Kernel-Based Machines for Abstract and Easy Modeling of Automatic Learning

Alessandro Moschitti

Computer Science and Engineering Department,
University of Trento,
Via Sommarive 18, Povo (TN), Italy
moschitti@disi.unitn.it
http://disi.unitn.it/moschitti

Abstract. The modeling of system semantics (in several ICT domains) by means of pattern analysis or relational learning is a product of latest results in statistical learning theory. For example, the modeling of natural language semantics expressed by text, images, speech in information search (e.g. Google, Yahoo,..) or DNA sequence labeling in Bioinformatics represent distinguished cases of successful use of statistical machine learning. The reason of this success is due to the ability to overcome the concrete limitations of logic/rule-based approaches to semantic modeling: although, from a knowledge engineer perspective, rules are natural methods to encode system semantics, noise, ambiguity and errors affecting dynamic systems, prevent such approached from being effective, e.g. they are not flexible enough.

In contrast, statistical relational learning, applied to representations of system states, i.e. training examples, can produce semantic models of system behavior based on a large number attributes. As the values of the latter are automatically learned, they reflect the flexibility of statistical settings and the overall model is robust to unexpected system condition changes. Unfortunately, while attribute weight and their relations with other attributes can be automatically learned from examples, their design for representing the target object (e.g. a system state) has to be manually carry out. This requires expertise, intuition and deep knowledge about the expected system behavior. A typical difficult task is for example the conversion of structures into attribute-value representations.

Kernel Methods are powerful techniques designed within the statistical learning theory. They can be used in learning algorithms in place of attributes, thus simplifying object representation. More specifically, kernel functions can define structural and semantic similarities between objects (e.g. states) at abstract level, replacing the similarity defined in terms of attribute overlap.

In this chapter, we provide the basic notions of machine learning along with latest theoretical results obtained in recent years. First, we show traditional and simple machine learning algorithms based on attribute-value representations and probability notions such as the Naive Bayes and the Decision Tree classifiers. Second, we introduce the PAC learning theory and the Perceptron algorithm to provide the readers with essential concepts of modern machine learning. Finally, we use the above background to illustrate a simplified theory of Support Vector Machines, which, along with the kernel methods, are the ultimate product of the statistical learning theory.

M. Bernardo and V. Issarny (Eds.): SFM 2011, LNCS 6659, pp. 458–503, 2011.

1 What Is Machine Learning?

In high school, in the mathematic or statistic classes, we have been taught techniques that, given a set of points, e.g. $\boldsymbol{x}_i$ and the values associated with them, i.e. y_i, attempt to derive the functions that best interpolates their relationships $\phi(\boldsymbol{x}_i, y)$. For example, linear or polynomial regression as shown in Figure 1. These techniques, e.g. least square fit, are the first examples of machine learning algorithms. When the output values, y_i, of the target function are finite and discrete, the regression problem is called classification, which is very interesting for the application on real scenarios, e.g. categorization of text documents in different topics.

Before introducing more advanced machine learning techniques, it is helpful to show an example of their usefulness in ICT. Let us suppose that a programmer is asked to write the following program: given some employee characteristics and a pre-defined employee level hierarchy, automatically assign to each new employee the adequate entry level. Moreover, suppose that (i) the rules to determine such entry level depends on many variables, e.g. achieved diplomas, previous working experiences, age and so on; and (ii) there is no formal document that explains how to produce such rule set. This is not an unrealistic situation as the target company may use such level information to only propose tasks to employees; thus the level may be heuristically assigned by the human resource department by using an informal algorithm.

The unlucky programmer would be soon in troubles as it is rather difficult to extract algorithmic information from people not used to think in terms of procedures and instructions. What might be the solution?

We note that, there is a lot of data about the link between variables (i.e. the employees) and the output of the target function (i.e. the entry level). The company keeps the data of employees along with their entry levels, thus the programmer may examine the data and try to hand-craft the rules from it. However, if the number of employees and the number of their characteristics are large, this would result in a very time consuming and boring task.

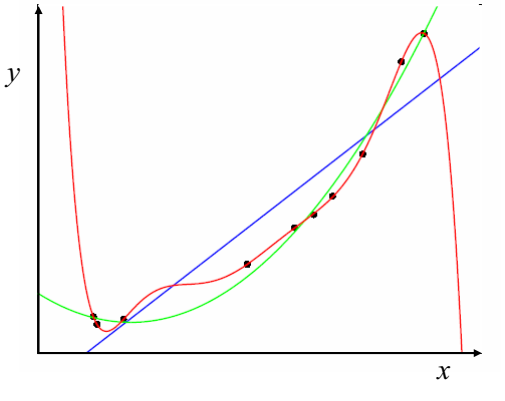

Fig. 1. Polynomial interpolation of as set of points $\langle x_i, y_i \rangle$

Machines have traditionally been built to perform such kind of job, thus, the solution should rely on writing an algorithm which automatically derives from examples the employee classification rules. This kind of algorithms are a special class of *machine learning methods* called example-driven or inductive learning models. They are standard in the sense that they can be applied to all problems in which there are some data examples and we need a classification function as output.

Given such tools, the lucky programmer can only re-write the examples from the employee database in an input format suitable for the target machine learning algorithm and run it to derive the classification function. The latter function unlikely will provide a correct entry level in all cases but if the commissioning company (as in this case) accepts an certain error rate in this procedure, the application of an automatic approach will be a feasible alternative to the hand-coding. Indeed, another output of the learning process is usually the expected error rate. This value can be estimated by measuring the number of classification mistakes that the classification function commits on a set of employee data (test set) not used for training.

We have introduced what learning models may offer to the solution of real problems. In the next section, we illustrate two simple ML approaches based on Decision Trees and naive probabilistic models. These will clarify the importance of kernel methods for more quickly and easily define the appropriate learning system.

1.1 Decision Trees

The introduction has shown that ML models derive a classification function from a set of training examples (e.g. the employee data) already categorized in the target class (e.g. the entry level). The input for the ML program is the set of examples encoded in a meaningful way with respect to the classification task, i.e. the level assignment. The variables describing the individual examples are usually called features and they capture important aspects of the classification objects, e.g. the employees. For instance, the study title is a relevant feature for the entry level whereas the preferred employee food is not relevant thus it should not be included in the example description.

The idea of decision tree classifier (DT) algorithm is inspired by a simple principle: the feature that correctly separates the highest number of training examples should be used before the others. To simplify such idea, suppose that we have only two levels (0 and 1) and also the features are binary (e.g. the employee has or not a master degrees). Intuitively, the decision tree algorithm finds the feature which splits the training set S in two subsets S_0 and S_1 such that the proportion of employees of level 0 is *higher* in S_0 than in S whereas the proportion of employees of level 1 is higher in S_1 than in S. This means that guessing the employee level in the two new sets is *easier* than in S. As we cannot hope to correctly separate all data with only one feature the algorithm will iteratively find other features that *best* separates S_0 and S_1.

Figure 2 illustrates the decision tree which a DT algorithm may generate. First, the *PhD* attribute is tested. In case the employee owns it the level is surely 1. Second, features such as *Previous Experiences* and *Intelligent Quotient* are tested. Finally, the tests on the leaves should output the final classification in case it had not been output on the internal nodes.

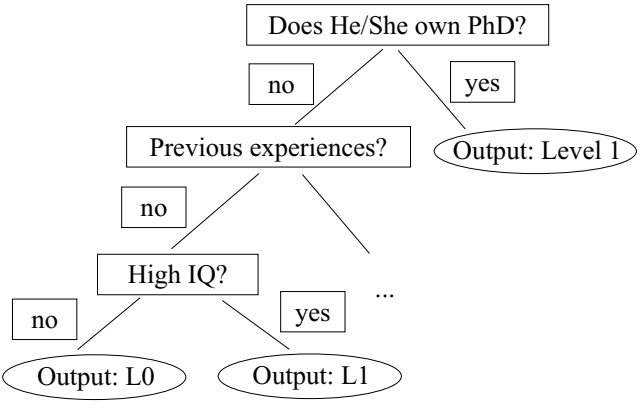

Fig. 2. Decision tree generated for the classification task of two employee levels

In order to find the most discriminative feature, DTs use the *entropy* quantity. In the general case, we have a set of classes $\{C_1, .., C_m\}$ distributed in the training set S with the probabilities $P(C_i)$, then the entropy H of P is the following:

$$H(P) = \sum_{i=1}^{m} -P(C_i) log_2(P(C_i)) \tag{1}$$

Suppose to select a feature f which assumes $\{a_1, .., a_n\}$ values in S. If we split the training examples according to f, we obtain n different subsets, i.e. $\{S_1, .., S_n\}$, whose average entropy is:

$$\bar{H}(P^{S_1}, .., P^{S_n}) = \sum_{i=1}^{m} \frac{H(P^{S_i})}{|S_i|} \tag{2}$$

where, P^{S_i} is the probability distribution of C_i on the S_i set and $H(P^{S_i})$ is the related entropy.

The DT algorithm evaluates Eq. 2 for each feature and selects the one which is associated with the highest value. Such approach uses the probability theory to select the most informative features and generate the tree of decisions. In the next section, we show another machine learning approach in which the probability theory is more explicitly applied for the design of the decision function.

1.2 Naive Bayes

We have pointed out that machine learning approaches are useful when the information about the target classification function (e.g. the commissioned program) is not explicitly available and is not completely accurate. Such aspects determine a degree of uncertainty, which results in an error rate.

Given the random nature of the expected outcome, the probability theory is well suited for the design of a classification function that aims to achieve the highest probability in producing correct results. Indeed, we can model the output of our target function as the probability to categorize an instance in a target class, given some parameters estimated from the training data.

More formally, let us indicate with E the classification example and let $\{C_1, .., C_m\}$ be the set of categories in which we want to classify such example. We are interested to evaluate the probability that E belongs to C_i, i.e. $P(C_i|E)$. In other words, we know the classifying example and we need to know its category. Our example E can be represented as a set of features $\{f_1, .., f_n\}$ but we do not know how to relate $P(C_i|f_1, .., f_n)$ to the training examples. Thus, we can use the Bayes' rule to derive a more useful probability form:

$$P(C_i|f_1, .., f_n) = \frac{P(f_1, .., f_n|C_i) \times P(C_i)}{P(f_1, .., f_n)}, \tag{3}$$

where

$$\sum_{i=1}^{m} P(C_i|f_1, .., f_n) = \sum_{i=1}^{m} \frac{P(f_1, .., f_n|C_i) \times P(C_i)}{P(f_1, .., f_n)} = 1$$

for definition of probability.

