Machine Learning algorithms for Data Mining

Author:
Dr. Luca ONETO

Supervisor:
Prof. Davide ANGUIA
Cosupervisor
Dr. Luca ONETO

November 3, 2011
# Contents

1 Machine Learning .......................................................... 3
   1.1 Learning Process .................................................. 3
   1.2 Learning Taxonomy ................................................ 4

2 Support Vector Machines ............................................... 7
   2.1 Linearly Separable Problems ...................................... 7
      2.1.1 Soft Margin Hyperplanes ................................ 10
   2.2 Non–Linear Support Vector Machines: the Kernel Trick .... 13
      2.2.1 The Kernel Trick ........................................... 13
      2.2.2 The Non–Linear SVM ...................................... 15
      2.2.3 Property of the kernel ................................... 16
   2.3 Extension to Multiclass Classification Problems .............. 17
      2.3.1 All Versus All .............................................. 18
      2.3.2 One Versus All ............................................ 18
      2.3.3 Augmented Binary ........................................ 18

3 Support Vector Regression ............................................... 20
   3.1 Linear Formulation ................................................ 20
   3.2 The KKT Conditions for the SVR ................................ 23
   3.3 Non–Linear Formulation ......................................... 23

4 Decision Tree .............................................................. 25
   4.1 Learning Decision Trees Using ID3 ............................... 28
      4.1.1 Specifying the Problem ................................... 28
      4.1.2 The basic idea .............................................. 28
      4.1.3 Entropy ...................................................... 29
      4.1.4 Information Gain ......................................... 29
      4.1.5 An Example Calculation .................................. 30
      4.1.6 The ID3 algorithm ......................................... 31
   4.2 Avoiding Overfitting .............................................. 31
   4.3 Appropriate Problems for Decision Tree Learning .......... 32

5 Clustering ................................................................. 33
   5.1 Distance measure .................................................. 33
   5.2 k–means clustering ................................................ 34
      5.2.1 Kernel k–means clustering ............................... 34
   5.3 Spectral clustering ................................................ 35

6 Model Selection and Error Estimation ............................... 39
   6.1 Is Model Selection a Problem? .................................. 40
   6.2 Bayesian Model Selection ......................................... 40
   6.3 Frequentist Model Selection ...................................... 41
   6.4 Model Selection and Error Estimation ......................... 41
      6.4.1 k–fold cross validation .................................... 42
      6.4.2 Bootstrap ..................................................... 42
## Contents

7 Practical laboratory sessions 44
  7.1 SVM ........................................................................................................ 44
      7.1.1 Part A - warm up ........................................................................ 44
      7.1.2 Part B - analysis ........................................................................ 45
  7.2 SVR ........................................................................................................ 46
  7.3 Decision Tree ......................................................................................... 47
  7.4 Clustering .............................................................................................. 48
      7.4.1 Kernel k-means ......................................................................... 48
      7.4.2 Spectral Clustering ................................................................... 49
Chapter 1

Machine Learning

Artificial Intelligence was introduced by [1] like *The Science and Engineering of Making Intelligent Machines* (e.g. chess playing).
The concept of computational intelligence born later in the 80s like the union of artificial intelligence and system that take inspiration from the nature. This is most of the time an heuristic approach but not always. In this context we find:

- Neural networks
- Fuzzy system
- Evolutionary Computation
- Swarm Intelligence
- Artificial Immune System

The concept of Machine learning instead born in the 90s like the union of Artificial Intelligence Computational Intelligence and Statistics [2]. In this field born a lot of Algorithms for learning:

- Support Vector Machine
- Decision Trees
- Statistical Pattern Recognition

A particular field of Machine Learning is the Data Mining, means the union of Machine Learning and Big Database.

1.1 Learning Process

The learning process can be view as the process of generating new knowledge. There are three process of learning (Figure 1.1):

- *Induction*: allows inferring b from a, where b does not follow necessarily from a. a might give us very good reason to accept b, but it does not ensure that b. For example, if all of the swans that we have observed so far are white, we may induce that all swans are white. We have good reason to believe the conclusion from the premise, but the truth of the conclusion is not guaranteed. (Indeed, it turns out that some swans are black.)
• **Deduction:** allows deriving \( b \) from \( a \) only where \( b \) is a formal consequence of \( a \). In other words, deduction is the process of deriving the consequences of what is assumed. Given the truth of the assumptions, a valid deduction guarantees the truth of the conclusion. For example, given that all bachelors are unmarried males, and given that this person is a bachelor, it can be deduced that this person is an unmarried male.

• **Transduction:** or transductive inference is reasoning from observed, specific (training) cases to specific (test) cases. In contrast, induction is reasoning from observed training cases to general rules, which are then applied to the test cases. The distinction is most interesting in cases where the predictions of the transductive model are not achievable by any inductive model. Note that this is caused by transductive inference on different test sets producing mutually inconsistent predictions.

• **Abduction:** allows inferring \( a \) as an explanation of \( b \). Because of this, abduction allows the precondition \( a \) to be abducted from the consequence \( b \). Deduction and abduction thus differ in the direction in which a rule like "\( a \) entails \( b \)" is used for inference. As such abduction is formally equivalent to the logical fallacy affirming the consequent or Post hoc ergo propter hoc, because there are multiple possible explanations for \( b \). For example, after glancing up and seeing the eight ball moving towards us we may abduce that it was struck by the cue ball. The cue ball’s strike would account for the eight ball’s movement. It serves as a hypothesis that explains our observation. There are in fact infinitely many possible explanations for the eight ball’s movement, and so our abduction does not leave us certain that the cue ball did in fact strike the eight ball, but our abduction is still useful and can serve to orient us in our surroundings. This process of abduction is an instance of the scientific method. There are infinite possible explanations for any of the physical processes we observe, but we are inclined to abduce a single explanation (or a few explanations) for them in the hopes that we can better orient ourselves in our surroundings and eliminate some of the possibilities.

A typical example of learning is the regression case. The example is show in Figure 1.2.

### 1.2 Learning Taxonomy

The Learning problems can be groped in the following big family:
CHAPTER 1. MACHINE LEARNING

Figure 1.2: Learning example

- **Supervised Learning**: for the learning phase we have \( n \) pattern of \( d \) features. We have the data \( \mathbf{x} \) and the label \( y \):

\[
\{ \mathbf{x}_i, y_i \}_{i=1}^n, \quad \mathbf{x}_i = [x_i^1, \ldots, x_i^d], \quad d = \text{dim} (\mathbf{x}_i)
\]  

In particular the supervised learning can be divided in sub family:

- Binary classification:
  
  \[
y_i \in \{0, 1\}, \quad \text{or,} \quad y_i \in \{-1, 1\}
  \]  

- Multi class classification:
  
  \[
y_i \in \{A, B, C, \ldots\}
  \]  

- Regression
  
  \[
y_i \in \mathbb{R}
  \]  

- One class, Novelty detection
  
  \[
y_i = 0
  \]  

- Ranking, Ordinal Regression
  
  \[
y_i \in \{A, B, C, \ldots\}, \quad A \leq B \leq C \leq \ldots
  \]

- **Unsupervised Learning**: there is only the data available but not the label.

\[
\{ \mathbf{x}_i \}_{i=1}^n
\]

Typical problems of this type are:
- Clustering
- Probability Density Estimation

- Semi-Supervised Learning: We have the data but only for part of the data we have the label. Usually the unlabeled data are much more than the labeled data

\[
\{x_i, y_i\}_{i=1}^{n_1}, \quad \{x_i\}_{i=1}^{n_2}, \quad n_1 << n_2 \tag{1.8}
\]

- Reinforcement Learning: we have the data but the label are available after some time

\[
\{x_i(1), x_i(2), \ldots, x_i(t), y_i\} \tag{1.9}
\]

A typical example is learning to play chess, we play some move \(x_i(\ldots)\) but only in the end we now if this move make me win the game.

- Active Learning: We have the data and we can ask for the label on demand (with a cost)

\[
\{x_i\}_{i=1}^{n}, \quad y_i \text{ on demand} \tag{1.10}
\]
Chapter 2

Support Vector Machines

After the first introductory Chapters, we start taking into account the classification problems and, in particular, one of the most used algorithms: the *Support Vector Machine* (SVM). The SVM is one of the best performing learning machines and rates many stories of successful applications in real-world problems [3]. The method was introduced in [4], but the first exhaustive tractation appears in [5]. The SVM is designed for solving biclass problems; anyhow, there exist many techniques for expanding the possible application of SVM to multiclass problems. In this Chapter, we briefly introduce the main concepts concerning the Support Vector Machines for biclass and multiclass classification. For a complete review, the reader can refer to [6, 7, 8, 9, 10, 2, 11].

2.1 Linearly Separable Problems

Let us consider a simple linearly separable biclass problem, like the one of Fig. 2.1. Let the hypothesis space be a set of linear classifiers:

\[ f(x) = w \cdot x + b. \]

(2.1)

For these data, many possible classifiers, which score no errors on the training data, are possible. Fig. 2.2 shows the classifiers that can be chosen according to the ERM principles; using SRM principles and the radius–margin bound, the only function of Fig. 2.3, characterized by a large margin, is selected. The Support Vector Machine algorithm allows to find, in a practical way, a separating hyperplane characterized by a low complexity (i.e., a large margin) and with a low error rate on the available patterns. The framework of SVM is exactly the same as the radius margin bound. In fact, we suppose that the coefficients

![Figure 2.1: Example of linearly separable set of points.](image-url)
Figure 2.2: Two of the classifiers chosen with ERM principle.

Figure 2.3: The maximum margin hyperplane chosen with SRM and the radius–margin bound.
**CHAPTER 2. SUPPORT VECTOR MACHINES**

\( \mathbf{w} \) and the bias \( b \) are scaled so that there is at least one pattern in the dataset for which

\[
y_j f(x_j) = 1 \tag{2.2}
\]

and, for all the points in the dataset, the following condition holds:

\[
y_i f(x_i) \geq 1. \tag{2.3}
\]

Please note that the previous conditions can be always satisfied in a linearly separable problem since there always exists an appropriate scaling factor for the coefficients.

From a practical point of view, the main drawback of the radius–margin bound is that there is no direct way for controlling the complexity through the tuning of the coefficients. In other word, we would like to express the margin \( M \) as a function of the solution, i.e. \( M(\mathbf{w}, b) \): the problem of finding a simple classifier would turn, in this case, in an optimization problem.

The first step to reach our objective is to write the equations of the two hyperplanes described by the conditions of Eq. (2.2):

\[
\begin{align*}
\mathcal{H}_1 : & \quad \mathbf{w} \cdot \mathbf{x} + b = 1 \\
\mathcal{H}_2 : & \quad \mathbf{w} \cdot \mathbf{x} + b = -1
\end{align*} \tag{2.4}
\]

Now, we can calculate the distance\(^1\) between \( \mathcal{H}_1 \) and \( \mathcal{H}_2 \), which is computed as the distance of each hyperplane from the origin \( \mathcal{O} \):

\[
d^2(\mathcal{H}_1, \mathcal{H}_2) = |d^2(\mathcal{H}_1, \mathcal{O}) - d^2(\mathcal{H}_2, \mathcal{O})| \tag{2.5}
\]

The distance between a point \( \mathbf{x}_p \) and a plane is:

\[
d = \frac{\mathbf{w} \cdot \mathbf{x} + b}{\|\mathbf{w}\|^2} \tag{2.6}
\]

Then we can say that:

\[
d^2(\mathcal{H}_1, \mathcal{H}_2) = \frac{2}{\|\mathbf{w}\|^2} = M(\mathbf{w}, b), \tag{2.7}
\]

where \( d^2(\cdot, \cdot) \) is the squared distance. Eq. (2.5) identifies the function describing the margin, which depends on the coefficients of the hyperplane and the bias. We can now express the optimization problem for finding the maximum margin:

\[
\min_{\mathbf{w}, b} \quad \frac{\|\mathbf{w}\|^2}{2}
\]

\[
y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, \quad \forall i = 1, \ldots, l \tag{2.9}
\]

which is called *primal formulation*, and which is a *Convex Constrained Quadratic Programming (CCQP)* problem.

