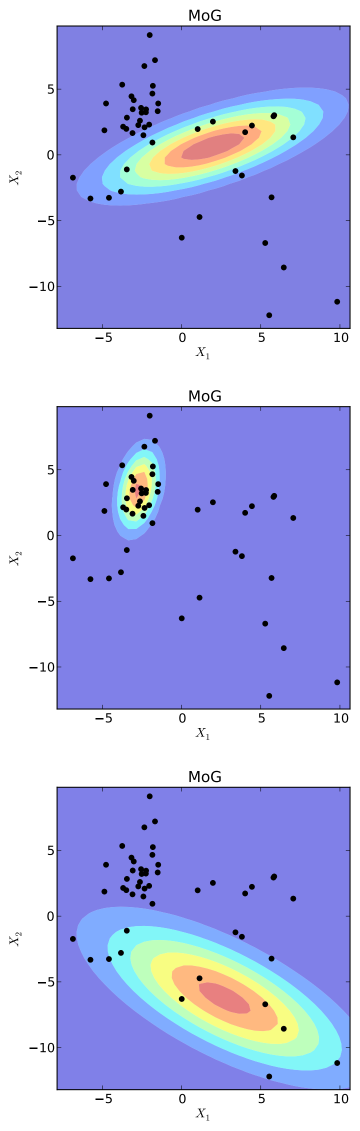


Figure 13.3: Result of MoG using Expectation Maximization after 20 iterations when estimating 3 clusters on the data in figure 13.2

DISCUSSION MoGs fit a wide range of real data well but they also have some limitations. Initialization of the parameters is sometimes an issue. It can make sense to initialize the parameters with a solution obtained by another algorithm, e.g. K-Means. Unfortunately, the EM algorithm is not stable. For more than two components the likelihood can grow to infinity by centering one component on a single data point and letting the covariance go to zero. Controlling for the variance or using appropriate priors can prevent these cases[2]. Other problems are that the solutions are not unique and convergence can be very slow convergence in some cases.

[2] Christopher M Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2007

SEE ALSO K-Means Clustering chapter 12

IV
Supervised Methods

14

Naive Bayes

APPLICATIONS Classification with probabilistic outputs

OBJECTIVE The goal of the Naive Bayes classifier is to obtain a probabilistic estimate of the label y given a data point $\mathbf{x}$, called $p(y|\mathbf{x})$. If one knew the joint probability $p(\mathbf{x}, y)$ of the data $\mathbf{x}$ and the label y it would be easy to compute $p(y|\mathbf{x})$. Usually $p(\mathbf{x}, y)$ is unknown but one can obtain $p(y|\mathbf{x})$ using basic probability rules via

$$\text{Symmetry} \qquad p(y, \mathbf{x}) = p(\mathbf{x}, y) \tag{14.1}$$

$$\text{Product Rule} \qquad p(y|\mathbf{x})p(\mathbf{x}) = p(\mathbf{x}|y)p(y) \tag{14.2}$$

$$\text{Bayes' Rule} \qquad p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \tag{14.3}$$

where $p(\mathbf{x}|y)$ is the probability of $\mathbf{x}$ conditioned on the label y (meaning the probability of $\mathbf{x}$ estimated for each possible value of y separately), $p(y)$ is the probability of a label and $p(\mathbf{x})$ is the probability of the data point $\mathbf{x}$. It is easy to estimate $p(y)$. If we have K labels $y \in \{1, 2, \dots, K\}$ we obtain

In high dimensional spaces it becomes difficult to obtain good estimates of probability distributions because the number of values a variable can take increases exponentially. Consider a $D - dimensional$ binary variable $\mathbf{x} \in \{0,1\}^D$, then $\mathbf{x}$ can take 2^D possible values. This phenomenon has been described as the *curse of dimensionality* [Bellman, 1957].

$$p(y = k) = \frac{N_k}{N} \tag{14.4}$$

where N_k is the number of occurrences of the label k in our training data set and N is the total number of data points in our training data set. In classification settings it does not matter that $p(\mathbf{x})$ cannot be estimated – it is the same for all labels y. The most difficult part is the estimation of $p(\mathbf{x}|y)$, especially when $\mathbf{x}$ is high-dimensional. Naive Bayes classifiers assume that the dimensions are *conditionally independent* and $p(\mathbf{x}|y)$ thus factorizes into

Conditional independence: The two binary random variables *rain* and *thunder* are not independent, so $p(\text{rain}, \text{thunder}) \neq p(\text{rain})p(\text{thunder})$. However when we have seen lightning, we thunder does not add information. So conditioned on the additional variable *lightning* the variables *rain* and *thunder* are independent, so $p(\text{rain}, \text{thunder}|\text{lightning}) = p(\text{rain}|\text{lightning})p(\text{thunder}|\text{lightning})$.

$$p(\mathbf{x}|y) = p(\mathbf{x}_1|y)p(\mathbf{x}_2|y)\dots p(\mathbf{x}_D|y) = \prod_{d=1}^{D} p(\mathbf{x}_d|y) \tag{14.5}$$

where $p(\mathbf{x}_1|y)$ is the probability of the dth dimension of $\mathbf{x}$ conditioned on y. Instead of 2^D values we now only have to estimate $2D$ states for each of the K labels.

PSEUDOCODE The pseudocode for training a multiclass Naive Bayes classifier on binary data in an iterative fashion is given in algorithm 11.

Algorithm 11 Naive Bayes for binary data

```
Input: Data $\mathbf{x}_1, \dots, \mathbf{x}_N \in \{0,1\}^D$, labels $y_1, \dots, y_N \in \{1, \dots, K\}$
Output: Class probability $p(y)$, class conditional probability $p(\mathbf{x}|y)$
1:  # Initialize counters for each class
2:  $\forall k, d: \; N_k = 0, \; N_{dk} = 0$
3:  for Data point $i = 1, \dots, N$ do
4:      $k = y_i$
5:      $N_k = N_k + 1$
6:      for Feature $d = 1, \dots, D$ do
7:          $N_{dk} = N_{dk} + 1$
8:      end for
9:  end for
10: # Use Laplace smoothing to avoid zero probabilities
11: $p(y) = (N_k + 1)/(N + 1), \quad p(\mathbf{x}|y) = (N_{dk} + 1)/(N_k + 1)$
```

DISCUSSION Although the conditional independence assumption is violated in almost all real world data sets, Naive Bayes Classifiers are often used. One reason is that it is often desirable to have a probabilistic model of your data. Another reason is that although the multivariate structure of $p(\mathbf{x}|y)$ is completely ignored, the Naive Bayes classifier often performs well in practice. Finally Naive Bayes classifiers are extremely simple to implement and scale well to large sets of data and computer systems. A common trick to avoid probability estimates of $p(y|\mathbf{x}) = 0$ is to use *Laplace smoothing* (adding 1 to all counts) in line 11 of algorithm 11.

EXAMPLE One of the most popular applications of Naive Bayes classifiers is text classification. In Figure 14.1 a simple example of email spam detection illustrates the Naive Bayes classifier.

SEE ALSO Nearest Centroid Classifier (chapter 15), Linear Discriminant Analysis (chapter 16)

Binary Bag-Of-Words Data

Word	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$
Dinner	0	0	0	0	1
Cinema	0	0	0	1	0
Buy	1	1	0	0	1
Viagra	1	0	1	0	0
Label	1	1	1	0	0

$p(\mathbf{x}|y)$

Word	$p(x|y=1)$	$p(x|y=0)$
Dinner	0	1/3
Cinema	0	1/3
Buy	2/3	1/3
Viagra	2/3	0

$p(y=0) = 0.4 \qquad p(y=1) = 0.6$

New data point

$$\mathbf{x}_n = [1, 0, 0, 0]^\top$$

Predictions

$$\prod_{d=1}^{D} p(y=1|\mathbf{x}_{nd}) = 0 \cdot 0.6 = 0$$

$$\prod_{d=1}^{D} p(y=0|\mathbf{x}_{nd}) = 0.5 \cdot 0.4 = 0.2$$

$\rightarrow \mathbf{x}_n$ was *non-spam*

Figure 14.1: Toy data example of spam classification with the Naive Bayes classifier (without Laplace smoothing); label $y = 1$ refers to *spam* and $y = 0$ to *non-spam*. A new data point is classified as *non-spam*, since $\prod_{d=1}^{D} p(y = 0|\mathbf{x}_{nd}) > \prod_{d=1}^{D} p(y = 0|\mathbf{x}_{nd})$.

15

Nearest-Centroid Classifier

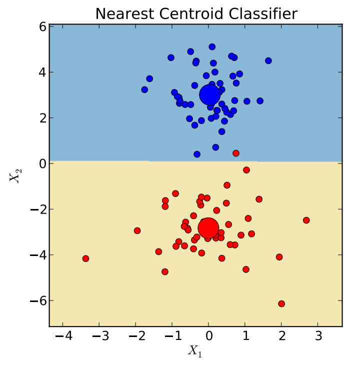


Figure 15.1: Two class example of NCC classification. Centroids μ_k are indicated by bigger circles. True labels are indicated as dot color, predicted labels as background colors. Data is drawn from a Gaussian distribution with isotropic covariance (same variance in all directions). Here NCC finds the optimal classification boundary.

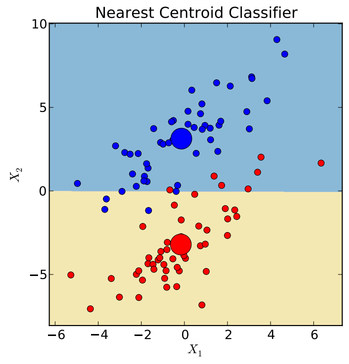


Figure 15.2: Same example as in Figure 15.1, except the covariance of features is not isotropic, instead the data distribution looks elliptical. In this case the decision boundary of NCC is not optimal.

APPLICATIONS Classification

OBJECTIVE The NEAREST-CENTROID CLASSIFIER (NCC) is the simplest classifier of them all. Given data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$ and corresponding class labels $y_1, \ldots, y_N \in \{1, \ldots, K\}$ computes the mean (also called centroid or prototype) of each class

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i=k} \mathbf{x}_i. \tag{15.1}$$

The prediction of the nearest centroid classifier is then simply the class corresponding to the closest centroids

$$\underset{k}{\text{argmin}} = \|\boldsymbol{\mu}_k - \mathbf{x}\|_2. \tag{15.2}$$

PSEUDOCODE The computation of the means $\boldsymbol{\mu}_k$ can be done iteratively as in algorithm 12, line 7. This way the solution can be computed in an online fashion with very low memory requirements.

Algorithm 12 Compute Class Centroids

```
Input: data $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$, labels $y_1, \ldots, y_N \in \{1, \ldots, K\}$
Output: Class means $\boldsymbol{\mu}_k,\ k \in \{1, \ldots, K\}$
1: # Initialize means and counters for each class
2: $\forall\, k: \quad \boldsymbol{\mu}_k = \mathbf{I} \cdot 0,\ N_k = 0$
3: # Iterative computation of class means
4: for Data point $i = 1, \ldots, N$ do
5:     # Update means and counters
6:     $k = y_i$
7:     $\boldsymbol{\mu}_k = \frac{N_k}{N_k+1}\, \boldsymbol{\mu}_k + \frac{1}{N_k+1}\, \mathbf{x}_i$
8:     $N_k = N_k + 1$
9: end for
```

The predicted class given a data point $\mathbf{x}$ is obtained as in algorithm 13 (see also Equation 15.2)

```
Algorithm 13 Nearest Centroid Prediction
Input: Data point $\mathbf{x} \in \mathbb{R}^D$, class centroids $\boldsymbol{\mu}_k$, $k \in \{1, \dots, K\}$
Output: Class membership $k^*$
1: # Compute nearest class centroid in discriminative subspace
2: $k^* = \text{argmin}_k \|\boldsymbol{\mu}_k - \mathbf{x}\|_2$.
```

DISCUSSION NCC can be seen as a linear classifier that finds a decision boundary orthogonal to the difference of means vector $\boldsymbol{\mu}_i - \boldsymbol{\mu}_j$ (see section C.2). It is efficient to train in an iterative fashion and does not have large memory requirements – only the centroids need to be stored. A severe disadvantage of NCC is that it ignores covariance structure. What this means is illustrated in Figure 15.1 and Figure 15.2: If the data comes from an isotropic distribution where all the information about class membership is in the distance from the centroids (and the direction in which this distance is measured is irrelevant) then NCC is optimal. All we need to know for optimal classification is the location of the data relative to all centroids. But if the data comes from a non-isotropic distribution as in Figure 15.2 ignoring the covariance structure will lead to a suboptimal decision boundary. This can lead to poor classification performance, as illustrated in figure 15.2. Methods that take covariance into account have a clear advantage over NCC, see e.g. chapters 16.

EXAMPLE Figure 15.2 shows NCC applied to toy data in a binary classification setting. Figure 15.3 shows NCC applied to a three class problem.

SEE ALSO Naive Bayes Classifier, Linear Discriminant Analysis

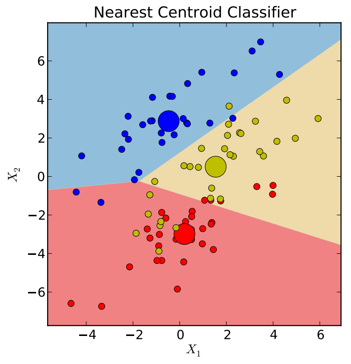


Figure 15.3: NCC applied to a three class problem.

16

Fisher Linear Discriminant Analysis

APPLICATIONS Classification

OBJECTIVE FISHER LINEAR DISCRIMINANT ANALYSIS [1] is a simple classifier that takes into account the shape of the class-conditional distributions, approximated by the covariance matrix. Given data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$ and corresponding class labels $\mathbf{y} = [y_1, \ldots, y_N] \in \{1, \ldots, K\}$ FLDA finds a subspace $[\mathbf{w}_1, \ldots, \mathbf{w}_K] = \mathbf{W} \in \mathbb{R}^{D \times K}$ in which the classes can be optimally separated. Optimality according to Fisher is defined as follows: The distance of the means of the classes should be maximized, while the variance within each class should be minimized. This is illustrated in figure 16.1. In order to formulate the objective, we first need the class centers (centroids) $\boldsymbol{\mu}_k$

[1] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936

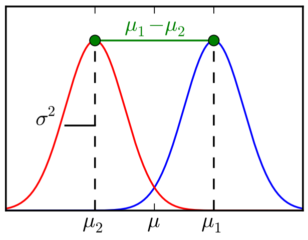


Figure 16.1: Two-class classification problem; the blue line indicates the probability distribution of one class, the red line that of the other class. Fisher LDA finds a subspace, such that when the data is projected onto it, the difference of class means $\mu_1 - \mu_2$ is maximized and the within class variance σ^2 is minimized.