We will choose for the example E the category C_i associated with the maximum $P(C_i|E)$. To evaluate such probabilities, we need to select a category i and count the number of examples that contain the whole set of features, $\{f_1, .., f_n\}$. Considering that in real scenarios, a training set may contain no more than 10,000 examples, we will unlikely be able to derive reliable statistics as n binary features determine 2^n different examples[1]. Thus, to make the Bayesian approach practical, we naively assume that features are independent. Given such assumption, Eq. 3 can be rewritten as:

$$P(C_i|f_1, .., f_n) = \prod_{k=1}^{n} \frac{P(f_k|C_i) \times P(C_i)}{P(f_1, .., f_n)} \tag{4}$$

As $P(f_1, .., f_n)$ is the same for each i, we do not need it to determine the category associated with the maximal probability. The $P(C_i)$ can be computed by simply counting the number of training examples labeled as C_i, i.e. $|C_i|$ and divide it by the total number of examples in all categories:

$$P(C_i) = \frac{|C_i|}{\sum_{j=1}^{m} |C_j|}$$

To estimate $P(f_k|C_i)$, we derive n_{ik}, i.e. the number of examples categorized as C_i that contain the feature f_k and we divide it by the C_i cardinality, i.e.

$$P(f_k|C_i) = \frac{n_{ik}}{|C_i|}$$

[1] If we assume uniform distribution, to have a chance that a target example of only 20 features is included in the training set, the latter has to have a size larger than 1 billion of examples.

Table 1. Probability distribution of $sneeze$, $cough$ and $fever$ features inside the Allergy, Cold and Healthy categories

Prob.	Allergy	Cold	Healthy
$P(C_i)$	0.05	0.05	0.9
$P(sneeze\|C_i)$	0.9	0.9	0.1
$P(cough\|C_i)$	0.7	0.8	0.1
$P(fever\|C_i)$	0.4	0.7	0.01

As an example of naive Bayesian classification suppose that we divide the healthy conditions of target patients in thee different categories: Allergy, Cold and Healthy. The features that we use to categorize such states are $f_1 = sneeze$, $f_2 = cough$ and $f_3 = fever$. Suppose also that we can derive the probability distribution of Table 1 from a medical database, in which f_1, f_2 and f_3 are annotated for each patient.

If we extract from our target patient the following feature representation $E = \{sneeze, cough, \sim fever\}$, where $\sim$ stands for not $fever$, we can evaluate his/her probabilities to be in each category i:

- $P(\text{Allergy}\,|E)$ = (0.05)(0.9)(0.7)(0.6)/P(E)=0.019/$P(E)$
- $P(\text{Cold}\,|E) = (0.05)(0.9)(0.8)(0.3)/P(E) = 0.01/P(E)$
- $P(\text{Healthy}\,|E) = (0.9)(0.1)(0.1)(0.99)/P(E) = 0.0089/P(E)$

According to the above table, the patient should be affected by allergy.

It is worth to note that such probabilities depend on the product of the probabilities of each feature. It may occur, especially when the training corpus is too small, that some of them never appear in the training examples of some categories. As a consequence, the estimation of the probability of a feature f in the category C_i, $P(f|C_i)$, will be 0. This causes the product of Eq. 4 of a category i to be 0, although the contributions of the other features in the product may be high. In general, assigning a probability equal 0 to a feature is a rough approximation as the real probability is just too small to be observed in the training data, i.e. we do not have enough data to find an occurrence of f.

To solve the above problem, smoothing techniques are applied. The idea is to give to the features which do not appear in the training data a small probability, α. To keep constant the overall feature probability mass, to the other features will be subtracted a small portion, β, of their probability such that the overall summation is still 1.

The simplest of such techniques is called Laplace smoothing. The new feature probability is the following:

$$P(f_k|C_i) = \frac{n_{ik} + a \times p_k}{|C_i| + a}$$

where p_k is a probability distribution and a is the size of a hypothetical set of examples, where we assume to have observed p_k. When we do not know any information about the not observed features, it is logical to assume a uniform distribution, i.e. $p_k = 1/a$ therefore $a = n$ and

$$P(f_k|C_i) = \frac{n_{ik} + 1}{|C_i| + n}.$$

The smoothing techniques improve the Naive Bayes model by providing a better estimation of the probability of the features not observed in the data. However, the independence assumption seems a serious limitation to the accuracy reachable by such approach. The next section illustrates more recent machine learning techniques, which do not need to make such assumptions. These are called Support Vector Machines and also offer the possibility to model object with abstract feature representations.

Exercise 1. Classify using a Naive Bayes learning algorithm and the probabilities in Table 1 all 8 possible examples, e.g. $\{sneeze, cough, \sim fever\}$, $\{sneeze, \sim cough, fever\}$,...

Exercise 2. Modify the probabilities in Table 1 to classify e.g. $\{sneeze, cough, \sim fever\}$ in class Cold with a Naive Bayes classifier.

Exercise 3. Define a new learning application using the Naive Bayes algorithm.

2 Probably Approximately Correct (PAC) Learning

So far, we have seen two different ML approaches, i.e. DT and Naive Bayes. They can be both applied to training examples to learn classification functions and estimate their accuracy on a test set of new examples. Intuitively, we may think that as the number of training examples increases the accuracy increases as well. Unfortunately, this is not generally true. For example, if we want to learn the difference between Allergy and Cold categories using only the $sneeze$ and $cough$ features, we will never reach high accuracy, no matter how many training examples we have available. This happens because such features do not deterministically separate the two classes.

Given such problems, we need some analytical results that helps us to determine (1) if our learning function is *adequate* for the target learning problem and (2) the probability of error according to the number of available training examples. The class of functions for which we have such analytical data is called the probably approximately correct (PAC) class.

The statistical learning theory provides mathematical tool to determine if a class of functions is PAC learnable. At the base of such result there is a new statistical quantity designed by two scientists, Vapnik and Chervonenkis, called VC-dimension. This gives a measure of the learning complexity and can be used to estimate the classification error.

In the next sections, we formally define the PAC function class, provide a practical example to derive the error probability of PAC functions and introduce the VC-dimension, which automatizes the estimation of such error.

2.1 Formal PAC Definition

The aim of ML is to learn some functions from a set (Tr) of training examples. These latter can be seen as data points that are associated with some discrete values $\mathcal{C} = \{C_1, .., C_n\}$, in case of classification problems or real number $\mathbb{R}$, in case of regression problem. We focus only on classification problems, i.e. on finding a function $f : X \rightarrow \mathcal{C}$ using $Tr \in X$. In general, the training examples are randomly drawn thus we need to deal with a probability distribution D on X.

The function f can be learned by using an algorithm, which can generate only a small subset of all possible functions. Such algorithm derives a function $h \in H$ from the examples, where H is the class of all possible hypotheses (functions) derivable with it. This suggests that h will hardly be equal to f, consequently, it is very useful to define a measure of its error.

A reasonable measure is the percentage of points for which f and h differ, i.e. the probability that given an example x, $P[f(x) \neq h(x)]$. Note that D is particularly important. As a trivial example, if the probability $D(x_0)$ of an element $x_0 \in X$ is 1 and $f(x_0) = h(x_0)$, the error rate will be 0, independently of the number of $x \in Tr$ for which $f(x) \neq h(x)$.

The above case is very rare and does not occur in practical situations. On the contrary, there is a large class of functions whose error decreases as the number of training examples increases. These constitute the PAC learnable functions. Their formal definition is the following:

- Let the function $f : X \rightarrow \mathcal{C}$ belongs to the class F, i.e. $f \in F$, where X is the domain and $\mathcal{C}$ is the codomain of f.
- Suppose that the training and the test documents $x \in X$ are generated with a probability D.
- Let $h \in H$ be the function that we learned from the examples provided that we can learn only functions in H, i.e. in the hypothesis space.
- The error of h, $error(h)$, is defined as $P[f(x) \neq h(x)]$, i.e. the percentage of miss-classified examples.
- Let m be the size of the training set, then F is a class of PAC learnable functions if there is a learning algorithm such that:
  - $\forall f \in F, \forall D \in X$ and $\forall \epsilon > 0, \delta < 1$
  - $\exists m$ such that $P[error(h) > \epsilon] < \delta$, i.e. the probability that the h's error is greater than ϵ is lower than δ.

In other words, a class of functions F is PAC learnable if we can find a learning algorithm which, given an enough number of training examples, produces a function h such that its error is greater than ϵ with a probability less than δ. Thus by choosing low values for ϵ and δ, we can have a low error (i.e. $< \epsilon$) with high probability (i.e. $1 - \delta$).

Next section clarifies the above idea with a practical example.

2.2 An Example of PAC Learnable Functions

Suppose that we need to learn the concept of medium-built people. Given such problem two very important features are the height and the weight of a person. One of these features alone would not be able to characterize the concept of medium-built body. For example, a person which has a height of 1,75 meters may be seen as medium person but if her/his weight is 130 kg we would immediately change our idea.

As the above two features assume real number values, we can represent people on a cartesian chart, where the X-axis and Y-axis correspond to height and the weight, respectively. Figure 3 illustrates such idea.

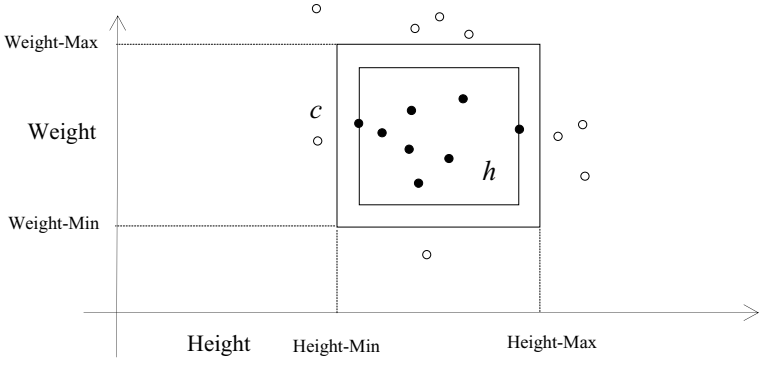

Fig. 3. The medium-built person concept on a cartesian chart

This representation implies that the medium-built person concept c is represented by a rectangle, which defines the maximum and minimum weight and height. Suppose that we have available some training examples, i.e. the measures of a set of people, which may or may not have a medium-build body, we can represent them in the chart. The white points, which are outside the rectangle c, are not medium-built people all the others (black points) are instead in such class.

