Machine Learning
A Practical Course

Disclaimer: This is work in progress!

Mikio L. Braun, Paul Buenau

started April 20, 2007,
last change July 26, 2007

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 thing 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 handfull of methods, implement them, and try to make them run. Tweaking parameters, getting your own idea why something work, and then go back to the theory to see whether other people have arrived at the same conclusions as you did.

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

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 artifical intelligence. Is AI (Artifical Intelligence) the same as artifical intelligence? 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

Machine learning and artifical intelligence are quite close in their ultimate goal: Designing algorithms or even machines which exhibit intelligent behavior. But machine learning does so with a twist: by construction general machines which are capable to learn what they should do from examples.

In principle, if the goal is to construct "inteligent" machines (whatever that is supposed to mean), one could approach it by adopting the standard engineering point of view: Study the problem at hand until you have sufficiently understood it in detail and devise a solution to the problem by hand.

Early attempts at machine learning have followed this idea: Understand how the human mind works and then implement the mechanisms in software.

However, in machine learning, the emphasis is slightly different. Instead of being required to understand the underlying problem thoroughly, the goal is "only" to construct an algorithm which can learn to perform the required task if presented with enough examples of what it is supposed to do.

Consider the problem of handwritten character recognition. The classical engineering appraoch would be to devise detectors for each character by hand. A "one" looks like a single vertical dash, "twos" and "fives" look like a snake, et cetera. Then you would tweak your detectors till you get sufficient performance.

The somewhat suprising realization is that the more general and therefore supposedly harder problem of constructing a "general learner" is possible. Take a neural net, or a support vector machine, collect a few hundred examples of each digit, and you can train a system which works well. And if you put some more tweaking into it, it will even work *very* well.

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 Xs and Ys, denoted by X_i, and Y_i.

In this book, n will always denote the total number of examples, such that the example inputs are given by the sequence $X_1, \ldots, X_n$, and the example outputs by $Y_1, \ldots, Y_n$.

2.2.2 Vectorial Data

For several reasons, vectorial data (that is, each 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 $X_1, \ldots, X_n$ can nicely be summarized in a $d \times n$-matrix. Now, there is some ambiguitiy here, as we could also encode the data set as a $n \times d$-matrix (such that the data points become *rows* of this matrix). However, we will stick with the first convention. To remember it, picture vectors as column vectors (as usual). If I have a sequence of those vectors and push them together, I'll get the matrix.

Another nice feature is that this allows to naturally "vectorize" vector-operations notationally. If $\mathbf{X}$ is the data set matrix to $X_1, \ldots, X_n$, and y is another vector, then

$$\mathbf{X}^\top y = (X_1^\top y, \ldots, X_n^\top y).$$

Dropping the index, changing the font to bold we get the operation on all vectors.

2.2.3 The Labels

One generally considers three distinct types of outputs. For *classification*, the output space is discrete, as one wants to predict membership in a number of (finite) classes. For *regression*, one wants to predict a real number. For *multivariate regression* output, one wants to predict a vector.

Distinguishing between only two classes 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 to classes directly, and that there exist effective schemes of combining two-class-classifiers into multi-class-classifiers (More on that in Section 4).

Multivariate regression can be similarily 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 hve advantages, although they are often more complex than the simpler cases. In practice, one has to see which works better (as always).

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 roman letters are usually reserved for matrices (whose entries are then denoted by the lower case letters: $A = (a_{ij})$). Sometimes people also denote vectors by lower case bold letters, although most of the objects are vectors which makes the typing quite unconvenient.

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 $\hat{a}$ means that $\hat{a}$ 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 roman 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 (sorry for that).

These conventions are summarized in this table:

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

2.3.1 Principles of Machine Learning

It's actually much less constructive than you would've think.

[picking hypothesis based on finite data]

[optimizing cost functions]

[maximum likelihood]

[bayesian inference]

Chapter 3

Unsupervised Learning

3.1 Dimension Reduction

Most of the interesting data sets are quite high-dimensional. Images is maybe the most extreme example (every pixel is his own 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 $X_1, \ldots, X_n \in \mathbb{R}^d$
Output: principal values $l_1, \ldots, l_d \in \mathbb{R}$,
principal directions $u_1, \ldots, u_d \in \mathbb{R}^d$

1: Compute mean $\bar{X} \leftarrow \frac{1}{n} \sum_{i=1}^{n} X_i$.
2: Compute scatter matrix

$$\mathbf{S} \leftarrow \sum_{i=1}^{n} (X_i - \bar{X})(X_i - \bar{X})^\top.$$

3: Compute eigenvectors u_i and eigenvalues l_i of S.

Algorithm 1 shows how to compute the principal components. The principal directions u_i (sorted together with l_i such that $l_1 \geq \ldots \geq l_n$) show the directions where the data has maximal variance. The directions are orthogonal (that is, $u_i^\top u_j = 0$ 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 $X_1, \ldots, X_n \in \mathbb{R}^d$,
principal directions $u_1, \ldots, u_m \in \mathbb{R}^d$
Output: transformed data points $X'_1, \ldots, X'_n \in \mathbb{R}^m$.

1: **for** $i \to 1$ to n **do**
2: $\quad X'_i \leftarrow (u_1^\top X_i, \ldots, u_m^\top X_i)^\top$
3: **end for**

Using the principal directions u_i, we can compute the low-dimensional representations of the data points X_i as shown in Algorithm 2.

Algorithm 3 Reconstructing projected data points in the original space

Input: transformed data points $X'_1, \ldots, X'_n \in \mathbb{R}^d$,
Output: reconstructed data points $X''_1, \ldots, X''_n \in \mathbb{R}^m$.

1: **for** $i \to 1$ to n **do**
2: $\quad X''_i \leftarrow \sum_{j=1}^{m} u_j [X'_i]_j$
3: **end for**

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 X''_i (Algorithm 3) have minimal distortion to the original points X_i. The distortion used here is the usual vector norm (squared). Therefore, PCA makes sense whenever the

usual (Euclidean) vector norm makes sense. This problem leads to an eigenvalue problem which can be solved on $O(n^3)$.

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 X_i are centered. Getting the projected coefficient for a one-dimensional direction can be computed by taking the scalar product with a vector u of lenght 1 which spans this subspace: $X^\top u$. We wish to minimize the reconstruction error, or alternatively, to capture as much variance along the first direction. Thus, we wish to compute maximize

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

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

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

since u has length 1.

