Handbook

Lab Course
Machine Learning and Data Analysis

Disclaimer: This is work in progress!

Mikio L. Braun Paul Bünau Daniel Bartz Jacob Kauffmann

started April 20, 2007,
last change March 31, 2023

Contents

Chapter 1

Preface

I always thought that the fastest way to learn something is to "hands-on", directly interacting with the matter at hand. The alternative, approaching the subject more from an abstract-theoretical point of view, is equally adequate only as long as you already know the underlying concepts. Otherwise, just trying things out seems to be the best way to acquire the necessary concepts as fast as possible.

This observation is maybe most true for a young field like machine learning which lacks a thorough theoritization. Methods are motivated using many different approaches, and there does not exist a single theory which is able to derive everything conclusively. Furthermore, methods often rely on heuristics, putting us in the somewhat curious situation that there exist many methods where you think you know why they work, without being able to prove exactly why.

So my advice to all those who want to get to know machine learning quickly is to expose themselves to concrete algorithms: Take a handful of methods, implement them, and try to make them run. Tweak the parameters, get your own idea why something works, and then go back to the theory to see whether other people have arrived at the same conclusions as you did.

Then, once you have first-hand experience and acquired an idea about how things work, you can go back to find out how people have analyzed the methods theoretically, and which general principles underlie the different methods.

This document tries to be a guide to exactly this approach. A number of methods from all areas of machine learning will be discussed. The emphasis is put on being able to implement the methods quickly, and on passing on a general intuition about the workings of the algorithms. Each method is listed with its name, field of application, the main idea, the implementation, and cross-references to other algorithms. Furthermore, some general tools like cross-validation will be discussed as well.

After all, I hope that I can get the important fact across that machine learning can also be fun.

Mikio L. Braun,
April 20, 2007 (with minor additions on December 18, 2007)

Chapter 2

Introduction

2.1 What is Machine Learning?

Actually, it is somewhat unclear where the exact boundaries of "machine learning" lie. What is the difference between machine learning and computer science? After all, computer science aims to automate cognitive processes, like computation. The next best candidate would be artificial intelligence. Is machine learning the same as artificial intelligence (AI)? Finally, when you actually look at a number of machine learning papers, it seems to be all about statistics and probability theory. What is the difference between ML and statistics?

2.1.1 An attempt at defining Machine Learning

In one sentence, machine learning studies the question how one can learn some task solely from examples of the desired behavior. Such a task might be for example, to classify objects into different categories. To be even more specific, consider the task of recognizing handwritten characters. Each character is scanned and represented by an image, and the task consists to correctly predict which character is shown in the image.

This approach has two characteristics which set it aside from both artificial intelligence (the example with the characters might be simple. Consider the task to understand spoken commands: the mapping to be learned is from the audio recording to possible meanings), and also from the way in which computer science usually addresses problems:

1. lack of explicit problem formalization
2. lack of explicit problem solving

Let us discuss both points in more detail.

2.1.2 Lack of Explicit Problem Formalization

Usually in computer science there is first an explicit, formalized description of the problem you are trying to solve. For example, the problem of sorting can be formalized as mapping

a set of objects on which an order is defined to a sequence such that the objects are in increasing order. Or, given a weighted graph, compute the shortest path between any given two nodes in the graph. These are problem definitions which are formally exact (well, maybe not in the informal tone stated above) in the sense that given an algorithm, I can *prove* that the algorithm solves the given problem, that it is in fact *correct*.

In principle, the problems one considers in machine learning are also formally defined. For example, the categorization task mentioned above has a clear formal description: From a finite set of examples, you should derive a function such that the *expected error* is minimal, that is, if I choose a point and the desired outcome at random, measure that error, and average over all such choices, the error should be minimal. This is in fact a formal definition, and you can even prove that it is correct.

However, note that at the same time, the problem definition is also extremely abstract. It does not even mention handwritten characters, or images. In fact, algorithms which solve some instantiation of this categorization problem may not work at all on other problems. So the bottom line is that there exists an abstract definition, but it does not capture the real nature of the problem.

So instead of a formal problem definition, we have a (possibly large) data set which *is the definition of the problem*. This approach has an important advantage: we do not need to have a full formal understanding of the problem in order to study it. Consider the problem of natural language processing. If we could actually formally define what the problem is, we will have solved a huge part of the problem, because we will have managed to define formally what meaning is, how to handle real world grammars, etc.

If you think about it, you see that the formal approach taken in theoretical computer science is really already part of the solution because the main remaining problem is in most cases to solve the problem *efficiently*. In contrast, in machine learning we're already happy if we can already solve the problem well. And by well we mean that we perform well on the actual data set defining the problem.

2.1.3 Lack of Explicit Problem Solving

Now the other characteristic is that we *do not try to solve the problem explicitly*. By this I mean that one does not try to fully design an algorithm by hand which solves the problem well. Instead, one usually designs a more general algorithm which is then tuned to the actual problem during a *training or learning step*.

Note that these two steps are actually independent. Take for example translation of natural languages. Instead of defining the problem formally, you collect a huge amount of examples to define the problem. But then, you could try to directly solve the problem, for example by designing grammars and parsers for both languages, developing a meta-language to express the concepts behind the languages and procedures for transforming between the natural languages and the meta-language. Such an approach has actually been studied.

One practical difference between AI and ML is that AI often tries do directly model cognitive processes whereas in ML we try to solve problems supposedly requiring intelligence

by learning from examples only.

While at first it seems counter-intuitive that you can solve problems easier by first solving a more general problem, it actually turns out that you can go a long way with simple models and raw computing power. When you as a human try to design a solution to a problem (think of the character recognition task), it will tend to end up with a solution of a few fairly powerful components, for example a procedure for extracting lines and interest-points like crossings, and rapid changes of directions. But there is a certain limit to the number of such components you can deal with mentally. On the other hand, a computer can combine thousands or hundred of thousands simple models (like small patches which it takes as templates) to build something more accurate and robust than what you could do by hand.

This is maybe also the downside of the current state-of-the-art of machine learning: Most algorithms are quite dumb when you look at how they work, and there is not really a lot of "intelligence" hidden in there, just sheer computing power.

If we said above that one does not try to explicitly solve the problem, we didn't mean that one does not invest any hand-tuning at all. In fact, for almost all real-world problems we actually have to put a lot of work in there to find the right kind of feature extractor and preprocessing to make an algorithm like the support vector machine perform well.

Machine learning is actually a hybrid approach: We try to solve the problem partially by transforming the raw data to make the interesting information pop up more prominently, but building on these handwritten partial solutions, you put an algorithm which is mostly agnostic to the actual application domain it is dealing with. This is also the reason why one can still learn a lot from the non-ML approaches which try to solve the problem explicitly. Their experience should not be ignored.

2.1.4 Proper Validation of Algorithms in Machine Learning

How machine learning algorithms work is the topic of this little guide. Before we start on that, it is important to briefly discuss how to evaluate such algorithms. We focus on the supervised learning task (as in the categorization problem mentioned above), but the same principles hold for other problem areas as well.

Recall that in theoretical computer science, the problem is often stated formally, and it is often quite easy to find an algorithm which is correct (for example, by explicitly enumerating the whole solution set). Therefore, one has to resort to other characteristics to measure the quality of an algorithm. Asymptotic run-time complexity is one example. You basically say that your algorithm is better than all these other algorithms which solve the same problem because it is faster. Memory requirements would be another choice.

Now, in machine learning, we have to take a step back. Actually solving the problem at all is important, and run-time is secondary. Only when it is accepted that the problem is in principle solved well, it becomes important how long it takes.

In order to measure the performance of the algorithm you have to test the algorithm against its definition, in this case a concrete data set. Now, you have to take a few precautions to get meaningful results.

First of all, you cannot test your algorithm on the same data set (or part of the data set) you have used to train you algorithm. The reason is that it is too easy to perform well: Simply memorize the whole data set and just recall everything.

The goal is to perform well on *unseen data*. Therefore, the first important rule is *that we have to evaluate your algorithm on data we haven't seen in the training step.* Practically, this means that we split the data set into a training and test set with no overlap. Typically, you iterate this procedure several times to also measure how stable your algorithm is.

Using this procedure you can now compare your algorithm against another algorithm: you iterate over random splits of your data set into training and test data sets and collect the resulting error rates. To properly compare these two numbers, you *should use some statistical test to see if the difference is really significant.* This is the second rule

Most algorithms also have some free parameters which control, for example, the complexity of the learned function. One commonly taken approach to adjusting these numbers is to evaluate different choices of parameters as described above and then taking the one which works best. If you do this, you *actually have to re-split the training set again.* This is the important third rule *the test set can really only be used for the final evaluation of your data set.* Otherwise, the reported results will be too good. You have chosen your parameters to be optimal on the test set, but how does this choice perform on further unseen data?

So to summarize: (1) Evaluate your algorithm on data you haven't used in training. (2) Compare the resulting performance measures using sound statistical tests. (3) Parameter tuning is part of the training, so you must not use the test data there either.

If you follow these three steps, you're already a large step towards properly validating algorithms. And being able to do this is important, irrespective whether you are "just" a practitioner (because you would want to use which algorithm works best, not just which looks most fancy), or a researcher (because ultimately, you'll have to defend your new method by comparing it against the state-of-the-art).

2.2 Data Types

Before we'll actually play around with some algorithms, let us talk about the data we will deal with. For some reason, machine learning is at the one time rich in data types (for example, we have vectors, strings, graphs, images, time series data et cetera), while at the same time, when you actually look at a machine learning paper, people will only talk about Xs and Ys.

2.2.1 Xs and Ys

The most basic machine learning task is that of prediction. For example, you want to learn how to predict whether an email contains spam or not. In machine learning, it is customary to denote with Xs the input objects which you have given, and with Ys the output you want to generate. In our case, X would denote the text of an email (or some

abstracted version of it), while Y denotes the binary piece of information "spam" or "no spam."

A core ingredient of the machine learning approach is that we don't want to design the algorithms which perform this prediction task by hand, for example by cleverly designing a number of stop-words and signatures identifying spam, but we directly tackle the more general problem of designing an algorithm which can learn this task from a set of examples. Assuming that we have collected 1000 spam emails and 1000 normal emails, the algorithm should automatically learn how to separate spam from "ham."

In machine learning we will therefore often deal with data sets which consist of examples of the mapping we want to learn. And since it is usually assumed that the examples are independent, they are just considered as an enumerated collection of $\mathbf{X}$s and Ys, denoted as pairs $(\boldsymbol{x}_i, y_i),\ i = 1, \ldots, n$ where $\boldsymbol{x}_i$ is typically a vectorial input and y_i a scalar target.

In this book, n will always denote the total number of examples, such that the example inputs are given by the sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, and the example targets by $y_1, \ldots, y_n$.

2.2.2 Vectorial Data

For several reasons, vectorial data (that is, each $\boldsymbol{x}_i$ is an d-dimensional vector) is the most basic data type occurring in machine learning.

Part of the reason is that vector spaces lend themselves to nice geometric intuition which can help in the design of new algorithms. And, of course, vector spaces are very useful. You can add vectors, compute a mean vector for a number of vectors, and compare two vectors if they are anywhere close to each other.

With vectorial data, a data set $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ can nicely be summarized in a $n \times d$-matrix. Now, there is some ambiguitiy here, as we could also encode the data set as a $d \times n$-matrix (such that the data points become *columns* of this matrix). However, we will stick with the first convention. To remember it, picture vectors as row vectors or lists. If we have a sequence of those vectors and stack them on top of each other, we get the matrix

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & \cdots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \cdots & x_d^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times d}$$

where $x_i^{(j)}$ denotes the ith dimension of vector $\boldsymbol{x}_j$. A nice feature is that this allows to naturally "matricize" vector-operations notationally. If $\mathbf{X}$ is the data set matrix to $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, and $\boldsymbol{v} \in \mathbb{R}^d$ is another vector, then we write

$$\mathbf{X}\boldsymbol{v} = \begin{bmatrix} \boldsymbol{x}_1^\top \boldsymbol{v} \\ \vdots \\ \boldsymbol{x}_n^\top \boldsymbol{v} \end{bmatrix}$$

which is a n-dimensional vector of dot products between each $\boldsymbol{x}_i$ and $\boldsymbol{v}$. Dropping the index, changing the typeface to uppercase we get the operation on all vectors. The logical next

step is to define a notation for matrix-matrix multiplication. Let $\mathbf{X} \in \mathbb{R}^{n \times d}, \mathbf{V} \in \mathbb{R}^{m \times d}$, then

$$\mathbf{X}\mathbf{V}^\top = \left(\mathbf{V}\mathbf{X}^\top\right)^\top = \begin{bmatrix} \boldsymbol{x}_1^\top \boldsymbol{v}_1 & \cdots & \boldsymbol{x}_1^\top \boldsymbol{v}_m \\ \vdots & \ddots & \vdots \\ \boldsymbol{x}_n^\top \boldsymbol{v}_1 & \cdots & \boldsymbol{x}_n^\top \boldsymbol{v}_m \end{bmatrix}$$

is the $n \times m$ matrix of pairwise dot products.