As we assumed that our hypothesis c has a rectangular shape whose edges are parallel to the axes, our ML algorithm should only learn h from the rectangle set, namely the set of hypotheses H. Additionally, since the error is defined as $P[f(x) \neq h(x)]$, we can evaluate it by dividing the area between the rectangles c and h by the area of c[2].

In order to design an effective algorithm, we need to exploit the training data. In this respect, a simple way is to avoid errors in the training set; hence our learning algorithm is the following:

Select the smallest rectangle having its edges parallel to the axes that includes all training examples corresponding to medium-built people.

Since it includes all positive points, it would not make mistakes on them on the training set. Selecting also the smallest rectangle also prevents to commit error on the more external negative points.

We would like to verify that this is a PAC algorithm. To do this, we fix an error ϵ, a target probability δ and evaluate the $P[error(h) > \epsilon]$, i.e. the probability of generating a bad hypothesis, h. (to be a PAC algorithm, such probability must be lower then δ). Since $P[error(h) > \epsilon]$, h correctly classifies one training example with a probability $< 1 - \epsilon$. This implies that, in the cartesian representation of Figure 4.A, the rectangle associated with a bad h is included in the smallest rectangle of $good$ hypotheses (i.e. the hypotheses of area equal to $1 - \epsilon$). Additionally, our algorithm produces a rectangle (a hypothesis) that includes all m training points.

Now, let us consider the four strips between c and h: a *bad* hypothesis cannot contemporary touch all four strips as shown by the frames B and C. It follows that, a

[2] It can be proven that this is true for any distribution D.

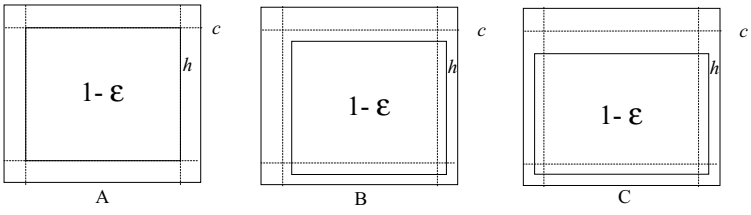

Fig. 4. Probabilities of bad and $good$ hypotheses

necessary condition for the existence of a bad hypothesis is to have all the m points at least outside of one of the 4 strips. *Necessary* means that it must happen each time we learn a bad hypothesis and, consequently, the probability of drawing m points out of at least one strip is higher than a hypothesis to be bad. In other words, the latter is upper bounded by the former probability. More in detail, the evaluation of the probability of the latter follows these steps:

1. the probability that a point x is out of one strip, $P(x \text{ out of 1 strip}) = (1 - \epsilon/4)$;
2. the probability that m points are out of one strip, $P(x \text{ out of 1 strip})^m = (1-\epsilon/4)^m$;
3. the probability that m points are out of 4 strips $< 4P(x \text{ out of 1 strip})^m = 4(1 - \epsilon/4)^m$;

Therefore, we can use the inequality, $P[error(h) > \epsilon] < 4(1 - \epsilon/4)^m < \delta$, to impose our δ requirement. From $\Rightarrow 4(1 - \epsilon/4)^m < \delta$, we can derive an upperbound[3] to m (satisfying our constraint):

$$m > \frac{ln(\delta/4)}{ln(1 - \epsilon/4)}$$

From Taylor's series, we know that

$$-ln(1 - y) = y + y^2/2 + y^3/3 + .. \Rightarrow (1 - y) < e^{(-y)}$$

We can apply the above inequality to $ln(1 - \epsilon/4)$ to obtain

$$m > \frac{ln(\delta/4)}{ln(1 - \epsilon/4)} \Rightarrow m > \frac{4ln(4/\delta)}{\epsilon}. \tag{5}$$

Eq. 5 proves that the medium-built people concept is PAC learnable as we can reduce the error probability as much as we want, provided that we have an enough number of training examples.

It is interesting to note that a general upperbound for PAC functions can be evaluated by considering the following points:

1. the probability that a bad hypothesis is consistent with m training examples (i.e. classifies them correctly) is $(1 - \epsilon)^m$;

[3] Consider that we divide by $ln(1 - \epsilon/4)$, which is always negative, thus we need to change the direction of the inequality.

2. the number of *bad* hypotheses is less than the total number of hypotheses $N \Rightarrow$
3. $P(h$ bad and consistent with m examples) = $N(1-\epsilon)^m < Ne^{-\epsilon^m} = Ne^{-m\epsilon} < \delta$. It follows that

$$m > \frac{1}{\epsilon}(ln\frac{1}{\delta} + lnN). \quad (6)$$

We can use Eq. 6 when N can be estimated. For example, if we want to learn a Boolean function of n variable, their number is $2^{2^n} > N \Rightarrow$ a rough upperboud of the needed m is $\frac{1}{\epsilon}(ln\frac{1}{\delta} + 2^n ln2)$.

In most cases the above bound is not useful and we need to derive one specific to our target problem as we did for the medium-built concept. However, when the feature space is larger than 2 the manual procedure may become much more complex. In the next section, we will see a characterization of PAC functions via VC dimension, which makes more systematic derivation of PAC properties.

2.3 The VC-Dimension

The previous section has shown that some function classes can be learned with any accuracy and this depends on the properties of the adopted learning algorithm. For example, the fact that we use rectangles as our hypothesis space (the one from which our algorithm selects h) instead of circles or lines impacts on the *learning capacity* of our algorithm.

Indeed, it is easy to show that using lines, we would have never been able to separate medium-built people from the others whereas the rectangle class is rather effective to do this. Thus, we need a property that allows us to determine which hypothesis class is more appropriate to learn a target function $f \in F$. Moreover, we note that, in most of the cases, we do not know the nature of the target f. We know only the training examples, consequently, our property should be derived only from them and by the function class H that we have available.

The Vapnik and Chervonenkis (VC) dimension aims to characterize functions from a learning perspective. The intuitive idea is that different function classes have different capacity in separating data points: some of them can just separate some configurations of points whereas others can separate a much larger number of configurations, i.e. they are in some sense more general purpose. The VC dimension captures this kind of property.

Intuitively, VC dimension, i.e. the learning capacity, determines the generalization reachable during learning:

- A function selected from a high class capacity is expected to *easily* separate the training points since it has the capacity to adapt to any training set. This will result on a learned function too specific to the used training data (i.e. it will overfit data). An immediate consequence is that the probability to correctly separate the test set will be lower.
- In contrast, a function that belongs to a low capacity class can separate a lower number of data configurations thus if it successful separates the current training points, the probability to well separate the test data will be higher.

The definition of VC dimension depends on the concept of shattering a set of points.

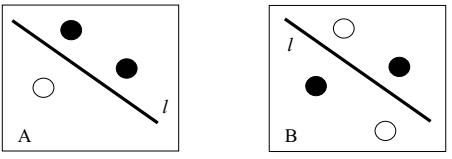

Fig. 5. VC dimension of lines in a bidimensional space

Definition 1. ***Shattered Sets***
Let us consider binary classification functions $f \in F$, $f : X \rightarrow \{0, 1\}$. We say that $S \subseteq X$ is shattered by a function class F if $\forall S' \subseteq S$, $\exists f \in F$:

$$f(x) = \begin{cases} 0 \textit{ iff } x \in S' \\ 1 \textit{ iff } x \in S - S' \end{cases} \tag{7}$$

The definition says that a set of points S is shattered by a function class F if for any assignment of the points in S into $\{0, 1\}$, we can find $f \in F$ that reproduces such assignments.

A graphical interpretation is given in Figure 5. In the frame A, we have 3 points represented in a two-dimensional space. The target function class L is the one of linear functions. For any assignment of points (white is 0 and black is 1), we can find a line $l \in L$ that separates them. From l we can derive the shattering function f by assigning $f(x_1, x_2)$=0 *iff* $x_2 < l(x_1)$ and 1 otherwise, i.e., if the point is under the line, we assign 0 to it and 1 otherwise. Consequently, a set of three points can be shattered by linear functions.

On the contrary, the 4 points in the frame B cannot be shattered. More precisely, there are not 4 points that can be shattered by linear functions since we can always draw a tetragon having such points as vertices and assign the same color to the opposite vertices. If the line assigned the same color to the opposite vertices there would always be a vertex on the same side of such two points with a different color.

Definition 2. ***VC dimension***
The VC dimension of a function class F is the maximum number of points that can be shattered by F.

Since Figure 5.A shows a set of tree points shattered by a linear function, such class has at least a VC dimension of 3 in the bidimensional space. We have also proved that 4 points cannot be shattered, consequently, the VC dimension of linear functions on the plane is exactly 3. Note that, selecting points that are linearly dependent, i.e., they lie on the same lines, will not work as we cannot hope to shatter them if we assign the same label to those external an a different color to the internal one. In particular it can be proven (see [16]) the following:

Theorem 1. *Consider a set of m points in $\mathbb{R}^n$ and choose any one of the points as origin, then they can be shattered by oriented hyperplanes if and only if the position vectors of the remaining points are linearly independent.*

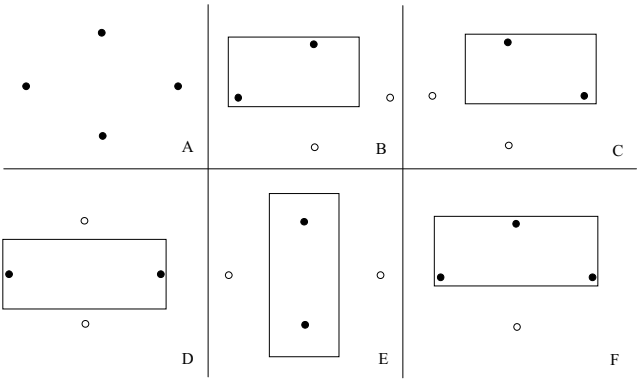

Fig. 6. VC dimension of (axis aligned) rectangles

As a consequence we have the following

Corollary 1. *The VC dimension of the class of functions composed by the set of oriented hyperplanes in $\mathbb{R}^n$ is n+1.*

Proof. We can always choose one of the points as origin of vectors and the remaining n points as their end such that the vectors are linearly independent. However, we can never choose $n + 1$ of such points (since no set of $n + 1$ vectors in $\mathbb{R}^n$ can be linearly independent).