By introducing \( l \) Lagrange multipliers \( \alpha_i \), we can write the following alternative formulation:

\[
\mathcal{L}_P = \frac{\|\mathbf{w}\|^2}{2} - \sum_{i=1}^{l} \alpha_i [y_i (\mathbf{w} \cdot \mathbf{x}_i + b) - 1], \tag{2.10}
\]

subject to the condition of positiveness of all the Lagrange multipliers (i.e., \( \alpha_i \geq 0 \)). The solution to the problem can be found by solving the following system (*Wolfe dual* [12]):

\[
\begin{align*}
\alpha_i & \geq 0, \quad \forall i = 1, \ldots, l \\
\frac{\partial \mathcal{L}_P}{\partial w_j} & = 0, \quad \forall j = 1, \ldots, n \\
\frac{\partial \mathcal{L}_P}{\partial b} & = 0
\end{align*} \tag{2.11}
\]

\(^1\)If not otherwise indicated, Euclidean distances are always used in this Chapter.
We obtain:

\[ w = \sum_{i=1}^{l} \alpha_i y_i x_i \]  

(2.12)

\[ \sum_{i=1}^{l} y_i \alpha_i = 0. \]  

(2.13)

By plugging the two previous conditions into the primal problem, we obtain the \textit{dual formulation}:

\[
\min_{\alpha} \quad \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i \cdot x_j - \sum_{i=1}^{l} \alpha_i \]

\[ \alpha_i \geq 0, \quad \forall i = 1, \ldots, l \]

\[ \sum_{i=1}^{l} y_i \alpha_i = 0. \]

(2.14)

In general, the solution of the dual problem is \textit{sparse}, in the sense that only a subset of the solution coefficients \( \alpha \) is strictly greater than 0.

Unfortunately, we are still missing a way for calculating the value of the bias \( b \), which does not appear explicitly in the dual formulation. The solution consists in using the \textit{Karush–Kuhn–Tucker} (\textit{KKT}) conditions \cite{13, 14}, which characterize any optimization problem and, for convex optimization formulations like the primal or the dual, give the sufficient and necessary conditions for finding the solution. In particular, one of the KKT conditions is the following:

\[ \alpha_i [y_i (w \cdot x_i + b) - 1] = 0. \]  

(2.15)

If we consider a pattern \( i \), corresponding to a \( \alpha_i > 0 \), it must be:

\[ y_i (w \cdot x_i + b) = 1, \]  

(2.16)

otherwise the KKT condition would be violated. Then, it is sufficient to consider one pattern with \( \alpha_i > 0 \) and to calculate the value of the bias simply using Eq. (2.16). The patterns which correspond to coefficients \( \alpha_i > 0 \) are the only ones involved in the calculation of \( w \) and \( b \) and are called \textit{Support Vectors} (\textit{SVs}), since they are the only ones of support for the final classification. These points are the ones which lie on the margin bounds, from a graphical point of view.

The maximum margin solution is used for classifying new patterns in the so-called \textit{feedforward phase}. The solution hyperplane can be express in terms of the solution coefficients \( \alpha \) by simply using the condition of Eq. (2.12), and we have:

\[ f(x) = \left( \sum_{i=1}^{l} \alpha_i y_i x_i \right) \cdot x + b. \]  

(2.17)

### 2.1.1 Soft Margin Hyperplanes

The SVM formulation for linearly separable problems is quite appealing, since it maps the problem of finding a maximum margin hyperplane in a CCQP problem, which is easy to solve with many optimization tools. Unfortunately, the most of the real–world problems are non–separable ones: in these cases, the SVM, as formulated in Eq. (2.8), is useless, since the contraints of Eq. (2.9) could never be satisfied using a linear classifier. The solution for these kind of problems has been proposed in \cite{5} and simply consists in a relaxation of the \( l \) constraints of Eq. (2.9) by means of \( l \) \textit{slack variables} \( \xi \). In Fig. 2.4, we show a non–separable dataset: the error on the pattern \( x_{\text{out}} \) is measured using a slack variable \( \xi_{\text{out}} \). The constraints, then, become:

\[ y_i (w \cdot x_i + b) \geq 1 - \xi_i, \quad \forall i = 1, \ldots, l \]  

(2.18)

\[ \xi_i \geq 0, \quad \forall i = 1, \ldots, l. \]  

(2.19)
CHAPTER 2. SUPPORT VECTOR MACHINES

Figure 2.4: A non–separable dataset. In particular, one error is scored ($x_{out}$).

With this formulation, for a generic $i$–th pattern, we have three possibilities:

- $\xi_i = 0$: the pattern is correctly classified;
- $0 < \xi_i < 1$: the pattern is correctly classified, but lies inside the margin;
- $\xi_i \geq 1$: the pattern is misclassified. Please note that $\xi_i = 1$ denotes a particular situation: in this case, the pattern lies exactly on the separating hyperplane.

If we simply replaced the constraint in the primal formulation of Eq. (2.8) with the ones of Eqns. (2.18) and (2.19), we would get an underfitting algorithm: in fact, the margin would get maximized regardless of the number of errors scored on the dataset. Anyhow, if the $i$–th pattern is misclassified, $\xi_i \geq 1$: then, $\sum_{i=1}^{l} \xi_i$ is an upper bound of the number of errors. Then, we can counterweight the maximization of the margin by trying to minimize the number of errors, i.e.:

\[
\begin{align*}
\min_{w, b, \xi} & \quad \frac{||w||^2}{2} + C \sum_{i=1}^{l} \xi_i \\
y_i (w \cdot x_i + b) & \geq 1 - \xi_i, \quad \forall i = 1, ..., l \\
\xi_i & \geq 0, \quad \forall i = 1, ..., l,
\end{align*}
\]

(2.20)

(2.21)

(2.22)

where $C$ is a hyperparameter which is tuned to balance the contribution of the margin and the error terms. In this sense, it expresses the SRM philosophy, since it allows to find a simple solution (i.e., large margin) with satisfactory performance in terms of error rate (i.e., a low $\sum \xi_i$ value). This SVM formulation is called soft margin primal problem or, briefly, the primal.\footnote{One could think that this definition is ambiguous, since it makes a mess between the linearly separable and the soft margin primal problem. Anyhow, since the former is seldom used in practice, from here further in this work, we will refer to the problem of Eq. (2.20) as the primal of SVM.} We can proceed as we have done for the separable case.
We can write the Lagrangian of the primal by introducing, in this case, two sets of Lagrange multipliers (one per constraint), \( \alpha \) and \( \mu \):

\[
L_P = \frac{\|w\|^2}{2} + C \sum_{i=1}^{l} \xi_i - \sum_{i=1}^{l} \alpha_i \left[ y_i (w \cdot x_i + b) - 1 + \xi_i \right] - \sum_{i=1}^{l} \mu_i \xi_i. \tag{2.23}
\]

We can formulate the KKT conditions for the Wolfe dual problem, consisting in the derivatives of \( L_P \) and the slackness conditions for \( \alpha_i \) and \( \mu_i \):

\[
\frac{\partial L_P}{\partial w_j} = 0 \rightarrow w_j = \sum_{i=1}^{l} \alpha_i y_i (x_i)_j, \quad \forall j = 1, ..., n \tag{2.24}
\]

\[
\frac{\partial L_P}{\partial b} = 0 \rightarrow \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{2.25}
\]

\[
\frac{\partial L_P}{\partial \xi_i} = 0 \rightarrow C - \alpha_i + \mu_i = 0, \quad \forall i = 1, ..., l \tag{2.26}
\]

\[
y_i (w \cdot x_i + b) \geq 1 - \xi_i, \quad \forall i = 1, ..., l \tag{2.27}
\]

\[
\alpha_i \left[ y_i (w \cdot x_i + b) - 1 + \xi_i \right] = 0, \quad \forall i = 1, ..., l \tag{2.28}
\]

\[
\mu_i \xi_i = 0, \quad \forall i = 1, ..., l \tag{2.29}
\]

\[
(C - \alpha_i) \xi_i = 0, \quad \forall i = 1, ..., l \tag{2.30}
\]

\[
\alpha_i, \xi_i, \mu_i \geq 0, \quad \forall i = 1, ..., l \tag{2.31}
\]

The dual\(^3\) of the SVM then becomes:

\[
\min_{\alpha} \quad \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i \cdot x_j - \sum_{i=1}^{l} \alpha_i \sum_{i=1}^{l} y_i \alpha_i = 0, \tag{2.32}
\]

which is the same formulation as Eq. (2.14) except for the upper bound of \( \alpha_i \). Analogously to the linearly separable case, the data corresponding to \( \alpha_i = 0 \) are correctly classified. For what concerns the Support Vectors, i.e. the data corresponding to \( \alpha_i > 0 \) we have to consider some cases:

- \( 0 < \alpha_i < C \): these data lie on the margin bounds and are also known as True Support Vectors (TSVs). In fact, from Eq. (2.28), \( y_i (w \cdot x_i + b) - 1 + \xi_i = 0 \). Moreover, from Eq. (2.26), since \( \alpha_i < C \), we have that \( \mu_i > 0 \). Then, from Eq. (2.29), we have that \( \xi_i = 0 \): this condition holds for data which are correctly classified (not the case, since \( \alpha_i \neq 0 \)) or which lie on the margin bounds. From Eq. (2.28), substituting \( \xi_i = 0 \), it is also possible to obtain the value of the bias \( b \) by considering one TSV.

- \( \alpha_i = C \): these patterns lie inside the margin or are misclassified by the classifier. These data are defined Bound Support Vectors (BSVs). Then, this set of data can be sub–splitted in two parts:

  - A set of samples which lie inside the margin. Among them, there are correctly classified and misclassified patterns;
  - A set of samples, which are misclassified and located outside the margin. For these patterns, \( y_i (w \cdot x_i + b) < -1 \).

The function for the feedforward phase of the soft margin SVM, obviously, is exactly the same as for the linearly separable problems.

\(^3\)Analogously to the primal problem, we will refer to Eq. (2.32) as the dual problem of SVM.
2.2 Non–Linear Support Vector Machines: the Kernel Trick

In the first Section of this Chapter, we analyzed the Support Vector Machine algorithm, starting from a very simple case: two set of data, which are linearly separable, i.e. which can be divided by a linear classifier without making any mistake. The next step consisted in reformulating the SVM so to contemplate possible mistakes: the soft margin SVM allows some patterns to be misclassified, and trades–off the number of errors and the complexity of the hyperplane.

Anyhow, in real world problems, a linear classifier could be characterized by an extremely poor performance. For example, let us introduce the problem of Fig. 2.5: there are two sets of data, which are extremely overlapped. In this case, a linear classifier in the space of the input patterns (i.e., $\mathbb{R}^n$) is not able to divide in an acceptable way the two classes. Now we have two possibilities:

- Switch to a non–linear classifier: the SVM formulation would have to be revised and the SVM problem would not be any more a CCQP problem, which, on the contrary, is one of the most appealing aspects of SVM;
- Map, in some way, the input patterns in a space which is higher dimensional than the original space $\mathbb{R}^n$.

The kernel functions describe the way to perform the mapping of the input data in a new space.