The number of principal components m used in the denoising has to be supplied by the user. Solving this issue is not trivial. Irrespective of the number of dimensions, PCA will find the optimal (with respect to the Euclidean norm) subspace. Thus, fewer dimensions means larger error, and more dimensions means more errors. By the way, the reconstruction error can be quickly read of as the sum of the remaining principal values $l_{m+1}, \ldots, l_d$

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 $l_{m+1} + \ldots + l_d$ 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), LLE, 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 $x_1, \ldots, x_n \in \mathbb{R}^d$, number of dimensions $d' \leq d$ for the embedding, parameter k or ϵ for graph construction

Output: embedded data $y_1, \ldots, y_n \in \mathbb{R}^{d'}$

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

$$D_{ij} \leftarrow \|x_i - x_j\|_{L_2}.$$

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

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

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

5: $A \leftarrow -\frac{1}{2} D_g$

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

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

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

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

10: $\quad y_i \leftarrow (\sqrt{\lambda_1} V_{i1}, \sqrt{\lambda_2} V_{i2}, \ldots, \sqrt{\lambda_{d'}} V_{id'})^\top$

11: **end for**

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 LLE, PCA, Kernel PCA

3.1.3 Local 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, PCA, Kernel PCA

Algorithm 5 LLE

Input: data points $x_1, \ldots, x_n \in \mathbb{R}^d$, number of dimensions $d' \leq d$ for the embedding, parameter k or ϵ for constructing the neighbourhoods

Output: embedded data $y_1, \ldots, y_n \in \mathbb{R}^{d'}$

for $i \leftarrow 1$ to n **do**
  Compute indices $\eta_{i1}, \ldots, \eta_{ik_i}$ of the neighbourhood of x_i by the k-nearest-neighbour or ϵ-ball rule.
end for
Initialize matrix of reconstruction weights $W \leftarrow \mathbf{0}_{n \times n}$.
for $i \leftarrow 1$ to n **do**
  Let $C \in \mathbb{R}^{k_i \times k_i}$ be the local covariance matrix.
  for $j \leftarrow 1$ to k_i **do**
    for $l \leftarrow j$ to k_i **do**
      $C_{jl} \leftarrow (x_i - x_{\eta_{ij}})^\top (x_i - x_{\eta_{il}})$
      $C_{lj} \leftarrow C_{jl}$
    end for
  end for
  $w \leftarrow C^{-1} \mathbf{1}_{k_i}$
  $w \leftarrow \frac{1}{w^\top \mathbf{1}_{k_i}} w$
  for $j \leftarrow 1$ to k_i **do**
    $W_{i\eta_{ij}} \leftarrow w_j$
  end for
end for
Let $M \in \mathbb{R}^{n \times n}$ 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**
    $M_{ij} \leftarrow \delta_{ij} - W_{ij} - W_{ji} + \sum_{l=1}^{n} W_{li} W_{lj}$
  end for
end for
Compute matrix $V \in \mathbb{R}^{n \times n}$ of eigenvectors (as columns) and associated eigenvalues $\lambda_1, \ldots, \lambda_n$ of M.
Sort eigenvalues and eigenvectors V such that $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.
for $i \leftarrow 1$ to n **do**
  $y_i \leftarrow (V_{i2}, V_{i3}, \ldots, V_{i(d'+1)})^\top$
end for

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 centres of clusters $\mu_1, \dots \mu_k$ in the data by minimizing the sum of the distances of datapoints to their respective cluster centre. More formally, the objective function can be written as

$$L(\{\mu_1, \dots, \mu_k\}, r) = \sum_{i=1}^{n} \|x_i - \mu_{r_i}\|$$

where r_i is the index of the cluster to which datapoint 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 centre. In the update step, the centres are set to the mean over their members.

Algorithm 6 K-means clustering

Input: data points $x_1, \dots, x_n \in \mathbb{R}^d$, number of clusters k, maximum number of iterations m.

Output: cluster centres $\mu_1, \dots, \mu_k \in \mathbb{R}^d$, assignment vector $r \in \mathbb{R}^n$

Choose random data points as initial cluster centres $\mu_1 \leftarrow x_{i_1}, \dots, \mu_k \leftarrow x_{i_k}$ where $i_j \neq i_l$ for all $j \neq l$.
$r \leftarrow \mathbf{0}_n$
$r' \leftarrow \mathbf{0}_n$
$i \leftarrow 0$
while $i < m$ **do**
  for $j \leftarrow 1$ to n **do**
    Find nearest cluster centre $r'_j \leftarrow \operatorname{argmin}_{1 \leq l \leq k} \|x_j - \mu_l\|_2$
  end for
  for $j \leftarrow 1$ to k **do**
    Compute new cluster centre $\mu_j \leftarrow \frac{1}{|\{l : r'_l = j\}|} \sum_{l : r'_l = j} x_l$
  end for
  if $r = r'$ **then**
    break
  end if
  $r \leftarrow r'$
  $i \leftarrow i + 1$
end while

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 centre 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 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

$$l(\{x_1, \ldots, x_n\}, r) = \sum_{i=1}^{n} \|x_i - \mu_{r_i}\|$$

where $x_1, \ldots, x_n$ are the datapoints, $r \in \mathbb{R}^n$ is the assignment vector and $\mu_1, \ldots, \mu_k$ are the cluster centres. The assignment vector $r \in \mathbb{R}^n$ 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 centre is the mean over its members, i.e.

$$\mu_i = \frac{1}{|\{j : r_j = i\}|} \sum_{j : r_j = i} x_j.$$

Let us introduce one further bit of notation that we will use to formulate the algorithm: if $r \in \mathbb{R}^n$ is an assignment vector, then we will denote by $r^{i=j} \in \mathbb{R}^n$ 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 $x_1, \ldots, x_n \in \mathbb{R}^d$, number of clusters k, assignment vector $r \in \mathbb{R}^n$, clustering cost function $l : \mathbb{R}^{d \times n} \times \mathbb{R}^n \to \mathbb{R}$ where the first argument is the data and the second argument is the assignment vector.

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

1: Compute initial clustering cost $l'_1 \leftarrow l(X, r)$.
2: **for** $i = 1$ to $k - 2$ **do**
3: Compute set C that contains the remaining cluster indices in 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 $l(X, r^{c_1=c_2})$ is minimal among all possible mergers, or formally,

$$(c_1, c_2) = \operatorname*{argmin}_{(c'_1, c'_2) \in C \times C, c'_1 \neq c'_2} l(X, r^{c'_1=c'_2}).$$

5: Store new assignments $r \leftarrow r^{c_1=c_2}$ and $R_{(i+1)j} \leftarrow r_j^{c_1=c_2}$ for all $1 \leq j \leq n$.
6: Compute new clustering cost $l'_{i+1} \leftarrow l(X, r)$.
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 merger 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.

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.