This Corollary is useful to determine the VC dimension of linear functions in an n dimensional space. Linear functions are the building block of Support Vector Machines, nevertheless, there are other examples of classifiers which have different VC dimension such as the rectangle class. The following example is useful to understand how to evaluate the VC dimension of geometric classifiers.

Example 1. **The VC dimension of rectangles with edges parallel to the axes**
To evaluate the VC dimension of rectangles, we (i) make a guess about its value, for instance 4, (ii) show that 4 points can be shattered by rectangles and (iii) prove that no set of 5 points can be shattered.

Let us choose 4 points that are not aligned like in Figure 6.A. Then, we give all possible assignments to the 4 points. For example, Figure 6.B shows two pairs of adjacent points, which have the same color. In Section 2.2, we established that points inside the rectangle belong to medium-built people, i.e., they are positive examples of such class. Without loss of generality, we can keep such assumption and use the black color to indicate that the examples are positive (or that they are assigned to 1). The only relevant aspect is that we need to be consistent with such choice for all assignments, i.e., we cannot change our classification algorithm while we are testing it on the point configurations.

From the above convention, it follows that given the assignments B, C, D, E and F in Figure 6, we need to find the rectangles that contain only black points and leave the

white points outside. The rectangles C, D, E, F separate half positive and half negative examples. It is worth noting that if we have 3 positive (or 3 negative) examples, finding the shattering rectangles is straight forward (see Frame E), consequently, we have proven that the VC dimension is at least 4.

To prove that is not greater than 4, let us consider a general 5 point set. We can create 4 different rankings of the points by sorting in ascending and descending order by their x-coordinate and by their y-coordinate. Then, we color the top point of each of the 4 lists in black and the 5th point in white. The latter will be included (by construction) in the rectangle of the selected 4 vertices. Since any rectangular hypothesis h that contains the four points must contain the previous rectangle, we cannot hope to exclude the 5th point from h. Consequently, no set of 5 points can be shattered by the rectangle class.

Finally, we report two theorems on the sample complexity, which, given a certain wished error, derive upper and lower bounds of the required number of training examples. We also report one theorem on the error probability of a hypothesis given the VC dimension of its class. These theorems make clear the link between VC dimension and PAC learning.

Theorem 2. *(upper bound on sample complexity, [15])*
Let H and F be two function classes such that $F \subseteq H$ and let A an algorithm that derives a function $h \in H$ consistent with m training examples. Then, $\exists c_0$ such that $\forall f \in F$, $\forall D$ distribution, $\forall \epsilon > 0$ and $\delta < 1$ if

$$m > \frac{c_0}{\epsilon}\Big(d \times ln\frac{1}{\epsilon} + \frac{1}{\delta}\Big)$$

then with a probability $1 - \delta$,

$$error_D(h) \leq \epsilon,$$

where d is the VC dimension of H and $error_D(h)$ is the error of h according to the data distribution D.

Theorem 3. *(lower bound on sample complexity, [15])*
To learn a concept class F whose VC-dimension is d, any PAC algorithm requires $m = O(\frac{1}{\epsilon}(\frac{1}{\delta} + d))$ examples.

Theorem 4. *(Vapnik and Chervonenkis, [64])*
Let H be a hypothesis space having VC dimension d. For any probability distribution D on $X \times \{-1, 1\}$, with probability $1 - \delta$ over m random examples S, any hypothesis $h \in H$ that is consistent with S has error no more than

$$error(h) \leq \epsilon(m, H, \delta) = \frac{2}{m}\big(d \times ln\frac{2e \times m}{d} + ln\frac{2}{\delta}\big),$$

provided that $d \leq m$ and $m \geq 2/\epsilon$.

Exercise 4. Compare the upper bounds on sample complexity of rectangles derived in Section 2.2 with the one derivable from Theorem 2.

Exercise 5. Evaluate the VC dimensions of triangles aligned and not aligned to the axes.

Exercise 6. Evaluate the VC dimension of circles.

3 Support Vector Machines

The previous section has shown that classification instances can be represented with numerical features. These can also be associated with the coordinates of points in an n-dimensional space, where a classification function can be modeled with geometrical objects, e.g., lines or hyperplanes. The latter constitute the basic building block of the statistical learning theory, which has produced Support Vector Machines (SVMs).

In this section, we first introduce the Perceptron algorithm, which can be considered the simplest SVM and then we define the theory and algorithms of more advanced SVMs. One of their important properties is the possibility to use kernel functions to deal with non linear classification problems. Thus, a conclusive section will introduce the kernel theory and its application to advanced learning tasks, e.g., the classification of syntactic-parse trees.

3.1 Perceptrons

Once objects are projected into a vector space, they can be simply classified by linear functions, e.g., Figure 5.A shows a line that separates black from white points. One advantage of such mathematical objects is their simplicity that allows us to design efficient learning algorithms, i.e., efficient approaches to find separating lines or hyperplanes in high dimensional spaces.

The reader may wonder if such simplicity limits the capability of the learning algorithms or if we can use them to learn any possible *learnable function*. It is clear that with only one hyperplane, we cannot learn any function. For example, Figure 5 shows four points that cannot be separated in the Frame B. However, this is not a definitive limitation of linear functions as:

1. By modeling our learning problem more effectively, i.e., by choosing more appropriate features, the target problem could become linearly separable. For example, the four points of the previous figure can be divided in a three-dimensional space. This means that we need just to add a significant feature to solve the problem.
2. We can use linear functions in cascade. The resulting function is more expressive and, depending on the number of levels in such cascade, we can design any function.

The thesis that linear functions are sufficient to derive any learnable relation from examples is supported by the observation that human beings' brain is structured with such sort of devices.

To clarify this point, let us consider an animal neuron shown in Figure 7. It is constituted by one set of inputs, i.e., the dendrites, which are connected to a cellular body, i.e., soma, via synapses. These are able to amplify or attenuate an incoming signal. The neuron output is transported by the axon, whose filaments are connected to the dendrites of other neurons. When a chemical signal is transmitted to the dendrites, it is amplified by the synapses before entering in the soma. If the overall signal, coming from different synapses, overcomes a certain threshold, the soma will launch a signal to the other neurons through the axon.

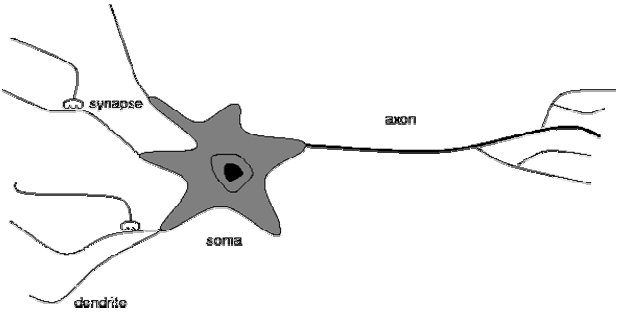

Fig. 7. An animal neuron

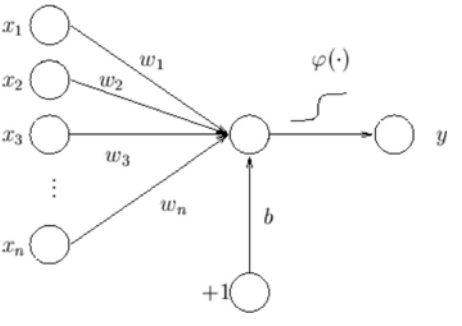

Fig. 8. An artificial neuron

The artificial version of the neuron is often referred to as *Perceptron* and can be sketched as in Figure 8. Each dendrite is an input x_i associated with a weight w_i. The product between the weights and the input signals are summed together and if such summation overcomes the threshold b the output y will be 1, otherwise it will be 0. The interesting aspect is that the output of such neuron can be modeled with a simple hyperplane whose equation is:

$$y = w_1 x_1 + .. + w_n x_n + b = \boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \tag{8}$$

where the final perceptron classification function output is obtained by applying the signum function to y, i.e.,

$$f(\boldsymbol{x}) = sgn(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{9}$$

Eq. 9 shows that linear functions are equivalent to neurons, which, combined together, constitute the most complex learning device that we know, i.e., the human brain. The signum function simply divides the data points in two sets: those that are over and those that are below the hyperplane. The major advantage of using linear functions is that given a set of training points, $\{\boldsymbol{x}_1, .., \boldsymbol{x}_m\}$, each one associated with a classification label y_i (i.e., $+1$ or -1), we can apply a learning algorithm that derives the vector $\boldsymbol{w}$ and the scalar b of a separating hyperplane, provided that at least one exists.

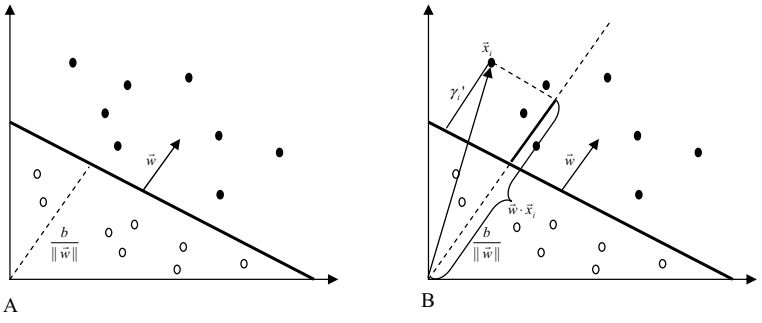

Fig. 9. Separating hyperplane and geometric margin

For example, Figure 9.A shows a set of training points (black positives and white negatives) along with a separating hyperplane in a 2-dimensional space. The vector $\boldsymbol{w}$ and the scalar $-b/||\boldsymbol{w}||$ are the gradient vector and the distance of such hyperplane from the origin, respectively. Indeed, from Eq. 8, $-b = \boldsymbol{w} \cdot \boldsymbol{x}$ thus $-b/||\boldsymbol{w}|| = \boldsymbol{w}/||\boldsymbol{w}|| \cdot \boldsymbol{x}$, where $\boldsymbol{x}$ is any point lying on the hyperplane and $\boldsymbol{w}/||\boldsymbol{w}|| \cdot \boldsymbol{x}$ is the projection of $\boldsymbol{x}$ on the gradient (i.e., the normal to the hyperplane).