2.2.3 The Labels

One generally considers three distinct types of outputs:

Classification: The output space is discrete, as one wants to predict membership to a number of (finite) $k \in \mathbb{N}$ classes, i.e.

$$y_i \in \{c_1, \ldots, c_k\}.$$

Regression: The prediction is a real value,

$$y_i \in \mathbb{R}.$$

Multivariate regression: One wants to predict a d'-dimensional vector,

$$\boldsymbol{y}_i \in \mathbb{R}^{d'}.$$

Distinguishing between only two classes $\{c_1, c_2\}$ is the simplest form of classification. Incidentally, it is also the variant which is used most. The reason is that it is hard to take care of more than two classes directly, and that there exist effective schemes of combining two-class-classifiers into multi-class-classifiers (More on that in Chapter 4).

Multivariate regression can be similarly reduced to a number of (normal) regression problems, by considering the output dimensions independently.

For both cases, there exist algorithm which directly take care of the multi-class or multivariate case, and which claim to have advantages, although they are often more complex than the simpler cases. In practice, one has to see which works better.

2.3 Notational Issues

Writing about machine learning brings certain notational problems, mainly because machine learning strongly depends on two independent branches of mathematics: probability theory and linear algebra.

Now in probability theory, random variables are usually denoted by upper case roman letters, while in linear algebra, upper case bold letters are usually reserved for matrices whose entries are then denoted by the lower case letters: $\mathbf{A} = (a_{ij})_{ij}$. Typically, matrices consist of real valued scalars, i.e. $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m, n \in \mathbb{N}$. We might write $\mathbf{A}_i$ for row i

of matrix $\mathbf{A}$ and $\mathbf{A}_{ij}$ for the scalar value in row i, column j. Vectors are denoted by lower case bold letters, $\boldsymbol{v} \in \mathbb{R}^d$ with $d \in \mathbb{N}$. The entries of a vector, however, are denoted by lower roman letters, v_i for $i \in \{1, \ldots, d\}$.

We will therefore use the following convention: lower case indicates fixed non-matrix object, upper case indicates random object, upper case bold indicates matrix. Furthermore, if a is some quantity, putting a hat over it $\widehat{a}$ means that $\widehat{a}$ is an estimate of a. From time to time, we will make use of correspondences between lower and upper case letters or the roman and the greek alphabet. For example, denoting objects $m \in M$, or having an index i run from 1 to n. Furthermore, we might denote an finite sample size object by a roman letter, and its asymptotic limit by the greek letter, for example, $l \to \lambda$, $a \to \alpha$.

However, invariably, the number of data points is n, and the examples are indexed by i.

These conventions are summarized in this table:

This...	can be...
$a, x_i, \alpha, \ldots$	scalars
$\boldsymbol{x}, \boldsymbol{v}$	vectors
$X, Y, \ldots$	random variables
$\mathbf{X}$	matrices
$\widehat{a}, \widehat{\boldsymbol{x}}, \ldots$	estimates of $a, \boldsymbol{x}, \ldots$
$a \in A$	trying to make use of corrspondence between cases.
$l \to \lambda, s \to \sigma$	estimated and asymptotic quantities.

Chapter 3

Unsupervised Learning

3.1 Dimensionality Reduction

Most of the interesting data sets are quite high-dimensional. Images is maybe the most extreme example (every pixel is one input dimension), but there exist other examples, for example in bioinformatics. Microarrays are able to measure concentration of a couple thousand different certain aminoacids in a cell at the same time.

Besides computational issues, high dimensional data bring their own problems for data analysis. Usually, the underlying problem is not really that high-dimensional such that there exist many correlation between individual dimensions, and ample opportunity to see structure which is only noise.

To continue the bioinformatics example, each dimension encodes the concentration of a certain gene in a cell (in a massively simplified view). One interesting question is which of the genes show interesting behavior. Now, with several thousand dimensions and only a few data points, it is inevitable that there will be many interesting dimensions – by pure chance.

In this first section we will discuss methods which try to reduce the dimension of the data while retaining the information contained in the data. Depending on what you define the terms "retain" and "information", you will end up with different ideas and algorithms. We will discuss a few classic choices.

3.1.1 Principal Component Analysis

▷ **Name** Principal Component Analysis (PCA)

▷ **Applications** dimension reduction, denoising, visualization

▷ **Method** The PCA method actually consists of three disjoint steps: Computing the principal components, projecting the data to obtain a low-dimensional representation, and reconstructing the original data set.

Algorithm 1 Compute Principal Components

Input: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$
Output: principal values $\lambda_1, \ldots, \lambda_d \in \mathbb{R}$,
principal directions $\mathbf{u}_1, \ldots, \mathbf{u}_d \in \mathbb{R}^d$

1: Compute mean $\boldsymbol{\mu} \leftarrow \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i$.
2: Compute covariance matrix

$$\mathbf{C} \leftarrow \frac{1}{n-1} \sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\top.$$

3: Compute eigenvalues λ_i and eigenvectors $\mathbf{u}_i$ of $\mathbf{C}$.
4: **return** $(\lambda_1, \ldots, \lambda_d)^\top$ and $(\mathbf{u}_1, \ldots, \mathbf{u}_d)$

Algorithm 1 shows how to compute the principal components. The principal directions $\mathbf{u}_i$ (sorted together with λ_i such that $\lambda_1 \geq \ldots \geq \lambda_n$) show the directions where the data has maximal variance. The directions are orthogonal, that is, $\mathbf{u}_i^\top \mathbf{u}_j = 0$ if $i \neq j$, which will come in handy for a number of computations.

Algorithm 2 Projecting to the low-dimensional sub-space

Input: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n'} \in \mathbb{R}^d$,
principal directions $\mathbf{u}_1, \ldots, \mathbf{u}_m \in \mathbb{R}^d$,
mean of training data $\boldsymbol{\mu} \in \mathbb{R}^d$
Output: transformed data points $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_{n'} \in \mathbb{R}^m$.

1: **for** $i \leftarrow 1$ **to** n' **do**
2: $\quad \boldsymbol{z}_i \leftarrow \big(\mathbf{u}_1^\top (\boldsymbol{x}_i - \boldsymbol{\mu}), \ldots, \mathbf{u}_m^\top (\boldsymbol{x}_i - \boldsymbol{\mu})\big)^\top$
3: **end for**
4: **return** $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_{n'}$

Using the leading m principal directions $\mathbf{u}_1, \ldots, \mathbf{u}_m$ (with $m \leq d$), we can compute the low-dimensional representations of the data points $\boldsymbol{x}_i, i = 1, \ldots, n$ as shown in Algorithm 2.

Finally, one may want to reconstruct the de-noised data points in the original space (for example with images). This is accomplished by Algorithm 3.

▷ **Discussion** The PCA algorithm is a old and trusted standard method from statistics. It is based on the idea of finding an m-dimensional subspace such that the reconstructed data points $\widehat{\boldsymbol{x}}_i$ (Algorithm 3) have minimal distortion to the original points $\boldsymbol{x}_i$. This leads to an eigenvalue problem which can be solved on $\mathcal{O}(d^3)$.

For general m-dimensional subspaces, this is not trivial to see. However, for the first principal component, it is rather easy: For simplicity, we'll assume that the $\boldsymbol{x}_i$ are centered. Getting the projected coefficient for a one-dimensional direction can be computed by taking the scalar product with a vector $\mathbf{u}$ of lenght 1 which spans this 1D subspace: $\boldsymbol{x}_i^\top \mathbf{u}$. We

Algorithm 3 Reconstructing projected data points in the original space

Input: transformed data points $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_{n'} \in \mathbb{R}^m$,
principal directions $\mathbf{u}_1, \ldots, \mathbf{u}_m \in \mathbb{R}^d$,
mean of training data $\boldsymbol{\mu} \in \mathbb{R}^d$

Output: reconstructed data points $\widehat{\boldsymbol{x}}_1, \ldots, \widehat{\boldsymbol{x}}_{n'} \in \mathbb{R}^d$.

1: **for** $i \leftarrow 1$ **to** n' **do**
2: $\quad \widehat{\boldsymbol{x}}_i \leftarrow \boldsymbol{\mu} + \sum_{j=1}^{m} z_j^{(i)} \mathbf{u}_j$
3: **end for**
4: **return** $\widehat{\boldsymbol{x}}_1, \ldots \widehat{\boldsymbol{x}}_{n'}$

wish to minimize the reconstruction error, or alternatively, to capture as much variance along the first direction. Thus, we wish to maximize

$$\sum_{i=1}^{n} (\boldsymbol{x}_i^\top \mathbf{u})^2 = \|\mathbf{X}\mathbf{u}\|^2 = \mathbf{u}^\top \mathbf{X}^\top \mathbf{X}\mathbf{u} = \mathbf{u}^\top \mathbf{S}\mathbf{u}$$

subject to $\|\mathbf{u}\| = 1$. However, this optimization problem is well known to be solved by the eigenvector corresponding to the largest eigenvalue λ, with $\mathbf{u}^\top \mathbf{S}\mathbf{u}$ being the eigenvalue, since

$$\mathbf{u}^\top \mathbf{S}\mathbf{u} = \mathbf{u}^\top (\lambda \mathbf{u}) = \lambda \|\mathbf{u}\|^2 = \lambda,$$

since $\mathbf{u}$ has length 1.

The number of principal components m used in the denoising has to be supplied by the user. Finding a good value for m is not trivial. Irrespective of the number of dimensions, PCA will find the optimal subspace with respect to the Euclidean norm. Thus, fewer dimensions implies larger error, and more dimensions means smaller error.

By the way, the reconstruction error can be quickly read of as the sum of the remaining principal values $\lambda_{m+1}, \ldots, \lambda_d$. If the data is particularily noise-free, and the data lives on a linear affine subspace, then it might be possible to set a threshold of distortion one is willing to tolerate, and then choose m such that $\lambda_{m+1} + \ldots + \lambda_d$ is smaller than the threshold.

However, in the presence of noise (and as a rule of thumb, all data is noisy), this will not be possible. Independent noise on each coordinate of the data leads to a uniform increase of the principal values along each direction, such that the distortion will never be negligible.

In such cases, one typically sees a "knee" in the sequence of principal values which indicates the transition from "signal" to "noise". This dimension is then often used as the "right" cut-off dimension.

Applying PCA for dimension reduction and denoising is pretty straightforward. For visualization, one will typically choose $m = 2$ or $m = 3$ and plot the resulting pictures. However, for most cases, other methods are better suited, in particular multi-dimensional scaling, LLE, and ISOMAP.

▷ **See also** Kernel PCA (a non-linear extension of PCA), Locally Linear Embedding, Isomap

3.1.2 Isomap

▷ **Name** Isomap

▷ **Applications** dimension reduction, visualization

▷ **Method** Isomap is a simple extension to Classical Scaling where the distances between data points are measured as shortest paths on a graph which is assumed to follow the high dimensional manifold of the data. From these distances, Classical Scaling reconstructs lower dimensional coordinate vectors by applying an eigendecomposition to the inner-product matrix that is reconstructed from the matrix of shortest paths.

Algorithm 4 Isomap

Input: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$, number of dimensions $d' \leq d$ for the embedding, parameter k or ϵ for graph construction

Output: embedded data $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n \in \mathbb{R}^{d'}$

1: Compute matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$ of Euclidean distances between the data points,

$$\mathbf{D}_{ij} \leftarrow \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_{\ell_2}.$$

2: Compute adjaceny matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ between data points using k-nearest-neighbour or ϵ-ball rule.

3: Compute matrix $\mathbf{D}_g \in \mathbb{R}^{n \times n}$ of pairwise distances as shortest paths on the graph defined by $\mathbf{A}$ using Floyd-Warshall-Algorithm (or modified Dijkstra-Algorithm).

4: Compute squared distances along the manifold: $(\mathbf{D}_g)_{ij} \leftarrow (\mathbf{D}_g)_{ij}^2$ for all $1 \leq i, j \leq n$.