2.2.1 The Kernel Trick

Let us suppose to know a function $\phi(\cdot)$, which is able to map any input pattern $\mathbf{x} \in \mathbb{R}^n$ in a new space $\phi(\mathbf{x}) \in \mathcal{H}$, where $\mathcal{H}$ is characterized by a dimensionality which is higher than $\mathbb{R}^n$ (even potentially infinite) and is called feature space. Summarizing, let us suppose to know a function $\phi(\cdot)$ such that:

$$\phi : \mathbb{R}^n \rightarrow \mathcal{H}. \quad (2.33)$$

If the dimensionality of $\mathcal{H}$ is sufficiently large, a classification through a linear classifier can perform better than in the original space or even make no mistakes. For example, let us suppose to know a mapping $\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ for the problem of Fig. 2.5 such that the patterns, in the feature space $\mathcal{H}$, lay out as in
Figure 2.6: Non–linear mapping through a function $\phi(\cdot)$.

Fig. 2.6: in this feature space, a classification using a linear plane turns out to be particularly effective. Unfortunately, such mappings are often unknown in an explicit form. Anyhow, it is possible to replace $x$ with $\phi(x)$ in the SVM formulation and the SVM framework is maintained. By simply analyzing Eqns. (2.17), (2.20) and (2.32), it is easy to see that there is only presence of dot products between input patterns. We define the following quantity:

$$K(a, b) = \phi(a) \cdot \phi(b),$$

(2.34)

where $K(\cdot, \cdot)$ is called kernel function. This apparently odd definition is the so–called kernel trick, and was first introduced by Aizerman [15]: it represents the core of SVM and of many other learning algorithms, which are called kernel methods. In fact, while $\phi(\cdot)$ is often unknown, many kernel functions can be retrieved in literature and can be used for an implicit mapping of the input patterns into the (unknown, in general) feature space: we only need the input patterns (in the input space) and a kernel function. Obviously, not all the functions are kernel functions. The necessary and sufficient conditions for a function to be a kernel are described by the Mercer’s theorem.

**Theorem 1** A function is a kernel if and only if, for any $g(x)$ such that

$$\int g(x)^2 dx < +\infty$$

then

$$\int K(x_i, x_j)g(x_i)g(x_j)dx_i dx_j \geq 0.$$  

(2.35)

It is particularly hard to verify, sometimes, if a possible kernel function satisfies the Mercer’s theorem: that is why kernel designing is a high interest topic in Machine Learning [10, 16]. The SVM algorithm does not explicitly require that the Mercer’s condition for the kernel function are satisfied; anyhow, if they are not satisfied, some problems during the optimization procedure could arise.
CHAPTER 2. SUPPORT VECTOR MACHINES

<table>
<thead>
<tr>
<th>Kernel Type</th>
<th>Kernel Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$K(x_i, x_j) = x_i \cdot x_j$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$K(x_i, x_j) = \left[\frac{x_i \cdot x_j}{n} + 1\right]^p$</td>
</tr>
<tr>
<td>Gaussian</td>
<td>$K(x_i, x_j) = \exp\left(-\gamma |x_i - x_j|^2\right)$</td>
</tr>
<tr>
<td>Laplacian</td>
<td>$K(x_i, x_j) = \exp\left(-\gamma |x_i - x_j|_1\right)$</td>
</tr>
</tbody>
</table>

Table 2.1: List of the most commonly used kernel functions.

2.2.2 The Non–Linear SVM

We are now ready for obtaining the general formulation of SVM, which makes use of the kernel trick. By simply replacing the dot products with a generic kernel function $K(\cdot, \cdot)$, we have the following:

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^{l} \alpha_i$$

$$0 \leq \alpha_i \leq C, \quad \forall i = 1, \ldots, l$$

$$\sum_{i=1}^{l} y_i \alpha_i = 0.$$  \hspace{1cm} (2.36)

If we define a matrix $Q$ such that its elements:

$$q_{ij} = y_i y_j K(x_i, x_j),$$  \hspace{1cm} (2.37)

we can write the problem of Eq. (2.36) in a more compact way:

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha + r^T \alpha$$

$$0 \leq \alpha_i \leq C, \quad \forall i = 1, \ldots, l$$

$$y^T \alpha = 0,$$  \hspace{1cm} (2.38)

where $r_i = -1, \forall i = 1, \ldots, l$. A list of the most used kernels is presented in Tab. 2.1: the linear kernel allows to obtain the linear formulation of Eq. (2.32), while the other non–linear kernels introduce new hyperparameters, which have to be tuned ($p$ for the polynomial and $\gamma$ for the Gaussian and the Laplacian kernels). The Gaussian and the Laplacian kernels are also known as Radial Basis Function (RBF) kernels. Many classical optimization techniques can be used for solving this problem: one of the most popular is the Sequential Minimal Optimization (SMO) algorithm [17, 18]. SMO simply splits large Quadratic Programming (QP) problems into a series of smallest QP problems, made up by only two patterns, which are then solved analytically. The feedforward function for the classification of a new pattern $x$ has the following formulation:

$$f(x) = \sum_{i=1}^{l} \alpha_i y_i K(x, x_i) + b.$$  \hspace{1cm} (2.41)

Since only SVs are involved in the feedforward phase, we can alternatively write:

$$f(x) = \sum_{i=1}^{l_{SV}} \alpha_i^{SV} y_i^{SV} K(x, x_i^{SV}) + b,$$  \hspace{1cm} (2.42)

where the superscript $SV$ is used to denote an indexing referred to the Support Vectors and $l_{SV}$ is the number of SVs, found during the training phase. The assignment to class ‘+1’ or ‘-1’ are made computing the sign of $f(x)$. If $f(x) = 0$, the pattern lies on the hyperplane and two possible ways for classifying the datum can be used:
• The pattern is assigned randomly to one of the two classes, which can be considered to have equal (or unequal) probability;
• The pattern is not classified but is considered as uncertain. When computing the error rate, in this case, the pattern is usually considered as misclassified.

2.2.3 Property of the kernel

Let’s take two kernel:

\[ K_1, K_2 : X^d, X^d \rightarrow \mathbb{R} \]  

(2.43)

There are some property of this kernel:

1. \( K(u, v) = K_1(u, v) + K_2(u, v) \)
2. \( K(u, v) = cK_1(u, v), c \in \mathbb{R}^+ \)
3. \( K(u, v) = K_1(u, v) + c, c \in \mathbb{R}^+ \)
4. \( K(u, v) = K_1(u, v)K_2(u, v) \)
5. \( K(u, v) = f(u)f(v) \)
6. \( K(u, v) = p(K_1(u, v)), p(.) = \text{polynomial with positive coefficient} \)
7. \( K(u, v) = e^{K_1(u,v)} \)
8. \( K(u, v) = \frac{K_1(u,v)}{\sqrt{K_1(u,v)K_1(u,v)}} \)

Note 1

\[ \frac{K_1(u,v)}{\sqrt{K_1(u,v)K_1(u,v)}} = \frac{\phi(u)\phi(v)}{\|\phi(u)\| \|\phi(v)\|} = \frac{\phi(v)}{\|\phi(u)\| \|\phi(v)\|} \]  

(2.44)

This is a typical normalized kernel.

Decomposition of the Gaussian Kernel

\[ K(u, v) = e^{-\gamma(u-v)^2} = e^{-\gamma(u^2-2uv+v^2)} = e^{-\gamma u^2}e^{-\gamma v^2} \left( 1 + \frac{2\gamma uv}{1!} + \frac{(2\gamma uv)^2}{2!} + \frac{(2\gamma uv)^3}{3!} + \cdots \right) = \]  

(2.45)

\[ e^{-\gamma u^2}e^{-\gamma v^2} \left( 1 \cdot 1 + \sqrt{\frac{2\gamma}{1!}}u \sqrt{\frac{2\gamma}{1!}}v + \sqrt{\frac{(2\gamma)^2}{2!}}u^2v + \sqrt{\frac{(2\gamma)^3}{3!}}u^3v^2 + \cdots \right) = \phi(u)\phi(v) \]  

(2.46)

then:

\[ \phi(u) = e^{-\gamma u^2} \left[ 1, \sqrt{\frac{2\gamma}{1!}}u, \sqrt{\frac{2\gamma}{2!}}u^2, \sqrt{\frac{(2\gamma)^2}{3!}}u^3, \cdots \right] \]  

(2.47)

An thanks to this we can say that the dimension of \( \phi(u) \) is \( \infty \).
Special Kernels

There are some special kernels:

- \( K(u, v) = K_s(u - v) \) Translation invariant (anisotropic stationary).
- \( K(u, v) = K_s(\|u - v\|) \) Translation invariant (isotropic stationary).
- \( K(u, v) = K_s(\|u - v\|) \), if \( \|u - v\| \geq A \Rightarrow K = 0 \) with compact support

Note 2 Building kernel is not a trivial task:

\[
K(u, v) = \frac{1}{2} \max(1 - \|u - v\|, 0)
\]

is a kernel in \( \mathbb{R}^1 \) but not in \( \mathbb{R}^2 \).

Kernel for non numerical data

Example:

\[
K(\text{car}', \text{cat}') = ?
\]

Let us define a \( \phi \) first a similarity function on substring of 2 characters:

\[
\lambda \in (0, 1) \leftrightarrow \text{similarity}
\]

\[
\begin{array}{cccccccc}
\phi(\text{cat}') & \lambda^2 & \lambda^3 & \lambda^2 & 0 & 0 & 0 & 0 & 0 \\
\phi(\text{car}') & \lambda^2 & 0 & 0 & 0 & 0 & \lambda^3 & \lambda^2 & 0 \\
\phi(\text{bat}') & 0 & 0 & \lambda^2 & \lambda^2 & \lambda^3 & 0 & 0 & 0 \\
\phi(\text{bar}') & 0 & 0 & 0 & \lambda^2 & 0 & 0 & \lambda^2 & \lambda^3 \\
\end{array}
\]

Then we can obtain:

\[
K(\text{car}', \text{cat}') = \phi(\text{car}')\phi(\text{cat}') = \lambda^4
\]

2.3 Extension to Multiclass Classification Problems

In the previous Sections, we introduced the linear Support Vector Machine for biclass classification purposes, and we analyzed its extensions to the soft margin and to the non–linear case: the latter allows us to make no mistakes even when the two classes of the dataset are highly overlapped (\( C \rightarrow +\infty \)). Moreover, the SVM training phase consists in a CCQP problem, for which efficient techniques can be retrieved in literature, and is characterized by a sparse solution, a particularly appealing aspect when memory is an issue.

However, many real world problems are characterized by more than two classes: e.g., one of the classic benchmarking datasets for multiclass classification algorithms is iris, in which three types of the iris flower must be distinguished using the dimensions of the petal and the sepal as inputs. While many other learning algorithms can tackle directly with both biclass and multiclass problems (although not having the same qualities as SVM...), the naïve SVM algorithm is designed only for biclass problems.

In this Section, we briefly analyze the most used techniques for using the SVM also in the case of multiclass classification datasets. We will suppose to have a dataset \( X \) with \( k \) classes.
2.3.1 All Versus All

The All vs. All (AVA) method consists in building \( \frac{k(k-1)}{2} \) training sets \( X_{uv} \), each one containing data only from two different classes, \( u \) and \( v \). These sets are used for training \( \frac{k(k-1)}{2} \) different biclass SVMs and the resulting models are saved for the feedforward phase. When a new pattern \( \mathbf{x} \) has to be classified, it is applied in input to all the biclass SVMs and the multiclass label is then found. In the case of conflicts among different classes, a Winner–Take–All (WTA) arbiter is used for assigning the pattern to the final label: the class corresponding to the highest \( f(\mathbf{x}) \) value is considered as the final \( \hat{y} \) of the datum.

Let us make a brief toy example. Let \( X \) be a multiclass dataset, with \( k = 4 \) classes and \( l = 4 \) patterns (one pattern per class). We want to apply the AVA technique to SVM: then, we need to define the different training sets for the biclass SVMs. The total number of sets is:

\[
\frac{k(k-1)}{2} = \frac{4 \cdot 3}{2} = 6.
\]

The sets are: \( X_{12} \) (i.e., class 1 vs. class 2), \( X_{13}, X_{14}, X_{23}, X_{24} \) and \( X_{34} \). Each set in this simple toy is composed by only 2 patterns (one for each class). The training sets are, then, used for building 6 SVMs, which classify new data during the feedforward phase.