The perceptron learning algorithm exploits the above properties along with the concept of functional and geometric margin.

Definition 3. *The **functional margin** γ_i of an example $\boldsymbol{x}_i$ with respect to a hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ is the product $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)$.*

Definition 4. *The **geometric margin** γ_i' of an example $\boldsymbol{x}_i$ with respect to a hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ is $y_i(\frac{\boldsymbol{w}}{||\boldsymbol{w}||} \cdot \boldsymbol{x}_i + \frac{b}{||\boldsymbol{w}||})$.*

It is immediate to see in Figure 9.B that the geometric margin γ_i' is the distance of the point $\boldsymbol{x}_i$ from the hyperplane as:

- $\frac{\boldsymbol{w}}{||\boldsymbol{w}||} \cdot \boldsymbol{x}_i$ is the projection of $\boldsymbol{x}_i$ on the line crossing the origin and parallel to $\boldsymbol{w}$;
- the distance of the hyperplane from the origin is subtracted to the above quantity, i.e., $\frac{b}{||\boldsymbol{w}||}$. It follows that we obtain the distance of $\boldsymbol{x}$ from the hyperplane.
- When the example $\boldsymbol{x}$ is negative, it is located under the hyperplane thus the product $\boldsymbol{w} \cdot \boldsymbol{x}_i$ is negative. If we multiply such quantity by the label y_i (i.e., -1), we make it positive, i.e., we obtain a distance.

Given the above geometric concepts, the algorithm of perceptron learning in Table 2, results very clear. At step $k = 0$, $\boldsymbol{w}$ and b are set to 0, i.e., $\boldsymbol{w}_0 = \mathbf{0}$ and $b_0 = 0$, whereas R is set to the maximum length of the training set vectors, i.e., the maximum among the distances of the training points from the origin. Then, for each $\boldsymbol{x}_i$, the functional margin $y_i(\boldsymbol{w}_k \cdot \boldsymbol{x}_i + b_k)$ is evaluated. If it is negative it means that $\boldsymbol{x}_i$ is not correctly classified by the hyperplane, i.e., y_i disagrees with the point position with respect to the hyperplane. In this case, we need to adjust the hyperplane to correctly classify the example. This can be done by rotating the current hyperplane (i.e., by summing $\eta y_i \boldsymbol{x}_i$

Table 2. Rosenblatt's perceptron algorithm

```
function Perceptron(training-point set: {x_1, .., x_m})
begin
   w_0 = 0; b_0 = 0; k = 0;
   R = max_{1≤i≤m} ||x_i||
   repeat
      no_errors = 1;
      for (i = 1 to m)
         if y_i(w_k · x_i + b_k) ≤ 0 then
            w_{k+1} = w_k + η y_i x_i;
            b_{k+1} = b_k + η y_i R^2;
            k = k + 1;no_errors = 0;
         end(if)
   until no_errors;
   return k, w_k and b_k ;
end
```

to $\boldsymbol{w}_k$) as shown in the charts A and B of Figure 10 and by translating the hyperplane of a quantity $\eta y_i R^2$ as shown in the chart C.

The perceptron algorithm always converges when the data points are linearly separable as stated by the following.

Theorem 5. *(Novikoff) Let S be a non-trivial training and let $\gamma > 0$ and $R = max_{1\leq i\leq m} ||\boldsymbol{x}_i||$. Suppose that there exists a vector $\boldsymbol{w}_{opt}$ such that $||\boldsymbol{w}_{opt}|| = 1$ and $y_i(\boldsymbol{w}_{opt} \cdot \boldsymbol{x}_i + b_{opt}) \geq \gamma \ \forall i = 1, .., m$. Then the number of mistakes made by the perceptron algorithm on S is at most $\left(\frac{2R}{\gamma}\right)^2$.*

This theorem proves that the algorithm converges in a finite number of iterations bounded by $\left(\frac{2R}{\gamma}\right)^2$ provided that a separating hyperplane exists. In particular:

- the condition $||\boldsymbol{w}_{opt}|| = 1$ states that normalized vectors are considered, i.e. $\boldsymbol{w}_{opt} = \frac{\boldsymbol{w}_{opt}}{||\boldsymbol{w}_{opt}||}$, thus the functional margin is equal to the geometric margin.
- $y_i(\boldsymbol{w}_{opt} \cdot \boldsymbol{x}_i + b_{opt}) \geq \gamma$ is equivalent to state that for such hyperplane the geometric margin of the data points are $\geq \gamma > 0$, i.e. any point is correctly classified by the hyperplane, $\boldsymbol{w}_{opt} \cdot \boldsymbol{x} + b_{opt} = 0$.

If the training data is not separable then the algorithm will oscillate indefinitely correcting at each step some misclassified example.

An interesting property showed by the Novikoff theorem is that the gradient $\boldsymbol{w}$ is obtained by adding vectors proportional to the examples $\boldsymbol{x}_i$ to $\mathbf{0}$. This means that $\boldsymbol{w}$ can be written as a linear combination of training points, i.e.,

$$\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i \tag{10}$$

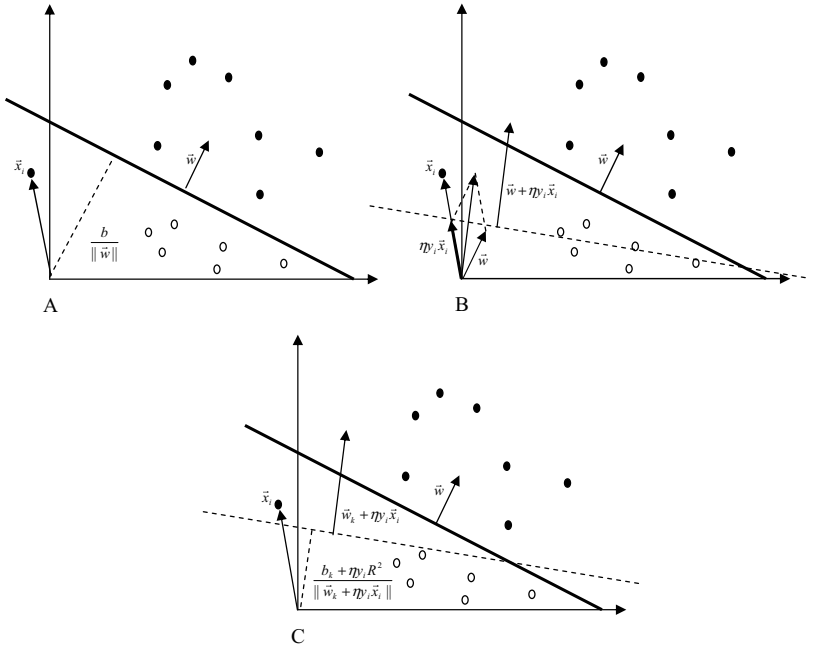

Fig. 10. Perceptron algorithm process

Since the sign of the contribution $\boldsymbol{x}_i$ is given by y_i, α_i is positive and is proportional (through the η factor) to the number of times that $\boldsymbol{x}_i$ is incorrectly classified. *Difficult points* that cause many mistakes will be associated with large α_i.

It is interesting to note that, if we fix the training set S, we can use the α_i as alternative coordinates of a dual space to represent the target hypothesis associated with $\boldsymbol{w}$. The resulting decision function is the following:

$$h(x) = sgn(\boldsymbol{w} \cdot \boldsymbol{x} + b) = sgn\left(\left(\sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i\right) \cdot \boldsymbol{x} + b\right) =$$

$$= sgn\left(\sum_{i=1}^{m} \alpha_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}) + b\right) \tag{11}$$

Given the dual representation, we can adopt a learning algorithm that works in the dual space described in Table 3.

Note that as the Novikoff's theorem states that the learning rate η only changes the scaling of the hyperplanes, it does not affect the algorithm thus we can set $\eta = 1$. On the contrary, if the perceptron algorithm starts with a different initialization, it will find a different separating hyperplane. The reader may wonder if such hyperplanes are all equivalent in terms of the classification accuracy of the test set; the answer is no: different hyperplanes may lead to different error probabilities. In particular, the next

Table 3. Dual perceptron algorithm

```
function Perceptron(training-point set: {x_1, .., x_m})
begin
   α = 0; b_0 = 0;
   R = max_{1≤i≤m} ||x_i||
   repeat
      no_errors = 1;
      for (i = 1 to m)
         if y_i( Σ_{j=1}^{m} α_j y_j (x_j · x) + b) ≤ 0 then
            α_i = α_i + 1;
            b = b + y_i R^2;
            no_errors = 0;
         end(if)
   until no_errors;
   return α and b ;
end
```

section shows that the maximal margin hyperplane minimizes an upperbound to the error probability on the space of all possible hyperplanes.

3.2 Maximal Margin Classifier

The PAC theory suggests that, for a class of target functions, a hypothesis h that is learned consistently with the training set provides low error probability and we can show an analytical bound for such error. This idea can be applied to hyperplanes to estimate the final error probability but also to improve the learning algorithm of linear classifiers. Indeed, one of the interesting results of the statistical learning theory is that to reduce such probability, we need to select the hyperplane (from the set of separating hyperplanes) that shows the maximum distance between positive and negative examples. To understand better this idea let us introduce some definitions:

Definition 5. *The **functional (geometric) margin distribution** of a hyperplane ($\boldsymbol{w}$,b) with respect to a training set S is the distribution of the functional (geometric) margins of the examples, i.e. $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \forall \boldsymbol{x}_i \in S$.*

Definition 6. *The **functional (geometric) margin of a hyperplane** is the minimum functional (geometric) margin of the distribution.*

Definition 7. *The **functional (geometric) margin of a training set** S is the maximum functional (geometric) margin of a hyperplane over all possible hyperplanes. The hyperplane that realizes such maximum is called the **maximal margin hyperplane**.*

Figure 11 shows the geometric margins of the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ (part A) and the geometric margin of the hyperplane (part B) whereas Figure 12 shows two separating hyperplanes that realize two different margins.