5: $\mathbf{A} \leftarrow -\frac{1}{2}\mathbf{D}_g$

6: Compute centering matrix $\mathbf{H} \leftarrow \mathbf{I}_n - \frac{1}{n}\mathbf{1}_{n \times n}$

7: Compute matrix $\mathbf{V} \in \mathbb{R}^{n \times p}$ of eigenvectors (as columns) and associated non-zero eigenvalues $\lambda_1, \ldots, \lambda_p$ of $\mathbf{HAH}$.

8: Sort eigenvalues and eigenvectors $\mathbf{V}$ such that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p > 0$.

9: **for** $i \leftarrow 1$ to n **do**

10: $\quad \boldsymbol{z}_i \leftarrow (\sqrt{\lambda_1}\mathbf{V}_{1i}, \ldots, \sqrt{\lambda_{d'}}\mathbf{V}_{di})^\top$

11: **end for**

12: **return** $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$

▷ **Discussion** Basically, the only thing that Isomap adds to Classical Scaling is a method for obtaining geodesic distances along the data manifold without making any further assumptions. However, if the matrix of shortest paths does not reflect the true geodesic

distances, Isomap will fail. To this end, it is important that the graph covers the data manifold as densely as possible (to avoid unnecessary detours in the shortest paths) without introducing shortcuts (which lead to incorrect geodesic distances). In particular, shortcuts can induce a distance matrix which is not embeddable in a Euclidean space. In that case, the inner-product matrix will have negative eigenvalues. Therefore, one has to choose the right k or ϵ for constructing the graph or use a different method altogether. This can be difficult (if not impossible) in the presence of outliers and noises or when some regions of the data are more densely sampled than others.

▷ **See also** Locally Linear Embedding, Principal Component Analysis, Kernel PCA

3.1.3 Locally Linear Embedding

▷ **Name** Locally-Linear-Embedding (LLE)

▷ **Applications** dimension reduction, visualization

▷ **Method** The LLE algorithms aims at finding a lower dimensional embedding of the data which preserves properties of the local neighbourhood of each datapoint. The solution of the resulting optimization problem over the embedding coordinates is found by solving an eigenvalue problem.

▷ **Discussion** In contrast to Isomap, which aims at recovering all pair-wise distances (globally), LLE merely tries to reproduce local characteristics in the embedded data. However, the choice of k or ϵ for defining the neighbourhood of each data point leads to a similar dilemma: small neighbourhoods can result in large reconstruction errors and weights that are not representative for the local geometry while large neighbourhoods may introduce shortcuts and thereby spoil the whole solution.

▷ **See also** Isomap, Principal Component Analysis, Kernel PCA

Algorithm 5 LLE

Input: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$,
number of dimensions $d' \leq d$ for the embedding,
parameter k or ϵ for constructing the neighborhoods,
regularization parameter for the local covariance matrices ν
Output: embedded data $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n \in \mathbb{R}^{d'}$

for $i \leftarrow 1$ **to** n **do**
  Compute indices $\eta_{i1}, \ldots, \eta_{ik_i}$ of the neighborhood of $\boldsymbol{x}_i$ by the k-nearest-neighbor or ϵ-ball rule.
end for
Initialize matrix of reconstruction weights $\mathbf{W} \leftarrow \mathbf{0}_{n \times n}$.
for $i \leftarrow 1$ **to** n **do**
  Let $\mathbf{C} \in \mathbb{R}^{k_i \times k_i}$ be the local covariance matrix.
  for $j \leftarrow 1$ **to** k_i **do**
    for $l \leftarrow j$ **to** k_i **do**
      $\mathbf{C}_{jl} \leftarrow (\boldsymbol{x}_i - \boldsymbol{x}_{\eta_{ij}})^\top (\boldsymbol{x}_i - \boldsymbol{x}_{\eta_{il}})$
      $\mathbf{C}_{lj} \leftarrow \mathbf{C}_{jl}$
    end for
  end for
  $\boldsymbol{w} \leftarrow (\mathbf{C} + \nu \mathbf{I})^{-1} \mathbf{1}_{k_i}$
  $\boldsymbol{w} \leftarrow \frac{1}{w^\top \mathbf{1}_{k_i}} \boldsymbol{w}$
  for $j \leftarrow 1$ **to** k_i **do**
    $\mathbf{W}_{i\eta_{ij}} \leftarrow w_j$
  end for
end for
Let $\mathbf{M} \in \mathbb{R}^{n \times n}$ be the matrix of the quadratic form that we want to minimize.
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $\mathbf{M}_{ij} \leftarrow \delta_{ij} - \mathbf{W}_{ij} - \mathbf{W}_{ji} + \sum_{l=1}^{n} \mathbf{W}_{li} \mathbf{W}_{lj}$
  end for
end for
Compute matrix $\mathbf{V} \in \mathbb{R}^{n \times n}$ of eigenvectors (as columns) and associated eigenvalues $\lambda_1, \ldots, \lambda_n$ of $\mathbf{M}$.
Sort eigenvalues and eigenvectors $\mathbf{V}$ such that $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.
for $i \leftarrow 1$ **to** n **do**
  $\boldsymbol{z}_i \leftarrow (\mathbf{V}_{i2}, \mathbf{V}_{i3}, \ldots, \mathbf{V}_{i(d'+1)})^\top$
end for
return $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$

3.2 Clustering

3.2.1 K-Means Clustering

▷ **Name** K-means clustering

▷ **Applications** clustering, quantization

▷ **Method** The K-means clustering algorithm aims at finding centers 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 center. More formally, the objective function can be written as

$$\mathcal{L}(\{\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k\}, \boldsymbol{r}) = \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{r_i}\|^2$$

where r_i is the index of the cluster to which datapoint $\boldsymbol{x}_i$ belongs to. At each iteration, the algorithm minimizes the loss function in two steps: in the assignment step, every datapoint is assigned to its nearest cluster center. In the update step, the centers are set to the mean over their members.

▷ **Discussion** This naïve version of K-means clustering suffers from two limitations: firstly, each datapoint belongs to exactly one cluster (so-called hard memberships) and all datapoints in one cluster have equal weight in the update step. Thus, outliers may drag their center into wrong directions. Secondly, clusters are spherical which is a strong parametric assumption.

▷ **See also** Hierarchical Clustering

3.2.2 Hierarchical Clustering

▷ **Name** Hierarchical clustering

▷ **Applications** clustering, quantization

▷ **Method** In contrast to standard clustering, where each datapoint is assigned to exactly one cluster (or prototype), hierarchical clustering aims at revealing a hierarchy of clusters in the data where clusters are joined together to form super-clusters. Most hierarchical clustering algorithms can be regarded as post-processing steps to standard clustering.

Agglomorative clustering methods build a hierarchy of clusters by step-wise merging of clusters. At each step, the algorithm merges those two clusters which leads to the smallest

Algorithm 6 K-means clustering

Input: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$,
number of clusters k,
maximum number of iterations m
Output: cluster centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k \in \mathbb{R}^d$,
assignment vector $\boldsymbol{r} \in \mathbb{R}^n$

Choose random data points as initial cluster centers $\boldsymbol{\mu}_1 \leftarrow \boldsymbol{x}_{i_1}, \ldots, \boldsymbol{\mu}_k \leftarrow \boldsymbol{x}_{i_k}$ where $i_j \neq i_l$ for all $j \neq l$.
$\boldsymbol{r} \leftarrow \mathbf{0}_n$
$\boldsymbol{r}' \leftarrow \mathbf{0}_n$
for $i \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** n **do**
    Find nearest cluster center $r'_j \leftarrow \operatorname{argmin}_{1 \leq l \leq k} \|\boldsymbol{x}_j - \boldsymbol{\mu}_l\|^2$
  end for
  for $j \leftarrow 1$ **to** k **do**
    Compute new cluster center $\boldsymbol{\mu}_j \leftarrow \frac{1}{|\{l : r'_l = j\}|} \sum_{l : r'_l = j} \boldsymbol{x}_l$
  end for
  if $\boldsymbol{r} = \boldsymbol{r}'$ **then**
    break
  end if
  $\boldsymbol{r} \leftarrow \boldsymbol{r}'$
end for
return $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k, \boldsymbol{r}$

increase in the original clustering cost function. This procedure is called step-wise optimal agglomorative clustering. For K-means clustering, the cost function (to be minimized) is

$$\mathcal{L}(\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}, \boldsymbol{r}) = \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{r_i}\|^2$$

where $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are the datapoints, $\boldsymbol{r} \in \mathbb{R}^n$ is the assignment vector and $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ are the cluster centers. The assignment vector $\boldsymbol{r}$ holds the cluster index of each datapoint, i.e. $r_i \in \{1, \ldots, k\}$ and $r_i = j$ if datapoint i belongs to cluster j. For each cluster $1, \ldots, k$, its center is the mean over its members, i.e.

$$\boldsymbol{\mu}_i = \frac{1}{|\{j : r_j = i\}|} \sum_{j:r_j=i} \boldsymbol{x}_j.$$

Let us introduce one further bit of notation that we will use to formulate the algorithm: if $\boldsymbol{r} \in \mathbb{R}^n$ is an assignment vector, then we will denote by $\boldsymbol{r}^{i=j} \in \mathbb{R}^n$ the assignment vector where the cluster i and j are merged, or, formally,

$$r_l^{i=j} = \begin{cases} r_l & : \text{ if } r_l \neq i \\ j & : \text{ otherwise.} \end{cases}$$

for $l = 1, \ldots, n$.

Algorithm 7 Step-wise optimal agglomorative clustering

Input: data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$,
number of clusters k,
assignment vector $\boldsymbol{r} \in \mathbb{R}^n$,
clustering cost function $\mathcal{L} : \mathbb{R}^{d \times n} \times \mathbb{R}^n \to \mathbb{R}$ where the first argument is the data and the second argument is the assignment vector.

Output: assignment matrix $\mathbf{R} \in \mathbb{R}^{(k-2) \times n}$ with one row for each step, vector of clustering cost values $\boldsymbol{l} \in \mathbb{R}^{k-1}$ after each step.

1: Compute initial clustering cost $l_1 \leftarrow \mathcal{L}(\mathbf{X}, \boldsymbol{r})$.
2: **for** $i = 1$ **to** $k - 2$ **do**
3: Compute set C that contains the remaining cluster indices in $\boldsymbol{r}$.
4: Find the two cluster indices $c_1, c_2 \in C$ such that if we merge the clusters c_1 and c_2 the cost function $\mathcal{L}(\mathbf{X}, \boldsymbol{r}^{c_1=c_2})$ is minimal among all possible merges, or formally,

$$(c_1, c_2) = \operatorname*{argmin}_{(c_1', c_2') \in C \times C, c_1' \neq c_2'} \mathcal{L}(\mathbf{X}, \boldsymbol{r}^{c_1'=c_2'}).$$

5: Store new assignments $\boldsymbol{r} \leftarrow \boldsymbol{r}^{c_1=c_2}$ and $\mathbf{R}_{(i+1)j} \leftarrow r_j^{c_1=c_2}$ for all $1 \leq j \leq n$.
6: Compute new clustering cost $l_{i+1} \leftarrow \mathcal{L}(\mathbf{X}, \boldsymbol{r})$.
7: **end for**

▷ **Discussion** Hierarchical cluster solutions are usually visualized as dendrogram plots. The clusters correspond to positions on the horizontal axis, the vertical axis shows the change in the clustering cost function due to the merging of clusters. Each cluster is represented as a line that grows from bottom to top, where a merge results in two lines joined together at a height corresponding to the increase in the clustering loss function. Figure 3.1 shows an example of a hierarchical cluster solution along with a dendrogram plot.

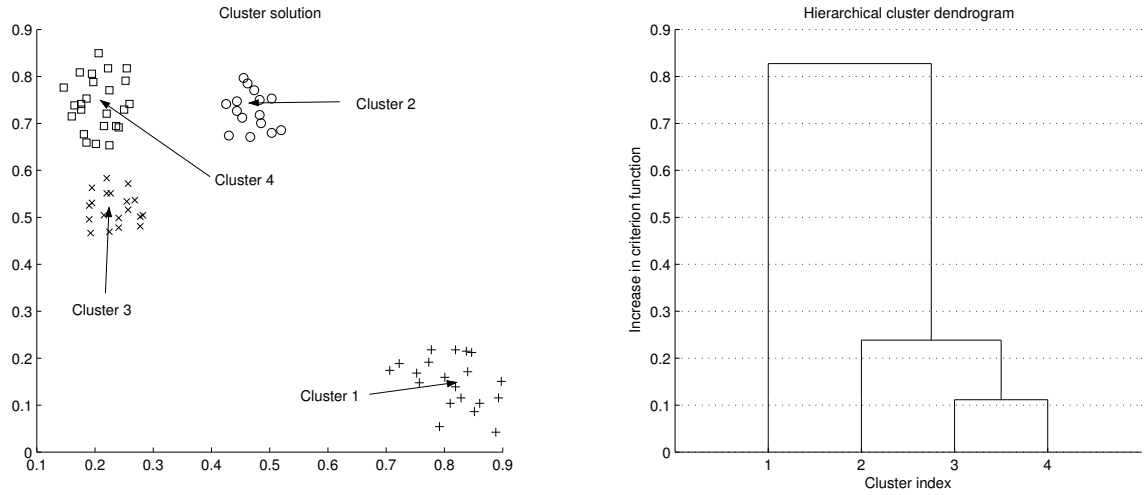

Figure 3.1: Hierarchical cluster solution. The left plot shows the most fine grained cluster structure, the right plot shows the hierarchical structure obtained from agglomorative merging of the four original clusters.