The AVA technique is usually characterized by a good performance in terms of error rate, since the granularity in classifying every class versus all the others helps reaching a high accuracy. Anyhow, AVA has two main drawbacks:

- The number of classifiers grows quadratically with the number of classes. This can be an issue in many applications. E.g., in a classic Optical Character Recognition (OCR) problem, the dataset is usually characterized by at least 36 classes (26 letters and 10 digits): in this case, 630 SVMs are necessary;

- The quality of the classification can be poor if, for some classes, few patterns are available: in OCR applications, it could be the case for letters like ‘X’ or ‘J’, which are more infrequent.

2.3.2 One Versus All

The One vs. All (OVA) technique consists in building \( k \) training sets \( X_u \): the patterns of \( X \) referred to class \( u \) are saved in \( X_u \) as ‘+1’ labelled data; the patterns of \( X \), referred to the classes other than \( u \), are assigned a ‘-1’ label. Then, the \( k \) datasets are used for building the biclass SVMs and the coefficients are saved for the feedforward phase. When a new pattern \( \mathbf{x} \) has to be classified, it is applied in input to all the biclass SVMs and the multiclass label is then found; in case of conflicts, a WTA arbiter assigns the label corresponding to the class with maximum value of the SVM feedforward function.

Recalling the toy example of the previous chapter, with OVA we have to create 4 sets, each one consisting of all the 4 original patterns: \( X_1 \) (i.e., class 1 vs. all the other classes), \( X_2, X_3 \) and \( X_4 \). The training sets are, then, used for building 4 SVMs.

With respect to AVA, the OVA technique allows to reduce the number of classifiers (which grows linearly with the number of classes) and circumvents the possible problem of having few data for some classes, since all the patterns in the original \( X \) are used for training all the SVMs. Unfortunately, in many real world problems, the performance (in terms of error rate) of OVA are poorer than AVA. Moreover, each SVM in OVA is trained over all the patterns of the original dataset: it causes a number of SVs per model which is generally higher than in the AVA method. This issue must be taken into account when the available memory is limited.

2.3.3 Augmented Binary

The Augmented Binary (AB) technique has been recently proposed [6] and represents an alternative to OVA e AVA.
Let us consider the dataset $X = \{(x_1, y_1), ..., (x_i, y_i)\}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \{1, 2, ..., k\}$, for $i = 1, ..., l$. We can always replicate $k$ times the $i$–th pattern, augmenting it by a vector $v^j \in \mathbb{R}^k$, $j = 1, ..., k$:

$$
(x_i, y_i) \rightarrow \begin{cases} (x_i | v^1, y^1_i) \\
(x_i | v^2, y^2_i) \\
\vdots \\
(x_i | v^k, y^k_i) \end{cases}
$$

(2.54)

where $v^j$ is such that its $s$–th component, for $s = 1, ..., k$, is

$$(v^j)_s = \begin{cases} +1 & \text{if } s = j \\
-1 & \text{if } s \neq j \end{cases}
$$

(2.55)

and the new target of the patterns is such that

$$y^j_i = \begin{cases} +1 & \text{if } j = y_i \\
-1 & \text{if } j \neq y_i \end{cases}
$$

(2.56)

At the end of the augmenting procedure, a new dataset $X_A \in \mathbb{R}^{kl \times (n+k)}$ is created: this is a biclass dataset, so it can be directly used for training one single conventional SVM. For the feedforward phase, a new pattern $x$ is augmented and replicated with the procedure of Eq. (2.54) and each replicate is classified. The class of the pattern is, then, assigned; in case of conflict, a WTA machine assigns the final label, exploiting the values of the feedforward function of SVM for the replicates of the pattern.

The main advantage of AB consists in the fact that it only necessitates one SVM; the main drawback is that the dimensionality of the training set increases noticeably. Let us recall the example of the OCR application of Section 2.3.1: let us suppose that the number of classes is 36, the number of features is $n = 100$ and the number of available patterns is $l = 1000$. Before the AB procedure, the original dataset is $X \in \mathbb{R}^{1000 \times 100}$; after AB, we have a new dataset $X_A \in \mathbb{R}^{36000 \times 136}$, which is a huge dimensionality. Anyhow, AB has shown to perform very well on microarray data, which are usually characterized by a small number of classes, few patterns available and an extremely large number of features [6].
Chapter 3

Support Vector Regression

The Support Vector Regression (SVR) presents many similarities with the conventional SVM for biclass classification. As we have done with the SVM, we start considering the linear formulation for SVR; then, we will switch to the non–linear one, based (again) on the kernel trick. The interested reader can find more details in [19].

3.1 Linear Formulation

Let us suppose to have a set of points, having almost linear dependency from a free variable: in time series, it could be the time (see Fig. 3.1). We are dealing in this Section with an estimator of a real valued function: in this case, it is wise to define a threshold such that estimations, which fall sufficiently near the true value (i.e., below the threshold), are considered as correct. For example, we can define a $\varepsilon$–tube: if a true value lies inside the tube, described by its estimation, is considered correctly estimated; otherwise, we have an error. As a first step, let us suppose that errors are not admitted ($\varepsilon$ is chosen sufficiently large to embed every pattern in the training set). The primal formulation of the SVR for this problem is the following:

$$
\min_{w,b} \frac{\|w\|^2}{2}
$$

$$
y_i - (w \cdot x_i + b) \leq \varepsilon, \quad \forall i = 1, ..., l
$$

$$
(w \cdot x_i + b) - y_i \leq \varepsilon, \quad \forall i = 1, ..., l
$$

Obviously, $\varepsilon$ is a hyperparameter, which needs to be tuned a priori. Anyhow, this is a very naïve formulation, since enlarging the tube for avoiding errors causes a hard underfitting effect. In a more realistic situation, some errors can be scored by the SVR (e.g., outlier values, i.e. values highly affected by noise). The approach, again, is similar to the classification case (see Fig. 3.2): we introduce some slack variables, which account for possible prediction errors. Obviously, in this case, we have to use $2l$ slack variables, instead of $l$, since we need to take into account both undershooting and overshooting errors. Then, the primal problem for linear Support Vector Regression is the following:

$$
\min_{w,b} \frac{\|w\|^2}{2} + C \sum_{i=1}^{l} (\xi_i + \xi_i^*)
$$

$$
y_i - (w \cdot x_i + b) \leq \varepsilon + \xi_i, \quad \forall i = 1, ..., l
$$

$$
(w \cdot x_i + b) - y_i \leq \varepsilon + \xi_i^*, \quad \forall i = 1, ..., l
$$

$$
\xi_i, \xi_i^* \geq 0, \quad \forall i = 1, ..., l.
$$
Figure 3.1: A time series.

Figure 3.2: A regression error ($y_{out}$) and the associated slack variable $\xi_{out}$. 
The loss function we used in this primal formulation is called $\varepsilon$–insensitive loss, since it derives directly from the definition of the $\varepsilon$–tube:

$$|\xi|_\varepsilon = \begin{cases} 
0 & \text{if } |\xi| \leq \varepsilon \\
|\xi| - \varepsilon & \text{if } |\xi| > \varepsilon
\end{cases}$$ (3.8)

meaning that only errors in prediction greater than $\varepsilon$ are taken into account in the primal problem. We are now interested in finding the dual form for the SVR. In this case, four sets of Lagrange multipliers are necessary: $\alpha$ and $\alpha^*$ are the Lagrange multipliers for the two constraints of Eqns. (3.5) and (3.6); $\eta$ and $\eta^*$ are the Lagrange multipliers for the positivity constraints of Eq. (3.7). The Lagrangian is:

$$L_P = \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l (\alpha_i - \alpha_i^*) (\alpha_j - \alpha_j^*) x_i \cdot x_j + \varepsilon \sum_{i=1}^l (\alpha_i + \alpha_i^*)$$ (3.9)

where the usual positivity constraint for the Lagrange multipliers is required.

We set to zero the partial derivatives of the Lagrangian and we obtain the following:

$$\frac{\partial L_P}{\partial w_j} = 0 \rightarrow w_j - \sum_{i=1}^l (\alpha_i - \alpha_i^*) (x_i)_j = 0$$ (3.10)

$$\frac{\partial L_P}{\partial b} = 0 \rightarrow \sum_{i=1}^l (\alpha_i - \alpha_i^*) = 0$$ (3.11)

$$\frac{\partial L_P}{\partial \xi_i} = 0 \rightarrow C - \alpha_i - \eta_i^* = 0$$ (3.12)

where the superscript ($*$) stands for both the overshooting and the undershooting slack variables and Lagrange multipliers. Substituting these conditions in Eq. (3.9), we obtain the dual formulation for SVR:

$$\min_{\alpha, \alpha^*} \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l (\alpha_i - \alpha_i^*) (\alpha_j - \alpha_j^*) x_i \cdot x_j + \varepsilon \sum_{i=1}^l (\alpha_i + \alpha_i^*)$$ (3.13)

subject to:

$$0 \leq \alpha_i, \alpha_i^* \leq C, \quad \forall i = 1, \ldots, l$$

$$\sum_{i=1}^l (\alpha_i - \alpha_i^*) = 0.$$  

The solution of the dual problem is sparse, in the sense that many ($\alpha_i - \alpha_i^*$) terms vanish. The value of $w$ can be obtained from Eq. (3.10), while the bias can be computed exploiting the KKT conditions (see next Section for more details). Then, the feedforward phase can be computed as:

$$f(x) = \sum_{i=1}^l (\alpha_i - \alpha_i^*) x_i \cdot x + b,$$ (3.14)

or with a sparse notation where only SVs are considered (as we have done for the classification case in Eq. (2.42)).
3.2 The KKT Conditions for the SVR

It is interesting to better analyze the KKT conditions in the case of the Support Vector Regression. Three KKT conditions has been already presented in Eqns. (3.10) – (3.12). The other KKT conditions, which hold for \( i = 1, ..., l \), are:

\[
\begin{align*}
\alpha_i [y_i - (\mathbf{w} \cdot \mathbf{x}_i + b) - \varepsilon - \xi_i] &= 0 \quad (3.15) \\
\alpha_i^* [(\mathbf{w} \cdot \mathbf{x}_i + b) - y_i - \varepsilon - \xi_i^*] &= 0 \quad (3.16) \\
\eta_i \xi_i &= 0 \quad (3.17) \\
\eta_i^* \xi_i^* &= 0 \quad (3.18) \\
(C - \alpha_i) \xi_i &= 0 \quad (3.19) \\
(C - \alpha_i^*) \xi_i^* &= 0 \quad (3.20) \\
\eta_i^{(*)}, \alpha_i^{(*)} &\geq 0. \quad (3.21)
\end{align*}
\]

These conditions give some information about the value of the coefficients \( \alpha^{(*)} \) for the patterns of the training set:

- \( \alpha_i^{(*)} = 0 \): the \( i \)-th pattern lies inside the \( \varepsilon \)-insensitive tube and, then, is considered to be correctly predicted;
- \( 0 < \alpha_i^{(*)} < C \): if one of the Lagrangian multipliers is in the range \((0, C)\), the pattern lies on one of the \( \varepsilon \)-tube boundaries. Let us suppose, for example, that \( \alpha_i \in (0, C) \). From Eq. (3.19), since \( \alpha_i < C \), the corresponding slack \( \xi_i = 0 \). Moreover, from Eq. (3.15):
  \[
y_i - (\mathbf{w} \cdot \mathbf{x}_i + b) = \varepsilon,
\]
  which means that the \( i \)-th pattern is a True Support Vector and can be used for computing the bias \( b \);
- \( \alpha_i^{(*)} = C \): the pattern lies outside the \( \varepsilon \)-tube and, then, is mispredicted.