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i=k} \mathbf{x}_i \tag{16.1}$$

where N_k is the number of data points in class k. With these centroids we can compute the *between-class covariance matrix* $\mathbf{B} \in \mathbb{R}^{D \times D}$ as

$$\mathbf{B} = \sum_{k=1}^{K} (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^\top, \qquad \boldsymbol{\mu} = \sum_{i}^{N} \mathbf{x}_i \tag{16.2}$$

where $\boldsymbol{\mu}$ is the overall class mean. Then we need the *within-class covariance matrix* $\mathbf{S} \in \mathbb{R}^{D \times D}$ which the sum of all k covariance matrices

$$\mathbf{S} = \frac{1}{K} \sum_{k=1}^{K} \quad \frac{1}{N_k} \sum_{i}^{N_k} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top \tag{16.3}$$

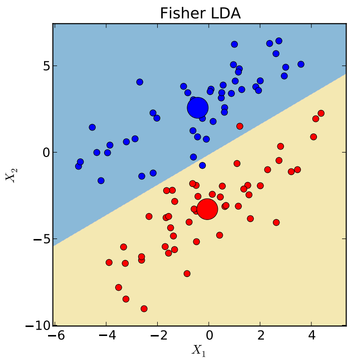


Figure 16.2: . Two class prediction problem. True labels are indicated as dot color, predicted labels as background colors. Note the difference in the decision boundary to standard nearest-neighbor classification (chapter 15).

The subspace $\mathbf{W}$ in which the data can now be optimally separated according to Fisher's objective is that which maximizes the Rayleigh quotient

$$\underset{\mathbf{W}}{\operatorname{argmax}} \frac{\mathbf{W}^\top \mathbf{B} \mathbf{W}}{\mathbf{W}^\top \mathbf{S} \mathbf{W}}. \tag{16.4}$$

PSEUDOCODE The pseudocode for finding the optimal subspace **W** is summarized in Algorithm 14.

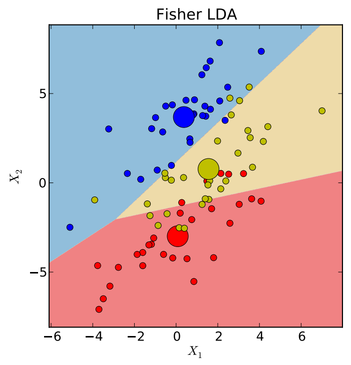
Algorithm 14 Fisher LDA

```
Input: Data $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$, Labels $y_1, \dots, y_N \in \{1, \dots, K\}$
Output: Discriminative Subspace $\mathbf{W}$, class means $[\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k]$
 1: # Compute overall mean
 2: $\boldsymbol{\mu} = 1/N \sum_i^N \mathbf{x}_i$
 3: for Class $k = 1, \dots, K$ do
 4:     # Compute class mean vectors
 5:     $\boldsymbol{\mu}_k = 1/N_k \sum_i^{N_k} \mathbf{x}_i$
 6:     # Compute within-class covariance matrix
 7:     $S_k = 1/N_k \sum_i^{N_k} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top$
 8: end for
 9: # Compute between-class covariance matrix $\mathbf{B}$
10: $\mathbf{B} = \sum_{k=1}^K (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^\top$
11: # Compute within-class covariance matrix $\mathbf{S}$
12: $\mathbf{S} = \sum_{k=1}^K S_k$
13: # Compute the first $k-1$ eigenvalues
14: $\mathbf{BW} = \mathbf{SW}\Lambda$
```

Algorithm 15 Fisher LDA Prediction

```
Input: Data point $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{W} \in \mathbb{R}^{D \times K}$, $\boldsymbol{\mu}_k$, $k \in \{1, \dots, K\}$
Output: Class membership $k^*$
 1: # Compute nearest class centroid in discriminative subspace
 2: $k^* = \text{argmin}_k \|\mathbf{W}^\top(\boldsymbol{\mu}_k - \mathbf{x})\|_2$.
```

Figure 16.3: Three class prediction problem. True labels are indicated as dot color, predicted labels as background colors.



DISCUSSION Fisher LDA is nearest-centroid classification w.r.t the Mahalanobis distance [2]. The algorithm requires to solve an eigenvalue equation which has cubic complexity. It is easy to show that Fisher's LDA is the bayes-optimal classifier if the data of each class originates from a Gaussian Distribution with equal covariance matrices, for a proof see section C.1. Using this generative model one can obtain probabilistic predictions.

[2] P Mahalanobis. On the generalized distance in statistics. *Proc. Nat. Inst. Sci. India*, 2:49–55, 1936

EXAMPLE Figure 16.2 shows Fisher LDA applied to a binary classification setting and Figure 16.3 in a three class setting.

SEE ALSO section C.1

17

Least Squares Regression

APPLICATIONS Regression

OBJECTIVE LINEAR REGRESSION (OLS) is almost as old as applied mathematics itself. The idea of OLS is that the target variable $\mathbf{Y} = [\mathbf{y}_1, \dots, \mathbf{y}_N] \in \mathbb{R}^{D_Y \times N}$ can be modeled as a linear combination of the features/dimensions of the data points $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{D_X, \times N}$:

$$\mathbf{Y} \approx \mathbf{W}^\top \mathbf{X} \tag{17.1}$$

The optimal linear mapping from data to targets minimizes the mean-squared-error[1] of the predictions

$$\begin{aligned} \underset{\mathbf{W}}{\operatorname{argmin}} \quad & \|\mathbf{Y} - \mathbf{W}^\top \mathbf{X}\|_2^2 & (17.2)\\ =& (\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{X}\mathbf{Y}^\top & (17.3) \end{aligned}$$

[1] Carl Friedrich Gauß. Theoria motus corporum coelestium in sectionibus conicis solem ambientium. Technical report, University of Göttingen, 1809; and Adrien-Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes*, chapter Sur la methode des moindres quarres. Firmin Didot, 1805

The solution is also known as the *pseudo-inverse* of the data matrix $\mathbf{X}$ multiplied by the target variables $\mathbf{Y}$, for a derivation see section B.3. If the number of features/dimensions of the data (rows of $\mathbf{X}$) becomes too large, it is crucial to constrain the complexity of the solution. Typically this is done by restricting the norm of the linear map $\mathbf{W}$. The most popular complexity control is known as *ridge regression*[2] and restricts the square of the euklidean norm $\|\mathbf{W}\|_2^2$. Adding this constraint to the original OLS objective results in an objective function that can be solved in closed form

[2] Andrey Nikolayevich Tychonoff. On the stability of inverse problems. *Doklady Akademii Nauk SSSR*, 39(5):195–198, 1943

$$\begin{aligned} \underset{\mathbf{W}}{\operatorname{argmin}} \quad & \|\mathbf{Y} - \mathbf{W}^\top \mathbf{X}\|_2^2 + \lambda \|\mathbf{W}\|_2^2 & (17.4)\\ =& (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}\mathbf{X}\mathbf{Y}^\top. & (17.5) \end{aligned}$$

Here λ controls the importance of the regularization: for small λ, only the error term determines the solution. For larger λ the solution $\mathbf{W}$ will have very small coefficients that are similar for all dimensions of $\mathbf{x}_n$. A derivation of the ridge regression solution can be found in section B.4.

Another view on Tikhonov regularization: If $\mathbf{X}$ has many more rows than columns, the matrix product $\mathbf{X}\mathbf{X}^\top$ will not have full rank, hence will not be invertible. By adding a small ridge of height λ the resulting matrix $\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I}$ becomes invertible.

PSEUDOCODE The pseudocode for linear ridge regression with vector valued target variables is shown in Algorithm 16.

Algorithm 16 Linear (Ridge) Regression

```
Input: $\mathbf{X} \in \mathbb{R}^{D_X \times N}$, $\mathbf{Y} \in \mathbb{R}^{D_Y \times N}$, ridge $\lambda$
Output: Weight matrix $\mathbf{W}$ for linear mapping $\mathbb{R}^{D_X} \to \mathbb{R}^{D_Y}$
1: Include offset parameters (row vector of $N$ ones)
2: $\mathbf{X} = \begin{bmatrix} \mathbf{1} \\ \mathbf{X} \end{bmatrix}$
3: $\mathbf{W} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}\mathbf{X}\mathbf{Y}^\top$
```

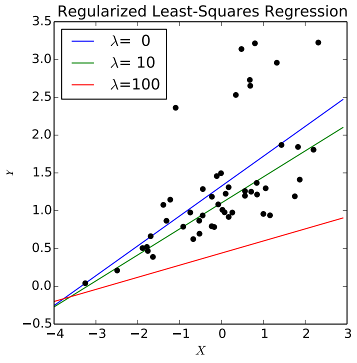


Figure 17.1: Toy data example with univariate data $\mathbf{x} \in \mathbb{R}^1$ and labels $\mathbf{y} \in \mathbb{R}^1$. Regularized Least-Squares regression (Equation 17.4) was used to estimate $y - axis$ offset and slope parameter of $\mathbf{x}$. Three different regularizers were used. The larger the regularizer λ the smaller the slope.

DISCUSSION It is probably fair so say that ordinary least squares regression is the most often used tool in applied mathematics. Some issues with OLS are that the squared error function suffers from high sensitivity to outliers. Assume that in one of the data points there was a really huge estimation error. Since the error is squared, this error can then dominate the whole cost function and effectively sabotage learning of $\mathbf{W}$. In order to circumvent this, one can employ other loss functions which scale only linearly for large values. However, the optimization becomes more complex then (and cannot be expressed as simple matrix algebra). Alternatives are error functions such as those used by support-vector machines in chapter 20.

Another drawback is that linear regression only estimates a linear mapping. If there is a non-linear relationship between features in $\mathbf{x}$ and targets $\mathbf{y}$ then linear regression will not find a sensible solution. In some cases one can use heuristics to construct non-linear features from $\mathbf{x}$, such that a linear combination of those predicts the targets better. But if we do not have any heuristic we can still resort to methods such as kernel ridge regression (chapter 23) or non-linear support vector machines (chapter 21) to learn arbitrary non-linear mappings from $\mathbf{x}$ to $\mathbf{y}$. Alternatively we can use random projections in order to emulate kernel functions[3].

[3] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NIPS*, 2007

Finally another disadvantage of OLS in its simples form is the need for matrix inversion. With high dimensional data and few data points this can lead to unstable solutions, or it might actually be difficult to invert the covariance matrix at all. One solution to stabilize the inversion is regularization. An alternative way of regularizing OLS is to take a low-rank approximation of the data matrix, e.g. by using only the strongest principal components of the data (see chapter 7). Another way of dealing with this setting is to use chapter 23.

SEE ALSO Kernel Ridge Regression (chapter 23), section B.3

18

Lasso

APPLICATIONS Regression, Shrinkage

OBJECTIVE The Lasso[1] (least absolute shrinkage and selection operator) fits a linear regression model, i.e. given data points $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$ with corresponding targets $\mathbf{y} = [\mathbf{y}_1, \ldots, \mathbf{y}_N] \in \mathbb{R}^{1 \times N}$ it tries to find $\mathbf{w} \in \mathbb{R}^D$ such that

$$\mathbf{y} \approx \mathbf{w}^\top \mathbf{X}, \tag{18.1}$$

and additionally, it minimizes the L_1-norm of $\mathbf{w}$, in contrast to ridge regression, where the L_2 norm is penalized.

The optimization problem is

$$\underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2N} \|\mathbf{y} - \mathbf{w}^\top \mathbf{X}\|_2^2 + \lambda \|\mathbf{w}\|_1, \tag{18.2}$$

i.e. we do least squares regression with an additional L_1-regularization constraint. The benefit of a L_1-regularization instead of L_2 is, that the solutions tend to be sparse, i.e. many entries of the solution vector equal to zero, yielding in a better interpretability and simpler models.

[1] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994

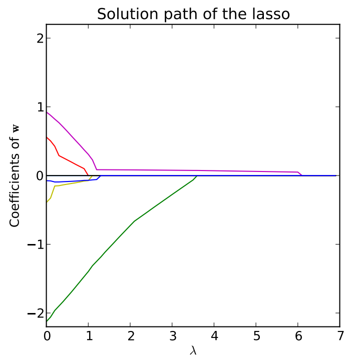


Figure 18.1: Typical solution path of the lasso. Each line corresponds to a dimension in the vector $\mathbf{w}$. Starting with the non-sparse full least squares solution for $\lambda = 0$, all coefficients go to zero when λ is increased.

The common way to solve the Lasso problem is the LARS (least angle regression) algorithm. It has the advantage, that it can calculate the solutions for all possible values of λ at the same time. On the other hand, LARS is not trivial to implement and a non-online optimization method. Algorithm 17 optimizes the Lasso in an online fashion[2] and is easy to implement at the same time.

[2] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187. Springer, 2010

DISCUSSION Algorithm 17 can be used to optimize the Lasso objective function in an online fashion. One critical point is the choice of the learning rate η in each step. In cases where we assume that the distribution of our data does not change, it is suitable to use a decaying learning rate to guarantee convergence. A popular choice is $\eta_t = \eta_0 t^{-\alpha}$, where t is the iteration number, η_0 the initial learning rate and $\alpha \in [0, 1]$ a decay parameter.

The last step in the algorithm is a threshold projection and optional but recommended, since it forces real sparsity for small entries in $\mathbf{w}$ in each iteration.