▷ **See also** K-Means Clustering

3.2.3 The EM Algorithm for Mixture Density Estimation

▷ **Name** Expectation-Maximization Algorithm for Mixture Density Estimation ("EM-Algorithm")

▷ **Applications** clustering, density estimation, quantization

▷ **Method** Actually, there is no such thing as "the" EM-Algorithm. It is rather a method to perform maximum-likelihood estimation of densities, when there exist "hidden variables" which prevent optimization.

Maximum-likelihood is a classical approach to determine the parameters of a parameterized density distribution. For example, assume that you have some points $x_1, \ldots, x_n \in \mathbb{R}$, and you think that these points are indepedent samples from a Gaussian distribution, but

you don't know the mean value μ and the variance σ^2. You want to estimate these two parameters by choosing them such that the joint probability of $x_1, \dots, x_n \in \mathbb{R}$ is maximized. This corresponds to assuming the explanation under which the data seems most plausible.

For Gaussians, this is pretty easy. The density function (also called the likelihood) is

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

If we assume that the observed points are independent, then we get the joint probability

$$p(x_1, \dots, x_n; \mu, \sigma^2) = \prod_{i=1}^{n} p(x_i; \mu, \sigma^2),$$

or for the log-likelihood (usually considered since all the exponentials just vanish, and you are left with quadratic functions only)

$$\log p(x_1, \dots, x_n; \mu, \sigma^2) = \sum_{i=1}^{n} \log p(x_i; \mu, \sigma^2) = -\frac{n}{2} \log 2\pi\sigma^2 - \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2}.$$

Maximizing with respect to μ and σ^2 (i.e. differentiating and setting to zero) leads to

$$\widehat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i, \qquad \widehat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \widehat{\mu})^2.$$

Anyhow, in this case (and also many others), maximum likelihood estimation is *easy*. You differentiate the log-likelihood and solve for the parameters you are interested in.

However, this is not possible for all kinds of density functions. We are interested in a mixture of Gaussians (and since this is more fun, we directly go to the multivariate case)

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k g(\boldsymbol{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \qquad \sum_{k=1}^{K} \pi_k = 1, \quad \pi_k \geq 0$$

with

$$g(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-d/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right).$$

In words: the probability for a point $\boldsymbol{x}$ is given as a mixture of Gaussians with different means and covariance matrices. The numbers π_k are called the class priors since they say how probable a class is.

We have to estimate three parameters: π_k, $\boldsymbol{\mu}_k$, and $\boldsymbol{\Sigma}_k$. It turns out that we cannot compute the maximizing parameters with closed forms in this case. Differentiating either of these equations leads to an equation which depends on the other quantities in a non-linear way. Also, gradient based optimization would need various constraints that are tricky to implement. For example, the covariance matrices $\boldsymbol{\Sigma}_k$ must be positive definite.

Algorithm 8 The EM Algorithm for Mixture of Gaussians

Input: Data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$,
number of mixture components $K \in \mathbb{N}$
Output: Means $\widehat{\boldsymbol{\mu}}_k$ and covariance matrices $\widehat{\boldsymbol{\Sigma}}_k$, $1 \leq k \leq K$.

{ Initialize }
$\widehat{\pi}_k \leftarrow 1/K$
$\widehat{\boldsymbol{\mu}}_k \leftarrow$ random points out of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$
$\widehat{\boldsymbol{\Sigma}}_k \leftarrow \mathbf{I}_d$
repeat
{ E-step }
  for $k \leftarrow 1$ **to** K **do**
    for $i \leftarrow 1$ **to** n **do**
      $\gamma_{ik} \leftarrow \dfrac{\widehat{\pi}_k g(\boldsymbol{x}_i; \widehat{\boldsymbol{\mu}}_k, \widehat{\boldsymbol{\Sigma}}_k)}{\sum_{k'=1}^{K} \widehat{\pi}_{k'} g(\boldsymbol{x}_i; \widehat{\boldsymbol{\mu}}_{k'}, \widehat{\boldsymbol{\Sigma}}_{k'})}$ {see (3.2.3)}
    end for
  end for
{ M-step }
  for $k \leftarrow 1$ **to** K **do**
    $n_k \leftarrow \sum_{i=1}^{n} \gamma_{ik}$
    $\widehat{\pi}_k \leftarrow n_k/n$
    $\widehat{\boldsymbol{\mu}}_k \leftarrow \frac{1}{n_k} \sum_{i=1}^{n} \gamma_{ik} \boldsymbol{x}_i$
    $\widehat{\boldsymbol{\Sigma}}_k \leftarrow \frac{1}{n_k} \sum_{i=1}^{n} \gamma_{ik} (\boldsymbol{x}_i - \widehat{\boldsymbol{\mu}}_k)(\boldsymbol{x}_i - \widehat{\boldsymbol{\mu}}_k)^\top$
  end for
until convergence

However, it turns out that estimation can be performed in an iterated manner once an intermediate quantity is introduced, which we call *responsibility* γ_{ik}. This γ_{ik} is given as the probability that data point $\boldsymbol{x}_i$ was generated by mixture component k. Once these *soft-assignments* are fixed for data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, we can estimate $\widehat{\pi}_k$, $\widehat{\boldsymbol{\mu}}_k$, and $\widehat{\boldsymbol{\Sigma}}_k$ by taking weighted variants of the usual maximum-likelihood estimators.

For reasons too obscure to be explained right now the first step is called the "E-step" (for expectation), while the second step is called "M-step" (for maximization).

It turns out that this introduction of a decoupling variable (in this case, the assignments to the clusters) is a general principle, which are applicable to a large number of probability models. The collection of all these algorithms is called the "EM Algorithm", although it is really more of a general idea than a concrete algorithm in the computer science sense.

The algorithm is summarized in Algorithm 8.

▷ **Discussion** One important point is the initialization of the algorithm. In Algorithm 8, we simply set $\widehat{\pi}_k$ and $\widehat{\boldsymbol{\Sigma}}_k$ to sane defaults and assign the cluster centers at random. Other options are possible, for example, setting $\widehat{\pi}_k$ to a random distribution (i.e. setting each $\widehat{\pi}_k$

to some positive value and normalizing everything such that they sum to one afterwards). Alternatively, one can even initialize the parameters with a solution obtained by k-means clustering.

Unfortunately, the EM algorithm is not stable. If you have more than two components, you can always drive the likelihood to infinity by centering one component on a single data point and letting the covariance go to zero. This seldom happens in practice, but we have *regularize* in order to prevent solutions where covariances tend to zero. Or, you directly go Bayesian and introduce sensible priors.

K-means clustering and EM are closely related. The algorithms structure is already very similar. In principle k-means can be interpreted as a crippled EM with covariance matrices fixed to identity matrices and considering hard assignments for the γ_{ik}.

▷ **See also** K-Means Clustering

▷ **Literature** Bishop, "Pattern Recognition and Machine Learning", pp. 435

3.3 Outlier Detection

Outliers are data points that are far away from the rest of the data. They are few and considered atypical, in the sense that they were generated in a different way than the normal data points. For example, outliers may arise due to measurement errors, unexpected artifacts or certain rare events. Many real data sets contain a small number of outliers, e.g. 1-5% of the observations.

Identifying the outliers in a data set is useful not only because they often correspond to curious and unique special cases, but also because they are problematic for a wide range of analysis methods. Therefore, semi-automatic outlier removal is a common pre-processing step. For two dimensions, looking at a scatter plot is good way to find outliers. In higher dimensions, however, this becomes impractical. Moreover, a univariate approach may not be able to distinguish between normal and extreme data points because it discards the dependencies between the dimensions.

The usual approach to outlier detection is to model the normal data points (i.e. the majority) and calculate, for each data point, an outlier score that can be thought of as the distance to normality. Finally, all data points with scores above a certain threshold are deemed outliers. The choice of this threshold is often guided by the distribution of the scores, manual inspection, or a fixed percentage of the data that one expects to reject.

In practice, we rarely have ground truth information about outliers, i.e. it is difficult to assess.

3.3.1 γ-index

▷ **Name** γ-index

▷ **Applications** outlier detection, ranking

▷ **Method** Given a data set $\mathbf{X}$, we want to find a function $f\colon \mathcal{X} \to \mathbb{R}$ such that $f(\boldsymbol{x}_i) < f(\boldsymbol{x}_j)$ iff $\boldsymbol{x}_i$ is more *typical*. This function would allows us to order data points from prototypes to outliers.

As a simplified example, assume our data set forms a single, connected, isotropic blob around a center point $\boldsymbol{\mu}$, and the density of points decreases with the distance to $\boldsymbol{\mu}$.[1] In that case, the Euclidean distance to $\boldsymbol{\mu}$ is a function that provides the desired ordering:

$$f(\boldsymbol{x}) = \|\boldsymbol{x} - \boldsymbol{\mu}\|_{\ell_2}$$

Now consider the case where our data comes from a multimodal distribution or aligns along a curved manifold. The measure f will fail in these cases. A better way to order such data is a score for local density: when a data point has no close neighbors, it cannot be *typical*. One particularly simple measure is the γ-index. It measures the mean distance to the k nearest neighbors:

$$\gamma(\boldsymbol{x}) = \frac{1}{k} \sum_{j=1}^{k} \|\boldsymbol{x} - \boldsymbol{z}_j(\boldsymbol{x})\|_{\ell_2}$$

where $\boldsymbol{z}_j(\boldsymbol{x}) \in \mathbf{X}$ is the j-nearest neighbor of $\boldsymbol{x}$. This score has a clear interpretation. $\gamma(\boldsymbol{x})$ will be small in dense regions and large in sparse regions. The only parameter to tune is the size of the neighborhood, k.

▷ **Discussion** There are clearly cases where the γ-index fails. For example, if the density is locally equal everywhere in the data, i.e. the same value is produced for every data point. Or if the metric does not capture the desired properties of the data. Although the ℓ_2-distance can effortlessly be replaced by any metric, it is often the very point of outlier detection to learn the right metric in the first place. Many methods exist that project data into a feature space where Euclidean distances become meaningful for the data at hand. This can be an engineered kernel, a neural network, diffusion process, Fourier transformation, latent variable model and so on. In this regard, the γ-index is a useful tool, but often does not solve the problem alone.

[1] One kind of process that produces data like that would be an isotropic Gaussian distribution.

Chapter 4

Supervised Learning

4.1 Introduction

Supervised learning is when we have labels. That is, every training example $\boldsymbol{x}_i \in \mathcal{X}$ is accompanied by the target variable $y_i \in \mathcal{Y}$ that we aim to predict from $\boldsymbol{x}_i$. In regression, $y_i \in \mathbb{R}$ is called *target* and is typically real-valued; in classification, our target is to predict discrete class *labels* $y_i \in \{c_1, \ldots, c_m\}$. For instance, if we want to build a system for optical character recognition (OCR), each example $\boldsymbol{x}_i$ could be an image of a digit and its label $y_i \in \{0, 1, \ldots, 9\}$ tells us which digit is shown. The presence of labels allows us to evaluate (or supervise) the performance of a predictor by comparing its output against the target whereas in unsupervised learning, one has to resort to subjective measures such as plausibility or beauty of the found solution. In fact, many supervised learning algorithms simply stem from minimizing the *loss function* on the training data.

In classification problems, the predominant notion of loss is the misclassification rate, i.e. the number of incorrectly predicted labels divided by the total number of examples. In practice, the loss function can also be a function of the data. For regression, it is less obvious what constitutes a correct prediction. Typical loss functions are the squared distance between target and prediction, (L_2 loss) or the absolute difference (L_1 loss).