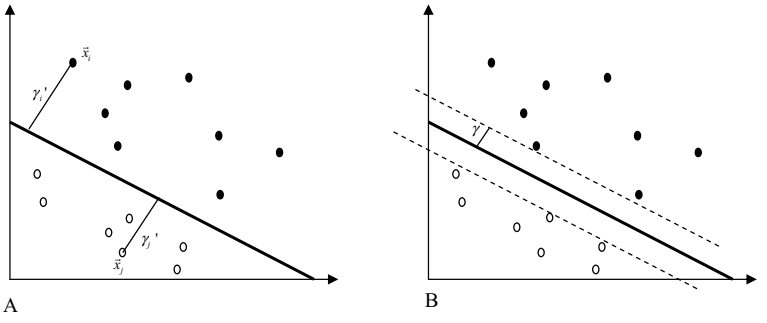

Fig. 11. Geometric margins of two points (part A) and margin of the hyperplane (part B)

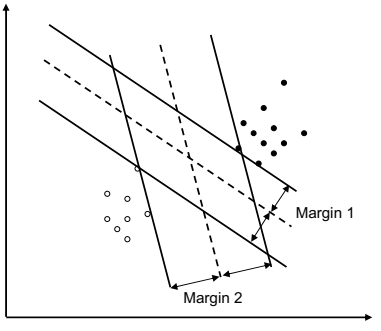

Fig. 12. Margins of two hyperplanes

Intuitively, the larger the margin of a hyperplane is, the lower the probability of error is. The important result of the statistical learning theory is that (i) analytical upperbounds to such error can be found; (ii) they can be proved to be correlated to the hyperplanes; and (iii) the maximal margin hyperplane is associated with the lowest bound.

In order to show such analytical result let us focus on finding the maximal margin hyperplane. Figure 13 shows that a necessary and sufficient condition for a hyperplane $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ to be a maximal margin hyperplane is that (a) two *frontier* hyperplanes (negative and positive frontiers) exist and (b) they hold the following properties:

1. their equations are $\boldsymbol{w} \cdot \boldsymbol{x} + b = k$ and $\boldsymbol{w} \cdot \boldsymbol{x} + b = -k$, i.e. they are parallel to the target hyperplane and are both located at a distance of k ($\frac{k}{||\boldsymbol{w}||}$ if $\boldsymbol{w}$ is not a normalized vector;
2. such equations satisfy the constraints $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq k \; \forall x_i \in S$, i.e. they both separate the data points in S; and
3. the distance of the hyperplane from such frontiers is maximal with respect to other frontiers.

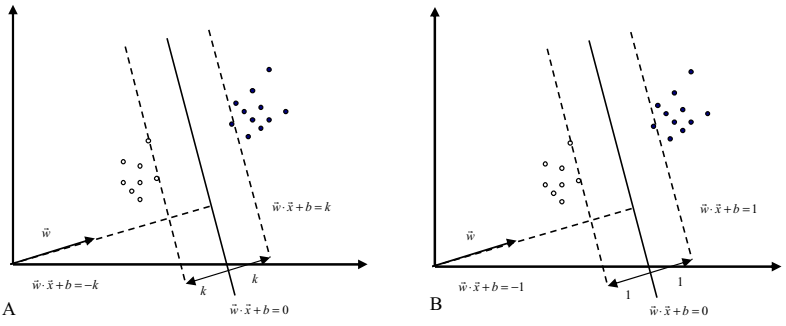

Fig. 13. Frontiers and Maximal Margin Hyperplane

First, property 1 follows from a simple consideration: suppose that: (i) the nearest positive example $\boldsymbol{x}^+$ is located at a distance of γ_i from a hyperplane h_1; (ii) the nearest negative example $\boldsymbol{x}^-$ is located at a distance of γ_j ($\neq \gamma_i$); and (iii) the h_1 margin is the minimum between γ_i and γ_j. If we select a hyperplane h_2 parallel to h_1 and equidistant from $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$, it will be at a distance of $k = \frac{\gamma_i+\gamma_j}{2}$ from both $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$. Since $k \geq min\{\gamma_i, \gamma_j\}$, the margin of h_2 equidistant from the frontier points is always greater or equal than other hyperplanes.

Second, the previous property has shown that the nearest positive examples is located on the frontier $\boldsymbol{w} \cdot \boldsymbol{x} + b = k$ thus all the other positive examples $\boldsymbol{x}^+$ have a functional margin $\boldsymbol{w} \cdot \boldsymbol{x}^+ + b$ larger than k. The same rational applies to the negative examples but, to work with positive quantities, we multiply $(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)$ by the label y_i, thus, we obtain the constrain $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b_k) \geq k$.

Finally, the third property holds since $\frac{k}{||\boldsymbol{w}||}$ is the distance from one of the two frontier hyperplanes which, in turn, is the distance from the nearest points, i.e. the margin.

From these properties, it follows that the maximal margin hyperplane can be derived by solving the optimization (maximization) problem below:

$$\begin{cases} max \quad \frac{k}{||\boldsymbol{w}||} \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 \quad \forall \boldsymbol{x}_i \in S, \end{cases} \tag{12}$$

where $\frac{k}{||\boldsymbol{w}||}$ is the objective function, $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) = 1 \quad \forall \boldsymbol{x}_i \in S$ are the set of linear equality constraints $h_i(\boldsymbol{w})$ and $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) > 1 \quad \forall \boldsymbol{x}_i \in S$ are the set of linear inequality constraints, $g_i(\boldsymbol{w})$. Note that (i) the objective function is quadratic since $||\boldsymbol{w}|| = \boldsymbol{w} \cdot \boldsymbol{w}$ and (ii) we can rescale the distance among the data points such that the maximal margin hyperplane has a margin of exactly 1. Thus, we can rewrite Eq. 12 as follows:

$$\begin{cases} max \quad \frac{1}{||\boldsymbol{w}||} \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 \quad \forall \boldsymbol{x}_i \in S \end{cases} \tag{13}$$

Moreover, we can transform the above maximization problem in the following minimization problem:

$$\begin{cases} min \quad ||\boldsymbol{w}|| \\ y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 \quad \forall \boldsymbol{x}_i \in S \end{cases} \tag{14}$$

Eq. 14 states that to obtain a maximal margin hyperplane, we have to minimize the norm of the gradient $\boldsymbol{w}$ but it does not provide any analytical evidence on the benefit of choosing such hyperplane. In contrast, the PAC learning theory provides the link with the error probability with the following theorem:

Theorem 6. *(Vapnik, 1982) Consider hyperplanes $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ in a $\mathbb{R}^n$ vector space as hypotheses. If all examples $\boldsymbol{x_i}$ are contained in a ball of radius R and*

$$\forall \boldsymbol{x}_i \in S, \quad y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1, \quad with \quad ||\boldsymbol{w}|| \leq A$$

then this set of hyperplanes has a VC-dimension d bounded by

$$d \leq min(R^2 \times A^2, n) + 1$$

The theorem states that if we set our hypothesis class H_A to be the set of hyperplanes whose $\boldsymbol{w}$ has a norm $\leq A$ then the VC dimension is less or equal than $R^2 \times A^2$. This means that if we reduce $||\boldsymbol{w}||$, we obtain a lower A and consequently a lower VC dimension, which in turn is connected to the error probability by the Theorem 4 (lower VC dim. results in lower error bound). This proves that, when the number of training examples is fixed, a lower VC-dimension will produce a lower error probability. In other words, as the maximum margin hyperplane minimizes the bound on the error probability, it constitutes a promising hypothesis for our learning problem.

Other interesting properties of the maximum margin hyperplane are derived from the optimization theory of convex functions over linear constraints. The main concepts of such theory relate on the following definition and theorem:

Definition 8. *Given an optimization problem with objective function $f(\boldsymbol{w})$, and equality constraints $h_i(\boldsymbol{w}) = 0$, $i = 1, .., l$, we define the Lagrangian function as*

$$L(\boldsymbol{w}, \boldsymbol{\beta}) = f(\boldsymbol{w}) + \sum_{i=1}^{l} \beta_i h_i(\boldsymbol{w}),$$

where the coefficient β_i are called Lagrange multipliers.

Theorem 7. *(Lagrange) A necessary condition for a normal point $\boldsymbol{w}^*$ to be a minimum of $f(w)$ subject to $h_i(\boldsymbol{w}) = 0$, $i = 1, .., l$, with f, $h_i \in C$ is*

$$\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{w}} = \mathbf{0} \tag{15}$$

$$\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} = \mathbf{0} \tag{16}$$

for some values of $\boldsymbol{\beta}^*$. The above conditions are also sufficient provided that $\partial L(\boldsymbol{\beta}^*)$ is a convex function of $\boldsymbol{w}$.

Proof. (necessity)
A continue function has a local maximum (minimum) when the partial derivatives are equal 0, i.e. $\frac{\partial f(\boldsymbol{w})}{\partial \boldsymbol{w}} = \mathbf{0}$. Since, we are in presence of constraints, it is possible that $\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}} \neq \mathbf{0}$. To respect such equality constraints, given the starting point $\boldsymbol{w}^*$, we can move only perpendicularly to $\frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$. In other words, we can only move perpendicularly to the subspace V spanned by the vectors $\frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}, i = 1, .., l$. Thus, if a point $\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$ lies on V, any direction we move causes to violate the constraints. In other words, if we start from such point, we cannot increase the objective function, i.e. it can be a minimum or maximum point. The V memberships can be stated as the linear dependence between $\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$ and $\frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}}$, formalized by the following equation:

$$\frac{\partial f(\boldsymbol{w}^*)}{\partial \boldsymbol{w}} + \sum_{i=1}^{l} \beta_i \frac{\partial h_i(\boldsymbol{w}^*)}{\partial \boldsymbol{w}} = \mathbf{0} \tag{17}$$

where $\exists i : \beta_i \neq 0$. This is exactly the condition 15. Moreover, Condition 16 holds since $\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} = (h_1(\boldsymbol{w}^*), h_2(\boldsymbol{w}^*), ..., h_l(\boldsymbol{w}^*))$ and all the constraints $h_i(\boldsymbol{w}^*) = 0$ are satisfied for the feasible solution $\boldsymbol{w}^*$. □

The above conditions can be applied to evaluate the maximal margin classifier, i.e. the Problem 14, but the general approach is to transform Problem 14 in an equivalent problem, simpler to solve. The output of such transformation is called dual problem and it is described by the following definition.