Algorithm 17 Online Lasso update step

```
Input: data point $\mathbf{x} \in \mathbb{R}^D$, target label $\mathbf{y}$, current vector $\mathbf{w} \in \mathbb{R}^D$,
    shrinkage parameter $\lambda > 0$, current learning rate $\eta$.
    Optional: threshold projection parameter $\theta$.
Output: updated $\mathbf{w} \in \mathbb{R}^D$
1: # calculate positive and negative part of $\mathbf{w}$
2: $\mathbf{u} = \max(\mathbf{w}, 0)$
3: $\mathbf{v} = \max(-\mathbf{w}, 0)$
4: # calculate gradients w.r.t. $\mathbf{u}$ and $\mathbf{v}$
5: $\text{grad}_{\mathbf{u}} = -(\mathbf{y} - \mathbf{w}^\top \mathbf{x})\mathbf{x} + \lambda$
6: $\text{grad}_{\mathbf{v}} = (\mathbf{y} - \mathbf{w}^\top \mathbf{x})\mathbf{x} + \lambda$
7: # gradient descent steps and projection to the nonnegative orthant
8: $\mathbf{u} \leftarrow \max(\mathbf{u} - \eta \cdot \text{grad}_{\mathbf{u}}, 0)$
9: $\mathbf{v} \leftarrow \max(\mathbf{v} - \eta \cdot \text{grad}_{\mathbf{v}}, 0)$
10: # update $\mathbf{w}$
11: $\mathbf{w} \leftarrow \mathbf{u} - \mathbf{v}$
12: # optional step: set entries smaller than $\theta$ to zero
13: $\forall i$ : if $|\mathbf{w}_i| < \theta$ then $\mathbf{w}_i \leftarrow 0$
```

SEE ALSO Least squares regression, Ridge regression (least squares regression using L_2 regularization)

19

Logistic Regression

APPLICATIONS Classification with probabilistic outputs

OBJECTIVE The goal of LOGISTIC REGRESSION (LR) is to predict the probability of a data point $\mathbf{x}$ belonging to one of K classes. Note that LR is *not a regression algorithm* – it is for classification. LR does not estimate a generative model of the data. Instead it directly estimates $p(y = k|\mathbf{x}, \mathbf{W})$, the probability that the label y is $k \in \{1, \ldots, K\}$ given a data point $\mathbf{x}$ and a linear transformation of the data $[\mathbf{w}_1, \ldots, \mathbf{w}_K] = \mathbf{W} \in \mathbb{R}^{D \times K}$. The conditional class likelihood for one data point $\mathbf{x}$ being of class $k \in \{1, \ldots, K\}$ is modeled as:

$$p_k(y|\mathbf{x}, \mathbf{W}) = \frac{e^{\mathbf{w}_k^\top \mathbf{x}}}{\sum_{j=1}^K e^{\mathbf{w}_j^\top \mathbf{x}}}. \tag{19.1}$$

The likelihood $p(y|\mathbf{x}, \mathbf{W})$ for all classes is then

$$p(y|\mathbf{x}, \mathbf{W}) = p_y(y|\mathbf{x}, \mathbf{W}) = \frac{e^{\mathbf{w}_y^\top \mathbf{x}}}{\sum_{j=1}^K e^{\mathbf{w}_j^\top \mathbf{x}}}. \tag{19.2}$$

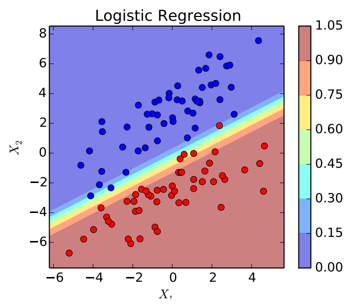


Figure 19.1: . Two class prediction problem. True labels are indicated as dot color, $p(y = \text{red}|\mathbf{x}, \mathbf{w})$ is indicated as colored background.

The optimal solution to LR is the $\mathbf{W}$ that maximizes the likelihood in eq. 19.2 or equivalently minimizes the negative log-likelihood. The first derivative of the negative log-likelihood for a single column $\mathbf{w}_k$ of $\mathbf{W}$ and a single data point $\mathbf{x}$ is

$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}_k} - \log(p(y|\mathbf{x}, \mathbf{W})) &= \frac{\partial}{\partial \mathbf{w}_k}\left[-\mathbf{w}_y^\top x + \log\left(\sum_{j=1}^K e^{\mathbf{w}_j^\top x}\right)\right] \\ &= \underbrace{\left(\frac{e^{\mathbf{w}_k^\top x}}{\sum_{j=1}^K e^{\mathbf{w}_j^\top x}} - \mathbf{I}_y(k)\right)}_{\text{Error}} \mathbf{x}\end{aligned} \tag{19.3}$$

where $\mathbf{I}_y$ is the indicator function of y, i.e. $\mathbf{I}_y(k) = 1$ for $k = y$ and $\mathbf{I}_y(k) = 0$ otherwise. So the gradient is simply the error of each of the k predictors times the data point. For a derivation of the gradient in eq. 19.3 see section B.6. As the objective function is convex, first order stochastic gradient descent will converge to the global optimum of $\mathbf{W}$.

PSEUDOCODE The pseudocode for multi class logistic regression using first order stochastic gradient descent[1] [Robbins and Monro, 1951] is shown in algorithm 18. Note that the label is provided as a K-dimensional boolean vector $y \in \{0,1\}^K$, where K is the number of classes and y_i is 0 except the element corresponding to the class of the data point $\mathbf{x}$.

[1] Often a second order optimization is used (Iterative Reweighted Least Squares); this algorithm however needs an estimate of the data covariance matrix, which is difficult for high-dimensional or very large data sets.

Algorithm 18 Multiclass Logistic Regression

```
Input: Data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$, labels $y_1, \ldots, y_N \in \{0,1\}^K$
Output: Weight matrix $\mathbf{W}$
 1: # Include bias terms
 2: $\mathbf{X} = [\mathbf{1}^N \ \ \mathbf{X}^\top]^\top$
 3: # Initialize weight matrix and counter
 4: $\mathbf{W} \leftarrow \mathbf{0}, \quad t \leftarrow 0$
 5: while not converged do
 6:     $t \leftarrow t + 1$
 7:     # Compute $\hat{y}_i = p(y|\mathbf{x}_i, \mathbf{W})$ on random data point $\mathbf{x}_i$
 8:     $z_i = e^{\mathbf{W}^\top \mathbf{x}_i}$
 9:     $\hat{y}_i = z_i / \|z_i\|_1$
10:     # Update weight vectors
11:     $\mathbf{W} \leftarrow \mathbf{W} + 1/t \ \ \mathbf{x}_i(\hat{y}_i - y_i)^\top$
12: end while
```

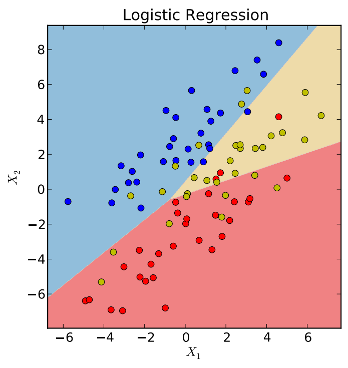


Figure 19.2: . Three class prediction problem. True labels are indicated as dot color, predicted labels as background colors.

DISCUSSION Logistic Regression is the workhorse of data scientists. It is easy to implement and yields easily interpretable results. Since the objective function is convex, the solution is guaranteed to be the global optimum for the training data. Also it is very efficient: Other classification algorithms such as linear discriminant analysis need to estimate the covariance matrix of the data and the mean, which are $K(D + D(D+1)/2)$ parameters for D-dimensional data and K classes. In contrast Logistic Regression needs only to estimate DK parameters and will thus need fewer data points to estimate these parameters.

EXAMPLE Figure 19.1 shows the result of Logistic Regression for a two class prediction problem. The color of the dots indicates the class membership. The color of the background indicates the prediction, i.e. $p(y = \text{red}|\mathbf{x}, \mathbf{W})$, the predicted probability that at a data point at this location belongs to the red class. Figure 19.2 shows an example for a three class problem. Here the background color indicates predicted class membership, that k that maximizes $p(y = k|\mathbf{x}, \mathbf{W})$.

SEE ALSO section B.6

20

Linear Support-Vector Machine

APPLICATIONS Maximum-Margin Classification

OBJECTIVE SUPPORT-VECTOR MACHINES (SVMs)[1] are among the most popular classification algorithms. In this chapter we will only discuss *linear* decision boundaries, non-linear SVMs are discussed in chapter 21. Like all linear classification algorithms also linear SVMs try to find a decision boundary (also called *hyperplane*), usually defined by its normal vector **w** between the data points of two classes. Linear classification and hyperplanes are illustrated in section C.2. The key idea in SVMs is that this decision boundary is placed such that the *margin* is maximized; the margin is the area between the decision boundary and the data points of each class (gray area in Figure 20.1). One could place the decision boundary differently, but this would automatically *decrease* the margin area – and importantly it would *increase* the length (or norm) of **w**. Placing the boundary such that the margin is maximal (and the length of **w** is minimal) ensures that the learned classification rule generalizes well to new data. The hyperplane with the maximal margin can be found by minimizing the error function

[1] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995

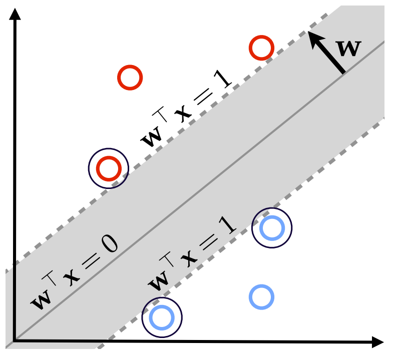

$\mathbf{w}$
$\mathbf{w}^\top\mathbf{x} = 1$
$\mathbf{w}^\top\mathbf{x} = 0$
$\mathbf{w}^\top\mathbf{x} = 1$

Figure 20.1: Two class prediction problem; red dots indicate positive examples (i.e. $y = +1$) and blue dots negative examples ($y = -1$). The hyperplane defined by **w** with the maximum margin (grey area) between hyperplane and the closest points to the hyperplane optimally separates the two classes. Encircled data points are referred to as *support-vectors*, they hold the decision boundary in place.

$$\mathcal{E}(\mathbf{w}) = \sum_{i=1}^{N} \max\left(0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i)\right) + \frac{1}{2}\lambda\|\mathbf{w}\|_2^2. \tag{20.1}$$

The first term ensures that the errors made by the decision rule are small. For correctly classified data that lies on the right side of the boundary and outside the margin, this term will return zero and have no effect on the objective function. For data that lies on the wrong side of the classification rule or just within the margin, this term will return the absolute error. These data points are called *support-vectors* (marked by gray circles in Figure 20.1): they hold the decision boundary in place. The other term, weighted with the Lagrangian multiplier λ constrains the hyperplane **w** to have a small norm. Usually optimization of SVMs involves heavy math. Here we use only standard first order stochastic gradient descent (SGD) [Robbins and Monro, 1951]. This optimization has some advantages: it is very easy to understand and implement;

also it scales to big data sets. The gradient of Equation 20.1 is

$$\frac{\partial \mathcal{E}(\mathbf{w})}{\partial \mathbf{w}} = \begin{cases} y_i \mathbf{w}^\top \mathbf{x}_i > 1 : & \lambda \mathbf{w} \\ y_i \mathbf{w}^\top \mathbf{x}_i \leq 1 : & \lambda \mathbf{w} - y_i \mathbf{x}_i \end{cases} \tag{20.2}$$

PSEUDOCODE The pseudocode for first order SGD optimization of a linear SVM is given in Algorithm 19.

Algorithm 19 Linear Support-Vector Machine

```
Input: Data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N), \mathbf{x} \in \mathbb{R}^D,\ y_i \in \{-1, +1\},\ \lambda$
Output: Normal vector $\mathbf{w}$ of decision hyperplane
 1: # Include bias term $\mathbf{X} = [\mathbf{1}_N \mathbf{X}^\top]^\top$
 2: # Initialize weight vector $\mathbf{w} = 0$, initialize counter $i = 0$
 3: while Not Converged do
 4:     $i = i + 1$
 5:     Pick a random data point $\mathbf{x}_i$
 6:     # Compute Gradient
 7:     if $y_i \mathbf{w}^\top \mathbf{x}_i > 1$ then
 8:         $\mathbf{g} = \lambda \mathbf{w}$
 9:     else
10:         $\mathbf{g} = \lambda \mathbf{w} - y_i \mathbf{x}_i$
11:     end if
12:     Update weight vector
13:     $\mathbf{w} \leftarrow \mathbf{w} - 1/i\ \mathbf{g}$
14: end while
```

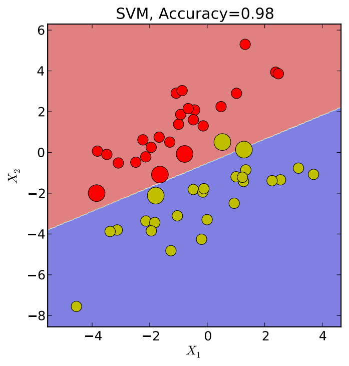


Figure 20.2: . Two class prediction problem. True labels are indicated as dot color, predicted labels are indicated as colored background. Support-Vectors are plotted as bigger dots.

DISCUSSION SMVs have been extremely successful in various applications and have become state-of-the-art in many fields. Computationally and in terms of classification performance SVMs are often similar to logistic regression (chapter 19). There are however reasons why logistic regression is often preferred: First SVMs produce non-probabilistic outputs which can be difficult to interpret, whereas logistic regression provides probabilistic outputs. Also the multiclass extension is conceptually more elegant in the logistic regression case.

EXAMPLE Figure 20.2 shows the result of a linear SVM for a two class prediction problem. The color of the dots indicates the class membership. The color of the background indicates the prediction. Support-Vectors are indicated as bigger dots – these dots fall inside the margin or are incorrectly classified.