Okay, let's try to be a bit more specific. 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

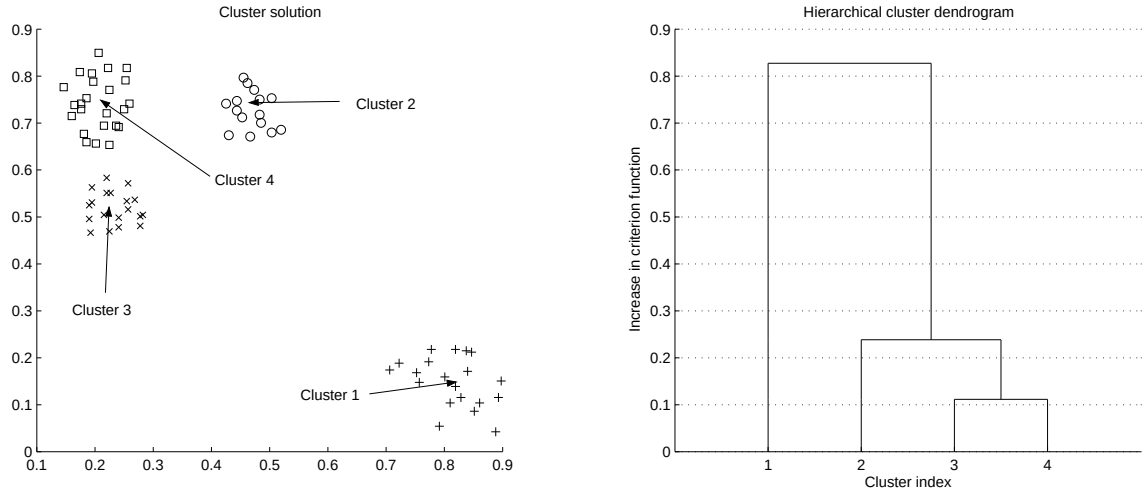

Figure 3.1: Hierarchical cluster solution. The left plot shows the most fine grained cluster structure, the right plot shows the hierarchical structure obtained from agglomorative merging of the four original clusters.

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 $x_1, \ldots, x_n \in \mathbb{R}$ 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 you assume that the observed points are independent, then you get

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

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

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

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

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

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)

$$p(x) = \sum_{k=1}^{K} \pi_k g(x; \mu_k, \Sigma_k), \qquad \sum_{k=1}^{K} \pi_k = 1, \quad \pi_k \geq 0.$$

with

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

In words: the probability for a point 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, μ_k, and Σ_k. It turns out that we cannot simply compute the maximum in this case. Differentiating either of these equations leads to an equation which depends on the other quantities in a non-linear way.

However, it turns out that estimation can be performed in an iterated manner once an intermediate quantity is introduced, which we'll call γ_{nk}. This γ_{nk} is given as the probability that data point n was generated by k. Once these "soft-assignments" of X_i to the data points $X_1, \ldots, X_n$ are fixed, we can estimate $\hat{\pi}_k$, $\hat{\mu}_k$, and $\hat{\Sigma}_k$ 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.

Anyway, the algorithm is summarized in Algorithm 8.

Discussion One important point is the initialization of the algorithm. In Algorithm 8, we simply set $\hat{\pi}_k$ and $\hat{\Sigma}_k$ to sane defaults and assign the cluster centers at random. Other options are possible, for example, setting $\hat{\pi}_k$ to a random distribution (i.e. setting each $\hat{\pi}_k$ 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 to be really sure, you have to make sure that the covariances do not 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_{nk}.

Algorithm 8 The EM Algorithm for Mixture of Gaussians

Input: Data points $X_1, \ldots, X_n \in \mathbb{R}^d$, number of components K
Output: Means $\hat{\mu}_k$ and covariance matrices $\hat{\Sigma}_k$, $1 \leq k \leq K$.

{ Initialize }
$\hat{\pi}_k \leftarrow 1/K$
$\hat{\mu}_k \leftarrow$ random points out of $X_1, \ldots, X_n$
$\hat{\Sigma}_k \leftarrow \mathbf{I}_d$
repeat
  { The "E"-step }
  for $k \leftarrow 1$ to K **do**
    for $n \leftarrow 1$ to N **do**
      Set $\gamma_{nk} \leftarrow \dfrac{\hat{\pi}_k g(X_n; \hat{\mu}_k, \hat{\Sigma}_k)}{\sum_{k'=1}^{K} \hat{\pi}_{k'} g(X_n; \hat{\mu}_{k'}, \hat{\Sigma}_{k'})}$ {see (3.2.3)}
    end for
  end for
  { The "M"-step }
  for $k \leftarrow 1$ to K **do**
    $N_k \leftarrow \sum_{n=1}^{N} \gamma_{nk}$
    $\hat{\pi}_k \leftarrow N_k/N$
    $\hat{\mu}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} X_n$
    $\hat{\Sigma}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (X_n - \hat{\mu}_k)(X_n - \hat{\mu}_k)^\top$
  end for
until "convergence"

See also k-means clustering.

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

Chapter 4

Supervised Learning

4.1 Introduction

Supervised learning is when we have labels. That is, every training example x is accompanied by the true value y that we aim to predict from x. In regression, the truth y is called *target* and is typically real-valued; in classification, our target is to predict discrete class *labels* y. For instance, if we want to build a system for optical character recognition (OCR), each example x is a picture of a handwritten digit and its label $y \in \{0, 1, \ldots, 9\}$ 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 true labels 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 *expected loss* 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 E-mail x is spam ($y = -1$) or ham ($y = +1$). 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.

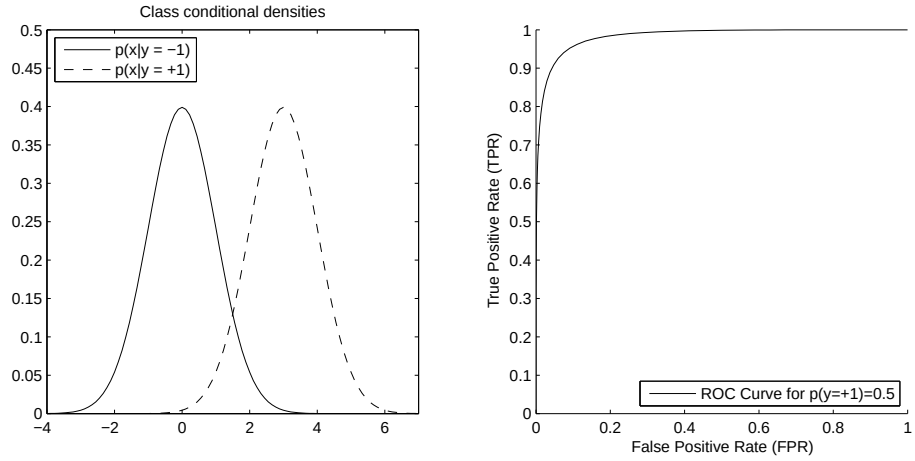

Figure 4.1: The left plot shows the class-conditional densities $p(x|y = +1)$ and $p(x|y = -1)$ for a one-dimensional classification problem. The right plot contains the ROC-curve obtained from varying the decision boundary.

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 known 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 4.2.1.