Moreover, there are different types of errors. Imagine, for example, we want to predict whether an email $\boldsymbol{x}_i$ is spam ($y_i = -1$) or ham ($y_i = +1$). While it is surely annoying to manually delete some spam emails from your inbox, it would be far worse, if the system stuffed precious ham into the spam bin. Conversely, in cancer diagnosis it is obviously better to issue some false alarms (*false positives*) than ignoring the possibility that a patient is sick (*false negatives*). The desired trade-off between false positives and false negatives depends on the application and the cost (or risk) involved. In many algorithms, this trade-off can be controlled by some parameter. We can thereby identify the relationship between false positives and true positives which is usually depicted as a ROC curve (receiver-operator-characteristic): percentages of false positives and true positivies are shown on the x- and y-axis respectively. The optimal curve would thus be a horizontal line at height 1 which is rarely attainable in practice. The area under the ROC curve (AUC) is a common performance measure for binary classifiers. Figure 4.1 shows a one-dimensional

classification problem along with the corresponding ROC-curve obtained from varying the decision threshold on the x-axis.

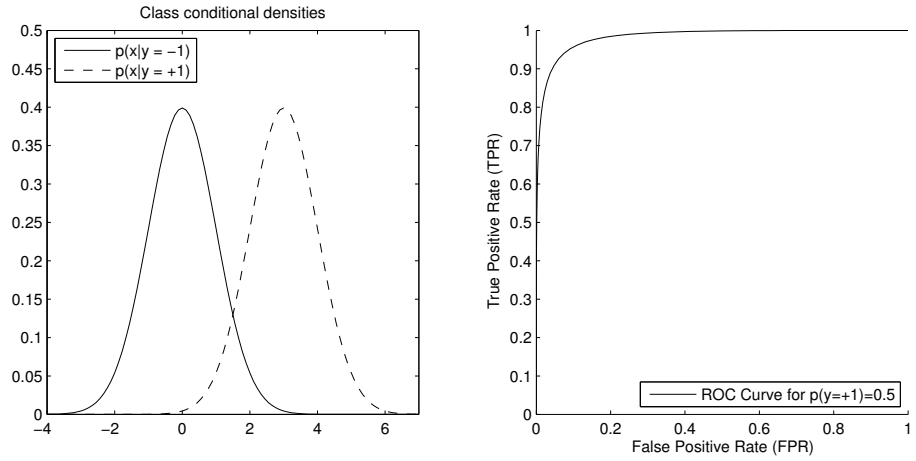

Figure 4.1: The left plot shows the class-conditional densities $p(x \mid y = +1)$ and $p(x \mid y = -1)$ for a one-dimensional classification problem. The right plot contains the ROC-curve obtained from varying the decision boundary.

Labelled training data is valuable but it comes at a peril: we are tempted to learn it by heart. Remember that our ultimate goal is to train a predictor which performs well on data that we have not observed in the training set. Thus, the error rate on the training set should only be regarded as an approximization to the *generalization error* which is the true performance on unseen data. We have to bear this in mind when we adjust the parameters of our classifier, the *model selection* step. Most classifiers come with a knob that controls the complexity of the learned function. If we follow the naïve approach of minimizing the error rate on the training set, we will inevitably choose a level of complexity that allows us to exactly reproduce the training labels. This behaviour is called *overfitting* and leads to poor generalisation performance since we will most likely fit properties of that particular sample, such as noise or other small sample artefacts that are not representative of the underlying data distribution. Figure 4.2 shows an example of training data with a decision boundary at three different levels of complexity: too simple (poor training and generalization performance), right fit (optimal generalization performance) and overfitting (best training but poor generalization performance). Selecting the right model without prior knowledge or abundant training data remains a difficult problem. The most common remedy is to employ the cross validation principle which is introduced in the following Section 4.2.2.

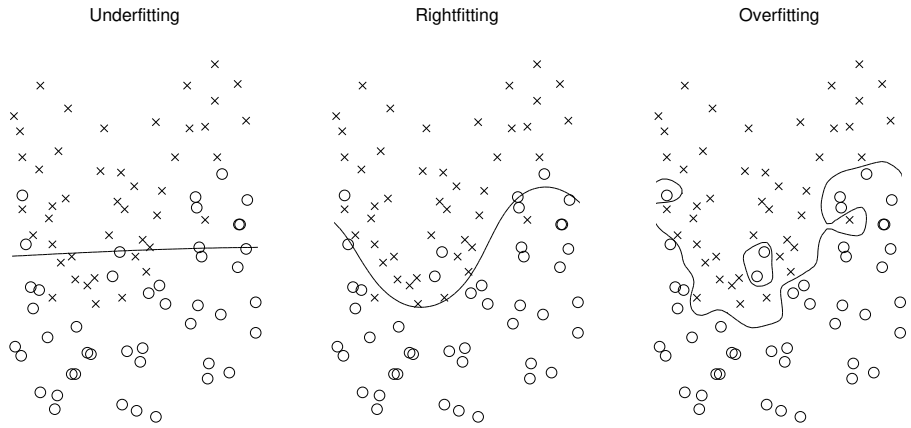

Figure 4.2: From left to right, the plots correspond to an underfitted, rightfitted and overfitted model. The models are kernel ridge regression classifiers with gaussian kernels where the kernel widths are set to $\sigma = 1, 0.05, 0.001$ respectively and fixed $\tau = 1$.

4.2 General Tools

4.2.1 ROC Curve

As mentioned above, ROC curve is a somewhat stable tool for evaluating binary classifiers, e.g. in the presence of class-imbalance or different cost for false positives and false negatives.

Although it is easy to understand and interpret, the implementation might be a bit dubious at first: What is the trade-off parameter? What are the candidate values? How many candidate values do we need? In a nutshell, the vector of prediction scores itself is the reasonable set of candidate values and, in fact, the ordering of these scores is enough to draw the curve.

Below (Algorithm 9), we show an efficient implementation. It might not be obvious why it answers all those questions, but it does. Note that the second input is the real-valued prediction score, i.e. the output of your classifier before applying the sign function!

4.2.2 Cross-validation

In the previous paragraph we have introduced the model selection problem: we want to find parameters for a classifiers that lead to good performance on unseen data (small generalization error) given only a limited amount of training data. If we simply choose those parameters which minimize the error rate on the training set, we are doomed to overfit it. That is because training and testing a classifier on the *same* data is not a good measure of its performance on unseen data. This observation leads to the cross-validation (CV) method which is a general principle applicable to a wide range of model selection problems. In the following, we will consider cross validation for adjusting parameters of a classifier where the loss is the misclassification rate (also called 0-1-loss).

Algorithm 9 ROC curve

Input: True labels $y_1, \ldots, y_n \in \{-1, +1\}$
Prediction scores $v_1, \ldots, v_n = f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_n) \in \mathbb{R}$.
Output: Pairs $(\mathrm{fpr}_i, \mathrm{tpr}_i)_{i=0}^n$ — the points on the ROC curve

$o \leftarrow \mathrm{argsort}(\boldsymbol{v}, \mathrm{descending} = \mathrm{true})$ { sort in descending order }
$\delta_{\mathrm{neg}} \leftarrow 1/\mathrm{sum}(\boldsymbol{y} == -1)$
$\delta_{\mathrm{pos}} \leftarrow 1/\mathrm{sum}(\boldsymbol{y} == +1)$
$(\mathrm{fpr}_0, \mathrm{tpr}_0) \leftarrow (0, 0)$
for $i = 1$ **to** n **do**
  $\mathrm{fpr}_i \leftarrow \mathrm{fpr}_{i-1} + \frac{1}{2}(1 - y_{o_i}) \cdot \delta_{\mathrm{neg}}$ { note: y is indexed by o }
  $\mathrm{tpr}_i \leftarrow \mathrm{tpr}_{i-1} + \frac{1}{2}(1 + y_{o_i}) \cdot \delta_{\mathrm{pos}}$
end for
return $(\mathrm{fpr}_i, \mathrm{tpr}_i)_{i=0}^n$

The basic scheme is that we randomly split the available data into training and test set, where the classifier is trained on the first and evaluated on the latter. As training data is scarce and we want to avoid artifacts from random splitting, the most popular practical method is to use repeated k-fold cross-validation: the data is randomly split into k parts of equal size. For each of the partitions $1 \leq j \leq k$, we train a classifier on the union of all the other partitions and evaluate the performance on partition j. The average misclassification rate over repetitions and folds serves as an estimate of the generalization performance for a particular set of parameters.

Let us now turn to a more formal treatment of the procedure. We have training data $\boldsymbol{x}_1, \ldots \boldsymbol{x}_n \in \mathbb{R}^d$ with corresponding class labels $y_1, \ldots, y_n \in \{-1, +1\}$. Our aim is to find parameters θ (via classifier training) for a function

$$f_{\theta;(x_1,y_1),\ldots,(x_n,y_n)} : \mathbb{R}^d \to \{-1, +1\}$$

which predicts the label given the data. The criterion for choosing θ is that the generalization error $g(\theta)$ is minimal. Imagine we have a set of candidates $\theta_1, \ldots, \theta_l$. Then, formally speaking, our rule for choosing θ^* is

$$\theta^* = \operatorname*{argmin}_{\theta_1,\ldots,\theta_l} g(\theta).$$

However, since the function g is usually unknown, we have to rely on estimates. The following Algorithm 10 computes an estimate $\hat{g}(\theta)$ of the generalization error for a parameter θ using r-times k-fold cross-validation.

4.3 Classical Methods

4.3.1 Ordinary Least Squares and Ridge Regression

▷ **Name** Ordinary Least Squares

Algorithm 10 Cross-validation

Input: training data $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$,
number of partitions k,
number of repetitions r,
parameter θ
Output: average error rate $\bar{e}$

Set $\bar{e} \leftarrow 0$
for $i = 1$ **to** r **do**
  Let $P_1, \ldots, P_k$ be a random partitioning of the training data of equal size (approximatively, since n may not be divisible by k), i.e. $|P_1| \approx \ldots \approx |P_k|$
  for $j = 1$ **to** k **do**
    Assemble training set $T_j \leftarrow \bigcup_{l \neq j} P_l$
    for each $(\boldsymbol{x}_i, y_i) \in P_j$ **do**
      Predict label $\widehat{y}_i \leftarrow f_{\theta;T_j}(\boldsymbol{x}_i)$ using classifier trained on T_j
      if $\widehat{y}_i \neq y_i$ **then**
        $\bar{e} \leftarrow \bar{e} + \frac{1}{|P_j|}$
      end if
    end for
  end for
end for
Set $\bar{e} \leftarrow \frac{1}{kr}\bar{e}$

▷ **Applications** Regression, Classification

▷ **Method** This method is almost as old as applied mathematics itself. Legend has it that Gauß himself invented this method when he was 18[1]. Anyway, the basic idea is simple: We assume that the dependency between the data $\mathbf{X}$ and labels $\boldsymbol{y}$ is not very complicated, but "just" linear. This means that

$$y = \sum_{i=1}^{d} w_i x_i, \qquad \text{or, in vector notation} \qquad y = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$$

for some weight vector $\boldsymbol{w} \in \mathbb{R}^d$. For simplicity, we have assumed that there is no offset $b \in \mathbb{R}$. On an actual data set this can be accomplished by centering $\mathbf{X}$ and $\boldsymbol{y}$ first.

Now, we wish to infer $\boldsymbol{w}$ from a finite set of examples: vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ (the inputs) and real numbers $y_1, \ldots, y_n \in$ (the outputs). In principle, this task is well-defined, as soon as one has as many data points as the dimensionality of the underlying vector space. In reality, however, the data will always be noisy, such that we will not be able to find a $\boldsymbol{w}$ such that the equation above is satisfied exactly.

Instead, we want to find a $\boldsymbol{w}$ such that the squared error between the predicted $\widehat{\boldsymbol{y}}$ and the actual $\boldsymbol{y}$ is minimal:

$$\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \right)^2 . \tag{4.1}$$

This means that we do not look for a $\boldsymbol{w}$ which fits the data points exactly, but which tries to find a good compromise between the noisy vectors.