3.3 Non–Linear Formulation

As in the case of SVM for classification, it is possible to see that, in both the dual formulation of Eq. (3.13) and the feedforward function of Eq. (3.14) of SVR, only dot products between the inputs can be retrieved. Then, by exploiting the kernel trick, it is possible to use kernel functions for performing an implicit non–linear mapping in a higher dimensional space, where the regression through a linear formulation performs better. Then, the dual formulation becomes:

\[
\min_{\alpha, \alpha^*} \quad \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} (\alpha_i - \alpha_i^*) (\alpha_j - \alpha_j^*) K(\mathbf{x}_i, \mathbf{x}_j) + \varepsilon \sum_{i=1}^{l} (\alpha_i + \alpha_i^*)
\]

\[
- \sum_{i=1}^{l} y_i (\alpha_i - \alpha_i^*)
\]

S.T. \( 0 \leq \alpha_i, \alpha_i^* \leq C, \quad \forall i = 1, ..., l \)

\[
\sum_{i=1}^{l} (\alpha_i - \alpha_i^*) = 0
\]

and the feedforward function is:

\[
f(\mathbf{x}) = \sum_{i=1}^{l} (\alpha_i - \alpha_i^*) K(\mathbf{x}_i, \mathbf{x}) + b.
\]
The most frequently used kernel functions are the same presented in Section 2.2.2 (Tab. 2.1) for the SVM. It is interesting to see that the same reliable and well-performing algorithms, used for solving the QP problem of the SVM, can be used also for finding the solution of SVR. We define the following quantities:

\[ Q = \{ q_{ij} \} = \{ K(x_i, x_j) \} \]  (3.25)

\[ \bar{\alpha} = \begin{bmatrix} \alpha \\ \alpha^* \end{bmatrix} \]  (3.26)

\[ \bar{Q} = \begin{bmatrix} Q & -Q \\ -Q & Q \end{bmatrix} \]  (3.27)

\[ \bar{r} = \begin{bmatrix} \varepsilon e - y \\ \varepsilon e + y \end{bmatrix} \]  (3.28)

\[ \bar{y} = \begin{bmatrix} e \\ -e \end{bmatrix} \]  (3.29)

where \( e \) is such that \( e_i = 1 \) for \( i = 1, ..., l \). Now it is possible to write the dual problem in a way which is similar to the dual SVM problem:

\[
\min_{\alpha} \quad \frac{1}{2} \bar{\alpha}^T \bar{Q} \bar{\alpha} + \bar{r}^T \bar{\alpha}
\]

\[ 0 \leq \bar{\alpha}_i \leq C, \quad \forall i = 1, ..., 2l \]

\[ \bar{y}^T \bar{\alpha} = 0, \]

where the only difference with the SVM for classification is that, for SVR, we have to solve a CCQP problem with \( 2l \) variables instead of \( l \).
Chapter 4
Decision Tree

The representation scheme we choose to represent our learned solutions and the way in which we learn those solutions are the most important aspects of a learning method. Now we look at decision trees [20], a simple but powerful representation scheme, and we look at the ID3 method for decision tree learning. Imagine you only ever do four things at the weekend: go shopping, watch a movie, play tennis or just stay in. What you do depends on three things: the weather (windy, rainy or sunny); how much money you have (rich or poor) and whether your parents are visiting. You say to your yourself: if my parents are visiting, we’ll go to the cinema. If they’re not visiting and it’s sunny, then I’ll play tennis, but if it’s windy, and I’m rich, then I’ll go shopping. If they’re not visiting, it’s windy and I’m poor, then I will go to the cinema. If they’re not visiting and it’s rainy, then I’ll stay in.

To remember all this, you draw a flowchart which will enable you to read off your decision. We call such diagrams decision trees. A suitable decision tree for the weekend decision choices would be as Figure 4.1.

We can see why such diagrams are called trees, because, while they are admittedly upside down, they start from a root and have branches leading to leaves (the tips of the graph at the bottom). Note that the leaves are always decisions, and a particular decision might be at the end of multiple branches (for example, we could choose to go to the cinema for two different reasons).

Armed with our decision tree, on Saturday morning, when we wake up, all we need to do is check (a) the weather (b) how much money we have and (c) whether our parent’s car is parked in the drive. The decision tree will then enable us to make our decision. Suppose, for example, that the parents haven’t turned up and the sun is shining. Then this path through our decision tree will tell us what to do (Figure 4.2), hence we run off to play tennis because our decision tree told us to. Note that the decision tree covers all eventualities. That is, there are no values that the weather, the parents turning up or the money situation could take which aren’t catered for in the decision tree. Note that, in this lecture, we will be looking at how to automatically generate decision trees from examples, not at how to turn thought processes into decision trees.

There is a link between decision tree representations and logical representations, which can be exploited to make it easier to understand (read) learned decision trees. If we think about it, every decision tree is actually a disjunction of implications (if ... then statements), and the implications are Horn clauses: a conjunction of literals implying a single literal. In the above tree, we can see this by reading from the root node to each leaf node:

- If the parents are visiting, then go to the cinema
- If the parents are not visiting and it is sunny, then play tennis
- If the parents are not visiting and it is windy and you’re rich, then go shopping
- If the parents are not visiting and it is windy and you’re poor, then go to cinema
- If the parents are not visiting and it is rainy, then stay in.

25
Figure 4.1: A suitable decision tree for the weekend decision choices.
Figure 4.2: Suppose, for example, that the parents haven’t turned up and the sun is shining.
Of course, this is just a re-statement of the original mental decision making process we described. Remember, however, that we will be programming an agent to learn decision trees from example, so this kind of situation will not occur as we will start with only example situations. It will therefore be important for us to be able to read the decision tree the agent suggests.

Decision trees don’t have to be representations of decision making processes, and they can equally apply to categorisation problems. If we phrase the above question slightly differently, we can see this: instead of saying that we wish to represent a decision process for what to do on a weekend, we could ask what kind of weekend this is: is it a weekend where we play tennis, or one where we go shopping, or one where we see a film, or one where we stay in? For another example, we can refer back to the animals example from the last lecture: in that case, we wanted to categorise what class an animal was (mammal, fish, reptile, bird) using physical attributes (whether it lays eggs, number of legs, etc.). This could easily be phrased as a question of learning a decision tree to decide which category a given animal is in, e.g., if it lays eggs and is homeothermic, then it’s a bird, and so on...

### 4.1 Learning Decision Trees Using ID3

#### 4.1.1 Specifying the Problem

We now need to look at how you mentally constructed your decision tree when deciding what to do at the weekend. One way would be to use some background information as axioms and deduce what to do. For example, you might know that your parents really like going to the cinema, and that your parents are in town, so therefore (using something like Modus Ponens) you would decide to go to the cinema.

Another way in which you might have made up your mind was by generalising from previous experiences. Imagine that you remembered all the times when you had a really good weekend. A few weeks back, it was sunny and your parents were not visiting, you played tennis and it was good. A month ago, it was raining and you were penniless, but a trip to the cinema cheered you up. And so on. This information could have guided your decision making, and if this was the case, you would have used an inductive, rather than deductive, method to construct your decision tree. In reality, it’s likely that humans reason to solve decisions using both inductive and deductive processes.

We can state the problem of learning decision trees as follows:

- We have a set of examples correctly categorised into categories (decisions). We also have a set of attributes describing the examples, and each attribute has a finite set of values which it can possibly take. We want to use the examples to learn the structure of a decision tree which can be used to decide the category of an unseen example.

Assuming that there are no inconsistencies in the data (when two examples have exactly the same values for the attributes, but are categorised differently), it is obvious that we can always construct a decision tree to correctly decide for the training cases with 100% accuracy. All we have to do is make sure every situation is catered for down some branch of the decision tree. Of course, 100% accuracy may indicate overfitting.

#### 4.1.2 The basic idea

In the decision tree above, it is significant that the ‘parents visiting’ node came at the top of the tree. We don’t know exactly the reason for this, as we didn’t see the example weekends from which the tree was produced. However, it is likely that the number of weekends the parents visited was relatively high, and every weekend they did visit, there was a trip to the cinema. Suppose, for example, the parents have visited every fortnight for a year, and on each occasion the family visited the cinema. This means that there is no evidence in favour of doing anything other than watching a film when the parents visit. Given that we are learning rules from examples, this means that if the parents visit, the decision is already made.
Hence we can put this at the top of the decision tree, and disregard all the examples where the parents visited when constructing the rest of the tree. Not having to worry about a set of examples will make the construction job easier.

This kind of thinking underlies the ID3 algorithm for learning decision trees, which we will describe more formally below. However, the reasoning is a little more subtle, as (in our example) it would also take into account the examples when the parents did not visit.

4.1.3 Entropy

Putting together a decision tree is all a matter of choosing which attribute to test at each node in the tree. We shall define a measure called information gain which will be used to decide which attribute to test at each node. Information gain is itself calculated using a measure called entropy, which we first define for the case of a binary decision problem and then define for the general case.

Given a binary categorisation, $C$, and a set of examples, $S$, for which the proportion of examples categorised as positive by $C$ is $p^+$ and the proportion of examples categorised as negative by $C$ is $p^-$, then the entropy of $S$ is:

$$
\text{Entropy}(S) = -p^+ \log_2(p^+) - p^- \log_2(p^-)
$$

The reason we defined entropy first for a binary decision problem is because it is easier to get an impression of what it is trying to calculate. Tom Mitchell puts this quite well:

*In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called entropy that characterizes the (im)purity of an arbitrary collection of examples.*

Imagine having a set of boxes with some balls in. If all the balls were in a single box, then this would be nicely ordered, and it would be extremely easy to find a particular ball. If, however, the balls were distributed amongst the boxes, this would not be so nicely ordered, and it might take quite a while to find a particular ball. If we were going to define a measure based on this notion of purity, we would want to be able to calculate a value for each box based on the number of balls in it, then take the sum of these as the overall measure. We would want to reward two situations: nearly empty boxes (very neat), and boxes with nearly all the balls in (also very neat). This is the basis for the general entropy measure, which is defined as follows:

**Definition 1** Given an arbitrary categorisation, $C$ into categories $c_1, \ldots, c_n$, and a set of examples, $S$, for which the proportion of examples in $c_i$ is $p_i$, then the entropy of $S$ is:

$$
\text{Entropy}(S) = \sum_{i=1}^{n} -p_i \log_2(p_i)
$$

This measure satisfies our criteria, because of the $-p \times \log_2(p)$ construction: when $p$ gets close to zero (i.e., the category has only a few examples in it), then the $\log(p)$ becomes a big negative number, but the $p$ part dominates the calculation, so the entropy works out to be nearly zero. Remembering that entropy calculates the disorder in the data, this low score is good, as it reflects our desire to reward categories with few examples in. Similarly, if $p$ gets close to 1 (i.e., the category has most of the examples in), then the $\log_2(p)$ part gets very close to zero, and it is this which dominates the calculation, so the overall value gets close to zero. Hence we see that both when the category is nearly - or completely - empty, or when the category nearly contains - or completely contains - all the examples, the score for the category gets close to zero, which models what we wanted it to. Note that $0 \times \log_2(0)$ is taken to be zero by convention.

4.1.4 Information Gain

We now return to the problem of trying to determine the best attribute to choose for a particular node in a tree. The following measure calculates a numerical value for a given attribute, $A$, with respect to a set of examples, $S$. Note that the values of attribute $A$ will range over a set of possibilities which we call
Values(A), and that, for a particular value from that set, v, we write $S_v$ for the set of examples which have value v for attribute A.

**Definition 2** The information gain of attribute $A$, relative to a collection of examples, $S$, is calculated as:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

The information gain of an attribute can be seen as the expected reduction in entropy caused by knowing the value of attribute A.