SEE ALSO Linear classification (section C.2), Logistic Regression (chapter 19), non-linear kernel SVMs (chapter 21)

21

Non-linear Support-Vector Machine

APPLICATIONS Non-linear classification

OBJECTIVE Some classification problems cannot be solved with linear classifiers. A famous toy data example is the XOR problem illustrated in Figure 21.1. The goal of NON-LINEAR SUPPORT-VECTOR MACHINES[1] is to learn a non-linear transformation of the data $\phi^*(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathcal{S}$ to some kernel feature space $\mathcal{S}$ such that the data can be classified accurately. At the same time the SVM objective tries to maximize the margin between data points and decision boundary (see chapter 20). It can be difficult to specify ϕ^* explicitely. Non-linear SVMs use the kernel trick (see section A.1) in order to express the predictions $\hat{y}_i = \phi^*(\mathbf{x}_i)$ as a linear combination of data points, or rather a linear combination of similarities between data points

[1] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995

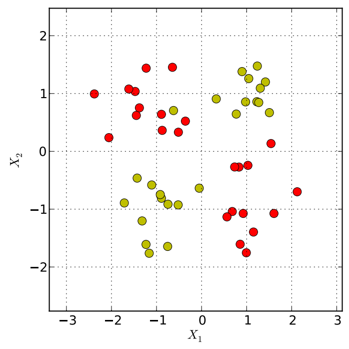


Figure 21.1: . The XOR problem, a two class nonlinear prediction problem. Data from one class (yellow dots) is drawn from a gaussian distribution around $[1, 1]^\top$ and $[-1, -1]^\top$ while data from the other class (red dots) were drawn from a gaussian distribution centered around $[1, -1]^\top$ and $[-1, 1]^\top$. Note that there is no linear classification boundary that separates the two classes.

$$\phi^*(\mathbf{x}_i) = \sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{x}_j)\mathbf{a}_j$$

where the kernel function $k(.,.)$ measures the similarity of data points in kernel feature space by computing inner products in $\mathcal{S}$

$$\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_i) \rangle_{\mathcal{S}}$$

The objective function minimized by SVMs depends on the data and on the weighting coefficients $\mathbf{a}$

$$\mathcal{E}(\mathbf{a}) = \frac{1}{2}\lambda \|\mathbf{a}\|_2^2 + \sum_{i=1}^{N} \max\left(0, 1 - y_i\left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{x}_j)\mathbf{a}_j\right)\right). \tag{21.1}$$

Again, as in chapter 20 we will use first order stochastic gradient descent (SGD) to optimize Equation 21.1. There are better methods, but none is easier to implement. All we need for this is the first derivative with respect to $\mathbf{a}$

$$\frac{\partial \mathcal{E}(\mathbf{a})}{\partial \mathbf{a}} = \begin{cases} y_i \mathbf{a}^\top k(\mathbf{X}_{j \neq i}, \mathbf{x}_i) > 1: & \lambda \mathbf{a} \\ y_i \mathbf{a}^\top k(\mathbf{X}_{j \neq i}, \mathbf{x}_i) \leq 1: & \lambda \mathbf{a} - y_i k(\mathbf{X}_{j \neq i}, \mathbf{x}_i) \end{cases} \tag{21.2}$$

PSEUDOCODE Algorithm 20 summarizes the algorithm for training a non-linear kernel SVM using first order stochastic gradient descent. We use the short cut $k(\mathbf{X}, \mathbf{x}_i)$ to denote the kernel function evaluated on all columns of $\mathbf{X}$.

Algorithm 20 Non-Linear Support-Vector Machine

```
Input: $(\mathbf{x}_i, y_i), i \in \{1, \dots, N\}, \mathbf{x}_i \in \mathbb{R}^D,\ y_i \in \{-1, +1\}$, Regularizer $\lambda$,
  Kernel $k(.,.)$ (for example kernel functions see section A.2)
Output: Dual coefficients $\boldsymbol{\alpha}$
  # Initialize coefficients $\boldsymbol{\alpha} = 0$, initialize counter $i = 0$
  while Not Converged do
    $i = i + 1$
    Pick a random data point $\mathbf{x}_i$
    Compute Gradient $\mathbf{g}$
    if $y_i \boldsymbol{\alpha}^\top k(\mathbf{X}, \mathbf{x}_i) > 1$ then
      $\mathbf{g} = \lambda \boldsymbol{\alpha}$
    else
      $\mathbf{g} = \lambda \boldsymbol{\alpha} - y_i k(\mathbf{X}, \mathbf{x}_i)$
    end if
    Update weight vector
    $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - 1/i\ \mathbf{g}$
  end while
```

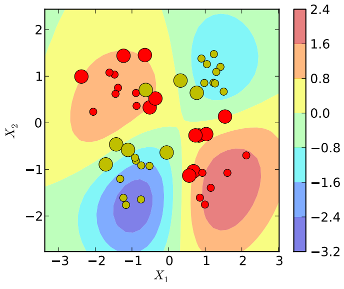


Figure 21.2: . Non-linear SVM applied to the XOR problem. True labels are indicated as dot color, SVM predictions are indicated as colored background. Support-Vectors are plotted as bigger dots and are located close to the decision boundary.

DISCUSSION Kernel SVMs with non-linear kernels (Equation A.9 or Equation A.8) can find arbitrarily complex decision boundaries and thus learn the data-label-relationship by heart. This overfitting to a training data set is useless (see section 6.1). Especially for non-linear methods it is crucial to control the complexity of the model by cross-validation during training – this is less of an issue with linear models. A computational drawback of kernel SVMs is that they require to store in memory all data points and each time a prediction or gradient step in the optimization is made, every training data point has to be accessed. This can become very slow. A simple solution is to not use all data points for training but only random subsets or clusters, extracted e.g. by K-Means (chapter 12). For an in depth review of kernel SVMs we refer to standard textbooks [Shawe-Taylor and Cristianini, 2004, Schölkopf and Smola, 2002].

Linear models – especially in low dimensions – are too simple to overfit as drastically as non-linear methods.

EXAMPLE Figure 21.2 shows the result of a non-linear SVM for the XOR two class prediction problem illustrated in Figure 21.1.

SEE ALSO Linear SVMs (chapter 20), kernel trick (section A.1), kernel functions (section A.2)

22

K-Nearest Neighbour Classification

APPLICATIONS Non-linear Classification

OBJECTIVE K-NEAREST NEIGHBOUR (KNN) is a very simple classifier that can learn non-linear classification rules. In contrast to other methods, no model is computed for k-NN – the data itself is the model. So the entire data set $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$ and the corresponding labels $\mathbf{y} \in \mathbb{Z}^{1 \times N}$is stored in memory and for each test data point $\mathbf{x}_i$ the distance $d(\mathbf{x}_i, \mathbf{x}_j)$ to all training data points $\mathbf{x}_j \in \mathbf{X}$ is computed. The distance function most often used is the euclidean distance so we compute

$$d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2 \qquad j \in [1, \dots, N] \tag{22.1}$$

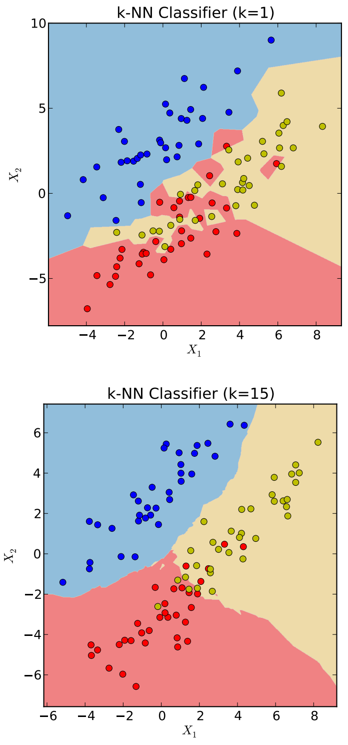


Figure 22.1: . An example of k-Nearest Neighbour classification on a linear classification task. Plotted are training data as dots, background color indicates k-NN predictions with $k = 1$ (*top*) and $k = 15$ (*bottom*). Increasing k has a smoothing effect on the prediction surface.

If the data points are not numerical we can apply standard techniques to make them numerical (**??**) or use distance functions that are tailored to non-numerical attributes (section C.4). Then the labels of the k data points with the smallest distances, in other words the labels of the k nearest neighbours of the test data point $\mathbf{x}_i$, are collected. The predicted label is simply the majority vote of the k nearest neighbours, so that label that occurred most often in the direct neighbourhood of $\mathbf{x}_i$. Ties are broken at random. The number of neighbours to consider is the only parameter of k-NN classification and has to be optimized using cross-validation (paragraph 6.2).

DISCUSSION K-NN is so simple it does not even need training. Instead of extracting rules from a data set (as all other methods in this book do), k-NN simply transfers the computational load to storing the training data and retrieving predictions. For large data sets this is not feasible. But of course one can combine k-NN classification with other methods in order to make it scalable to larger data sets. Computing a number of clusters on the data for each class and using the cluster centers as new data points can decrease the memory demands as well as computational complexity for obtaining predictions substantially. The performance of k-NN – liike that of clustering algorithms (e.g. chapter 12) and nearest-centroid classification (chapter 15) depends critically on whether or not the distance function used is appropriate for the type of data at hand. Using the right distance functions (see section C.4) or whitening the data (corresponding to using the Mahalanobis [Mahalanobis, 1936] distance, see **??**) can help in cases where the standard euclidean distance does not yield satisfactory results. Equivalent to whitening is transforming the data linearly by PCA (chapter 7) and rescaling the inverse eigenvalues. In analogy one can perform kernel PCA (see chapter 8), this corresponds to using a non-linear distance function.

PSEUDOCODE Algorithm 21 shows the pseudocode for k-Nearest Neighbour prediction.

Algorithm 21 K-Nearest Neighbour Prediction

```
Input: Test data point $\mathbf{x}_i \in \mathbb{R}^D$, training data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}$,
    corresponding labels $\mathbf{y} = [y_1, \ldots, y_N]^\top \in \mathbb{R}^N$, number of neighbours $K$
Output: Predicted label $\mathbf{y}_i$
1:  for $\mathbf{x}_j in \mathbf{X}$ do
2:      # Compute distance between test data $\mathbf{x}_i$ and training data $\mathbf{x}_j$
3:  end for
4:  Initialize a $K$-dimensional zero vector $\gamma$
5:  for $k$ in  do
6:      Count label of the $k$-nearest neighbor
7:      # $\gamma_{\mathbf{y}_k} \leftarrow \gamma_{\mathbf{y}_k} + 1$
8:  end for
9:  Predicted label is that with most votes (ties are broken at random)
10: $\mathbf{y}_i = \mathrm{argmax}_\gamma$
```

EXAMPLE Figure 22.1 shows the standard linear classification example we used in previous chapters to illustrate classification performance. We predicted the training data points (indicated as background color) with $k = 1$ and $k = 15$ neighbours. The effect of increasing k is intuitively that the regions in which a particular label is assigned become more contingent. Figure 22.2 shows a typical non-linear classification example. The effect of increasing k is the same as in the linear classification problem, but note that k-NN perfectly captures the structure of the data and can correctly classify test data, even though there was no training at all prior to prediction.

SEE ALSO Kernel Trick (section A.1)

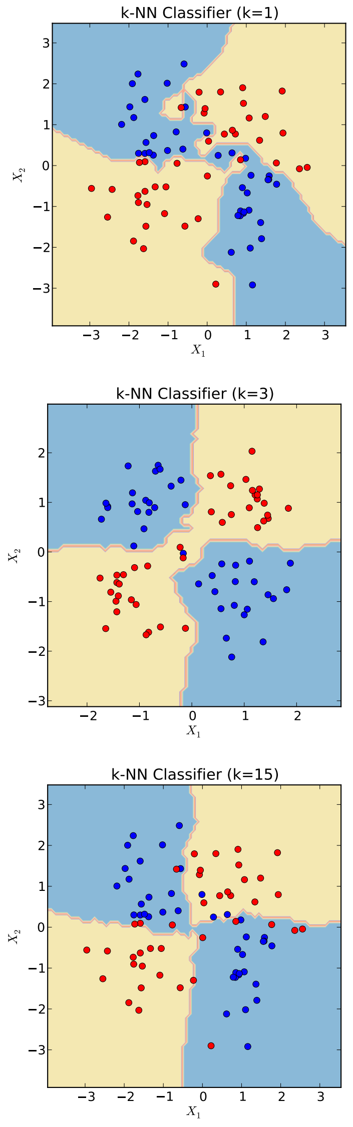


Figure 22.2: . An example of k-Nearest Neighbour classification on a non-linear classification task, the XOR problem. Plotted are training data as dots, background color indicates k-NN predictions with $k = 1$ (*top*), $k = 3$ (*middle*) and $k = 15$ (*bottom*). Increasing k has a smoothing effect on the prediction surface.

23

Kernel Ridge Regression

APPLICATIONS Non-linear Regression

OBJECTIVE KERNEL RIDGE REGRESSION (KRR) is a kernel version of regularized least-squares regression (chapter 17). Remember the solution to ridge regression (Equation 17.4). In KRR the kernel trick (see section A.1) is used to express $\mathbf{w}$ as a linear combination of data points, i.e. $\mathbf{w} = \mathbf{X}\boldsymbol{\alpha}$. The solution to KRR, the coefficients $\boldsymbol{\alpha}$, are obtained by

$$\boldsymbol{\alpha} =(\underbrace{\mathbf{X}^\top\mathbf{X}}_{\mathbf{K}}+\lambda\mathbf{I})^{-1}\mathbf{y}^\top \tag{23.1}$$

which looks very similar to the standard ridge regression solution, only that instead of inverting the covariance matrix $\mathbf{X}\mathbf{X}^\top$, KRR inverts the *kernel matrix* $\mathbf{K}$. For a derivation of Equation 23.1 see section A.1. Note that the solution to linear kernel ridge regression in Equation 23.1 can be extended straightforwardly from linear functions $f(\mathbf{x}) = \mathbf{w}^\top\mathbf{x}$ to non-linear functions $f(\mathbf{x}) = \phi^*(\mathbf{x})$ by replacing the kernel function with a non-linear kernel (see section A.2). KRR is closely related to a technique known as *Kriging* [Krige, 1951] in the geosciences or *Gaussian Processes* (GPs) [Rasmussen and Williams, 2005] in the Machine Learning community. KRR can compute predictions $f(\mathbf{x})$, GPs allow to compute the probability of these predictions. For a training set indexed by $j = [1, 2, \ldots, J]$ and a test data indexed by $i = [1, 2, \ldots, I]$ this estimate of $p(y_i|\mathbf{x}_j)$ is modeled as a Gaussian distribution

KRR predictions are the mean function of Gaussian Processes (GP) or Kriging. These techniques allow to compute confidence intervals for the predictions.

$$p(y_i|\mathbf{x}_j) \sim \mathcal{N}\left(\boldsymbol{\mu}_i, \Sigma_i\right)) . \tag{23.2}$$

The output of KRR is mean $\boldsymbol{\mu}_i$ and the covariance Σ_i can be computed from the kernel of the training data $\mathbf{K}_{jj} \in \mathbb{R}^{J\times J}$, the kernel between test and training data $\mathbf{K}_{ij} \in \mathbb{R}^{I\times J}$ and the kernel between testpoints $\mathbf{K}_{ii} \in \mathbb{R}^{I\times I}$

$$\boldsymbol{\mu}_i = \sum_{j=1}^{J}\mathbf{K}_{ij}\boldsymbol{\alpha}_j \qquad \Sigma_i = \mathbf{K}_{ii} - \mathbf{K}_{ij}\left(\mathbf{K}_{jj} + \lambda\mathbf{I}\right)^{-1}\mathbf{K}_{ij}^\top . \tag{23.3}$$

PSEUDOCODE The pseudocode for computing the coefficients $\boldsymbol{\alpha}$ is given in algorithm 22. Algorithm 23 shows how to compute predictions of KRR and confidence intervals as by the Kriging or GP framework.