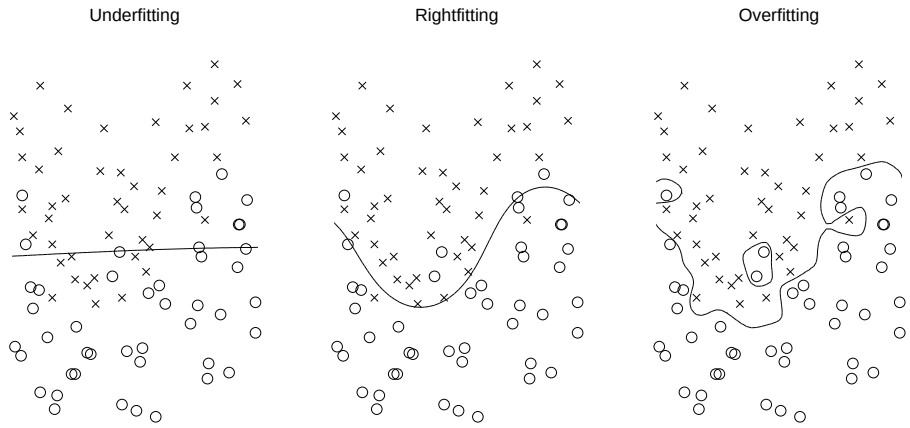

Figure 4.2: From left to right, the plots correspond to an underfitted, rightfitted and overfitted model. The models are kernel ridge regression classifiers with gaussian kernels where the kernel widths are set to $\sigma = 1, 0.05, 0.001$ respectively and fixed $\tau = 1$.

4.2 General Tools

4.2.1 Crossvalidation

In the previous chapter 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).

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 m-fold cross-validation: the data is randomly split into m parts of equal size. For each of the partitions $1 \leq i \leq m$, we train a classifier on the union of all the other partitions and evaluate the performance on partition i. The average missclassification 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 $x_1, \ldots x_n \in \mathbb{R}^d$ with corresponding class labels $y_1, \ldots, y_n \in \{-1, +1\}$. Our aim is to find parameters θ for a function (a.k.a. training a classifier)

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

which predicts the label given the data. The criterion for choosing θ is that the generaliza-

tion 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 9 computes an estimate $\hat{g}(\theta)$ of the generalization error for a parameter θ using r-times m-fold cross-validation.

Algorithm 9 Cross-validation

Input: training data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$, number of partitions m, number of repetitions r, parameter θ
Output: average error rate $\bar{e}$

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

4.3 Classical Methods

4.3.1 Decision Trees

▷ **Name** Decision trees (CART)

Applications Classification, Data Mining

Method The principal idea of decision trees for classification is to partition the input space $\mathbb{R}^d$ into set of regions $R_1, \ldots, R_m$ where each region is associated with a class label that is predicted if an unseen datapoint falls into it. This is more or less true for any classification algorithm, but in the case of decision trees, the regions are defined in a

computationally inexpensive way that lends itself to easy interpretation: the region R_i of a new datapoint x can be determined by simply checking a number of inequalities on its dimensions $x_1, \ldots, x_d$ which corresponds to traversing a binary graph based on the outcome of each test. Thus, each inner node corresponds to a question, that we ask the datapoint (is the value of your d-dimension lower than x_0?) and the leaves (or terminal nodes) represent the partitions. Thus, for each leaf, we have an associated class label.

The objective function to be minimized for such a partition $R_1, \ldots, R_m$ based on the training data is the so-called *impurity* which is a measure of how well the partitioning divides the input space into regions such that the classes do not mix. The latter is obviously desirable if predictions are solely based on membership to partitions. Let us consider the binary case in detail. We are given training data $x_1, \ldots, x_n \in \mathbb{R}^d$ and labels $y_1, \ldots, y_n \in \{-1, +1\}$. The impurity of a region R is defined as

$$I(R) = -p_1(R)\log(p_1(R)) - (1 - p_1(R))\log(1 - p_1(R))$$

where $p_1(R)$ is the ratio of training points from class $+1$ falling into R, that is

$$p_1(R) = \frac{|\{x_i \in R : y_i = +1 \wedge 1 \leq i \leq n\}|}{|R|}$$

and a training point x_i is a member of region R_j if and only if

$$x_i \in R_j \iff \left[\left(\bigwedge_{k=1}^{n_j^1} x_{d'_{jk}} \leq c_{jk}\right) \wedge \left(\bigwedge_{k=n_j^1+1}^{n_j^2} x_{d'_{jk}} \geq c_{jk}\right)\right] \tag{4.1}$$

where $1 \leq d'_{jk} \leq d$ for $k = 1, \ldots, n_j^2$.

During training we try to find a partitioning that minimizes the following cost function L, which is the total weighted impurity plus a regularization term,

$$L(R_1, \ldots, R_m) = \left(\sum_{i=1}^{m} |R_i| I(R_i)\right) + \lambda m.$$

The parameter λ controls the trade off between impurity and complexity. The minimization of L is accomplished in a two-stage process: first, we build a partitioning by repeated splitting that best minimizes the weighted impurity under some mild constraints. Then, we prune this partitioning back by step-wise merging of those nodes, that least increase the impurity (thereby reducing λm) until we have found a minimum of L.

The first stage corresponds to Algorithm 10, the pruning is carried out by Algorithm 11. Formally, the representation of the decision tree is as follows. Let $T = \{t_1, \ldots, t_r\}$ be the set of nodes of the tree. Each leaf corresponds to a partition of the data and each inner-node represents a question that we ask in order to determine the partition of a datapoint.

For each leaf of the tree, we have the following functions.

- label : $T \to \{-1, +1\}$ is the label.

- data : $T \to \mathcal{P}(\mathbb{R}^d)$ is the training data associated with this leaf.

And for every inner node (i.e. non-leaf), we have the following functions.

- dim : $T \to \mathbb{N}$ returns the dimension on which we condition.
- left : $T \to T$ returns the left child node.
- right : $T \to T$ returns the right child node.
- cutpoint : $T \to \mathbb{R}$ returns the cutpoint x_0.

For instance, if we want to determine the partition (and thus predict the label) of a datapoint x, we start traversing a given tree T from its root node. Let $t \in T$ be the current node. If t is a leaf, then t corresponds to the partition of x and label(t) is our prediction of the label. Otherwise, we check whether the value of dimension dim(t) of x is smaller than cutpoint(t). If so, then we continue traversing from its left child $t = \text{left}(t)$, otherwise, we follow the right branch $t = \text{right}(t)$.

Discussion

4.3.2 Ordinary Least Squares and Ridge Regression

▷ **Name** Ordinary Least Squares

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 Xs and Ys is not very complicated, but "just" linear. This means that

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

for some weights w_i. (For simplicity, we have assumed that there is no offset $\langle w, x \rangle + b$. On an actual data set this can be accomplished by centering X and Y first.)

Now, we wish to infer w from a finite set of examples: vectors $X_1, \ldots, X_n$ (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 w such that the equation above is satisfied exactly.

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

Algorithm 10 Build full CART tree

Input: data points $x_1, \ldots, x_n \in \mathbb{R}^d$, labels $y_1, \ldots, y_n \in \{-1, +1\}$, maximum depth of the decision tree m, minimum number l of datapoints in a partitioning

Output: set of nodes T, root node $t_0 \in T$.