### 4.1.5 An Example Calculation

As an example, suppose we are working with a set of examples, $S = \{s1, s2, s3, s4\}$ categorised into a binary categorisation of positives and negatives, such that $s1$ is positive and the rest are negative. Suppose further that we want to calculate the information gain of an attribute, $A$, and that $A$ can take the values $\{v1, v2, v3\}$. Finally, suppose that:

- $s1$ takes value $v2$ for $A$
- $s2$ takes value $v3$ for $A$
- $s3$ takes value $v3$ for $A$
- $s4$ takes value $v1$ for $A$

To work out the information gain for $A$ relative to $S$, we first need to calculate the entropy of $S$. To use our formula for binary categorisations, we need to know the proportion of positives in $S$ and the proportion of negatives. These are given as: $p^+ = 1/4$ and $p^- = 3/4$. So, we can calculate:

$$\text{Entropy}(S) = -(1/4) \log_2(1/4) - (3/4) \log_2(3/4) = -(1/4)(-2) - (3/4)(-0.415) = 0.5 + 0.311 = 0.811$$  \hspace{1cm} (4.4)

Next, we need to calculate the weighted Entropy($S_v$) for each value $v = v1, v2, v3, v4$, noting that the weighting involves multiplying by ($|S_v|/|S|$). Remember also that $S_v$ is the set of examples from $S$ which have value $v$ for attribute $A$. This means that:

$$S_{v1} = \{s4\}, S_{v2} = \{s1, s2\}, S_{v3} = \{s3\}.$$  \hspace{1cm} (4.5)

We now have need to carry out these calculations:

- $(|S_{v1}|/|S|) \times \text{Entropy}(S_{v1}) = (1/4) \times (-0/1) \log_2(0/1) - (1/1) \log_2(1/1)) = (1/4)(0 - 1) = 0$  \hspace{1cm} (4.6)
- $(|S_{v2}|/|S|) \times \text{Entropy}(S_{v2}) = (2/4) \times (-1/2) \log_2(1/2) - (1/2) \log_2(1/2)) = (1/2)(-1/2 - 1/2) = 1/2$  \hspace{1cm} (4.7)
- $(|S_{v3}|/|S|) \times \text{Entropy}(S_{v3}) = (1/4) \times (0/1) \log_2(0/1) - (1/1) \log_2(1/1)) = (1/4)(0 - 1) = 0$  \hspace{1cm} (4.8)

Note that we have taken $0 \log_2(0)$ to be zero, which is standard. In our calculation, we only required $\log_2(1) = 0$ and $\log_2(1/2) = -1$. We now have to add these three values together and take the result from our calculation for Entropy($S$) to give us the final result:

$$\text{Gain}(S, A) = 0.811 - (0 + 1/2 + 0) = 0.311$$  \hspace{1cm} (4.9)

We now look at how information gain can be used in practice in an algorithm to construct decision trees.
4.1.6 The ID3 algorithm

The calculation for information gain is the most difficult part of this algorithm. ID3 performs a search whereby the search states are decision trees and the operator involves adding a node to an existing tree. It uses information gain to measure the attribute to put in each node, and performs a greedy search using this measure of worth. The algorithm goes as follows:

Given a set of examples, $S$, categorised in categories $c_i$, then:

1. Choose the root node to be the attribute, $A$, which scores the highest for information gain relative to $S$.
2. For each value $v$ that $A$ can possibly take, draw a branch from the node.
3. For each branch from $A$ corresponding to value $v$, calculate $S_v$. Then:
   - If $S_v$ is empty, choose the category $c_{\text{default}}$ which contains the most examples from $S$, and put this as the leaf node category which ends that branch. If $S_v$ contains only examples from a category $c$, then put $c$ as the leaf node category which ends that branch. Otherwise, remove $A$ from the set of attributes which can be put into nodes. Then put a new node in the decision tree, where the new attribute being tested in the node is the one which scores highest for information gain relative to $S_v$ (note: not relative to $S$). This new node starts the cycle again (from 2), with $S$ replaced by $S_v$ in the calculations and the tree gets built iteratively like this. The algorithm terminates either when all the attributes have been exhausted, or the decision tree perfectly classifies the examples.

The diagram of Figure 4.3 should explain the ID3 algorithm further.

4.2 Avoiding Overfitting

As we discussed in the previous lecture, overfitting is a common problem in machine learning. Decision trees suffer from this, because they are trained to stop when they have perfectly classified all the training
data, i.e., each branch is extended just far enough to correctly categorise the examples relevant to that branch. Many approaches to overcoming overfitting in decision trees have been attempted. As summarised by Tom Mitchell, these attempts fit into two types:

- Stop growing the tree before it reaches perfection.
- Allow the tree to fully grow, and then post-prune some of the branches from it.

The second approach has been found to be more successful in practice. Both approaches boil down to the question of determining the correct tree size.

### 4.3 Appropriate Problems for Decision Tree Learning

It is a skilled job in AI to choose exactly the right learning representation or method for a particular learning task. As elaborated by Tom Mitchell, decision tree learning is best suited to problems with these characteristics:

- The background concepts describe the examples in terms of attribute-value pairs, and the values for each attribute range over finitely many fixed possibilities.
- The concept to be learned (Mitchell calls it the target function) has discrete values.
- Disjunctive descriptions might be required in the answer.

In addition to this, decision tree learning is robust to errors in the data. In particular, it will function well in the light of (i) errors in the classification instances provided (ii) errors in the attribute-value pairs provided and (iii) missing values for certain attributes for certain examples.
Chapter 5

Clustering

Cluster analysis or clustering is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis used in many fields, including machine learning, data mining, pattern recognition, image analysis and bioinformatics.

There are many types of clustering: Hierarchical algorithms find successive clusters using previously established clusters. These algorithms usually are either agglomerative (‘bottom-up’) or divisive (‘top-down’). Agglomerative algorithms begin with each element as a separate cluster and merge them into successively larger clusters. Divisive algorithms begin with the whole set and proceed to divide it into successively smaller clusters. Partitional algorithms typically determine all clusters at once, but can also be used as divisive algorithms in the hierarchical clustering. Density-based clustering algorithms are devised to discover arbitrary-shaped clusters. In this approach, a cluster is regarded as a region in which the density of data objects exceeds a threshold. Subspace clustering methods look for clusters that can only be seen in a particular projection (subspace, manifold) of the data. These methods thus can ignore irrelevant attributes. The general problem is also known as Correlation clustering while the special case of axis-parallel subspaces is also known as Two-way clustering, co-clustering or biclustering: in these methods not only the objects are clustered but also the features of the objects, i.e., if the data is represented in a data matrix, the rows and columns are clustered simultaneously. They usually do not however work with arbitrary feature combinations as in general subspace methods. But this special case deserves attention due to its applications in bioinformatics. Many clustering algorithms require the specification of the number of clusters to produce in the input data set, prior to execution of the algorithm.

5.1 Distance measure

An important step in most clustering is to select a distance measure, which will determine how the similarity of two elements is calculated. This will influence the shape of the clusters, as some elements may be close to one another according to one distance and farther away according to another. For example, in a 2-dimensional space, the distance between the point \((x = 1, y = 0)\) and the origin \((x = 0, y = 0)\) is always 1 according to the usual norms, but the distance between the point \((x = 1, y = 1)\) and the origin can be 2, \(\sqrt{2}\) or 1 if you take respectively the 1–norm, 2–norm or infinity-norm distance. Common distance functions:

- The Euclidean distance (also called distance as the crow flies or 2–norm distance). A review of cluster analysis in health psychology research found that the most common distance measure in published studies in that research area is the Euclidean distance or the squared Euclidean distance.

- The Manhattan distance (1–norm)

- The maximum norm (infinity norm)
• The Mahalanobis distance corrects data for different scales and correlations in the variables
• The angle between two vectors can be used as a distance measure when clustering high dimensional data. See Inner product space.
• The Hamming distance measures the minimum number of substitutions required to change one member into another.

Another important distinction is whether the clustering uses symmetric or asymmetric distances. Many of the distance functions listed above have the property that distances are symmetric (the distance from object A to B is the same as the distance from B to A). In many other case this property can not holds any more.

5.2 k–means clustering

The k–means algorithm assigns each point to the cluster whose center (also called centroid) is nearest. The center is the average of all the points in the cluster that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster.

The algorithm steps are:

1. Choose the number of clusters, k.
2. Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.
3. Assign each point to the nearest cluster center, where ”nearest” is defined with respect to one of the distance measures discussed above.
4. Recompute the new cluster centers.
5. Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn’t changed).

The main advantages of this algorithm are its simplicity and speed which allows it to run on large datasets. Its disadvantage is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments (the k-means++ algorithm addresses this problem by seeking to choose better starting clusters). It minimizes intra-cluster variance, but does not ensure that the result has a global minimum of variance. Another disadvantage is the requirement for the concept of a mean to be definable which is not always the case. For such datasets the k-medoids variant is appropriate. An alternative, using a different criterion for which points are best assigned to which centre is k-medians clustering.

5.2.1 Kernel k–means clustering

In the algorithm presented before we can note that we have to compute, in practice, two quantities: the mean of the data of every cluster and the distance between a point and the different center. Then let’s try to put this concept in a more rigorous formulation. Let

\[ X = \{ x_1, \ldots, x_n \} \]  

the data and \( k \leq n \) the number of cluster. Then we denote:

\[ I_j, \quad j \in \{1, \ldots, k\} \]
The set of index that belong to the class \( j \). Then the center of the different cluster are:

\[
c_j = \frac{\sum_{i \in I_j} x_i}{|I_j|}, \quad j \in \{1, \ldots, k\}
\]

(5.3)

where \(|I_j|\) is the cardinality (number of element) of \( I_j \). The last thing that we have to compute is the distance between a new point \( x \) and the center \( c_j \):

\[
D(c_j, x) = ||c_j - x||^2
\]

(5.4)

Now we make explicit the last term:

\[
D(c_j, x) = ||c_j - x||^2 = (c_j - x)^T(c_j - x) = \\
= \left( \frac{\sum_{i \in I_j} x_i}{|I_j|} - x \right)^T \left( \frac{\sum_{i \in I_j} x_i}{|I_j|} - x \right) = \\
= \frac{1}{|I_j|^2} \sum_{u \in I_j} \sum_{v \in I_j} x_u^T x_v - 2 \frac{|I_j|}{|I_j|^2} \sum_{u \in I_j} x_u^T x + x^T x
\]

(5.5)

(5.6)

(5.7)

(5.8)

And now let us project our points \( x \) with an unknown function \( \phi \) in another space \( \phi(x) \). The centers became:

\[
c_j = \frac{\sum_{i \in I_j} \phi(x_i)}{|I_j|}, \quad j \in \{1, \ldots, k\}
\]

(5.9)

And we can not compute \( c_j \) if we do not know \( \phi \). But we do not need this because:

\[
D(c_j, x) = \frac{1}{|I_j|^2} \sum_{u \in I_j} \sum_{v \in I_j} \phi(x_u)^T \phi(x_v) - 2 \frac{|I_j|}{|I_j|^2} \sum_{u \in I_j} \phi(x_u)^T \phi(x) + \phi(x)^T \phi(x) = \\
= \frac{1}{|I_j|^2} \sum_{u \in I_j} \sum_{v \in I_j} K(x_u, x_v) - 2 \frac{|I_j|}{|I_j|^2} \sum_{u \in I_j} K(x_u, x) + K(x, x)
\]

(5.10)

(5.11)

In this way we have the kernel extension of the k–means clustering.