Definition 9. *Let $f(\boldsymbol{w})$, $h_i(\boldsymbol{w})$ and $g_i(\boldsymbol{w})$ be the objective function, the equality constraints and the inequality constraints (i.e. $\leq$) of an optimization problem, and let $L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ be its Lagrangian, defined as follows:*

$$L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = f(\boldsymbol{w}) + \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{w}) + \sum_{i=1}^{l} \beta_i h_i(\boldsymbol{w})$$

*The **Lagrangian dual problem** of the above primal problem is*

$$maximize \quad \theta(\boldsymbol{\alpha}, \boldsymbol{\beta})$$

$$subject \ to \quad \boldsymbol{\alpha} \geq \mathbf{0}$$

where $\theta(\boldsymbol{\alpha}, \boldsymbol{\beta}) = inf_{w \in W} \ L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$

The *strong duality theorem* assures that an optimal solution of the dual is also the optimal solution for the primal problem and vice versa, thus, we can focus on the transformation of Problem 14 according to the Definition 9.

First, we observe that the only constraints in Problem 14 are the inequalities[4] $[g_i(\boldsymbol{w}) = -(y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1)] \geq 0 \quad \forall \boldsymbol{x}_i \in S$.

[4] We need to change the sign of the inequalities to have them in the normal form, i.e. $g_i(\cdot) \leq 0$.

Second, the objective function is $\boldsymbol{w} \cdot \boldsymbol{w}$. Consequently, the primal Lagrangian[5] is

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} - \sum_{i=1}^{m} \alpha_i [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1], \tag{18}$$

where α_i are the Lagrange multipliers and b is the extra variable associated with the threshold.

Third, to evaluate $\theta(\boldsymbol{\alpha}, \boldsymbol{\beta}) = inf_{w \in W} \ L(\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$, we can find the minimum of the Lagrangian by setting the partial derivatives to 0.

$$\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\alpha})}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i \tag{19}$$

$$\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\alpha})}{\partial b} = \sum_{i=1}^{m} y_i \alpha_i = 0 \tag{20}$$

Finally, by substituting Eq. 19 and 20 into the primal Lagrangian we obtain

$$\begin{aligned} L(\boldsymbol{w}, b, \boldsymbol{\alpha}) &= \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} - \sum_{i=1}^{m} \alpha_i [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1] = \\ &= \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} - \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} + \sum_{i=1}^{m} \alpha_i \\ &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} \end{aligned} \tag{21}$$

which according to the Definition 9 is the optimization function of the dual problem subject to $\alpha_i \geq 0$. In summary, the final dual optimization problem is the following:

$$\begin{aligned} &maximize \quad \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} \\ &subject \ \ to \quad \alpha_i \geq 0, \quad i = 1, .., m \\ &\sum_{i=1}^{m} y_i \alpha_i = 0 \end{aligned}$$

where $\boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i$ and $\sum_{i=1}^{m} y_i \alpha_i = 0$ are the relation derived from eqs. 19 and 20. Other conditions establishing interesting properties can be derived by the Khun-Tucker theorem. This provides the following relations for an optimal solution:

[5] As $\boldsymbol{w} \cdot \boldsymbol{w}$ or $\frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w}$ is the same optimization function from a solution perspective, we use the $\frac{1}{2}$ factor to simplify the next computation.

$$\begin{aligned}
\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{w}} &= \mathbf{0} \\
\frac{\partial L(\boldsymbol{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} &= \mathbf{0} \\
\alpha_i^* g_i(\boldsymbol{w}^*) &= 0, \quad i = 1, .., m \\
g_i(\boldsymbol{w}^*) &\leq 0, \quad i = 1, .., m \\
\alpha_i^* &\geq 0, \quad i = 1, .., m
\end{aligned}$$

The third equation is usually called Karush-Khun-Tucker condition and it is very interesting for Support Vector Machines as it states that $\alpha_i^* \times [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1] = 0$. On one hand, if $\alpha_i^* = 0$ the training point $\boldsymbol{x}_i$ does not affect $\boldsymbol{w}$ as stated by Eq. 19. This means that the separating hyperplane and the associated classification function do not depend on such vectors. On the other hand, if $\alpha_i^* \neq 0 \Rightarrow [y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) - 1] = 0$ $\Rightarrow y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) = -1$, i.e. $\boldsymbol{x_i}$ is located on the frontier. Such data points are called support vectors (SV) as they support the classification function. Moreover, they can be used to derive the threshold b by evaluating the average between the projection of a positive and a negative SV on the gradient $\boldsymbol{w}^*$, i.e.:

$$b^* = -\frac{\boldsymbol{w}^* \cdot \boldsymbol{x}^+ + \boldsymbol{w}^* \cdot \boldsymbol{x}^-}{2}$$

The error probability upperbound of SVMs provides only a piece of evidence of the maximal margin effectiveness. Unfortunately, there is no analytical proof that such approach produces the *best* linear classifier. Indeed, it may exist other bounds lower than the one derived with the VC dimension and the related theory. Another drawback of the maximal margin approach is that it can only be applied when training data is linearly separable, i.e. the constraints over the negative and positive examples must be satisfied. Such $hard$ conditions also define the name of such model, i.e., Hard Margin Support Vector Machines. In contrast, the next section introduces the Soft Margin Support Vector Machines, whose optimization problem relaxes some constraints, i.e., a certain number of errors on the training set is allowed.

3.3 Soft Margin Support Vector Machines

In real scenario applications, training data is often affected by noise due to several reasons, e.g. classification mistakes of annotators. These may cause the data not to be separable by any linear function. Additionally, the target problem itself may be not separable in the designed feature space. As result, the Hard Margin SVMs will fail to converge.

In order to solve such critical aspect, the Soft Margin SVMs have been designed. Their main idea is to allow the optimization problem to provide solutions that can violate a certain number of constraints. Intuitively, to be as much as possible consistent with the training data, such number of errors should be the lowest possible. This trade-off between the separability with highest margin and the number of errors can be encoded by (a) introducing *slack variables* ξ_i in the inequality constraints of Problem 14 and (b) the number of errors as quantity to be minimized in the objective function. The resulting optimization problem is

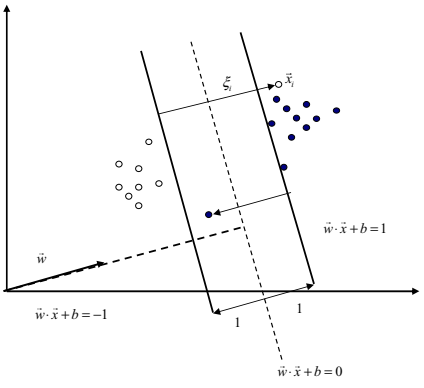

Fig. 14. Soft Margin Hyperplane

$$\begin{cases} min \quad ||\boldsymbol{w}|| + C\sum_{i=1}^{m}\xi_i^2 \\ y_i(\boldsymbol{w}\cdot\boldsymbol{x_i}+b) \geq 1-\xi_i, \quad \forall i = 1,..,m \\ \xi_i \geq 0, \quad i = 1,..,m \end{cases} \tag{22}$$

whose the main characteristics are:

- The constraint $y_i(\boldsymbol{w}\cdot\boldsymbol{x_i}+b) \geq 1-\xi_i$ allows the point $\boldsymbol{x}_i$ to violate the hard constraint of Problem 14 of a quantity equal to ξ_i. This is clearly shown by the outliers in Figure 14, e.g. $\boldsymbol{x}_i$.
- If a point is misclassified by the hyperplane then the slack variable assumes a value larger than 1. For example, Figure 14 shows the misclassified point x_i and its associated slack variable ξ_i, which is necessarily > 1. Thus, $\sum_{i=1}^{m}\xi_i$ is an upperbound to the number of errors. The same property is held by the quantity, $\sum_{i=1}^{m}\xi_i^2$, which can be used as an alternative bound[6].
- The constant C tunes the trade-off between the classification errors and the margin. The higher C is, the lower number of errors will be in the optimal solution. For $C \rightarrow \infty$, Problem 22 approximates Problem 14.
- Similarly to the hard margin formulation, it can be proven that minimizing $||\boldsymbol{w}|| + C\sum_{i=1}^{m}\xi_i^2$ minimizes the error probability of classifiers. Even though these are not perfectly consistent with the training data (they do not necessarily classify correctly all the training data).
- Figure 15 shows that by accepting some errors, it is possible to find better hypotheses. In the part A, the point $\boldsymbol{x}_i$ prevents to derive a good margin. As we accept to mistake $\boldsymbol{x}_i$, the learning algorithm can find a more suitable margin (part B).

As it has been done for the hard optimization problem, we can evaluate the primal Lagrangian:

$$L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{w}\cdot\boldsymbol{w} + \frac{C}{2}\sum_{i=1}^{m}\xi_i^2 - \sum_{i=1}^{m}\alpha_i[y_i(\boldsymbol{w}\cdot\boldsymbol{x_i}+b) - 1 + \xi_i], \tag{23}$$

[6] This also results in an easier mathematical solution of the optimization problem.

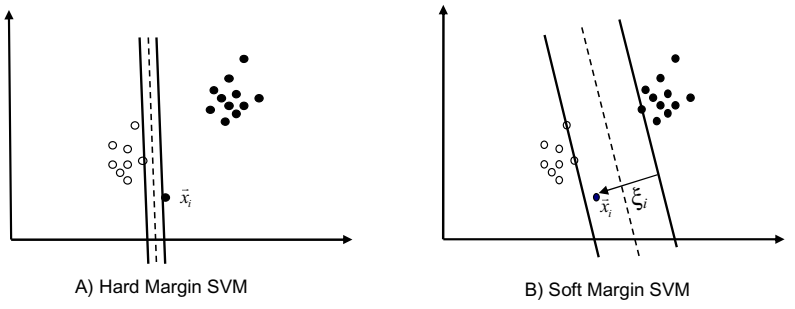

Fig. 15. Soft Margin vs. Hard Margin hyperplanes

where α_i are Lagrangian multipliers.