The actual solution is found as follows: First, note that we can write Equation (4.1) more compactly using matrix notation. Recall that $\mathbf{X}$ is the $n \times d$ matrix whose rows are the $\boldsymbol{x}_i$, and $\boldsymbol{y}$ the vector whose entries are the targets y_i. Then,

$$\sum_{i=1}^{n} \left(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i \right)^2 = \|\mathbf{X}\boldsymbol{w} - \boldsymbol{y}\|^2.$$

Taking gradients w.r.t. $\boldsymbol{w}$, we obtain

$$\frac{\partial}{\partial \boldsymbol{w}} \|\mathbf{X}\boldsymbol{w} - \boldsymbol{y}\|^2 = \frac{\partial}{\partial \boldsymbol{w}} (\boldsymbol{w}^\top \mathbf{X}^\top \mathbf{X} \boldsymbol{w} - 2\boldsymbol{w}^\top \mathbf{X}^\top \boldsymbol{y} + \boldsymbol{y}^\top \boldsymbol{y}) = 2\mathbf{X}^\top \mathbf{X}\boldsymbol{w} - 2\mathbf{X}^\top \boldsymbol{y} \stackrel{!}{=} 0,$$

or

$$\mathbf{X}^\top \mathbf{X}\boldsymbol{w} \stackrel{!}{=} \mathbf{X}^\top \boldsymbol{y}.$$

We'll assume that $\mathbf{X}^\top \mathbf{X}$ is invertible, and get

$$\boldsymbol{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{y}.$$

[1] `http://en.wikipedia.org/wiki/Least_squares`

Algorithm 11 Ordinary Least Squares

Input: Input features $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$
Output labels $y_1, \ldots, y_n$.
Output: Output weight vector $\boldsymbol{w}$
1: Set $\mathbf{X} \leftarrow (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)^\top$. {Store $\boldsymbol{x}_i$ in rows of $\mathbf{X}$}
2: Set $\boldsymbol{y} \leftarrow (y_1, \ldots, y_n)^\top$.
3: Center $\mathbf{X}$ and $\boldsymbol{y}$ if not already the case.
4: $\boldsymbol{w} \leftarrow (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{y}$.

▷ **Discussion** OLS more or less behaves as expected. The idea of minimizing the squared error is also quite general. The reason why this works is that the squared error is minimized by the expectation: Consider a random variable X. The number α which minimizes the expected square error $\mathbb{E}[(X-\alpha)^2]$ is given by $\alpha = \mathbb{E}[X]$. Therefore, squared error is usefull to remove any additive zero-mean noise.

The maybe biggest drawback of the squared error is its sensitivity to outliers. Assume that in one of the Y_i there was a really huge estimation error, leading to a very large value of Y_i. Since the error is squared, this error can then dominate the whole cost function and effectively sabotage learning of $\boldsymbol{w}$. 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).

We have assumed that the data is already centered. There are at least three ways to deal with this practically. The first is to center the data "by hand". This makes the prediction a bit more complex as new points have to be transformed first, and the prediction afterwards. The second alternative is to explicitly optimize for the offset b as well. This is in principle also possible, but the formulas become a bit more complex. The third alternative is to transform the X by adding an extra dimension which is always set to 1. This way, the offset is computed automatically. This last alternative becomes a bit problematic if regularization is used (see below).

Ordinary least squares can be extended to fitting linear-combinations of functions quite easily. For example, in order to fit third-order polynomials, we transform X as follows:

$$X \mapsto (1, X, X^2, X^3).$$

The fitted function will then be

$$f(x) = w_0 + w_1 X + w_2 X^2 + w_3 X^3,$$

a polynomial of degree three as promised. In this setting, the matrix $\mathbf{X}$ will contain entries

$$\mathbf{X} = \begin{bmatrix} 1 & X_1 & X_1^2 & X_1^3 \\ \vdots & & \ddots & \vdots \\ 1 & X_n & X_n^2 & X_n^3 \end{bmatrix}$$

and is sometimes called the *design matrix*.

Finally, in particular for high-dimensional data, the matrix $\mathbf{X}^\top\mathbf{X}$ can be close to singular. In such situations, one stabilizes the solution by regularizing the weight vector $\boldsymbol{w}$.

The matrix $\mathbf{X}^\top\mathbf{X}$ has been introduced as the covariance matrix (see section 3.1.1). The eigenvalues of this matrix are therefore the principal values. For high-dimensional data, often many correlations between individual coordinates exist, such that there are only a few large principal values. Small principal values make estimation of weights difficult, in particular since the data is known to contain noise.

Therefore, one stabilizes the solution by restricting the size of the $\boldsymbol{w}$, and thereby also the amount of fluctuation possible. Computationally, this is done by adding a regularization term to the original cost-function:

$$\min_{\boldsymbol{w}} \|\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\|^2 + C\|\boldsymbol{w}\|^2.$$

In order to optimize this function, a good compromise has to be found between the goodness of the fit, and the size of the weight vector $\boldsymbol{w}$.

Interestingly, the solution does not differ too much from the original problem:

$$\widehat{\boldsymbol{w}} = (\mathbf{X}^\top\mathbf{X} + C\mathbf{I})^{-1}\mathbf{X}^\top\boldsymbol{y}$$

The term $C\mathbf{I}$ "pushes" the eigenvalues of $\mathbf{X}^\top\mathbf{X}$ up, away from 0 and therefore stabilizes the solution.

This algorithm is also known as *Ridge Regression*. The choice of the regularization constant becomes much more important in the context of kernelized functions, see .

▷ **See also** Kernel Ridge Regression

4.4 Kernel methods

4.4.1 The Kernel Trick

Linear methods (that is, methods which learn a linear discriminant function, or regression) have a certain appeal: The underlying mathematics is often not overly complex, and there is a good geometric intuition about what is happening.

On the other hand, linear methods are just not flexible enough, as many interesting phenomena are actually non-linear. So what can we do? Ideally, we would like to have the best of both worlds: Easy mathematics, well-defined optimization problems *and* powerful discrimination functions.

In the discussion of ordinary least squares, we have already seen one way of dealing with non-linear data: By transforming the data using a finite set of basis functions, we can fit non-linear functions.

In a certain sense, the *kernel trick* amounts to taking this idea to its extreme. Instead of a fixed number of basis functions, we use a set of kernel functions, one for each new data

point. There is another important ingredient: The transformation of the feature points is only performed *implicitly*. This allows us not only to use n basis functions on n data points, but also to use potentially *infinite-dimensional* feature spaces.

The kernel trick boils down to replacing scalar products between data points by a function k. In order for such function to be admissible replacements, the kernel has to satisfy the important condition that for any set of points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, the matrix $\mathbf{K} = (k(\boldsymbol{x}_i, \boldsymbol{x}_j))_{i,j=1}^n$ has to be positive definite, just as would be the case if $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$.

It can be shown that in this case, k can be interpreted as a scalar-product on transformed features $\Phi(\boldsymbol{x})$ where $\Phi\colon \mathcal{X} \to \mathcal{F}$:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \langle \Phi(\boldsymbol{x}), \Phi(\boldsymbol{x}') \rangle_{\mathcal{F}}.$$

Now, let us consider replacing the usual linear function $f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$ by a kernel. Since we can only replace scalar products between data points $\boldsymbol{x}_i$, we first represent the weight vector $\boldsymbol{w}$ as a linear combination of data points: $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$. That this is actually possible is the result of so-called *representer theorem*.

$$f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle = \left\langle \sum_{i=1}^n \alpha_i \boldsymbol{x}_i, \boldsymbol{x} \right\rangle = \sum_{i=1}^n \alpha_i \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle$$

Now, let $\Phi\colon \mathcal{X} \to \mathcal{F}$ be map into a feature space $\mathcal{F}$ such that the dot product in $\mathcal{F}$ induces kernel k. The representer theorem still holds: $\boldsymbol{w} \in \mathcal{F}$ will have the form[2] $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \Phi(\boldsymbol{x}_i)$. Then,

$$f(\boldsymbol{x}) = \langle \boldsymbol{w}, \Phi(\boldsymbol{x}) \rangle_{\mathcal{F}} = \sum_{i=1}^n \alpha_i \langle \Phi(\boldsymbol{x}_i), \Phi(\boldsymbol{x}) \rangle_{\mathcal{F}} = \sum_{i=1}^n \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}) \tag{4.2}$$

So, introducing the kernel enhances the power of linear methods. The last formula is just a linear prediction on transformed data. Two typical examples are polynomial kernels and Gaussian kernels:

Polynomial kernel:	$k(\boldsymbol{x}, \boldsymbol{x}') = (\langle \boldsymbol{x}, \boldsymbol{x}' \rangle + 1)^d$
Gaussian kernel:	$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\dfrac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right)$

The main problem in the case of kernel methods is proper regularization. For the Gaussian kernel, the feature space is potentially infinite-dimensional, and methods will directly overfit severly without proper regularization.

Computing Distance-Based Kernels Efficiently

In order to compute distance-based kernels like the Gaussian kernel efficiently, one should rather not start by computing all n^2 pairwise distances in big loops. In interpreted languages like Python, such explicit computations are inherently slower than the matrix multiplications (which are usually performed using some highly optimized toolbox like BLAS).

[2]The $\alpha_1, \ldots, \alpha_n$ will change, depending on the kernel k.

But even if you are coding in C, there is a reason not compute the distances explicitly. As we will see below, one can reduce the problem of computing the squared distances to some matrix algebra. Now, this matrix algebra basically has the same theoretical complexity as computing the distances explicitly, however, for large numbers of training examples or high dimensional data, there is an additional effect to take into account, namely L1 and L2-cache misses. The data becomes to large to fit in those caches, and every naïve implementation will likely not be very good in terms of memory locality, leading to a huge number of cache misses and severe performance degradation. (If you don't believe me, benchmark your own naïve implementation of matrix-matrix multiplication against those of ATLAS. I doubt that you have a chance).

So by reducing the problem to matrix algebra, you can directly take advantage of the highly-optimized existing linear algebra routines, leading to extremely efficient implementations.

After this rather lengthy preamble, let us come to the trick. It is very easy, for two vectors $\boldsymbol{x}$, and $\boldsymbol{x}'$, making it "vectorized" over a set of vectors is bit harder.

Recall that the squared norm is just the scalar product $\|\boldsymbol{x}\|^2 = \langle \boldsymbol{x}, \boldsymbol{x} \rangle$. Then for $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^d$,

$$\|\boldsymbol{x} - \boldsymbol{z}\|^2 = \langle \boldsymbol{x} - \boldsymbol{z}, \boldsymbol{x} - \boldsymbol{z} \rangle = \langle \boldsymbol{x}, \boldsymbol{x} \rangle - 2\langle \boldsymbol{x}, \boldsymbol{z} \rangle + \langle \boldsymbol{z}, \boldsymbol{z} \rangle.$$

Now, assume that we have two matrices $\mathbf{X}, \mathbf{Z} \in \mathbb{R}^{n \times d}$ which have the same number of columns and whose rows are the data vectors. All pairwise scalar products are easily computed by $\mathbf{X}\mathbf{Z}^\top$. For the individual norms, we take the squared row sums $\|\boldsymbol{x}_i\|^2 = \sum_{j=1}^{d} [\mathbf{X}]_{ij}^2$.

Then, do figure out how to map the square sums on columns or rows of the scalar-products, and you are done, basically.

4.4.2 Kernel Ridge Regression

▷ **Name** Kernel Ridge Regression

▷ **Applications** Classification, Regression

▷ **Method** Kernel Ridge Regression (KRR) amounts to kernelized ridge regression (cf. Section 4.3.1).

Let us retrace the transformation step by step. Recall that scalar products between the data points are replaced by the kernel evaluation. Therefore, $\mathbf{X}\mathbf{X}^\top$ is replaced by the *kernel matrix* $\mathbf{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. The other ingredient is representing $\boldsymbol{w}$ as a linear combination of the $\boldsymbol{x}_i$: $\boldsymbol{w} = \mathbf{X}^\top \boldsymbol{\alpha}$.

With these two terms, we can transform the ridge regression cost function and obtain

KRR:

$$\min_{\boldsymbol{w}} \|\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\|^2 + C\|\boldsymbol{w}\|^2 \xrightarrow{\boldsymbol{w}=\mathbf{X}^\top\boldsymbol{\alpha}} \min_{\boldsymbol{\alpha}} \|\boldsymbol{y} - \mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha}\|^2 + C\boldsymbol{\alpha}^\top\mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha}$$
$$\xrightarrow{\mathbf{X}\mathbf{X}^\top=\mathbf{K}} \min_{\boldsymbol{\alpha}} \|\boldsymbol{y} - \mathbf{K}\boldsymbol{\alpha}\|^2 + C\boldsymbol{\alpha}^\top\mathbf{K}\boldsymbol{\alpha}$$

The solution to this optimization problem is surprisingly simple:

$$\widehat{\boldsymbol{\alpha}} = (\mathbf{K} + C\mathbf{I})^{-1}\boldsymbol{y}.$$

The vector $\boldsymbol{\alpha}$ contains the coefficients of the learned function $f(\boldsymbol{x})$ which you evaluate in order to make predictions, see Equation (4.2).