Let $T = \{t_0\}$ be a tree with a single leaf where $\text{data}(t_0) \leftarrow \{x_1, \ldots, x_n\}$ and $\text{label}(t_0)$ is the label that appears most often in the whole training set.
for $i = 1$ to m **do**
  Let $t' \leftarrow \perp$
  Let $I' \leftarrow 0$
  for each leaf node $t \in T$ **do**
    if $|\,\text{data}(t)| \geq l$ and $p_1(\text{data}(t)) \notin \{0, 1\}$ **then**
      for $j = 1$ to d **do**
        for each possible cutpoint c between two neighbouring datapoints along dimension j **do**
          Let $\hat{L} \leftarrow \{x_k \in \text{data}(t) : x_{kj} \leq c\}$ be the training data in t that lies to the left of the cutpoint c
          Let $\hat{I} \leftarrow |\,\text{data}(t)|\, I(\text{data}(t)) - \Big(|\hat{L}|\, I(\hat{L}) + (|\,\text{data}(t)| - |\hat{L}|)\, I(\text{data}(t) \setminus \hat{L})\Big)$ be the reduction in impurity resulting from cutting at c along dimension j
          if $\hat{I} > I'$ **then**
            Assign new greatest reduction $I' \leftarrow \hat{I}$
            Store parameters of the split $t' \leftarrow t$, $d' \leftarrow j$, $c' \leftarrow c$
          end if
        end for
      end for
    end if
  end for
  if $t' = \perp$ **then**
    break
  else
    Create two new leaf nodes t_L and t_R and set $\text{left}(t') \leftarrow t_L$, $\text{right}(t') \leftarrow t_R$. Note that t' is now an inner node.
    Let $\text{cutpoint}(t') \leftarrow c'$
    Let $\text{data}(t_L) \leftarrow \{x_k \in \text{data}(t') : x_{kd'} \leq c'\}$
    Let $\text{data}(t_R) \leftarrow \text{data}(t') \setminus \text{data}(t_L)$
    Let $\text{label}(t_L)$ and $\text{label}(t_R)$ be the label that occurs most often in $\text{data}(t_L)$ and $\text{data}(t_R)$ respectively
    Add the new nodes to the tree $T \leftarrow T \cup \{t_L, t_R\}$
  end if
end for

Algorithm 11 Prune CART tree

Input: tree T, root node $t_0 \in T$, regularization parameter λ
Output: pruned tree T, root node $t_0 \in T$

while $|T| > 1$ **do**
  Compute loss $L' \leftarrow \sum_{\text{leaf } l' \in T} (|\operatorname{data}(l')|\, I(\operatorname{data}(l')) + \lambda)$
  $T' \leftarrow \perp$
  for each inner node $t \in T$ **do**
    Let $\hat{T} \leftarrow T$ and let $\hat{t} \in \hat{T}$ be the node corresponding to $t \in T$
    Merge the data at node $\hat{t} \in \hat{T}$,

$$\operatorname{data}(\hat{t}) \leftarrow \operatorname{data}(\operatorname{left}(\hat{t})) \cup \operatorname{data}(\operatorname{right}(\hat{t})),$$

    make it a leaf node and remove all its former children from $\hat{T}$.
    Compute loss $\hat{L} \leftarrow \sum_{\text{leaf } l' \in \hat{T}} (|\operatorname{data}(l')|\, I(\operatorname{data}(l')) + \lambda)$
    if $\hat{L} < L'$ **then**
      Assign new minimal loss $L' \leftarrow \hat{L}$
      $T' \leftarrow \hat{T}$
    end if
  end for
  if $T' \neq \perp$ **then**
    $T \leftarrow T'$
  else
    break
  end if
end while

Instead, we want to find a w such that the squared error between the predicted Ys and the actual Ys is minimal:

$$\min_w \frac{1}{n} \sum_{i=1}^{n} (Y_i - \langle w, X_i \rangle)^2 . \tag{4.2}$$

This means that we do not look for a 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 (4.2) more compactly using matrix notation. Recall that $\mathbf{X}$ is the $d \times n$ matrix whose columns are the X_i, and Y the vector whose entries are the labels Y_i.1 Then,

$$\sum_{i=1}^{n} (Y_i - \langle w, X_i \rangle)^2 = \|Y - \mathbf{X}^\top w\|^2.$$

Taking gradients w.r.t. w, we obtain

$$\partial_w \|Y - \mathbf{X}^\top w\|^2 = \partial_w (Y^\top Y - 2Y^\top \mathbf{X}^\top w + w^\top \mathbf{X}\mathbf{X}^\top w) = -2\mathbf{X}Y + 2\mathbf{X}\mathbf{X}^\top w \stackrel{!}{=} 0,$$

or

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

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

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

(The matrix $(\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{X}$ is also known as the pseudo-inverse.)

Algorithm 12 Ordinary Least Squares

Input: Input features $X_1, \ldots, X_n \in \mathbb{R}^d$
Output labels $Y_1, \ldots, Y_n$.
Output: Output weight vector w
1: Center X and Y if not already the case.
2: Set $\mathbf{X} \leftarrow (X_1, \ldots, X_n)$. {Store X_i in columns of $\mathbf{X}$}
3: Set $Y \leftarrow (Y_1, \ldots, Y_n)^\top$.
4: $w \leftarrow (\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{X}Y$.

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 $E(X - \alpha)^2$ is given by $\alpha = EX$. 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 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

$$\mathbf{X} = \begin{pmatrix} 1 & \cdots & 1 \\ X_1 & \cdots & X_n \\ X_1^2 & \cdots & X_n^2 \\ X_1^3 & \cdots & X_n^3 \end{pmatrix}$$

and is sometimes called the *design matrix*.

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

The matrix $\mathbf{X}\mathbf{X}^\top$ has been introduced as the covariance matrix (see page 11). 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 w, and thereby also the amount of fluctuation possible. Computationally, this is done by adding a regularization term to the original cost-function:

$$\min_w \|Y - \mathbf{X}^\top w\|^2 + C\|w\|.$$

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 w.

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

$$\hat{w} = (\mathbf{X}\mathbf{X}^\top + C\mathbf{I})^{-1}\mathbf{X}Y.$$

The term $C\mathbf{I}$ "pushes" the eigenvalues of $\mathbf{X}\mathbf{X}^\top$ 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 $x_1, \ldots, x_n$, the matrix $\mathbf{K} = (k(x_i, x_j))_{i,j=1}^n$ has to be positive definite, just as would be the case if $k(x, z) = \langle x, z\rangle$.

It can be shown that in this case, k can be interpreted as a scalar-product on transformed features $\Phi(x)$:

$$k(x, z) = \langle \Phi(x), \Phi(z)\rangle.$$

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

Then,

$$f(x) = \langle w, x\rangle = \langle \sum_{i=1}^{n} \alpha_i X_i, x\rangle = \sum_{i=1}^{n} \alpha_i \langle X_i, x\rangle \rightsquigarrow \sum_{i=1}^{n} \alpha_i k(X_i, x) = \sum_{i=1}^{n} \alpha_i \langle \Phi(X_i), \Phi(x)\rangle$$

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:

$$k(x, z) = (\langle x, z\rangle + 1)^d$$
$$k(x, z) = \exp\left(-\frac{\|x - z\|^2}{2w}\right)$$

The main problem in the case of kernel methods is proper regularization. For the rbf-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 rbf-kernel efficiently, one should rather not start by computing all n^2 pairwise distances in big loops. In interpreted languages like Matlab, such explicit computations are inherently slower than the matrix multiplications (which are usually performed using some highly optimized Toolbox like ATLAS).