Algorithm 22 Kernel Ridge Regression - Training

```
Input: Kernel matrix $\mathbf{K} \in \mathbb{R}^{N\times N}$,labels $[y_1,\dots,y_N]^\top \in \mathbb{R}^U$, ridge $\kappa$
Output: $\boldsymbol{\alpha}$
1: # Compute dual coefficients
2: $\boldsymbol{\alpha} = (\mathbf{K} + \mathbf{I}\kappa)^{-1}\mathbf{y}^\top$
```

Algorithm 23 Gaussian Process - Prediction

```
Input: Kernel of training data $\mathbf{K}_{jj} \in \mathbb{R}^{J\times J}$, kernel between test and
    training data $\mathbf{K}_{ij} \in \mathbb{R}^{I\times J}$, the kernel between testpoints $\mathbf{K}_{ii} \in \mathbb{R}^{I\times I}$,
    coefficients $\boldsymbol{\alpha}$, regularizer $\lambda$
Output: Parameters of $p(y|\mathbf{X}_{\text{train}}) \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$
1: Compute mean of predicted label distribution $p(y|\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{test}})$
2: $\boldsymbol{\mu} = (\mathbf{K}_{ij}\boldsymbol{\alpha})^\top$
3: Compute covariance $\Sigma$
4: $\Sigma = \mathbf{K}_{ii} - \mathbf{K}_{ij}(\mathbf{K}_{jj} + \kappa\mathbf{I})^{-1}\mathbf{K}_{ij}^\top$
```

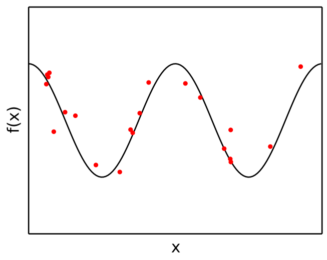


Figure 23.1: . An example of a non-linear prediction problem: the function to be learned $f(\mathbf{x}) = \cos(\mathbf{x})$ is plotted in black. But we can only see training data at the positions indicated as red dots.

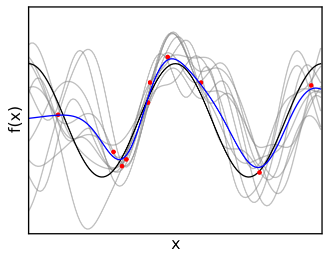


Figure 23.2: . Same data as in Figure 23.1. The KRR prediction (the mean $\boldsymbol{\mu}$ of the GP) is plotted in blue, and data sampled from the GP according to Equation 23.2 is plotted in grey. Note that the variance of these samples is large where we did not see data points; where the algorithm saw data points the variance is small and the mean accurately reflects the true underlying function $\cos(\mathbf{x})$.

DISCUSSION Kernel ridge regression may be the kernel methods which can be implemented most easily. The downside is that it does not scale very well – like all kernel methods: Inversion of the kernel matrix might be practical up to a few thousand data points. KRR also does not produce sparse solutions, all of the $\boldsymbol{\alpha}$ will be non-zero. But there are some tricks to speed up training KRR models, see section C.3. These thoughts aside, KRR is just as powerful as Support Vector Machines. KRR can also be seen as a part of GPs or Kriging; using these algorithms one can actually compute *confidence estimates* of the predictions which is a decisive advantage compared to non-probabilistic methods like kernel SVMs (chapter 21) or the standard version of KRR.

SEE ALSO Kernel Trick (section A.1)

24

Random Kitchen Sinks

APPLICATIONS Non-linear Regression

OBJECTIVE RANDOM KITCHEN SINKS (RKS) [1,2] is a technique for non-linear regression. Similar to kernel methdos, RKS aims at linearizing a non-linear problem. Unlike kernel methods, which essentially use the data points as basis vectors to model a non-linear function, RKSs use a random set of basis vectors $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_C]$, $\mathbf{w}_c \in \mathbb{R}^{D \times K}$, $\mathbf{w} \sim \mathcal{P}$, where the vectors $\mathbf{w}_c$ are drawn from a distribution $\mathcal{P}$. The data points $\mathbf{x}_n \in \mathbb{R}^D$ are transformed into *random fourier features* $\mathbf{z}_n$ by projecting them onto this random basis and applying a non-linear function ϕ to the mapped data

[1] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NIPS*, 2007

[2] Ali Rahimi and Benjamin Recht. Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning. In *NIPS*, 2008

$$\mathbf{z}_n = \phi(\mathbf{W}^\top \mathbf{x}_n), \tag{24.1}$$

where ϕ is usually the Fourier Transform of a shift-invariant kernel function (see section A.2). After the linear and the non-linear transform non-linear problems can be solved as a linear regression problem (see chapter 17). In the simplest case we can consider non-linear regression with with a quadratic loss function. Given a univariate target variable $y_n \in \mathbb{R}$, $n = 1, \dots, N$ the non-linear function can be modelled as a linear combination of the featurized data

$$\underset{\mathbf{a}}{\operatorname{argmin}} \quad \frac{1}{N} \sum_{n=1}^{N} (y_n - \mathbf{a}^\top \mathbf{z}_n)^2 \qquad \text{subject to } \|\mathbf{a}\|_2 < \rho, \tag{24.2}$$

where the norm constraint ensures robust solutions and controls the complexity of the solution (see Equation 17.4 in chapter 17). Other loss functions that the quadratic loss can be used, but for the mean squared loss, the optimal $\mathbf{a}$ is simply the standard ridge regression solution

$$\mathbf{a} = (\mathbf{Z}\mathbf{Z}^\top + \mathbf{I}\lambda)^{-1}\mathbf{Z}\mathbf{y} \tag{24.3}$$

where $\mathbf{Z} = [\mathbf{z}_1, \dots, \mathbf{z}_N]$ is a matrix containing the featurized data points, $\mathbf{y} = [y_1, \dots, y_N]^\top$ is a vector containing the labels and λ is a regularization constant, inversely proportional to the norm constraint ρ.

Predictions of the model can be obtained by computing

$$\hat{y}_n = \mathbf{a}^\top \mathbf{z}_n. \tag{24.4}$$

PSEUDOCODE The pseudocode for a weighted sum of random kitchen sinks using ridge regression, as proposed in [Rahimi and Recht, 2008], is shown in Algorithm 24.

Algorithm 24 Random Kitchen Sink Regression

Input: $\mathbf{X} \in \mathbb{R}^{D\times N}$, $\mathbf{y}^\top \in \mathbb{R}^{1\times N}$, ridge λ, Fourier-Transform of a kernel function ϕ, number of random basis functions K, distribution $\mathcal{P}$ to draw basis functions from

Output: Random Basis $\mathbf{W} \in \mathbb{R}^{D\times K}$, Weight vector $\mathbf{a} \in \mathbb{R}^K$ for linear mapping from feature space to labels

```
1: # Create random basis
2: $\mathbf{W} \sim \mathcal{P}$
3: # Featurize input data
4: $\mathbf{Z} = [\phi(\mathbf{W}^\top \mathbf{x}_1), \ldots, \phi(\mathbf{W}^\top \mathbf{x}_N)]$
5: # Do linear regression in feature space
6: $\mathbf{a} = (\mathbf{Z}\mathbf{Z}^\top + \lambda \mathbf{I})^{-1}\mathbf{Z}\mathbf{y}$
```

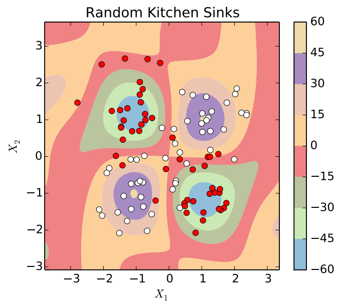


Figure 24.1: Non-linear classification of the XOR-problem using RKS. Using RKS regression with $K = 100$ random basis functions drawn from a zero mean gaussian distribution with unit variance $\mathcal{N}(0,1)$ and a non-linearity corresponding to the fourier transform of a gaussian kernel function $\phi = e^{\sqrt{-1}\mathbf{W}^\top \mathbf{x}}$ RKS finds a solution comparable to that of a non-linear SVM (see chapter 21)

DISCUSSION Using kernel methods for very large data sets can become infeasible as the size of the kernel matrix grows with the number of data points. A simple solution is to use only a random sample of data points to compute the kernel matrix [Achlioptas et al., 2001]. In practice one samples only as much data points as can be conveniently processed. The idea of RKSs is somewhat similar, but instead of randomly sampling data points, we can adjust the problem size to the computational resources by choosing an appropriate number of random basis functions K. An advantage of RKS is its simple implementation and the fact that solutions to Equation 24.3 can be obtained with fast off-the-shelf linear algebra routines. A potential disadvantage of RKS is that the solutions might be difficult to interpret – as are the solutions for kernel methods. Figure 24.1 shows a toy data classification example for the XOR-problem as in chapter 21.

SEE ALSO Kernel Ridge Regression (chapter 23), Ridge Regression (chapter 17)

Appendix

A

Kernels

A.1 Kernel Trick

The KERNEL TRICK in machine learning has become a standard tool for data analysis. A KERNEL FUNCTION $k(\mathbf{x}_i, \mathbf{x}_j)$ is a *similarity measure* between data points. The kernel matrix

$$\mathbf{K} \in \mathbb{R}^{N \times N}, \quad \mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) \tag{A.1}$$

PRO KERNEL METHODS

+ Accurate Predictions
+ Algorithms independent of kernel
+ Complexity independent of data dimensionality

CONTRA KERNEL METHODS

– Difficult too interpret kernel space
– Complexity grows with number of samples

→ For Big Data (many samples):

- large memory demands
- slow training / testing

containins all pairwise similarities between data points. Some examples for $k(.,.)$ are listed in section A.2. The "trick" essentially has two main aspects to it. First $k(.,.)$ does not necessarily compute the similarity between data points in the original space of the data, but on some non-linearly mapped version of the data. However this mapping is never computed explicitely. Instead the kernel function computes both, the data transformation and the similarity in one step. The second aspect, which can lead to substantial computational advantages, is that the function of interested that is learned by a kernel algorithm is expressed as a linear combination of training data points.

There is a huge and often intimidatingly technical body of literature on kernels. The theoretically inclined reader is referred to the standard textbooks [Schölkopf and Smola, 2002, Shawe-Taylor and Cristianini, 2004] or review papers [Müller et al., 2001, Hofmann et al., 2008]. For the practitioner it is enough to know first *why and when* one should make use of it and second *how* to implement it. In order to get an intuition about the kernel trick we here consider just linear Ridge Regression (chapter 17) as an example and turn it into linear kernel ridge regression; but the argument extends straightforwardly to the non-linear case.

Why should one use the kernel trick?

Roughly speaking the kernel trick is useful if ...

1. *... the data has more dimensions than samples.*

2. *... the problem requires (unknown) non-linear transformations of the data.*

In the following we explain these settings and why kernels help here.

High-dimensional data, few data points Consider a data set $\mathbf{X} \in \mathbb{R}^{D \times N}$ with D dimensions / features and N data points / samples and corresponding labels $\mathbf{Y} \in \mathbb{R}^{1 \times N}$. If the goal of an analysis is to estimate a linear regression model in this case the solution $\mathbf{w} \in \mathbb{R}^{D \times 1}$ has D parameters (or $D+1$ if we include an offset parameter). The regularized ordinary least solution is then given by Equation 17.4 as

$$(\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}\mathbf{X}\mathbf{Y}^\top$$

If we are in a high-dimensional / low sample size setting, then $D > N$ and we have more parameters to estimate than we have samples available. . As we have only N data points, we can be sure that the solution $\mathbf{w}$ – although it lives in D dimensions – can have only N degrees of freedom. It is difficult to illustrate this in low-dimensional scenarios but you can think of the data living in two dimensions but all points in the data set fall on a line – so the dimensionality is two but the data can be described by just one number, the position on the line.

In other words the covariance matrix $\mathbf{X}\mathbf{X}^\top$ that needs to be inverted will not be full rank and thus not be invertible. The solution of ridge regression is to make it invertible by adding a ridge.