Algorithm 12 Kernel Ridge Regression

Input: Input features $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$,
output labels $y_1, \ldots, y_n \in \mathbb{R}$,
kernel function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

Output: Weight vector $\boldsymbol{\alpha}$

1: Compute kernel matrix $\mathbf{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for $1 \leq i, j \leq n$.
2: $\boldsymbol{\alpha} \leftarrow (\mathbf{K} + C\mathbf{I})^{-1}\boldsymbol{y}$.

▷ **Discussion** Kernel ridge regression may be the kernel methods which can be implemented most easily.[3] The downside is that it does not scale very well: Inversion of the kernel matrix might be practical up to a few thousand data points. In cases of huge data sets, other algorithms exist (for example, conjugate gradients). KRR also does not produce sparse solutions, all of the α_i will have non-zero value.

These thoughts aside, KRR is just as powerfull as Support Vector Machines (for example).

Efficient Leave-One-Out Cross-Validation

An interesting additional property of KRR is that the automatic selection of C 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 $\mathbf{S} = \mathbf{K}(\mathbf{K} + C\mathbf{I})^{-1}$ be the *hat* matrix. Then the leave-one-out cross-validation error for the squared loss is given by

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

The next insight is that $\mathbf{S}\boldsymbol{y}$ 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

[3] And we don't say that just because one of the authors has written his thesis about it.

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,

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

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

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

4.4.3 Support Vector Machines

▷ **Name** Support Vector Machine

▷ **Applications** Classification

▷ **Method** The Support Vector Machine (SVM) in its original form computes nothing more than a separating hyperplane (that is, a linear separation) between two classes. What brought SVMs to fame are the following features:

- The separating hyperplane is chosen to maximize the "margin". Geometrically, this means that the decision boundary tries to be as far from the given points as possible. Thus, even when there is some noise on the data set, they are classified robustly.
- One can show that such maximum margin hyperplanes have nice statistical features. In particular, their "Vapnik-Chervonenkis"-dimension (a measure for the complexity of a set of functions) does not depend on the dimension of the underlying space. Put differently, if the data points are contained in a ball of finite radius and can be separated by a hyperplane with a large margin, one can prove that the solution converges as more points become available.
- Using the kernel trick, one can easily extend the algorithm to produce non-linear decision boundaries.
- SVMs can be learned efficiently. Modern implementation easily scale up to hundreds of thousands or even several million data points.

There exist several different versions of SVMs which choose slightly different norms for different parts. The original support vector machine cannot deal with misclassified points and will therefore be omitted.

Support Vector Machines learn a function

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i k(\boldsymbol{x}_i, \boldsymbol{x}) - b.$$

This basically means that a new function is placed around every data point, and in principle, as $n \to \infty$, the set of functions which can be represented like this becomes more and more complex.

Now Support Vector Machines are special since the solutions are *sparse*, which means that some of the α_i are zero. If a coefficient α_i is zero it means that we do not need $\boldsymbol{x}_i$ for prediction. The remaining data points where $\alpha_i \neq 0$ are called "support vectors", giving the method its name.[4]

Algorithmically, the SVM is learned by solving the following quadratic optimization problem (corresponding to the 1-Norm Soft Margin, see below).

$$\begin{aligned} &\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_j) \alpha_j y_j \\ &\text{subject to } 0 \leq \alpha_i \leq C, \quad 1 \leq i \leq n \\ &\sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned} \tag{4.3}$$

The offset b is chosen such that $y_i f(X_i) = 1$ for any i with $0 < \alpha_i < C$. Note that the objective function is a quadratic function in $\boldsymbol{\alpha} \in \mathbb{R}^n$ due to the second term of the objective function, while the constraints are all linear.

Actually, as we will discuss below, this is the *dual* problem to the original optimization problem. While the dual problem is easier to optimize, the original problem is easier to understand. For now, let us just say that the optimization problem minimizes the error of a misprediction while keeping the weight vector small. The error of the prediction at $\boldsymbol{x}_i$ is measured by

$$\max(0, 1 - y_i f(\boldsymbol{x}_i)).$$

Let's take a closer look at this function. If $y_i f(\boldsymbol{x}_i) \geq 1$, then the error is zero. Note that this implies that y_i and $f(\boldsymbol{x}_i)$ have the same sign (recall that in classification y_i is either $+1$ or -1). On the other hand, if $y_i f(\boldsymbol{x}_i) < 1$, the deviation is penalized linearly. In other words, if $y_i = 1$, then $f(\boldsymbol{x}_i)$ should be larger than 1, otherwise we have an error.

The next problem is how to optimize the above problem. One solution is to pass the whole problem to some generic quadratic optimizer like Matlab's `quadprog`. However, it is instructive to see that the problem can actually be solved by hand, as we will discuss next.

The Sequential Minimal Optimization Algorithm

This algorithm due to Microsoft's John Platt (inventor of the ClearType™ subpixel smoothing for LCD displays, among other things).

The crucial insight was that it is possible to solve the problem iteratively by considering only two variables α_i and α_j at a time. Just one variable won't work due to the constraint

[4]The input space $\mathcal{X}$ does not need to be a vector space. To be absolutely clear, not $\boldsymbol{x}_i \in \mathcal{X}$ is the support vector, but $\Phi(\boldsymbol{x}_i) \in \mathcal{F}$ in feature space, where $\mathcal{F}$ must be a vector space.

$\sum_i y_i \alpha_i = 0$. If you take two variables, this constraint effectively removes one dimension such that the resulting optimization problem is one-dimensional and can be solved in closed form.

Algorithm 13 summarizes the computation necessary to compute the restricted problem (where we have assumed for simplicity that we are optimizing α_1, and α_2). The computations might actually look quite complicated but can be derived using basic computations. The optimization is made difficult because you have to make sure that the box constraints $0 \leq \alpha_i \leq C$ are not violated by clipping the result accordingly if the minimum lies outside of the box.

So in principle, you could pick two coordinates of the αs, and in each step, it is guaranteed that you will get closer to the true solution—only very slowly if you don't choose the coordinates correctly.

We need a heuristic to choose the coordinates. It is based on the concept of "Karush-Kuhn-Tucker"-conditions. These are conditions which indicate that a certain point is optimal. We will concentrate on points for which the KKT-conditions are violated, thereby hopefully getting closer to the true solution.

For the problem above, the KKT-conditions read

$$\begin{aligned} y_i f(\boldsymbol{x}_i) &\geq 1 \qquad \text{if } \alpha_i = 0 \\ y_i f(\boldsymbol{x}_i) &\leq 1 \qquad \text{if } \alpha_i = C \\ y_i f(\boldsymbol{x}_i) &= 1 \qquad \text{if } 0 < \alpha_i < C. \end{aligned}$$

So, we choose the first coordinate whenever

$$(\alpha_i < C \text{ and } y_i f(\boldsymbol{x}_i) < 1 - \varepsilon) \text{ or } (\alpha_i > 0 \text{ and } y_i f(\boldsymbol{x}_i) > 1 + \varepsilon),$$

where ε is a (small) tolerance level. The second coordinate is chosen at random such that it does not coincide with the first one. The resulting algorithm is summarized in Algorithm 14.

▷ **Discussion** There are a lot of things one could say about SVMs. The most important once have already been stated above: SVMs compute decision boundaries which can be shown to be statistically robust (although it is not easy to exactly quantify how robust). Using kernels, SVMs can be adapted to many different applications. Finally, the optimization problem can be solved efficiently such that it is possible to train on up to several million data points.

The statistical properties are clearly beyond the scope of this guide. We will therefore focus on the practical aspects and discuss the optimization problem a bit more.

The original optimization problem is given by the following problem (initially formulted for the case of a linear kernel (that is, $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z}$). This is also known as the *primal*:

$$\begin{aligned} \min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}^\top \boldsymbol{x} + b) \geq 1 - \xi_i, \quad 1 \leq i \leq n \\ & \xi_i \geq 0, \quad 1 \leq i \leq n \end{aligned} \tag{4.4}$$

Algorithm 13 Optimizing for α_1, α_2

Input: Input points $\boldsymbol{x}_1, \boldsymbol{x}_2$
Input labels $y_1, y_2 \in \{\pm 1\}$
Kernel function $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$
Prediction errors $E_1 = f(\boldsymbol{x}_1) - y_1$, $E_2 = f(\boldsymbol{x}_2) - y_2$
Old parameters α_1^{old}, α_2^{old}, b^{old}
Output: Updated parameters α_1^{new}, α_2^{new}, b^{new}
or "no changes"

{ Compute box constraints }
if $y_1 = y_2$ **then**
  $L \leftarrow \max(0, \alpha_1^{\text{old}} + \alpha_2^{\text{old}} - C), \quad H \leftarrow \min(C, \alpha_1^{\text{old}} + \alpha_2^{\text{old}})$
else
  $L \leftarrow \max(0, \alpha_2^{\text{old}} - \alpha_1^{\text{old}}), \quad H \leftarrow \min(C, C + \alpha_2^{\text{old}} - \alpha_1^{\text{old}})$
end if
if $L = H$ **then return** "no changes"
{ Compute updated αs }
$D \leftarrow 2k(\boldsymbol{x}_1, \boldsymbol{x}_2) - k(\boldsymbol{x}_1, \boldsymbol{x}_1) - k(\boldsymbol{x}_2, \boldsymbol{x}_2),$
if $D \geq 0$ **then return** "no changes"
$\alpha_2^{\text{new}'} \leftarrow \alpha_2^{\text{old}} - \frac{y_2(E_1 - E_2)}{D}$
$$\alpha_2^{\text{new}} \leftarrow \begin{cases} H & \text{if } \alpha_2^{\text{new}'} > H \\ L & \text{if } \alpha_2^{\text{new}'} < L \\ \alpha_2^{\text{new}} & \text{otherwise} \end{cases}$$
$\alpha_1^{\text{new}} \leftarrow \alpha_1^{\text{old}} + y_1 y_2 (\alpha_2^{\text{old}} - \alpha_2^{\text{new}}),$
if $|\alpha_2^{\text{old}} - \alpha_2^{\text{new}}| < 10^{-5}$**then return** "no changes"
{ Compute new b }
$b_1 \leftarrow b^{\text{old}} + E_1 + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(\boldsymbol{x}_1, \boldsymbol{x}_1) + y_2(\alpha_2^{\text{new}} - \alpha_2^{\text{old}})k(\boldsymbol{x}_1, \boldsymbol{x}_2)$
$b_2 \leftarrow b^{\text{old}} + E_2 + y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(\boldsymbol{x}_1, \boldsymbol{x}_2) + y_2(\alpha_2^{\text{new}} - \alpha_2^{\text{old}})k(\boldsymbol{x}_2, \boldsymbol{x}_2)$
$$b^{\text{new}} \leftarrow \begin{cases} b_1 & 0 < \alpha_1 < C \\ b_2 & 0 < \alpha_2 < C \\ (b_1 + b_2)/2 & \text{else} \end{cases}$$

Algorithm 14 The SMO algorithm for SVMs with simplified heuristic

Input: Input points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$
Input labels $y_1, \ldots, y_n \in \{\pm 1\}$
Kernel function $k \colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$
Regularization constant $C > 0$
Maximum number of passes $P > 0$
Tolerance level $tol > 0$
Output: Learned parameter vector $\boldsymbol{\alpha}$, and offset b

Initialize all $\alpha_i \leftarrow 0$.
Set $p \leftarrow 0$ {counts passes over the whole datasets without changes}
while $p < P$ **do**
  $a \leftarrow 0$ {counts changed $\boldsymbol{\alpha}$s}
  for $i = 1$ **to** n **do**
    Calculate $E_i = f(\boldsymbol{x}_i) - y_i$
    if $(y_i E_i < -tol$ **and** $\alpha_i < C)$ **or** $(\ y_i E_i > tol$ **and** $\alpha_i > 0)$ **then**
      Select $j \neq i$ at random
      Calculate $E_j = f(\boldsymbol{x}_j) - y_j$
      Compute updated α_i^{new}, α_j^{new}, and b^{new}.
      if successfully updated **then**
        Increment a.
      end if
    end if
  end for
  if $a = 0$ **then**
    Increment p.
  else
    $p = 0$
  end if
end while

The first thing we can quickly see is that we want to minimize the norm of $\boldsymbol{w}$ and the sum of the ξ_is. It can be shown that the norm of $\boldsymbol{w}$ is related to the size of the margin (that is, the amount of separateness of the two classes). Minimizing $\boldsymbol{w}$ amounts to maximizing the margin.