### 5.3 Spectral clustering

A promising alternative that has recently emerged in a number of fields is to use spectral methods for clustering [21, 22, 23]. Here one uses the top eigenvectors of a matrix derived from the distance between points. Such algorithms have been successfully used in many applications including computer vision and machine learning. But despite the empirical success different authors still disagree on exactly which eigenvectors to use and how to derive a cluster from them. Also the analysis of these algorithm has tended to focus on simplified algorithms that only use one eigenvector at the time. Here we report a work that is built upon [23, 24] who analyzed algorithms that uses the k eigenvectors simultaneously and give condition in which algorithm can be expected to do well.

Given a set of points

\[
S = \{s_1, \ldots, s_n\} \in \mathbb{R}^d
\]

(5.12)

that we want to cluster into \( k \) subset:
1. Form the affinity matrix $A \in \mathbb{R}^{n \times n}$ defined by:

$$A_{ij} = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \quad \text{if} \quad i \neq j$$

(5.13)

$$A_{ij} = 0 \quad \text{if} \quad i = j$$

(5.14)

2. Define $D$ to be the diagonal matrix whose $(i,i)$–element is the sum of the $A$’s $i$–th rows and construct the matrix $L = D^{-1/2}AD^{-1/2}$

3. Find the $k$ $(x_1, \ldots, x_k)$ largest eigenvector of $L$ (chosen to be orthogonal in case of repeated eigenvalues) and form the matrix $X = [x_1, \ldots, x_k] \in \mathbb{R}^{n \times k}$ by stacking the eigenvector in columns.

4. Form the matrix $Y$ from $X$ by renormalizing each of $X$’s row to have unit length

$$Y_{i,j} = X_{i,j}/\left(\sum_j X_{ij}^2\right)^{1/2}$$

(5.15)

5. Threating each row of $Y$ ad a point in $\mathbb{R}^k$, cluster them with k–means or any other algorithm.

6. Finally assign the original point $s_i$ to the cluster $j$ if and only if row $i$ of the matrix $Y$ was assigned to cluster $j$

Here the parameter $\sigma$ controls how rapidly the affinity $A_{ij}$ falls off with the distance between $s_i$ and $s_j$. At first sight, this algorithm seems to make little sense. Since we run k–means why we not just apply k–means directly on the data? In the Figure 5.1 show and example. The natural cluster in $\mathbb{R}^2$ do not correspond t convex region and then k–means find the unsatisfactory clustering of Figure 5.2. But when we map the point to $\mathbb{R}^k$ they form a tight cluster of Figure 5.3 for which our methods obtains the good clustering of Figure 5.1.
Figure 5.1: Non convex region

Figure 5.2: k–means clustering
Figure 5.3: Point mapped in $\mathbb{R}^k (Y)$
The problem of learning is often decomposed into the tasks of fitting parameters to some training data, and then selecting the best model using heuristic or principled methods, collectively referred to as model selection methods. Model selection methods range from simple yet powerful crossvalidation based methods to the optimization of cost functions penalized for model complexity, derived from performance bounds or Bayesian priors. Some think that the problem is solved, others that it is not a problem at all! For Bayesian theoreticians, the problem of model selection is circumvented by averaging all models over the posterior distribution. For risk minimization theoreticians (called frequentists by the Bayesians) the problem is solved by minimizing performance bounds. For practitioners, the problem is solved using cross-validation. However, looking more closely, most theoretically grounded methods of solving or circumventing model selection have at least one hyper-parameter left somewhere, which ends up being optimized by cross-validation. Cross-validation seems to be the universally accepted ultimate remedy.

In its broadest sense, model selection designates an ensemble of techniques used to select a model, that best explains some data or phenomena, or best predicts future data, observations or the consequences of actions. This broad definition encompasses both scientific and statistical modeling. In this lectures, we address only the problem of statistical modeling and are mostly concerned with supervised learning from independently and identically distributed (i.i.d.) data. The desired properties of the predictor chosen by a model selection procedure are: good generalization performance, fast training/prediction, and ease of interpretation of the predictions. Even though all of these aspects are important in practice, we will essentially focus on the first aspect: obtaining the best possible generalization performance.

The parametrization of $f$ differentiates the problem of model selection from the general machine learning problem. Instead of parameterizing $f$ with one set of parameters, the model selection framework distinguishes between parameters and hyper-parameters. We adopt the simplified notation $f(x; \alpha, \theta)$ for a model of parameters $\alpha$ and hyper-parameters $\theta$. The hyper-parameters may include indicators of presence or absence of features, choice of preprocessing methods, choice of algorithm or model sub-class (e.g., linear models, neural networks, kernel methods, etc.), algorithm or model sub-class parameters (e.g., number of layers and units per layer in a neural network, maximum degree of a polynomial, bandwidth of a kernel), choice of post-processing, etc. We also refer to the parameters of the prior $\mathcal{P}(f)$ in Bayesian/MAP learning and the parameters of the regularizer $W[f]$ in risk minimization as hyper-parameters even if the resulting predictor is not an explicit function of those parameters, because they are used in the process of learning.

In what follows, we relate the problem of model selection to that of hyper-parameter selection, taken in is broadest sense and encompassing all the cases mentioned above. We refer to the adjustment of the model parameters $\alpha$ as the first level of inference. When data are split in several subsets for the purpose of training and evaluating models, we call $n_{tr}$ the number of training examples used to adjust $\alpha$. If the hyper-parameters $\theta$ are adjusted from a subset of data of size $n_{va}$, we call the examples used to adjust them at this second level of inference the ‘validation sample’. Finally we call $n_{te}$ the number of test examples used to evaluate the final model. The corresponding empirical estimates of the expected risk $R(f)$, denoted $R_{tr}(f)$, $R_{va}(f)$, and $R_{te}(f)$, will be called respectively training error,
validation error, and test error.

6.1 Is Model Selection a Problem?

It is legitimate to first question whether the distinction between parameters and hyper-parameters is relevant. Splitting the learning problem into two levels of inference may be convenient for conducting experiments. For example, combinations of preprocessing, feature selection, and postprocessing are easily performed by fixing $\theta$ and training $\alpha$ with off-the-shelf programs. But, the distinction between parameters and hyper-parameters is more fundamental. For instance, in the model class of kernel methods $f(x) = \sum_i \alpha_i K(x_i, x, \theta)$, why couldn’t we treat both $\theta$ and $\alpha$ as regular parameters? One common argument is that, for fixed values of $\theta$, the problem of learning $\alpha$ can be formulated as a convex optimization problem, with a single unique solution, for which powerful mathematical programming packages are available, while the overall optimization of $\theta$ and $\alpha$ in nonconvex. Another compelling argument is that, splitting the learning problem into several levels might also benefit to the performance of the learning machine by alleviating (but not eliminating) the problem of over-fitting.

Consider for example the Gaussian radial basis function kernel $K(x_i, x_j, \theta) = \exp(-\|x_i - x_j\|^2/2\theta^2)$. The function $f(x) = \sum_i \alpha_i K(x_i, x, \theta)$ is a universal approximator if $\theta$ is let to vary and if the sum runs over the training examples. If both $\theta$ and $\alpha$ are optimized simultaneously, solutions with a small value of $\theta$ might be picked, having zero training error but possibly very poor generalization performance. The model class $\mathcal{F}$ to which $f$ belongs has infinite capacity $C(\mathcal{F})$. In contrast, for a fixed value of the hyper-parameter $\theta^o$, the model $f(x) = \sum_i \alpha_i K(x_i, x, \theta^o)$ is linear in its parameters $\alpha$ and has a finite capacity. In addition, the capacity of $f(x) = \sum_i \alpha_i K(x_i, x, \theta)$ of parameter $\theta$ for fixed values $\alpha^o$ and $x_i$ is very low. Hence, using multiple levels of inference may reduce over-fitting, while still searching for solutions in a model class of universal approximators. This last idea has been taken one step further in the method of structural risk minimization (Vapnik, 1979), by introducing new hyper-parameters in learning problems, which initially did not have any. Consider for instance the class of linear models $f(x) = w^T x$. It is possible to introduce hyper-parameters by imposing a structure in parameter space. A classical example is the structure $\|w\| \leq A$, where $\|w\|$ denotes the Euclidean norm and $A$ is a positive hyper-parameter. For increasing values of $A$ the space of parameters is organized in nested subsets.

6.2 Bayesian Model Selection

In the Bayesian framework, there is no model selection per se, since learning does not involve searching for an optimum function, but averaging over a posterior distribution. For example, if the model class $\mathcal{F}$ consists of models $f(x; \alpha, \theta)$, the Bayesian assumption is that the parameters $\alpha$ and hyper-parameters $\theta$ of the model used to generate the data are drawn from a prior $\mathcal{P}(\alpha, \theta)$. After observing some data $D$ the predictions should be made according to:

$$E_{\alpha, \theta}(y|x, D) = \int \int f(x; \alpha, \theta) \mathcal{P}(\alpha, \theta|D) d\alpha d\theta$$  \hspace{1cm} (6.1)$$

Hence there is no selection of a single model, but a summation over models in the model class $\mathcal{F}$, weighed by $\mathcal{P}(\alpha, \theta|D)$. The problem is to integrate over $\mathcal{P}(\alpha, \theta|D)$. A two-level decomposition can be made by factorizing $\mathcal{P}(\alpha, \theta|D) = \mathcal{P}(\alpha|\theta, D) \mathcal{P}(\theta|D)$:

$$E_{\alpha, \theta}(y|x, D) = \int \left( \int f(x; \alpha, \theta) \mathcal{P}(\alpha|\theta, D) d\alpha \right) \mathcal{P}(\theta|D) d\theta$$  \hspace{1cm} (6.2)$$

Bayesian model selection decomposes the prior $\mathcal{P}(\alpha, \theta)$ into parameter prior $\mathcal{P}(\alpha|\theta)$ and a ‘hyper-prior’ $\mathcal{P}(\theta)$. 
6.3 Frequentist Model Selection

While Bayesians view probabilities as being realized in the idea of ‘prior’ and ‘posterior’ knowledge of distributions, frequentists define probability in terms of frequencies of occurrence of events. In this section, the ‘frequentist’ approach is equated with risk minimization. There are obvious ties between the problem of model selection and that of performance prediction. Performance prediction is the problem of estimating the expected risk or generalization error $R(f)$. Model selection is the problem of adjusting the capacity or complexity of the models to the available amount of training data to avoid either under–fitting or over–fitting. Solving the performance prediction problem would also solve the model selection problem, but model selection is an easier problem. If we find an ordering index $r(f)$ such that for all pairs of functions $r(f_1) < r(f_2) \Rightarrow R(f_1) < R(f_2)$, then the index allows us to correctly carry out model selection. Arguably, the tightness of the bound is of secondary importance in obtaining a good ranking index. Bounds of the form $r(f) = R_{te}(f) + \epsilon(C/n_{te})$, where $C$ characterizes the capacity or complexity of the model class, penalizes complex models, but the penalty vanishes as $n_{te} \to \infty$. Some learning algorithms, for example, SVM or boosting, optimize a guaranteed risk rather than the empirical risk $R_{te}(f)$, and therefore provide some guarantee of good generalization. Algorithms derived in this way have an embedded model selection mechanism.

6.4 Model Selection and Error Estimation

Methods based on the splitting procedure of the data are favored by practitioners because they work well in many real cases and are simple to implement. Here we present a rigorous statistical analysis of two well known out–of–sample methods for model selection and error estimation: the k–fold Cross Validation (KCV) and the Bootstrap (BTS). The philosophy of the two methods is similar: part of the data is left out from the training set and is used for estimating the error of the classifier that has been found during the learning phase. The splitting of the original training data is repeated several time, in order to average out unlucky cases, therefore the entire procedure produces several classifier (one for each data splitting). Note that, from the point of view of our analysis, it is not statistically correct to select one of the trained classifiers to perform the classification of new samples because, in this case, the samples of the validation (or test) set would not be i.i.d. anymore. For this reason, every time a new sample is received, the user should randomly select one of the classifiers, so that the error estimation bounds, one for each trained classifier, can be safely averaged.