In geometric terms, this means that if $D > N$ the solution $\mathbf{w}$ has to lie in the space spanned by the data points in $\mathbf{X}$. If all data points (columns of $\mathbf{X}$) fall on a line, the solution $\mathbf{w}$ has to lie on that line, too. More formally this means that $\mathbf{w}$ can be expressed as a linear combination of data points

The main intuition behind kernel methods: The solution of a problem is expressed as a linear combination of data points.

$$\mathbf{w} = \mathbf{X}\boldsymbol{\alpha} \tag{A.2}$$

In our example of ridge regression this can be easily derived from the objective function. As described in section B.4, the derivative of the ridge regression error function is

$$\begin{aligned} \frac{\partial \mathcal{E}_{RR}(\mathbf{w})}{\partial \mathbf{w}} &= -2\mathbf{X}\mathbf{y}^\top + 2\mathbf{X}\mathbf{X}^\top\mathbf{w} + \lambda 2\mathbf{w} \\ \rightarrow \mathbf{w} &= \mathbf{X}\underbrace{\frac{1}{\lambda}(\mathbf{y} - \mathbf{X}^\top\mathbf{w})}_{\boldsymbol{\alpha}} \end{aligned} \tag{A.3}$$

and we see that the optimal $\mathbf{w}$ is a **linear combination** $\boldsymbol{\alpha}$ of all data points $\mathbf{X}$. This is probably the most important insight about kernel methods. Using Equation A.3 we can derive the solution of the ridge regression objective function by

$$\begin{aligned} \boldsymbol{\alpha} &= \frac{1}{\lambda}(\mathbf{y} - \mathbf{X}^\top\mathbf{w}) \\ \lambda\boldsymbol{\alpha} &= \mathbf{y} - \mathbf{X}^\top\mathbf{X}\boldsymbol{\alpha} \\ \mathbf{y} &= (\mathbf{X}^\top\mathbf{X} + \mathbf{I}\lambda)\boldsymbol{\alpha} \\ \rightarrow \boldsymbol{\alpha} &= (\underbrace{\mathbf{X}^\top\mathbf{X}}_{\mathbf{K}} + \mathbf{I}\lambda)^{-1}\mathbf{y}^\top \end{aligned} \tag{A.4}$$

Equation A.4 contains the kernel matrix $\mathbf{K}$ computed with a linear kernel function, where $\mathbf{K}_{ij} = \mathbf{x}^\top \mathbf{x}$. This is a very special case but one advantage of kernel methods is, once you have derived a kernel algorithm and you have a set of valid kernel functions (see e.g. section A.2), you can simply replace the kernel function and obtain a new algorithm. This allows to construct powerful analysis tools for non-linear problems. What this means is explained in the next paragraph.

Non-linear problems We refer to a non-linear problem as an analysis setting in which the structure or relationship of interest in a data set cannot be described by a linear combination of features of the data. A very simple example of a non-linear relationship between the data x and a target variable $y = f(x) = \cos(x)$ is plotted in Figure A.1. Other examples are shown in Figure 8.1 or Figure 21.2. The most straightforward solution to non-linear problems is to extract features of interest and learn a linear combination of these features. But often it is not easy to come up with a non-linearity that describes the data well. Finding a good non-linear feature is especially difficult when dealing with high-dimensional data. In the example shown in Figure A.1 it is clear that the best feature would be $\cos(x)$.

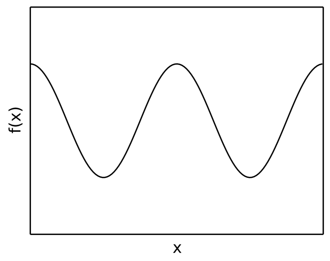


Figure A.1: Example of a non-linear relationship between one-dimensional data points $x \in \mathbb{R}^1$ and a target variable $f(x) = \cos(x)$. Note that $f(x)$ cannot be modeled as a linear function of x. If the non-linearity is unknown and data high-dimensional, kernel methods can help to capture complex structure in the data.

Kernel methods circumvent the need for finding non-linear features explicitly. Instead one uses a generic kernel function that computes a non-linear function of the data features. This non-linear function is expressed as a weighted sum of similarities between data points. The idea is to use a mapping $\phi(\mathbf{x})$ that projects a data point to a *kernel feature space* and express ϕ^* – the non-linear function of interest (e.g. the $f(x) = \cos(x)$) – as in Equation A.2 by

$$\phi^* = \sum_{i=1}^{N} \phi(\mathbf{x}_i)\boldsymbol{\alpha}_i. \tag{A.5}$$

Computing the prediction for a new data point $\phi^*(\mathbf{x}_j)$ is then done by computing the inner product (denoted $\langle . \rangle$) between $\phi(\mathbf{x}_j)$ and a weighted sum of all $\phi(\mathbf{x}_{i \neq j})$

$$\phi^*(\mathbf{x}_j) = \sum_{i=1}^{N} \langle \phi(\mathbf{x}_j), \phi(\mathbf{x}_i) \rangle \boldsymbol{\alpha}_i = \sum_{i=1}^{N} k(\mathbf{x}_j, \mathbf{x}_i)\boldsymbol{\alpha}_i = \mathbf{K}_{j,:} \tag{A.6}$$

where $\mathbf{K}_{j,:}$ denotes the jth column of the kernel matrix, corresponding to the similarity between $\mathbf{x}_j$ and all $\mathbf{x}_{i \neq j}$. The kernel trick here is that one never has to compute or optimize ϕ explicitely. Instead one uses Equation A.5 – the fact that in a high-dimensional feature space the solution can be cast as a linear expansion of data points – in combination with the kernel function $k(\mathbf{x}_j, \mathbf{x}_i)$ that directly computes the inner product of $\phi(\mathbf{x}_j)$ and $\phi(\mathbf{x}_i)$.

Implementation of Kernel Algorithms

All kernel methods have in common that they require three main steps. Given some data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]$ and corresponding labels $\mathbf{Y} = [\mathbf{y}_1, \ldots, \mathbf{y}_N] \in \mathbb{R}^{E \times N}$ these steps are summarized in algorithm 25.

Algorithm 25 Kernel Algorithms

```
Input: $\mathbf{X}_{\text{Test}}, \mathbf{X}_{\text{Train}}, \mathbf{A}$, kernel function $k(.,.)$
Output: Data projected on non-linear principal components $\hat{\mathbf{X}}$
1: Compute Kernel
2: $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_i)$
3: Compute weights $\mathbf{A} \in \mathbb{R}^{N \times E}$ for each training sample
4: $\mathbf{A} = \text{someKernelAlgorithm}(\mathbf{K}, \mathbf{Y})$
5: Compute predictions
6: $\hat{\mathbf{y}}_j = \sum_{i=1}^{N} (k(\mathbf{x}_j, \mathbf{x}_i)\mathbf{A}_{:,i}) = \mathbf{K}_{j,:}\mathbf{A}$
```

A major advantage of kernel methods is that kernel functions (step 1) and algorithm (step 3) are separated and can be replaced easily. This modularity makes kernel methods extremely flexible. In most cases, the coefficient matrix $\mathbf{A}$ will have only one column, but for kernel ridge regression with multidimensional outputs it will have as many columns as output dimensions.

A.2 Kernel Functions

Some popular kernel functions are:

Linear Kernel

$$k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j \tag{A.7}$$

Polynomial Kernel

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j + c)^p \tag{A.8}$$

Gaussian Kernel

$$k(\mathbf{x}_i, \mathbf{x}_j) = e^{||\mathbf{x}_i - \mathbf{x}_j||_2^2 / -2\sigma} \tag{A.9}$$

A.3 PCA for high-dimensional data

Principal Component Analysis of a data set $\mathbf{X} \in \mathbb{R}^{D \times N}$ (D dimensions, N samples) is based on an eigendecomposition of the covariance matrix $\mathbf{X}\mathbf{X}^\top$. If the data has many more dimensions than samples $\mathbf{X} \in \mathbb{R}^{D \times N}$ will not be full rank and thus will not be invertible. In this case one can easily compute PCA on the *linear kernel matrix* $\mathbf{X}^\top\mathbf{X}$ instead. The relationship between the eigenvectors of $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}\mathbf{X}^\top$ is explained best using Singular Value Decomposition (SVD). By SVD we can decompose $\mathbf{X}$ into

$$\mathbf{X} = \mathbf{E}\mathbf{S}\mathbf{F}^\top \tag{A.10}$$

Now we see that the covariance matrix can be written as

$$\mathbf{X}\mathbf{X}^\top = \mathbf{E}\mathbf{S}\mathbf{F}^\top(\mathbf{E}\mathbf{S}\mathbf{F}^\top)^\top = \mathbf{E}\mathbf{S}\mathbf{F}^\top\mathbf{F}\mathbf{S}^\top\mathbf{E}^\top = \mathbf{E}\mathbf{S}^2\mathbf{E}^\top \tag{A.11}$$

and the linear kernel matrix can be written as

$$\mathbf{X}^\top\mathbf{X} = (\mathbf{E}\mathbf{S}\mathbf{F}^\top)^\top\mathbf{E}\mathbf{S}\mathbf{F}^\top = \mathbf{F}\mathbf{S}^\top\mathbf{E}^\top\mathbf{E}\mathbf{S}\mathbf{F}^\top = \mathbf{F}\mathbf{S}^2\mathbf{F}^\top \tag{A.12}$$

So $\mathbf{E}$ are the eigenvectors of $\mathbf{X}\mathbf{X}^\top$, $\mathbf{F}$ are the eigenvectors of $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{S}$ are the (square root of) the eigenvalues of $\mathbf{X}^\top\mathbf{X}\ \mathbf{and}\ \mathbf{X}\mathbf{X}^\top$. More importantly we can obtain the eigenvectors of $\mathbf{X}\mathbf{X}^\top$ from those of $\mathbf{X}^\top\mathbf{X}$ and vice versa. The relationship is

$$\mathbf{E}\mathbf{S} = \mathbf{X}\mathbf{F}^\top \tag{A.13}$$

This means that we can obtain the linear PCA solution (see Equation 7.1) either way, via the eigendecomposition of $\mathbf{X}^\top\mathbf{X}$ or that of $\mathbf{X}\mathbf{X}^\top$. In practice one should go for the solution that is faster to compute:

If there are more dimensions than samples ($N \ll D$)

$\rightarrow$ Compute PCA on linear kernel matrix $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{N\times N}$

If there are more samples than dimensions ($D \ll N$)

$\rightarrow$ Compute PCA on covariance matrix $\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{D\times D}$

B

Gradients

B.1 Derivation Principal Component Analysis

The objective function of PCA is to maximize the variance of the data along each principal component

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmax}}\, \mathbf{w}^\top \mathbf{X}\mathbf{X}^\top \mathbf{w} \tag{B.1}$$

In order to obtain a sensible solution, one needs to constrain **w**

$$\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w} = 1 \tag{B.2}$$

Combining these two yields the Lagrangian

$$\mathcal{L} = \mathbf{w}^\top \mathbf{X}\mathbf{X}^\top \mathbf{w} + \lambda(1 - \mathbf{w}^\top \mathbf{w}) \tag{B.3}$$

Setting the derivative w.r.t. **w** to zero yields

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{w}} &= 2\mathbf{X}\mathbf{X}^\top \mathbf{w} - 2\lambda \mathbf{w} = 0 \\ &\Rightarrow \mathbf{X}\mathbf{X}^\top \mathbf{w} = \lambda \mathbf{w} \end{aligned} \tag{B.4}$$

This is a standard eigenvalue problem. **w** is the eigenvector of $\mathbf{X}\mathbf{X}^\top$ corresponding to the largest eigenvalue

SEE ALSO chapter 7

B.2 Derivation Canonical Correlation Analysis

We here consider only two variables $\mathbf{X}, \mathbf{Y}$ and only the first canonical direction. We assume that the data is centered, i.e. $\sum_{i=1}^{N} \mathbf{X}_i = \mathbf{Y}_i = 0$. The objective function of CCA then is

$$\begin{aligned} \underset{\mathbf{w}_x, \mathbf{w}_Y}{\operatorname{argmax}} \quad & \mathbf{w}_x^\top \mathbf{X}\mathbf{Y}^\top \mathbf{w}_y, \\ \text{s.t. } & \mathbf{w}_x^\top \mathbf{X}\mathbf{X}^\top = \mathbf{w}_x^\top \mathbf{X}\mathbf{X}^\top = 1. \end{aligned} \tag{B.5}$$

Combining the objective with the constraints into one Lagrangian we get

$$\mathcal{L}(\mathbf{w}_x, \mathbf{w}_Y) = \mathbf{w}_x^\top \mathbf{X}\mathbf{Y}^\top \mathbf{w}_y - \frac{1}{2}\alpha(\mathbf{w}_x^\top \mathbf{X}\mathbf{X}^\top \mathbf{w}_x - 1) - \frac{1}{2}\beta(\mathbf{w}_Y^\top \mathbf{Y}\mathbf{Y}^\top \mathbf{w}_Y - 1) \tag{B.6}$$

$$= \mathbf{w}_x^\top \mathbf{C}_{xy}\mathbf{w}_y - \frac{1}{2}\alpha(\mathbf{w}_x^\top \mathbf{C}_x \mathbf{w}_x - 1) - \frac{1}{2}\beta(\mathbf{w}_Y^\top \mathbf{C}_Y \mathbf{w}_Y - 1) \tag{B.7}$$

The partial derivatives of $\mathcal{L}(\mathbf{w}_x, \mathbf{w}_Y)$ w.r.t. $\mathbf{w}_x$, $\mathbf{w}_Y$ are

$$\frac{\partial \mathcal{L}(\mathbf{w}_x, \mathbf{w}_Y)}{\partial \mathbf{w}_x} = \mathbf{C}_{xy}\mathbf{w}_Y - \alpha \mathbf{C}_X \mathbf{w}_x \tag{B.8}$$

$$\frac{\partial \mathcal{L}(\mathbf{w}_x, \mathbf{w}_Y)}{\partial \mathbf{w}_Y} = \mathbf{C}_{xy}^\top \mathbf{w}_x - \beta \mathbf{C}_Y \mathbf{w}_Y \tag{B.9}$$

From the constraints

$$\mathbf{w}_x^\top \mathbf{C}_X \mathbf{w}_x = \mathbf{w}_Y^\top \mathbf{C}_Y \mathbf{w}_Y = 1 \tag{B.10}$$

follows that

$$\alpha = \beta \tag{B.11}$$

Given $\alpha = \beta$ the partial derivatives become 0 at

$$\begin{aligned} \mathbf{C}_{xy}\mathbf{w}_Y &= \alpha \mathbf{C}_X \mathbf{w}_x \\ \mathbf{C}_{xy}^\top \mathbf{w}_x &= \alpha \mathbf{C}_Y \mathbf{w}_Y \end{aligned} \tag{B.12}$$

We just need to arrange these two equations in one single equation

$$\begin{bmatrix} 0 & C_{xy} \\ C_{xy}^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{w}_x \\ \mathbf{w}_Y \end{bmatrix} = \alpha \begin{bmatrix} C_X & 0 \\ 0 & C_Y \end{bmatrix} \begin{bmatrix} \mathbf{w}_x \\ \mathbf{w}_Y \end{bmatrix}. \tag{B.13}$$

This is just a *generalized eigenvalue equation*.