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 naive 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 naive 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.

So, after this rather lengthy preamble, let us come to the trick. It is very easy, for two vectors x, and z, making it "vectorized" over a set of vectors is bit harder.

Recall that the squared norm is just the scalar product $\|x\|^2 = \langle x, x\rangle$. Then,

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

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

Then, do figure out how to map the square sums on columns or rows of the scalar-products (or some serious repmat'ing in Matlab), and you're 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 (p. 28).

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

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

$$\begin{aligned} &\min_w \|Y - \mathbf{X}^\top w\|^2 + C\|w\|^2 \\ &\qquad \Downarrow \;\; w = \mathbf{X}\alpha \\ &\min_\alpha \|Y - \mathbf{X}^\top\mathbf{X}\alpha\|^2 + C\alpha^\top\mathbf{X}^\top\mathbf{X}\alpha \\ &\qquad \Downarrow \;\; \mathbf{X}^\top\mathbf{X} = \mathbf{K} \\ &\min_\alpha \|Y - \mathbf{X}\alpha\|^2 + C\alpha^\top\mathbf{K}\alpha \end{aligned}$$

The solution to this optimization problem is surprisingly simple:

$$\hat{\alpha} = (\mathbf{K} + C\mathbf{I})^{-1}Y.$$

Kernel Ridge Regression

Input: Input features $X_1, \ldots, X_n \in \mathbb{R}^d$,
Output labels $Y_1, \ldots, Y_n \in \mathbb{R}$,
Kernel function k.

Output: Weight vector α

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

Discussion Kernel ridge regression may be the kernel methods which can be implemented most easily[2]. The downside is that it does not scale very well: Inversion of the

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

kernel matrix might be practical up to a few thousand data points. In such cases, other algorithms exist (for example, conjugate gradients). KRR also does not produce sparse solutions, all of the α will have non-zero data.

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 $\mathbf{S} = \mathbf{K}(\mathbf{K} + C\mathbf{I})$. Then,

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

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

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

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 $\mathbf{U}^\top Y$ leads to further speed-up.

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

Large Scale Kernel-Ridge-Regression

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, like all kernel methods, learn a function

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

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 most of the α_i are zero. If a coefficient α_i is zero it means that we do not need X_i for prediction. The remaining data points are called "support vectors", giving the method its name.

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

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

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 α 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 X_i is measured by

$$\max(0, 1 - y_i f(X_i)).$$

Let's take a closer look at this function. If $y_i f(X_i) \geq 1$, then the error is zero. Note that this implies that y_i and $f(X_i)$ have the same sign (recall that in classification y_i is either $+1$ or -1). On the other hand, if $y_i f(X_i) < 1$, the deviation is penalized linearily. In other words, if $y_i = 1$, then $f(X_i)$ 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$^{\text{TM}}$ subpixel smoothing for LCD displays, among other things, homepage `http://research.microsoft.com/~jplatt/`).

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 $\sum_i y_i \alpha_i = 0$. 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

$$\begin{aligned} Y_i f(X_i) &\geq 1 \qquad \text{if } \alpha_i = 0 \\ Y_i f(X_i) &\leq 1 \qquad \text{if } \alpha_i = C \\ Y_i f(X_i) &= 1 \qquad \text{if } 0 < \alpha_i < C. \end{aligned}$$

So, we choose the first coordinate whenever

$$(\alpha_i < C \text{ and } Y_i f(X_i) < 1 - \varepsilon) \text{ or } (\alpha_i > 0 \text{ and } Y_i f(X_i) > 1 + \varepsilon),$$

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.

Algorithm 13 Optimizing for α_1, α_2

Input: Input points X_1, 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 $E_1 = f(X_1) - Y_i$, $E_2 = f(X_2) - Y_2$
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 }
$\kappa \leftarrow 2k(X_1, X_2) - k(X_1, X_1) - k(X_2, X_2),$
if $\kappa \geq 0$ **then return** "no changes"
$\alpha_2^{\text{new}'} \leftarrow \alpha_2^{\text{old}} - \frac{Y_2(E_1 - E_2)}{\kappa}$

$$\alpha_2^{\text{new}} \leftarrow \begin{cases} H & \alpha_2^{\text{new}'} > H \\ L & \alpha_2^{\text{new}'} < L \\ \alpha_2^{\text{new}} & \text{else} \end{cases}$$

$\alpha_1^{\text{new}} \leftarrow \alpha_1^{\text{old}} + Y_1 Y_2 (\alpha_2^{\text{old}} - \alpha_2^{\text{new}}),$
if $|\alpha_2^{\textbf{old}} - \alpha_2^{\textbf{new}}| < 10^{-5}$ **then return** "no changes"
{ Compute new b }
$b_1 \leftarrow b^{\text{old}} + E_1 + Y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(X_1, X_1) + Y_2(\alpha_2^{\text{new}} - \alpha_2^{\text{old}})k(X_1, X_2)$
$b_2 \leftarrow b^{\text{old}} + E_2 + Y_1(\alpha_1^{\text{new}} - \alpha_1^{\text{old}})k(X_1, X_2) + Y_2(\alpha_2^{\text{new}} - \alpha_2^{\text{old}})k(X_2, X_2)$

$$b^{\text{new}} \leftarrow \begin{cases} b_1 & 0 < \alpha_1 < C \\ b_2 & 0 < \alpha_2 < C \\ (b_1 + b_2)/2 & \text{else} \end{cases}$$

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

Algorithm 14 The SMO algorithm for SVMs with simplified heuristic

Input: Input points $X_1, \ldots, X_n$
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 α, 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 αs}
  for $i = 1$ to n **do**
    Calculate $E_i = f(X_i) - Y_i$
    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 $E_j = f(X_j) - Y_j$
      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

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, $k(x, z) = x^\top z$). This is also known as the 1-:

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

The first thing we can quickly see is that we want to minimize the norm of w and the sum of the ξ_is. It can be shown that the norm of w is related to the size of the margin (that is, the amount of separatedness of the two classes). Minimizing 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

$$\min_{w,b} \frac{1}{2} w^\top w + C \sum_{i=1}^{n} \max(0, 1 - Y_i f(X_i)). \tag{4.5}$$

The maximum in the second term has already been explained above: it measures whether $f(X_i)$ 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 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

$$\xi_i = \max(0, 1 - Y_i f(X_i)).$$

This equality constraint is replaced by two inequality constraints

$$Y_i f(X_i) \geq 1 - \xi_i, \qquad \xi_i \geq 0,$$

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 OK 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 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 possible to transform the problem such that (a) only scalar products between points $X_i^\top X_j$ 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:

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

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

$$f(x) + \sum_{i=1}^{s} \lambda_i g_i + \sum_{j=1}^{t} \mu_j h_j.$$

Now, if you take this unconstraint optimization problem and minimize it in x, you obtain a new function f' which depends on λ and μ. 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

$$\begin{aligned} &\max_{\lambda,\mu} f'(\lambda, \mu) = \inf_x \left(f(x) + \sum_{i=1}^{s} \lambda_i g_i + \sum_{j=1}^{t} \mu_j h_j \right) \\ \text{subject to } &\lambda_i \geq 0, \qquad 1 \leq i \leq s. \end{aligned}$$

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 Bayesian Methods

4.5.1 Belief Propagation in Markov Random Fields

Markov Random Fields are a particular class of Graphical Models and Belief Propagation is an algorithm for exact inference in such models. A graphical model is a loosely defined visual notation for specifying the structure of a joint distribution in terms of an annotated graph.