Depending on the chosen loss function, we can apply different statistical tools to estimate the classifier error. When dealing with the hard loss, we are considering sums of Bernoulli random variables, so we can use the well–known one–sided Clopper–Pearson bound [26, 27]. Given $t = n_{te} R_{te}(f)$ misclassifications, and defining $R(f) = R_{te}(f) + \epsilon$, then the errors follow a Binomial distribution:

$$B(t; n_{te}, R(f)) = \sum_{j=0}^{t} \binom{n_{te}}{j} R(f)^j (1 - R(f))^{n_{te} - j}$$

so we can bound the generalization error by computing the inverse of the Binomial tail:

$$\epsilon^*(R_{te}(f), n_{te}, \delta) = \max_{\epsilon} \{ \epsilon : B(t; n_{te}, R_{te}(f) + \epsilon) \geq \delta \}$$

and, therefore, with probability $(1 - \delta)$:

$$R(f) \leq R_{te}(f) + \epsilon^*(R_{te}(f), n_{te}, \delta).$$

We propose another tight bound, which was conceived by Hoeffding in [28], and has been neglected in the literature, mainly because it cannot be put in closed form (it holds for any bounded loss function). With
our notation, the bound is:

\[ \mathbb{P}[R(f) - R_{te}(f) > \epsilon] \leq \left[ \left( \frac{1 - R_{te}(f) - \epsilon}{1 - R_{te}(f)} \right)^{1-R_{te}(f)} \left( \frac{R_{te}(f) - \epsilon}{R_{te}(f)} \right)^{R_{te}(f)} \right]^{nte}. \] (6.6)

By equating the right part of Eq. (6.6) to \( \delta \) and solving it numerically, we can find the value \( \epsilon^* \), as a function of \( \delta \), \( m \) and \( R_{te}(f) \), that can be inserted in Eq. (6.5). There is another version of this bound that can be computed in close from:

\[ \mathbb{P}[R(f) - R_{te}(f) > \epsilon] \leq \exp\left[-2nte\epsilon^2\right] \] (6.7)

Then imposing \( \exp[-2nte\epsilon^2] = \delta \) we have:

\[ R(f) \leq R_{te}(f) + \sqrt{\frac{\log \frac{1}{\delta}}{2nte}} \] (6.8)

With probability \((1 - \delta)\).

### 6.4.1 k–fold cross validation

If our target is to perform both the model selection and the error estimation of the final classifier, a nested KCV is required, where \( k - 2 \) subsets are used, in turn, for the training phase, one is used as a validation set to optimize the hyperparameters and the last one as a test set to estimate the generalization error. Note that \( O(k^2) \) training steps are necessary in this case.

To guarantee the statistical soundness of the KCV approach, one of the \( k \) trained classifiers must be randomly chosen before classifying a new sample. This procedure is seldomly used in practice because, usually, one retrains a final classifier on the entire training data; however, as pointed out by many authors, we believe that this heuristic procedure is the one to blame for unexpected and inconsistent results of the KCV technique in the small–sample setting.

If the correct procedure is performed, then the generalization error can be bounded by

\[ R(f) \leq \frac{1}{k} \sum_{j=1}^{k} \left[ R_{n/k}^j(f) + \epsilon^* \left( R_{n/k}^j(f), n/k, \delta \right) \right] \] (6.9)

where \( R_{n/k}^j(f) \) is the error performed by the \( j \)-th optimal classifier on the corresponding test set, composed of \( n/k \) samples, and \( f \) is the randomly selected classifier.  

It is interesting to note that, for the LOO procedure, \( n/k = 1 \) so the bound becomes useless, in practice, for any reasonable value of the confidence \( \delta \). This is another hint that the LOO procedure should be used with care, as this result adds more concerns on its reliability, especially in the small–sample setting, which is the elective setting for LOO.

### 6.4.2 Bootstrap

The BTS method is a pure resampling technique: at each \( j \)-th step, a training set, with the same cardinality of the original one, is built by sampling the patterns with replacement, while the remaining \( m_j \) data, which consists, on average, of approximately 36.8% of the original dataset, are used to compose the validation set. The procedure is then repeated several times \((N_B \in [1, (2n-1)])\) in order to obtain statistically sound results [29]. As for KCV, if the user is interested in performing the error estimation of the trained classifiers, a nested Bootstrap is needed, which makes use of the \( n - n_j \) samples identified at the \( j \)-th step, for building
the validation set, by sampling them with replacement. After performing the model selection, we will be left with \( N_B \) different models for which the average generalization error can be expressed as:

\[
R(f) \leq \frac{1}{N_B} \sum_{j=1}^{N_B} \left[ R_{n_j}^j(f) + \epsilon^*(R_{n_j}^j(f), n_j, \delta) \right].
\] (6.10)

As can be seen by comparing (6.9) and (6.10), the two procedures are equivalent except for the different splitting of the original dataset.
Chapter 7

Practical laboratory sessions

7.1 SVM

In the folder ‘SVM’ you will find a series of file:

- main.m: an example of the use of the SVM routines
- SVM_learn.m: a function that perform the learning phase of SVM given the data $X$ the label $Y$ a kernel function the hyperparameter of SVM $C$ and the hyperparameter of the kernel $\gamma$.

$X \in \mathbb{R}^{n \times d}, \ n \ \text{pattern}, \ d \ \text{dimension}$ (7.1)

- SVM_forward.m: function that perform the forward phase of SVM
- SMO_ab.m: a function that solve the quadratic programming problem of SVM
- k_linear.m: an implementation of a linear kernel
- k_gaussian.m: an implementation of a gaussian kernel

7.1.1 Part A - warm up

Using the previous routines try to do the following step:

- using the linear kernel what happened if we change the hyperparameter $C$?
- using the gaussian kernel what happened if we change the hyperparameter $C$? And if we change the hyperparameter $\gamma$?
- try to implement a polynomial kernel and try to use this kernel by varying the hyperparameter $C$ and the degree of the polynomial kernel. What happened?
- in SVM learn there is a call to SMO_ab.m a function that solve a quadratic programming problem of SVM. Try to use the routine ‘quadprog’ (see help in Matlab) and see what happened.
- Try to change the dataset by creating two circles, one inside the other. Assign the label +1 the the internal circle and −1 to the other one. What kernel performs better?
7.1.2 Part B - analysis

Let’s go to the UCI dataset and chose a dataset that you like (Iris, Banana, etc...) or if you want try to think to a toy dataset (Archimedean spiral) with some noise (function ‘randn’ in Matlab see Help) and randomly split the data in two group (70% for learn and 30% for test). By varying the hyperparameter C and gamma (or the degree of the polinomial kernel) what happened to the error on the test set? Plot the trend of the test error and the train error in function of the different hyperparameter with the different kernel:

- linear
- polynomial
- gaussian

What do you see in these function? Any connections with the theoretical conclusions that we have discuss in class?
7.2 SVR

In the folder ‘SVR’ you will find a series of file:

- main.m: an example of the use of the SVR routines
- SVR_learn.m: a function that perform the learning phase of SVR given the data $X$ the target $Y$ a kernel function, the hyperparameter of SVR $C$, the hyperparameter epsilon and the hyperparameter of the kernel $\gamma$.

$$X \in \mathbb{R}^{n \times d}, \quad n \text{ pattern, } d \text{ dimension}$$ (7.2)
- SVR_forward.m: function that perform the forward phase of SVR
- SMO_ab.m: a function that solve the quadratic programming problem of SVM
- k_linear.m: an implementation of a linear kernel
- k_gaussian.m: an implementation of a gaussian kernel

Using the previous routines try to do the following step:

- using the gaussian kernel what happened if we change the hyperparameter $C$?
- using the gaussian kernel what happened if we change the hyperparameter epsilon?
- using the gaussian kernel what happened if we change the hyperparameter gamma?
- try to implement a polynomial kernel and try to use this kernel by varying the hyperparameter $C$ and the degree of the polynomial kernel. What happened?
- Try to change the dataset by creating some strange function that you like (with some noise function ‘randn’ in Matlab see Help):
  - There is some correlation between the magnitude of the noise and the different hyperparameter?
  - What happened if we use a gaussian kernel with a gamma too large if we have noise?

Let’s go to the UCI dataset and chose a dataset that you like or if you want try to think to a toy dataset with some noise and randomly split the data in two group (70% for learn and 30% for test). By varying the hyperparameter $C$ and gamma (or the degree of the polynomial kernel) what happened to the error on the test set?

Plot the trend of the test error and the train error in function of the different hyperparameter ($C$, epsilon, gamma) with the gaussian kernel.

What do you observe?
7.3 Decision Tree

In this case we will use some tools already developed in Matlab. In particular we will use the following function:

- `classregtree`: a function used for learning the tree
- `view`: a function for showing the shape of the tree (useful for the interpretation of the learning procedure)
- `prune`: a function for cutting the tree (reduce the complexity)
- `eval`: function that perform the forward phase

An example of the use of this function is showed in ‘main.m’ in the folder ‘Decision Tree’. For more explanation see the Help in Matlab.

Let’s go to the UCI dataset and choose a dataset (Iris etc...) that you like or if you want try to think to a toy dataset with some noise and randomly split the data in two groups (70% for learn and 30% for test). Now perform the following step:

- learn the tree with the 70% of the data
- look at the tree, and check if the tree is too complex or not
- try to prune the tree and look at what happened at the error on the test set
- plot the curve of the error in function of the number of level pruned in the tree
- this plot confirms or is in contrast with what we have explained in class?

Now add some more noise to the data of train (‘randn’ function on Matlab). You can perform this both in a real dataset and in the toy dataset. Varying the amount of noise:

- what happened to the tree?
- how change the optimal prune in function of the noise?
- the results are reasonable?
7.4 Clustering

In the folder ‘Clustering’ you will find two subfolder ‘kkmeans’ and ‘spectral clustering’. Let’s start with the folder ‘kkmeans’ that contains the following files:

- main.m: an example of the use of the kkmeans routine
- kkmeans.m: a function that implement the kkmeans
- k_linear.m: an implementation of a linear kernel
- k_gaussian.m: an implementation of a gaussian kernel

Instead the folder ‘spectral clustering’ contains the following files:

- main.m: an example of the use of the spectral clustering routine
- spectralclustering.m: a function that implement the spectralclustering
- k_gaussian.m: an implementation the gaussian routine for the affinity matrix $A$

7.4.1 Kernel k-means

Let’s start with the kernel k-means. Take the file ‘main.m’ and try to perform this steps:

- change the number of cluster $k$ and see what happened with a linear kernel
- run different time the same routine with a linear kernel, the same $k$ and the same data. In this way we start with different initial solution, the result is the same?
- implement a routine that chose the best solution based on different initial solution. What index of quality we have to use for selection the best solution?
- use a gaussian kernel and repeat the previous step. What do you observe?
- with the gaussian kernel what is the impact of gamma on the solution?
- how we have to chose the best gamma?

Now that we have a bit of confidence with this hyperparameters, let’s chose another toy dataset (with a non convex region) for example two clouds of data that lies on two circles one inside the other.

- what happened if we use a linear kernel?
- what happened if we use a gaussian kernel and we select a ‘good’ gamma?

Now try to think to other case where the k-means can work, for example the dataset that you can find in the script ‘datasettoy.m’. In this case we are able to find the correct cluster with the kernel k-means?
7.4.2 Spectral Clustering

Now let’s analyze the spectral clustering example.

- change the number of cluster $k$, what happened?
- change the gamma, what happened?
- if we do not normalize the data what happened?
- if we use a different affinity matrix what happened?
- if we change the problem by adding more cluster what happened?
- what is the relation between gamma and the number of cluster in the example of the script ‘main.m’?

Now let’s try to solve the problem that with the kernel k-means we was not able to solve. In this case the spectral clustering works? Is more difficult to find the correct value of gamma for make the things works?
Bibliography