SEE ALSO chapter 10

B.3 Derivation Least Squares Regression

To minimize the least-squares loss function in eq. Equation 17.2

$$\mathcal{E}_{lsq}(w) = (\mathbf{y} - \mathbf{w}^\top \mathbf{X})^2 = \mathbf{y}\mathbf{y}^\top - 2\mathbf{w}^\top \mathbf{X}\mathbf{y}^\top + \mathbf{w}^\top \mathbf{X}\mathbf{X}^\top \mathbf{w}$$

we compute derivative w.r.t. $\mathbf{w}$

$$\frac{\partial \mathcal{E}_{lsq}(\mathbf{w})}{\partial \mathbf{w}} = -2\mathbf{X}\mathbf{y}^\top + 2\mathbf{X}\mathbf{X}^\top \mathbf{w} \tag{B.14}$$

set it to zero and solve for w

$$\begin{aligned} -2\mathbf{X}\mathbf{y}^\top + 2\mathbf{X}\mathbf{X}^\top \mathbf{w} &= 0 \\ \mathbf{X}\mathbf{X}^\top \mathbf{w} &= \mathbf{X}\mathbf{y}^\top \\ \mathbf{w} &= (\underbrace{\mathbf{X}\mathbf{X}^\top}_{\text{Cov. Mat.}})^{-1}\mathbf{X}\mathbf{y}^\top \end{aligned} \tag{B.15}$$

SEE ALSO chapter 17

B.4 Derivation of Ridge Regression

For the euklidian norm $p = 2$ the error function of regularized linear regression is

$$\mathcal{E}_{RR}(\mathbf{w}) = (\mathbf{y} - \mathbf{w}^\top \mathbf{X})^2 + \lambda ||\mathbf{w}||_2^2 \tag{B.16}$$

Computing the derivative w.r.t. w yields

$$\frac{\partial \mathcal{E}_{RR}(\mathbf{w})}{\partial \mathbf{w}} = -2\mathbf{x}\mathbf{y}^\top + 2\mathbf{X}\mathbf{X}^\top \mathbf{w} + \lambda 2\mathbf{w}. \tag{B.17}$$

Setting eq. B.17 to zero and rearranging terms the optimal $\mathbf{w}$ is

$$\begin{aligned} 2\mathbf{X}\mathbf{X}^\top \mathbf{w} + \lambda 2\mathbf{w} =& 2\mathbf{X}\mathbf{y}^\top \\ (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})\mathbf{w} =& \mathbf{X}\mathbf{y}^\top \\ \mathbf{w} =& (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}\mathbf{X}\mathbf{y}^\top \end{aligned} \tag{B.18}$$

SEE ALSO chapter 17

B.5 Gradients for NMF

We can reformulate the objective of NMF in Equation 9.1

$$\begin{aligned} ||\mathbf{X} - \mathbf{W}\mathbf{H}||_{\text{Fro}}^2 =& \operatorname{Tr}\left((\mathbf{X} - \mathbf{W}\mathbf{H})(\mathbf{X} - \mathbf{W}\mathbf{H})^\top\right) \\ =& \operatorname{Tr}\left(\mathbf{X}\mathbf{X}^\top - \mathbf{X}(\mathbf{W}\mathbf{H})^\top - \mathbf{W}\mathbf{H}\mathbf{X}^\top - (\mathbf{W}\mathbf{H})^\top \mathbf{W}\mathbf{H}\right) \\ =& \operatorname{Tr}\left(\mathbf{X}\mathbf{X}^\top\right) - \operatorname{Tr}\left(\mathbf{X}(\mathbf{W}\mathbf{H})^\top\right) \\ & - \operatorname{Tr}\left(\mathbf{W}\mathbf{H}\mathbf{X}^\top\right) - \operatorname{Tr}\left((\mathbf{W}\mathbf{H})^\top \mathbf{W}\mathbf{H}\right) \end{aligned} \tag{B.19}$$

and then take the partial derivatives w.r.t. $\mathbf{H}$, $\mathbf{W}$

$$\begin{aligned} & \frac{\partial}{\partial \mathbf{H}} ||\mathbf{X} - \mathbf{W}\mathbf{H}||_{\text{Fro}}^2 \\ =& \frac{\partial}{\partial \mathbf{H}} \operatorname{Tr}\left(\mathbf{X}\mathbf{X}^\top\right) - \operatorname{Tr}\left(\mathbf{X}(\mathbf{W}\mathbf{H})^\top\right) - \operatorname{Tr}\left(\mathbf{W}\mathbf{H}\mathbf{X}^\top\right) - \operatorname{Tr}\left((\mathbf{W}\mathbf{H})^\top \mathbf{W}\mathbf{H}\right) \\ =& \mathbf{W}^\top \mathbf{W}\mathbf{H} - \mathbf{W}^\top \mathbf{X} \end{aligned} \tag{B.20}$$

The additive update rules for gradient descent on the matrices $\mathbf{H}$ is

$$\mathbf{H} \leftarrow \mathbf{H} - \eta \left(\mathbf{W}^\top \mathbf{W}\mathbf{H} - \mathbf{W}^\top \mathbf{X}\right)$$

Choosing $\eta = \mathbf{H} \oslash (\mathbf{W}^\top \mathbf{W}\mathbf{H})$ will transform the additive update into multiplicative updates

$$\begin{aligned}\mathbf{H} \leftarrow &\mathbf{H} - \mathbf{H} \oslash (\mathbf{W}^\top \mathbf{W}\mathbf{H}) \odot \left(\mathbf{W}^\top \mathbf{W}\mathbf{H} - \mathbf{W}^\top \mathbf{X}\right) \\ \leftarrow &\mathbf{H} - \mathbf{H} \oslash (\mathbf{W}^\top \mathbf{W}\mathbf{H}) \odot \mathbf{W}^\top \mathbf{W}\mathbf{H} + \mathbf{H} \oslash (\mathbf{W}^\top \mathbf{W}\mathbf{H}) \odot \mathbf{W}^\top \mathbf{X} \\ \leftarrow &\mathbf{H} - \mathbf{H} + \mathbf{H} \oslash (\mathbf{W}^\top \mathbf{W}\mathbf{H}) \odot \mathbf{W}^\top \mathbf{X} \\ \leftarrow &\mathbf{H} \odot \mathbf{W}^\top \mathbf{X} \oslash (\mathbf{W}^\top \mathbf{W}\mathbf{H})\end{aligned} \tag{B.21}$$

SEE ALSO chapter 9

B.6 Gradient of Logistic Regression

$$L(z_i) = \frac{e^{z_i}}{\sum_{i=1}^K e^{z_i}} \text{ and } z_i = \mathbf{w}_i^\top \mathbf{x}$$

$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}_k} - \log(p(y|\mathbf{x}, \mathbf{w})) &= \frac{\partial}{\partial \mathbf{w}_k} - \log\left(\frac{e^{z_j}}{\sum_{i=1}^K e^{z_i}}\right) \\ &= \frac{\partial}{\partial \mathbf{w}_k} - \left(\log\left(e^{z_j}\right) - \log\left(\sum_{i=1}^K e^{z_i}\right)\right) \\ &= \frac{\partial}{\partial \mathbf{w}_k} \log\left(\sum_{i=1}^K e^{z_i}\right) - \frac{\partial}{\partial \mathbf{w}_k} z_j \\ &= \frac{\partial}{\partial \mathbf{w}_k} \log\left(\sum_{i=1}^K e^{\mathbf{w}_i^\top \mathbf{x}}\right) \frac{\partial}{\partial \mathbf{w}_k} - \mathbf{w}_j^\top \mathbf{x} \\ &= \frac{1}{\sum_{i=1}^K e^{\mathbf{w}_i^\top \mathbf{x}}} \frac{\partial}{\partial \mathbf{w}_k} e^{\mathbf{w}_k^\top \mathbf{x}} - \mathbf{I}(y)_k \mathbf{x} \\ &= \frac{e^{\mathbf{w}_i^\top \mathbf{x}}}{\sum_{i=1}^K e^{\mathbf{w}_i^\top \mathbf{x}}} \mathbf{x} - \mathbf{I}(y)_k \mathbf{x} \\ &= \frac{e^{\mathbf{w}_k^\top \mathbf{x}}}{\sum_{i=1}^K e^{\mathbf{w}_i^\top \mathbf{x}}} \mathbf{x} - \mathbf{I}(y)_k \mathbf{x} \\ &= \underbrace{\left(L(\mathbf{w}_k^\top \mathbf{x}) - \mathbf{I}(y)_k\right)}_{\text{Error}} \mathbf{x}\end{aligned}$$

SEE ALSO Chapter 19

C

Miscalleneous

C.1 FLDA is Bayes-Optimal

It is easy to show that Fisher's Linear Discriminant Analysis is Baye's-Optimal if the assumptions of the algorithm are met: the data of each class has to be drawn from a Gaussian Distribution with Equal Covariance.

Given binary labels $\{+,-\}$ and data $\mathbf{x} \in \mathbb{R}^D$ the optimal bayesian classifier computes

$$p(+|\mathbf{x}) = \frac{p(\mathbf{x}|+)p(+)}{p(\mathbf{x})} \tag{C.1}$$

Linear Discriminant Analysis (LDA) models $p(\mathbf{x}|+)$, the distribution of the data given a positive label, as gaussian distributions

$$p(\mathbf{x}|+) = \mathcal{N}(\boldsymbol{\mu}_+, \mathbf{S}_+) = \frac{e^{-1/2(\mathbf{x}-\boldsymbol{\mu}_+)^\top \mathbf{S}_+^{-1}(\mathbf{x}-\boldsymbol{\mu}_+)}}{(2\pi)^{(D/2)}\sqrt{|\mathbf{S}_+|}} \tag{C.2}$$

$$\tag{C.3}$$

and the prior class probability

$$p(+) = \frac{N_+}{N} \tag{C.4}$$

where $\mathbf{S}_+$ denotes the covariance matrix of all data from the positive class and $\boldsymbol{\mu}_+$ denotes the mean of all data from the positive class. For negative labels the distributions are defined in complete analogy :

$$p(\mathbf{x}|-) = \mathcal{N}(\boldsymbol{\mu}_-, \mathbf{S}_-) = \frac{e^{-1/2(\mathbf{x}-\boldsymbol{\mu}_-)^\top \mathbf{S}_-^{-1}(\mathbf{x}-\boldsymbol{\mu}_-)}}{(2\pi)^{(D/2)}\sqrt{|\mathbf{S}_-|}} \tag{C.5}$$

$$\tag{C.6}$$

and the prior class probability

$$p(-) = \frac{N_-}{N}. \tag{C.7}$$

The assumptions of LDA are that the data $\mathbf{x}$ of each class comes from a gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_+, \mathbf{S}_+)$ and that the covariance matrices for each class are the same, meaning $\mathbf{S}_+ = \mathbf{S}_- = \mathbf{S}_\pm$). If these assumptions are met LDA is the **bayes-optimal classifier**. Show that the log-ratio of $p(+|\mathbf{x})$ and $p(-|\mathbf{x})$ is equivalent to the LDA decision rule

$$\log\left(\frac{p(+|\mathbf{x})}{p(-|\mathbf{x})}\right) = \log\left(\frac{N_+}{N_-}\right) - \frac{1}{2}(\mu_+ + \mu_-)^\top \mathbf{S}_\pm^{-1}(\mu_+ - \mu_-) + \mathbf{x}^\top \mathbf{S}_\pm^{-1}(\mu_+ - \mu_-) \tag{C.8}$$

Remember that

$$\mathbf{x}^\top \mathbf{A}\mathbf{a} - \mathbf{x}^\top \mathbf{A}\mathbf{b} = \mathbf{x}^\top \mathbf{a}(\mathbf{a} - \mathbf{b}) \tag{C.9}$$

$$\text{and } \mathbf{a}^\top \mathbf{A}\mathbf{a} - \mathbf{b}^\top \mathbf{A}\mathbf{b} = (\mathbf{a} - \mathbf{b})^\top \mathbf{A}(\mathbf{a} + \mathbf{b}) \tag{C.10}$$

$$\begin{aligned}
\mathcal{L}(\mathbf{w}) &= \log\left(\frac{p(+|\mathbf{x})}{p(-|\mathbf{x})}\right) \\
&= \log\left(\frac{p(\mathbf{x}|+)p(+)\ \ p(\mathbf{x})}{p(\mathbf{x}|-)p(-)\ \ p(\mathbf{x})}\right) \\
&= \log\left(\frac{(2\pi)^{D/2}|\mathbf{S}_\pm|^{1/2}e^{-1/2(\mathbf{x}-\mu_+)^\top \mathbf{S}_\pm^{-1}(\mathbf{x}-\mu_+)}\ \ N_+/N}{(2\pi)^{D/2}|\mathbf{S}_\pm|^{1/2}e^{-1/2(\mathbf{x}-\mu_-)^\top \mathbf{S}_\pm^{-1}(\mathbf{x}-\mu_-)}\ \ N_-/N}\right) \\
&= \log\left(\frac{(2\pi)^{D/2}|\mathbf{S}_\pm|^{1/2}}{(2\pi)^{D/2}|\mathbf{S}_\pm|^{1/2}}\right) + \log\left(\frac{e^{-1/2(\mathbf{x}-\mu_+)^\top \mathbf{S}_\pm^{-1}(\mathbf{x}-\mu_+)}}{e^{-1/2(\mathbf{x}-\mu_-)^\top \mathbf{S}_\pm^{-1}(\mathbf{x}-\mu_-)}}\right) \\
&\quad + \log\left(\frac{N_+/N}{N_-/N}\right) \\
&= \log\left(\frac{N_+}{N_-}\right) - \frac{1}{2}(\mathbf{x}-\mu_+)^\top \mathbf{S}_\pm^{-1}(\mathbf{x}-\mu_+) \\
&\quad + \frac{1}{2}(\mathbf{x}-\mu_-)^\top \mathbf{S}_\pm^{-1}(\mathbf{x}-\mu_-) \\
&= \log\left(\frac{N_+}{N_-}\right) - \frac{1}{2}\left(\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mathbf{x} - \mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_+ - \mu_+^\top \mathbf{S}_\pm^{-1}\mathbf{x}^\top + \mu_+^\top \mathbf{S}_\pm^{-1}\mu_+\right) \\
&\quad + \frac{1}{2}\left(\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mathbf{x} - \mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_- - \mu_-^\top \mathbf{S}_\pm^{-1}\mathbf{x}^\top + \mu_-^\top \mathbf{S}_\pm^{-1}\mu_-\right) \\
&= \log\left(\frac{N_+}{N_-}\right) - \frac{1}{2}\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mathbf{x} + \frac{1}{2}\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_+ + \frac{1}{2}\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_+ - \frac{1}{2}\mu_+^\top \mathbf{S}_\pm^{-1}\mu_+ \\
&\quad + \frac{1}{2}\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mathbf{x} - \frac{1}{2}\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_- - \frac{1}{2}\mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_- + \frac{1}{2}\mu_-^\top \mathbf{S}_\pm^{-1}\mu_- \\
&= \log\left(\frac{N_+}{N_-}\right) + \mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_+ - \mathbf{x}^\top \mathbf{S}_\pm^{-1}\mu_- + \frac{1}{2}\mu_-^\top \mathbf{S}_\pm^{-1}\mu_- - \frac{1}{2}\mu_+^\top \mathbf{S}_\pm^{-1}\mu_+ \\
&= \log\left(\frac{N_+}{N_-}\right) - \frac{1}{2}(\mu_+ - \mu_-)^\top \mathbf{S}_\pm^{-1}(\mu_+ - \mu_-) + \mathbf{x}^\top \mathbf{S}_\pm^{-1}(\mu_+ - \mu_-)
\end{aligned}$$