A Markov Random Field (MRF) is composed of a set of latent (unobserved) discrete random variables $X = \{X_1, \ldots, X_n\}$ along with a set of corresponding variables $Y_1, \ldots, Y_n$ for which observations $y_1, \ldots y_n$ are available and a graph $G = (X, E)$ on the latent variables X.

The graphical representation of an MRF is defined as follows: latent variables are depicted by circles, observations as black dots where each latent variable X_i is connected to its corresponding observable Y_i by a line. The edges in G correspond to lines between the latent variables. See Figure 4.3 for an example.

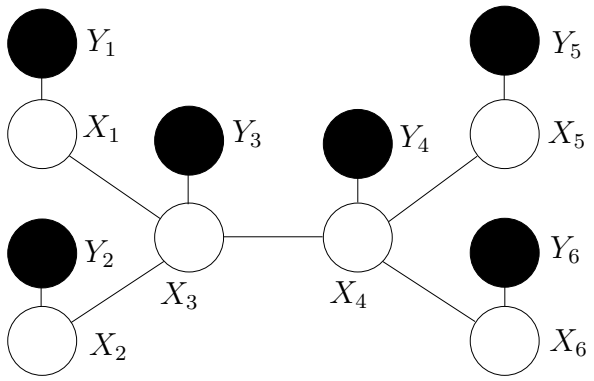

Figure 4.3: Markov Random Field with latent variables $X_1, \ldots, X_5$ and observable variables $Y_1, \ldots Y_5$.

Each line in the graphical representation corresponds to a factor in the joint distribution $p(X_1, \ldots, X_n)$ of the observed variables: The presence of an edge $(i, j) \in E$ between between latent variables X_i and X_j indicates that there is a factor $\psi_{i,j}(X_i, X_j)$ in the joint distribution which is proportional to the compatibility of the values of X_i and X_j. Moreover, the lines connecting latent variables X_i with their associated observable variables Y_i represent the evidence functions $\phi_i(X_i, y_i)$ for which we adopt the shorter notation $\phi_i(X_i)$ since the observation y_i is constant. See Figure 4.4 for an illustration.

The unnormalized joint distribution specified by an MRF is the product of evidence and compatibility functions

$$p(X_1, \ldots, X_n) \propto \prod_{(i,j)\in E} \psi_{ij}(X_i, X_j) \prod_{i=1}^{n} \phi_i(X_i).$$

The evidence factors represent the knowledge that we gain from observing the data $y_1, \ldots, y_n$ while the compatibility functions measure the pair-wise plausibility of the latent variables. Therefore the joint distribution from the MRF in Figure 4.3 is

$$p(X_1, \ldots, X_6) \propto \phi_1(X_1) \cdots \phi_6(X_6)\psi_{1,3}(X_1, X_3) \\ \psi_{2,3}(X_2, X_3)\psi_{3,4}(X_3, X_4)\psi_{4,5}(X_4, X_5)\psi_{4,6}(X_4, X_6).$$

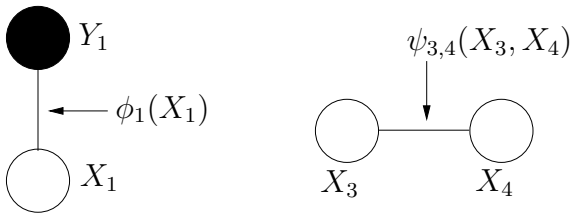

Figure 4.4: Two types of factors in an MRF: compatibility functions $\psi_i j(X_i, X_j)$ and evidence $\phi_i(X_i)$ correspond to lines in the graph.

Inference in a Markov Random Field boils down to computing the marginal distribution $p(X_i)$ for each latent variable given the observations. For instance, if we the latent variable are discrete with z states, $X_i \in \{1, \ldots, z\}$, then by the sum-rule of probability we have the marginal distributions

$$p(X_i = x) \propto \sum_{X_1=1}^{z} \cdots \sum_{X_{i-1}=1}^{z} \sum_{X_{i+1}=1}^{z} \cdots \sum_{X_n=1}^{z} p(X_1, \ldots, X_{i-1}, x, X_{i+1}, \ldots, X_n). \tag{4.6}$$

However, this apparently horrendous to compute since it envolves summing up z^{n-1} evaluations of the joint distribution! This where the Belief Propagation (BP) algorithm comes in. The BP algorithm belongs to the class of message passing algorithms, where a costly computation is broken down into many small problems which together yield the solution by clever exchange of preliminary results (the messages).

Standard Belief Propagation is an efficient algorithm for computing exact marginal distributions for Markov Random Fields whose graph structure is a tree. Loopy belief propagation is an extension of standard BP to arbitrary MRFs. We start off by illustrating the core idea of the Standard BP algorithm on the simple chain MRF depicted in Figure 4.5 where again each discrete latent variable can have z states, $X_i \in \{1, \ldots, z\}$. Imagine

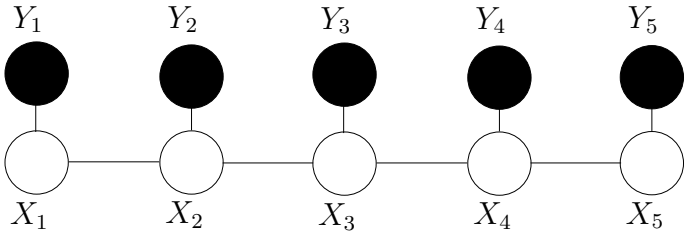

Figure 4.5: Simple chain MRF.

we want to compute the marginal distribution of X_3. Then, by using the sum rule of probability as in Equation (4.6), we get

$$p(X_3 = x) \propto \sum_{X_1=1}^{z} \sum_{X_2=1}^{z} \sum_{X_4=1}^{z} \sum_{X_5=1}^{z} \phi_1(X_1) \cdots \phi_5(X_5) \psi_{1,2}(X_1, X_2) \cdots \psi_{4,5}(X_4, X_5).$$

Note that only $\phi_5(X_5)$ and $\psi_{4,5}(X_4, X_5)$ depend on X_5, thus by the distributive law we can write

$$p(X_3 = x) \propto \sum_{X_1=1}^{z} \sum_{X_2=1}^{z} \sum_{X_4=1}^{z} \Bigg[\phi_1(X_1) \cdots \phi_4(X_4) \psi_{1,2}(X_1, X_2) \cdots \psi_{3,4}(x, X_4) \Bigg[\sum_{X_5=1}^{z} \phi_5(X_5) \psi_{4,5}(X_4, X_5) \Bigg] \Bigg].$$