Now, the ξ_i are so called *slack* variables, which are a standard trick in the area of optimization to re-write constraints to fit into formal criteria.

In this case, originally, the optimization problem read

$$\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i f(\boldsymbol{x}_i)). \tag{4.5}$$

The maximum in the second term has already been explained above: it measures whether $f(\boldsymbol{x}_i)$ predicts the correct class by outputting at least 1 (or -1). Smaller (or larger) values are penalized.

While the optimization problem in the last display is mathematically correct, it is not easy to see that it is a quadratic optimization problem in $\boldsymbol{w}$. After all, the maximum is a piecewise linear function. The solution is to translate the maximum term into the actual amount of measured error, and some linear constraints.

The amount of error is given by

$$\xi_i = \max(0, 1 - y_i f(\boldsymbol{x}_i)).$$

This equality constraint is replaced by two inequality constraints

$$y_i f(\boldsymbol{x}_i) \geq 1 - \xi_i, \qquad \xi_i \geq 0,$$

and minimizing over ξ_i. The optimal ξ_i is then given as either the error, or 0, if f predicts correctly.

The final insight is that it is okay to have constraints which depend on other variables you are optimizing. In fact, any linear combination of variables can occur in a constraint.

In summary, the optimization problem (4.4) is actually just a reformulation of the original problem (4.5), which shows that the SVM tries to find a comprimise between the size of $\boldsymbol{w}$ (related to the margin), and the errors ξ_i.

Let us now look how the *dual* optimization problem (4.3) is related to (4.4). In principle, it is possible to directly optimize (4.4), however, it is easier to transform the problem such that (a) only scalar products between points $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ are used, opening the possibility of applying the kernel trick, and such that (b) the constraints in the optimization problem are easier.

This is accomplished by computing the dual of the original optimization problem. This step is closely related to the use of Lagrange multipliers. As you certainly know, you can identify the extremal points of a differential function by finding the roots of the first derivatives. In the presence of constraints, it is no longer that easy since the extrema typically lie on the border of the set of feasible points, where the derivative is typically anything but zero.

In this situation, the use of Lagrange multipliers permits to transform the problem such that one ends up with an optimization problem with simpler constraints. More formally, assume that you have an optimization problem like this[5]:

$$\begin{aligned} \min_{\boldsymbol{x}} \; & f(\boldsymbol{x}) \\ \text{subject to } & g_i(\boldsymbol{x}) \leq 0, \qquad 1 \leq i \leq s, \\ & h_j(\boldsymbol{x}) = 0, \qquad 1 \leq j \leq t. \end{aligned}$$

Adding the constraints using *Lagrangian multipliers* leads to the new cost function

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\boldsymbol{x}) + \sum_{i=1}^{s} \lambda_i g_i(\boldsymbol{x}) + \sum_{j=1}^{t} \mu_j h_j(\boldsymbol{x}).$$

This function is called the *Lagrange dual* function. If f is convex and obeys additional mild conditions, one can show that the *maximum* of the Lagrange dual under the constraints $\lambda_i \geq 0$ is the same as the solution to the original optimization problem.

In summary, the dual problem is given by

$$\begin{aligned} \max_{\boldsymbol{\lambda}, \boldsymbol{\mu}} \; & \mathcal{L}(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{x} \left(f(\boldsymbol{x}) + \sum_{i=1}^{s} \lambda_i g_i(\boldsymbol{x}) + \sum_{j=1}^{t} \mu_j h_j(\boldsymbol{x}) \right) \\ \text{subject to } & \lambda_i \geq 0, \qquad 1 \leq i \leq s. \end{aligned}$$

Often, the dual function is easier to solve than the original function. Intuitively, this is the case since the constraints are easier, and part of the optimization has already been carried out. The remaining step is thus "just" to find the correct Lagrange multipliers.

While the SVM can in practice be trained very efficiently, the same does not hold for model selection. Normally, you have to pick the regularization constant plus any parameters the kernel functions have. While there exist methods to compute the leave-one-out error without performing full re-training, in general one will have to resort to cross-validation.

Given the complexity of writing a large scale SVM solver, several libraries exist, for example libsvm, svmligth, or svmtorch. There exist libraries which provide a standardized front end to all these libraries, for example the shogun toolbox.

4.5 Neural Networks

▷ **Name** Neural Network

▷ **Applications** Classification, Regression

[5]Here, $\boldsymbol{x}$ is not a data point, but denotes the variable that we aim to optimize for – in case of SVMs this will be $\boldsymbol{w}$.

▷ **Method** In 2006, a fast training algorithm for deep nets was developed for the first time (**?**). A few years later it became clear that deep nets with ReLU activations can be trained efficiently with backpropagation (**?**). The ReLU is a particularly simple neuron: it computes

$$\mathrm{ReLU}(\boldsymbol{x})_i = \max(0, \boldsymbol{w}_i^\top \boldsymbol{x} + b_i),$$

which is a linear function with negative outputs clipped to zero and where index i indicates the ith neuron output. A pseudo-code (with dropout, described below) is given in Algorithm 16. A layer in a Neural Network consists of many such neurons and the network of many such layers. For classification tasks, the last layer is a softmax function:

$$\mathrm{softmax}(\boldsymbol{x})_i = \frac{\exp(\boldsymbol{w}_i^\top \boldsymbol{x} + b_i)}{\sum_{i'} \exp(\boldsymbol{w}_{i'}^\top \boldsymbol{x} + b_{i'})},$$

which outputs a value near 1 for its largest input and all outputs sum to one. The pseudo-code for the softmax is given in Algorithm 17 and the whole forward pass is given in Algorithm 15.

Algorithm 15 Neural Network forward pass

Input: Input matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$,
parameters $(\mathbf{W}, \boldsymbol{b})_\ell$ for every layer $\ell = 1, \ldots, L$
Output: Probabilities $\widehat{\boldsymbol{y}}_1, \ldots, \widehat{\boldsymbol{y}}_n \in \mathbb{R}^k$

$\mathbf{Z} \leftarrow \mathbf{X}$
for $\ell = 1$ **to** $L - 1$ **do**
  $\mathbf{Z} \leftarrow \texttt{relu}(\mathbf{Z}, \mathbf{W}_\ell, \boldsymbol{b}_\ell)$ {see Algorithm 16}
end for
$\widehat{\mathbf{Y}} \leftarrow \texttt{softmax}(\mathbf{Z}, \mathbf{W}_L, \boldsymbol{b}_L)$ {see Algorithm 17}

Algorithm 16 ReLU with dropout

Input: Input/feature matrix $\mathbf{Z} \in \mathbb{R}^{n \times \text{in}}$,
parameters $\mathbf{W} \in \mathbb{R}^{\text{in} \times \text{out}}$ and $\boldsymbol{b} \in \mathbb{R}^{\text{out}}$
dropout rate $p \in (0, 1)$
Output: New feature matrix $\mathbf{Z} \in \mathbb{R}^{n \times \text{out}}$

if currently training **then**
  Sample $\boldsymbol{\delta} \in \{0, 1\}^{\text{out}_\ell}$ with Bernoulli sampled $\delta_j \sim B(1 - p)$
  $\mathbf{Z} \leftarrow \boldsymbol{\delta} \odot \max(0, \mathbf{Z}\mathbf{W}_\ell + \boldsymbol{b}_\ell)$ {The $\max(0, \cdot)$ applies element-wise}
else if currently testing **then**
  $\mathbf{Z} \leftarrow \max(0, (1 - p) \cdot \mathbf{Z}\mathbf{W}_\ell + \boldsymbol{b}_\ell)$
end if

The weights and biases are usually initialized randomly and the forward pass does not do anything useful at initialization time. However, the gradient of the loss function

Algorithm 17 Softmax layer

Input: Feature matrix $\mathbf{Z}$
parameters $\mathbf{W}$ and $\boldsymbol{b}$
Output: Predicted probabilities $\widehat{\mathbf{Y}} \in (0,1)^{n \times k}$
1: $\mathbf{Z} \leftarrow \mathbf{Z}\mathbf{W}_L + \boldsymbol{b}_L$
2: $\widehat{\mathbf{Y}} \leftarrow \exp(\mathbf{Z}) / \sum_i \exp(\mathbf{Z}_{:,i})$

provides an error signal that can be used for updating the parameters *a little bit*, and it was empirically shown that (stochastic) gradient descent (SGD) on these networks often yields good solutions. An update step is given in Algorithm 18. For the computation of the gradient, autograd libraries[6] are typically used. Iterating this update procedure (with

Algorithm 18 Neural Network update

Input: Input matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$,
targets $\mathbf{Y} \in \{0,1\}^{n \times k}$,
parameters $(\mathbf{W}, \boldsymbol{b})_\ell$ for every layer $\ell = 1, \ldots, L$,
learning rate $\lambda \in \mathbb{R}^+$,
loss function $\mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}}) \in \mathbb{R}$
Output: Updated parameters $(\mathbf{W}, \boldsymbol{b})_\ell$
1: Compute the prediction $\widehat{\mathbf{Y}} = \texttt{forward}(\mathbf{X})$
2: Compute the gradients $\nabla_{\mathbf{W}_\ell} \mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}})$ and $\nabla_{\boldsymbol{b}_\ell} \mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}})$ for every layer
3: **for** $\ell = L$ **to** 1 **do**
4: $\quad \mathbf{W}_\ell \leftarrow \mathbf{W}_\ell - \eta \cdot \nabla_{\mathbf{W}_\ell} \mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}})$
5: $\quad \boldsymbol{b}_\ell \leftarrow \boldsymbol{b}_\ell - \eta \cdot \nabla_{\boldsymbol{b}_\ell} \mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}})$
6: **end for**

sufficiently small step size η) will usually converge to a good solution.

One issue with Neural Networks is overfitting. Consider minimizing a loss function like cross-entropy,

$$\mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{c=1}^{k} \mathbf{Y}_{ic} \cdot \log(\widehat{\mathbf{Y}}_{ic}),$$

where labels for sample $\boldsymbol{x}_i$ are $\boldsymbol{y}_i \in \{0,1\}^k$ with a *one-hot-encoding*. If the model does exactly what we ask for, it will just remember the training data and have little reason to generalize. We introduce here two methods to avoid this behavior.

- *Dropout (?):* Each neuron activation (except for the last layer) is set to zero with probability p at training time. The idea is to avoid neurons to be too strongly coupled and make the upper layers more robust against random effects in the lower layers. At test time, dropout is turned off. Because the amount of activations in each layer

[6] For example `https://github.com/HIPS/autograd`

will increase now, weights are multiplied by $1 - p$ (the probability of the neuron to be activated during training). Depending on the choice of p, convergence may slow down severely.

- *Weight Decay:* This is just another name for ridge regularization as introduced in section 4.3.1. Because we train the Neural Network with gradient descent, we can add any differentiable regularization term to the loss. For weight decay, a new loss is defined as

$$\widetilde{\mathcal{L}}(\mathbf{Y}, \widehat{\mathbf{Y}}) = \mathcal{L}(\mathbf{Y}, \widehat{\mathbf{Y}}) + \lambda \sum_{\ell=1}^{L} \sum_{i=1}^{\mathrm{in}_\ell} \|\boldsymbol{w}_{\ell,i}\|^2$$

  which adds a squared penalty to the weight vector of every neuron in every layer. Parameter λ has to be chosen carefully as to avoid putting to much emphasis on weight decay.

▷ **Discussion** Neural Networks have been around for quite some time. John von Neumann already described circuits that simulate neural networks from the brain. It was known that these programs can approximate *any* function with arbitrary precision, but it was unclear how to train them on complex problems. The research was almost abandoned in the 90s and 00s in favor of kernel machines. The latter have provable generalization bounds, are easy to optimize and simply worked better at the time. But these desirable properties come at the cost of two problems:

1. The kernel is not learned, but has to be engineered. It usually consists of an engineered feature space in which the problem at hand becomes (almost) linear. This basically means we have to identify the important features by hand, whilst it would be very nice to learn them as well from data.

2. Universal kernels exist, but they are not efficient. The training part is quadratic in the number of samples, $\mathcal{O}(n^2)$. This is bad, but would be okay as long as the prediction function was efficient. However, the number m of support vectors grows to the extreme for complex problems, and the cost of a single prediction is $\mathcal{O}(m)$, which can also be bad.

Neural Networks address exactly these two problems: We learn the features $\Phi(\boldsymbol{x})$ from data and we trade generalization guarantees for a fixed prediction runtime, $\mathcal{O}(1)$.

Bibliography