C.2 Nearest Centroids and Linear Classification

Nearest Centroid Classification (NCC, see chapter 15) can be seen as a simple linear classifier. Understanding this relationship can be helpful for understanding linear classification in general and the weaknesses of NCC. Consider data points $\mathbf{x} \in \mathbb{R}^2$ from classes Δ and o with corresponding centroids μ_Δ μ_o. NCC classifies a new data point $\mathbf{x}$ as that class whose centroid is closest in terms of euclidean distance.

$$\begin{aligned} \text{distance}(\mathbf{x}, \mu_\Delta) &> \text{distance}(\mathbf{x}, \mu_o) \\ \|\mathbf{x} - \mu_\Delta\|_2 &> \|\mathbf{x} - \mu_o\|_2 \\ \Leftrightarrow \|\mathbf{x} - \mu_\Delta\|_2^2 &> \|\mathbf{x} - \mu_o\|_2^2 \\ \Leftrightarrow \mathbf{x}^\top \mathbf{x} - 2\mu_\Delta^\top \mathbf{x} + \mu_\Delta^\top \mu_\Delta &> \mathbf{x}^\top \mathbf{x} - 2\mu_o^\top \mathbf{x} + \mu_o^\top \mu_o \\ \Leftrightarrow \mu_\Delta^\top \mathbf{x} - \mu_\Delta^2/2 &< \mu_o^\top \mathbf{x} - \mu_o^2/2 \\ \Leftrightarrow 0 &< \underbrace{(\mu_o - \mu_\Delta)}_{\mathbf{w}}{}^\top \mathbf{x} - 1/2 \underbrace{(\mu_o^\top \mu_o - \mu_\Delta^\top \mu_\Delta)}_{\mathbf{b}} \end{aligned} \tag{C.11}$$

So using the NCC rule to assign data points to the class of the closest centroid is equivalent to projecting the data point on $\mathbf{w}$, the vector connecting the two centroids, and setting a threshold $\mathbf{b}$ in order to decide which centroid is closest:

$$\mathbf{w}^\top \mathbf{x} - \mathbf{b} = \begin{cases} > 0 & \text{if } \mathbf{x} \text{ belongs to class } \circ \\ < 0 & \text{if } \mathbf{x} \text{ belongs to class } \Delta \end{cases} \tag{C.12}$$

This linear classification setting is often visualized as in Figure C.1, the vector $\mathbf{w}$ in combination with the threshold $\mathbf{b}$ defines a decision boundary between the two classes

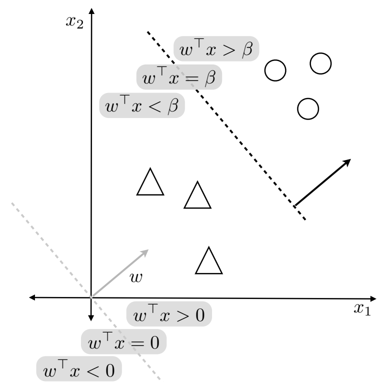


Figure C.1: Hyperplane.

Using a different distance function (such as the Mahalanobis distance [Mahalanobis, 1936]) will different decision boundaries, which can lead to better classification performance in case the covariance of the data features is not isotropic (see chapter 16).

C.3 *Efficient Leave-One-Out Cross-Validation for KRR*

An interesting additional property of kernel ridge regression (chapter 23) is that the automatic selection of the regularization parameter λ (not for other hyperparameters such as the kernel width in a gaussian kernel function) can be performed efficiently (meaning faster than explicitly computing cross-validation).

The leave-one-out cross-validation error can be computed in closed form: Let $\mathbf{S} = \mathbf{K}(\mathbf{K} + \lambda \mathbf{I})^{-1}$. Then,

$$\text{err} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\mathbf{y}_i - [\mathbf{S}\mathbf{y}^\top]_i}{1 - \mathbf{S}_{ii}} \right)^2 .$$

The next insight is that $\mathbf{S}Y$ can be computed without inverting $\mathbf{K}$ each time by computing the eigendecomposition of $\mathbf{K}$ first. Let $\mathbf{K} = \mathbf{U}\mathbf{L}\mathbf{U}^\top$, meaning that $\mathbf{U}$ is an orthogonal matrix ($\mathbf{U}\mathbf{U}^\top = \mathbf{U}^\top\mathbf{U} = \mathbf{I}$), whose columns are the eigenvectors of $\mathbf{K}$, and $\mathbf{L}$ is the diagonal matrix which contains the corresponding eigenvalues on the diagonal. Then,

$$\mathbf{K}(\mathbf{K} + \lambda\mathbf{I})^{-1} = \mathbf{U}\mathbf{L}(\mathbf{L} + \lambda\mathbf{I})^{-1}\mathbf{U}^\top$$

Here, $\mathbf{L} + \lambda\mathbf{I}$ is a diagonal matrix, such that the inverse can be computed just by inverting the diagonal elements. Pre-computing $\mathbf{U}^\top\mathbf{y}^\top$ leads to further speed-up.

The choice of kernel parameters (like the widths for rbf-kernels) must, however, be performed by explicit cross-validation.

C.4 Distance Functions

Real-Valued Data

Many algorithms (such as clustering algorithms, see K-Means Clustering chapter 12 or k-NN Classification chapter 22) need a **distance function** $d(\mathbf{x}_i, \mathbf{x}_j)$ that measures the distance between two data points. For different types of data

- For real valued data $\mathbf{x} \in \mathbb{R}^D$ we can use the Euclidean distance

$$d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2^2 = \sqrt{\sum_{d=1}^{D} (\mathbf{x}_{id} - \mathbf{x}_{jd})^2} \tag{C.13}$$

- Less sensitive to outliers is the **city block distance** or $\mathcal{L}_1$ norm

$$d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_1 \tag{C.14}$$

Non-numerical Data

Often data sets contain attributes that are not numerical. For *ordinal data* that has some order or ranking one can vectorize the data as explained in **??** and Equation C.15 and then compute a distance function for real-valued data (**??**). Instead of vectorizing these attributes (as explained in **??**) one can compute the relevant information for an algorithm directly on the raw data.

- For ordinal variables such as $\mathbf{x}_i \in \{\text{low}, \text{medium}, \text{high}\}^D$ we can transform the values into real-valued numbers (for three possible values e.g. $1/3, 2/3, 3/3$) and then apply distance functions for real-valued data

$$\mathbf{x}_{id} = \begin{cases} 1/3 & \text{if } \mathbf{x}_{id} = \text{low} \\ 2/3 & \text{if } \mathbf{x}_{id} = \text{medium} \\ 3/3 & \text{if } \mathbf{x}_{id} = \text{high} \end{cases} \tag{C.15}$$

- For categorial variables $\mathbf{x} \in \{\text{red}, \text{green}, \text{blue}\}^D$ we can use a binary coding for the differences

  $$d(\mathbf{x}_i, \mathbf{x}_j) = \sum_d^D \mathbf{x}_{id} \neq \mathbf{x}_{jd} \tag{C.16}$$

  This metric is called **Hamming Distance**

Bibliography

Dimitris Achlioptas, Frank McSherry, and Bernhard Schölkopf. Sampling techniques for kernel methods. In *NIPS*, pages 335–342, 2001.

Richard Ernest Bellman. *Dynamic programming*. Princeton University Press, 1957.

James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13: 281–305, 2012. URL http://dl.acm.org/citation.cfm?id=2188395.

Christopher M Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2007.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187. Springer, 2010.

François Chollet. *Deep Learning with Python*. Manning, November 2017. ISBN 9781617294433.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

N. M. Dempster, A.P. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society B*, 39:185–197, 1977.

C.H.Q. Ding, T. Li, and Wei Peng. On the equivalence between non-negative matrix factorization and probabilistic latent semantic indexing. *Computational Statistics & Data Analysis*, 52(8):3913–3927, 2008.

RO Duda, PE Hart, and DG Stork. Pattern classification. *Wiley*, 2001.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *NeurIPS*, pages 2962–2970, 2015.

R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

Carl Friedrich Gauß. Theoria motus corporum coelestium in sectionibus conicis solem ambientium. Technical report, University of Göttingen, 1809.

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

T Hastie, R Tibshirani, and J H Friedman. *The Elements of Statistical Learning*. Springer New York, 2003.

Thomas Hofmann, Bernhard Schölkopf, and Alex J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 36(3):1171–1220, 2008.

Harold Hotelling. Relations between two sets of variates. *Biometrika*, 28(3):321–377, 1936.

C. Jordan. Essai sur la Géometrie à *n* dimensions. *Bull. Soc. Math. France*, 3:103–174, 1875. Tome III, Gauthiers-Villars, Paris, 1962, 79-149.

Jon R Kettenring. Canonical analysis of several sets of variables. *Biometrika*, 58(3):433–451, 1971.

Daniel G. Krige. A statistical approach to some basic mine valuation problems on the witwatersrand. *Journal of the Chemical, Metallurgical and Mining Society of South Africa*, 52(6):119–139, 1951.

Sebastian Lapuschkin, Stephan Wäldchen, Alexander Binder, Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Unmasking clever hans predictors and assessing what machines really learn. *Nature Communications*, 10(1):1096, 2019. DOI: 10.1038/s41467-019-08987-4. URL https://doi.org/10.1038/s41467-019-08987-4.

D.D. Lee and H.S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–91, 1999.

D.D. Lee and H.S. Seung. Algorithms for non-negative matrix factorization. In *In NIPS*, pages 556–562, 2000.

Adrien-Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes*, chapter Sur la methode des moindres quarres. Firmin Didot, 1805.

James Lighthill. Artificial intelligence: A general survey. In *Artificial Intelligence: a paper symposium, Science Research Council*, 1973.

S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

P Mahalanobis. On the generalized distance in statistics. *Proc. Nat. Inst. Sci. India*, 2:49–55, 1936.

J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research*, 11:19–60, 2010.

S. Mika, B. Schölkopf, A.J. Smola, K.-R. Müller, M. Scholz, and G. Rätsch. Kernel PCA and de–noising in feature spaces. In *NIPS*, pages 536–542, 1999.

Klaus-Robert Müller, Sebastian Mika, G Ratsch, K Tsuda, and B Bernhard Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201, Jan 2001. DOI: 10.1109/72.914517.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. The MIT Press, 1 edition, 2012. ISBN 0262018020,9780262018029. URL http://gen.lib.rus.ec/book/index.php?md5=8ecfeeb2e1f9a19c770fba1ff85fa566.

K Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2:559–572, 1901.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NIPS*, 2007.

Ali Rahimi and Benjamin Recht. Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning. In *NIPS*, 2008.

Carl Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2005.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400—407, 1951.

F Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

S T Roweis and L K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–6, 2000.

Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller, editors. *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, volume 11700 of *Lecture Notes in Computer Science*. Springer, 2019. ISBN 978-3-030-28953-9. DOI: 10.1007/978-3-030-28954-6. URL https://doi.org/10.1007/978-3-030-28954-6.

Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On challenges in machine learning model management. volume 41 of *5*, 12 2018a.

Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *PVLDB*, 11(12):1781–1794, 2018b.

B Schölkopf, A J Smola, and KR Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(6):1299–1319, 1998.

Bernhard Schölkopf and Alex J Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2002. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8684.

Steven L. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010. DOI: 10.1002/asmb.874. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/asmb.874.

D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. pages 2503–2511, 2015.

John Searle. Minds, brains and programs. *Behavioral and Brain Sciences*, 3(3):417–457, 1980.

Burr Settles. Active learning literature survey. Technical Report 1648, University ofWisconsin–Madison, 2010.

J Shawe-Taylor and N Cristianini. *Kernel methods for pattern analysis*. Cambridge University Press, 2004.

Jasper Snoek, Hugo Larochelle, and Rp Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *Nips*, pages 1–9, 2012. ISSN 10495258. DOI: 2012arXiv1206.2944S. URL http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdfhttps://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Trans. Neural Networks*, 9(5):1054–1054, 1998. DOI: 10.1109/TNN.1998.712192. URL https://doi.org/10.1109/TNN.1998.712192.

J B Tenenbaum, V de Silva, and J C Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500): 2319–23, 2000.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

Michael Tomasello. *The cultural origins of human cognition*. Harvard University Press, 2003.

A Turing. The imitation game. *MIND*, pages 1–28, 1950.

Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

Andrey Nikolayevich Tychonoff. On the stability of inverse problems. *Doklady Akademii Nauk SSSR*, 39(5):195–198, 1943.

Vladimir Vapnik. *Estimation of dependencies based on empirical data*. Springer Series in Statistics. Springer-Verlag, New York, 1982.

U von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

Joseph Weizenbaum. Eliza - a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.