Applying this exercise again and pulling out the constant $\phi_3(x)$ yields

$$p(X_3 = x) \propto \phi_3(x) \sum_{X_1=1}^{z} \sum_{X_2=1}^{z} \Bigg[\phi_1(X_1) \phi_2(X_2) \psi_{1,2}(X_1, X_2) \psi_{2,3}(X_2, x) \Bigg[\sum_{X_4=1}^{z} \phi_4(X_4) \psi_{3,4}(x, X_4) \Bigg[\sum_{X_5=1}^{z} \phi_5(X_5) \psi_{4,5}(X_4, X_5) \Bigg] \Bigg] \Bigg].$$

Now since the sum over X_4 is independent of X_1 and X_2 we can rewrite the whole summation as a product of two factors. Note that the sum over X_1 decomposes in reverse order,

$$p(X_3 = x) \propto \phi_3(x) \Bigg[\sum_{X_2=1}^{z} \phi_2(X_2) \psi_{2,3}(X_2, x) \Bigg[\sum_{X_1=1}^{z} \phi_1(X_1) \psi_{1,2}(X_1, X_2) \Bigg] \Bigg] \Bigg[\sum_{X_4=1}^{z} \phi_4(X_4) \psi_{3,4}(x, X_4) \Bigg[\sum_{X_5=1}^{z} \phi_5(X_5) \psi_{4,5}(X_4, X_5) \Bigg] \Bigg]. \tag{4.7}$$

The next thing to observe is that the two inner-most sums can be regarded as functions of X_2 and X_4 respectively. Hence we can think of them as z-dimensional vectors. Now if we introduce $(z \times z)$-matrices Ψ_{lm} and z-dimensional vectors Φ_i to represent the compatibility and evidence functions respectively,

$$(\Psi_{lm})_{ij} = \psi_{l,m}(i,j) \qquad\qquad (\Phi_i)_j = \phi_i(X_i = j)$$

we can use matrix notation to rewrite the inner-most sum of the first factor in Equation (4.7) as

$$\sum_{X_1=1}^{z} \phi_1(X_1)\psi_{1,2}(X_1, X_2) = \Psi_{12}^\top \Phi_1.$$

Putting it all together, we can express the z-dimensional vector $(p_3)_i = p(X_3 = i)$ representing the unnormalized marginal distribution of X_3 as in Equation (4.7) by

$$p_3 = \Phi_3 \odot \left[\Psi_{23}^\top \left[\Phi_2 \odot \Psi_{12}^\top \Phi_1\right]\right] \odot \left[\Psi_{34}\left[\Phi_4 \odot \Psi_{45}\Phi_5\right]\right].$$

where $\odot$ denotes element-wise multiplication of vectors. Because the compatiblity relation is symmetric we have $\Psi_{ij} = \Psi_{ji}^\top$ and thus

$$p_3 = \Phi_3 \odot \underbrace{\left[\Psi_{23}^\top \left[\Phi_2 \odot \underbrace{\Psi_{12}^\top \Phi_1}_{=:m_{12}}\right]\right]}_{=:m_{23}} \odot \underbrace{\left[\Psi_{43}^\top \left[\Phi_4 \odot \underbrace{\Psi_{54}^\top \Phi_5}_{=:m_{54}}\right]\right]}_{=:m_{43}}$$

which leads to the following interpretation: the vectors m_{ij} are called *messages* which are sent from node X_i to X_j because the computation of each message m_{ij} only requires local information available at the sending node X_i: the evidence function Φ_i, the compatibility function Ψ_{ij} and all messages received from its neighbours except for X_j. Figure 4.6 depicts the message flow in the graph towards X_3.

More formally, we can write the messages as

$$m_{ij} = \Psi_{ij}^\top \left[\Phi_i \odot \bigodot_{\substack{l:(l,i)\in E,\\ l\neq j}} m_{li}\right] \tag{4.8}$$

and the marginal vectors can be expressed as

$$p_i = \Phi_i \odot \bigodot_{l:(l,i)\in E} m_{li}, \tag{4.9}$$

that is, the evidence at X_i multiplied by all the messages received from its neigbours. The above equations hold for all MRFs that are trees, i.e. where the graph G has no loops.

In most applications, we want to compute all the marginals $p_1, \ldots, p_n$. In order to do that, we see from Equation (4.9) that every node needs to receive all the messages from

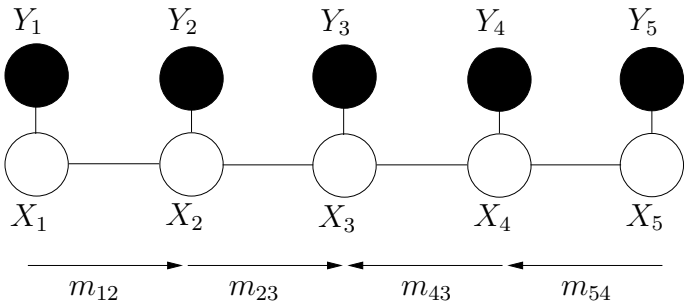

Figure 4.6: Messages flowing towards X_3.

its neighbours. Those can be computed efficiently if we start sending messages from the leaves of the tree (since the leaves are always ready to send their message) and then keep on sending and storing the received messages at each node. In Figure 4.7 you can see the message flow for a more complex graph. The following Algorithm 15 formalizes this idea. The normalization ensures numerical stability for large MRFs.

Algorithm 15 Belief Propagation: Node X_j receives a message from X_i

Input: message m_{ij} received from adjacent node X_i, graph G, evidence Φ_j, compatibility functions Ψ_{jl} for all neighbours l

1: Store received message $M_j \leftarrow M_j \cup m_{ij}$.
2: **for** each neigbour $X_l : (l, j) \in E, l \neq i$ except for X_i **do**
3: **if** all required messages were received, $m_{kj} \in M_j$ for all $(k, j) \in E, k \neq l$ **then**
4: Compute message m_{jl} using Equation 4.8.
5: Normalize message $m_{jl} \leftarrow m_{jl}(\mathbf{1}^\top m_{jl})^{-1}$.
6: Send message m_{jl} to neighbour X_l.
7: **end if**
8: **end for**

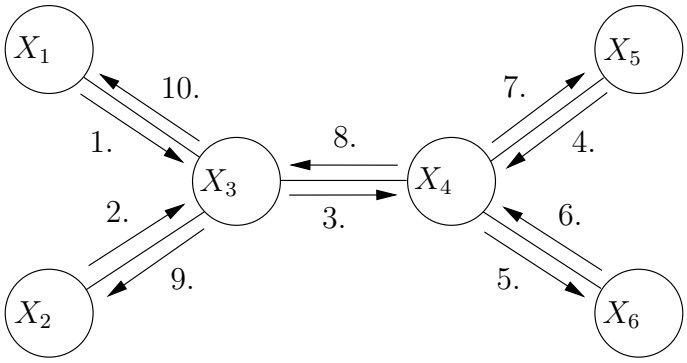

Figure 4.7: Message flow initiated from the leaves. The arrows depict the messages and the numbers next to them indicate the order in which they are sent.

Index