The dual problem is obtained by imposing stationarity on the derivatives respect to $\boldsymbol{w}$, $\boldsymbol{\xi}$ and b:

$$\begin{aligned}
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha})}{\partial \boldsymbol{w}} &= \boldsymbol{w} - \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{x}_i \\
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha})}{\partial \boldsymbol{\xi}} &= C\boldsymbol{\xi} - \boldsymbol{\alpha} = \boldsymbol{0} \\
\frac{\partial L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha})}{\partial b} &= \sum_{i=1}^{m} y_i \alpha_i = 0
\end{aligned} \tag{24}$$

By substituting the above relations into the primal, we obtain the following dual objective function:

$$\begin{aligned}
w(\boldsymbol{\alpha}) &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} + \frac{1}{2C} \boldsymbol{\alpha} \cdot \boldsymbol{\alpha} - \frac{1}{C} \boldsymbol{\alpha} \cdot \boldsymbol{\alpha} = \\
&= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} - \frac{1}{2C} \boldsymbol{\alpha} \cdot \boldsymbol{\alpha} = \\
&= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \big(\boldsymbol{x_i} \cdot \boldsymbol{x_j} + \frac{1}{C} \delta_{ij} \big),
\end{aligned} \tag{25}$$

where the Kronecker's delta, δ_{ij} = 1 if $i = j$ and 0 otherwise. The objective function above is subject to the usual constraints:

$$\begin{cases} \alpha_i \geq 0, \quad \forall i = 1, .., m \\ \sum_{i=1}^{m} y_i \alpha_i = 0 \end{cases}$$

This dual formulation can be used to find a solution of Problem 22, which extends the applicability of linear functions to classification problems not completely linearly separable. The separability property relates not only to the available class of hypotheses, e.g. linear vs. polynomial functions, but it strictly depends on the adopted features. Their

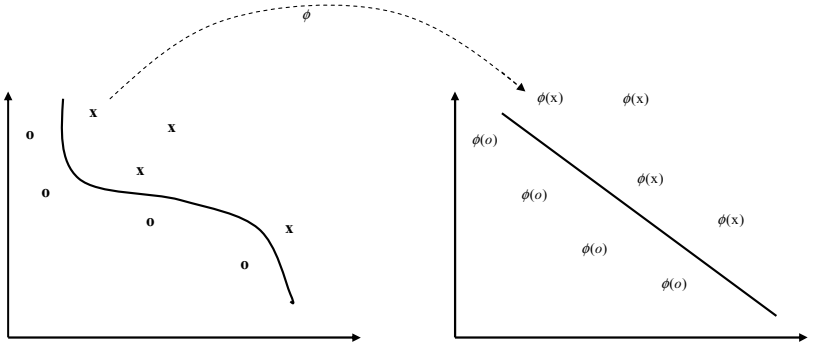

Fig. 16. A mapping ϕ that makes separable the initial data points

roles is to provide a map between the examples and vectors in $\mathbb{R}^n$. Given such mapping, the scalar product provides a measure of the *similarity* between pairs of examples or, according to a more minimalist interpretation, it provides a partitioning function based on such features.

The next section shows that, it is possible to directly substitute the scalar product of two feature vectors with a similarity function between the data examples. This allows for avoiding explicit feature design and consequently enabling the use of similarly measures called kernel functions. These, in turn, define implicit feature spaces.

4 Kernel Methods

One of the most difficult step on applying machine learning is the feature design. Features should represent data in a way that allows learning algorithms to separate positive from negative examples. The features used by SVMs are used to build vector representations of data examples and the scalar product between them. This, sometimes, simply counts the number of common features to measure how much the examples are similar. Instead of encoding data in feature vectors, we may design kernel functions that provide such similarity between examples avoiding the use of explicit feature representations. The reader may object that the learning algorithm still requires the supporting feature space to model the hyperplane and the data points, but this is not necessary if the optimization problem is solved in the dual space.

The real limit of the kernel functions is that they must generate a well defined inner product vector space. Such property will hold if the Mercer's conditions are satisfied. Fortunately, there are many kernels, e.g. polynomial, string, lexical and tree kernels that satisfy such conditions and give us the possibility to use them in SVMs.

Kernels allow for more abstractly defining our leaning problems and in many cases allow for solving non linear problems by re-mapping the initial data points in a separable space as shown by Figure 16. The following example illustrates one of the case in which a non-linear function can be expressed in a linear formulation in a different space.

Example 2. Overcoming linear inseparability
Suppose that we want to study the force of interactions between two masses m_1 and m_2. m_1 is free to move whereas m_2 is blocked. The distance between the two masses is indicated with r and their are subject to the Newtown's gravity law:

$$f(m_1, m_2, r) = C\frac{m_1 m_2}{r^2},$$

Thus mass m_1 naturally tends to move towards m_2.

We apply a force f_a of inverse direction with respect to f to m_1. As a result, we note that sometimes m_1 approaches m_2 whereas other times it gets far from it. To study such phenomenon, we carry out a set of experiments with different experimental parameters, i.e. m_1, m_2, r and f_a and we annotate the result of our action: success if m_1 gets closer to m_2 (or does not move) and failure otherwise.

Each successful experiment can be considered a positive example whereas unsuccessful experiments are considered negative examples. The parameters above constitute feature vectors representing an experiment. We can apply SVMs to learn the classification of new experiments $\langle f_a, m_1, m_2, r\rangle$ in successful or unsuccessful, i.e. if $f_a - f(m_1, m_2, r) \geq 0$ or otherwise, respectively. This means that SVMs have to learn the gravitational law function, $f(m_1, m_2, r)$, but, since this is clearly non-linear, hard margin SVMs will not generally converge and soft margin SVMs will provide inaccurate results.

The solution for this problem is to map our initial feature space in another vector space, i.e. $\langle f_a, m_1, m_2, r\rangle \rightarrow \langle ln f_a, ln(m_1), ln(m_2), ln(r)\rangle = \langle k, x, y, z\rangle$. Since $ln\big(f(m_1, m_2, r)\big) = ln(C) + ln(m_1) + ln(m_2) - 2ln(r) = c + x + y - 2z$, we can express the logarithm of gravity law with a linear combination of the transformed features in the new space. In more detail, points above (or lying on) the ideal hyperplane $k-(c+x+y-2z) = 0$, i.e. points that satisfy $k-(c+x+y-2z) \geq 0$ (or equivalently that satisfy $f_a - f(m_1, m_2, r) \geq 0$), are successful experiments whereas points below such hyperplane are unsuccessful. The above passages prove that a separating hyperplane of the training set always exists in the transformed space, consequently SVMs will always converge (with an error dependent on the number of training examples).

4.1 The Kernel Trick

Section 3.1 has shown that the Perceptron algorithm, used to learn linear classifiers, can be adapted to work in the dual space. In particular, such algorithm (see Table 3) clearly shows that it only exploits feature vectors in the form of scalar product. Consequently, we can replace feature vectors $\boldsymbol{x}_i$ with the data objects o_i, substituting the scalar product $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$ with a kernel function $k(o_i, o_j)$, where o_i are the initial objects mapped into $\boldsymbol{x}_i$ using a feature representation, $\phi(.)$. This implies that $\boldsymbol{x}_i \cdot \boldsymbol{x}_j = \phi(o_i) \cdot \phi(o_j) = k(o_i, o_j)$.

Similarly to the Perceptron algorithm, the dual optimization problem of Soft Margin SVMs (Eq. 25) uses feature vectors only inside a scalar product, which can be substituted with $k(o_i, o_j)$. Therefore, the kernelized version of the soft margin SVMs is

$$\begin{cases} maximize \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \big(k(o_i, o_j) + \frac{1}{C}\delta_{ij}\big) \\ \alpha_i \geq 0, \quad \forall i = 1, .., m \\ \sum_{i=1}^{m} y_i \alpha_i = 0 \end{cases}$$

Moreover, Eq. 10 for the Perceptron appears also in the Soft Margin SVMs (see conditions 24), hence we can rewrite the SVM classification function as in Eq. 11 and use a kernel inside it, i.e.:

$$h(x) = sgn\left(\sum_{i=1}^{m} \alpha_i y_i k(o_i, o_j) + b\right)$$

The data object *o* is mapped in the vector space trough a feature extraction procedure $\phi : o \rightarrow (x_1, ..., x_n) = \boldsymbol{x}$, more in general, we can map a vector $\boldsymbol{x}$ from one feature space into another one:

$$\boldsymbol{x} = (x_1, ..., x_n) \rightarrow \boldsymbol{\phi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), ..., \phi_n(\boldsymbol{x}))$$

This leads to the general definition of kernel functions:

Definition 10. *A kernel is a function k, such that* $\forall\ \boldsymbol{x},\boldsymbol{z} \in X$

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{\phi}(\boldsymbol{x}) \cdot \boldsymbol{\phi}(\boldsymbol{z})$$

where $\boldsymbol{\phi}$ *is a mapping from* X *to an (inner product) feature space.*

Note that, once we have defined a kernel function that is effective for a given learning problem, we do not need to find which mapping ϕ corresponds to. It is enough to know that such mapping exists. The following proposition states the conditions that guarantee such existence.

Proposition 1. *(Mercer's conditions)*
Let X be a finite input space and let $K(\boldsymbol{x}, \boldsymbol{z})$ *be a symmetric function on X. Then* $K(\boldsymbol{x}, \boldsymbol{z})$ *is a kernel function if and only if the matrix*

$$k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{\phi}(\boldsymbol{x}) \cdot \boldsymbol{\phi}(\boldsymbol{z})$$

is positive semi-definite (has non-negative eigenvalues).

Proof. Let us consider a symmetric function on a finite space $X = \{x_1, x_2, ..., x_n\}$

$$\boldsymbol{K} = \big(K(x_i, x_j)\big)_{i,j=1}^{n}$$

Since $\boldsymbol{K}$ is symmetric there is an orthogonal matrix $\boldsymbol{V}$ such that $\boldsymbol{K} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}'$ where $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues λ_t of $\boldsymbol{K}$, with corresponding

eigenvectors $\boldsymbol{v}_t = (v_{ti})_{i=1}^n$, i.e., the columns of $\boldsymbol{V}$. Now assume that all the eigenvalues are non-negatives and consider the feature mapping:

$$\boldsymbol{\phi} : \boldsymbol{x}_i \rightarrow \left(\sqrt{\lambda_t} v_{ti}\right)_{t=1}^n \in \mathbb{R}^n, i = 1, .., n.$$

It follows that

$$\boldsymbol{\phi}(\boldsymbol{x}_i) \cdot \boldsymbol{\phi}(\boldsymbol{x}_j) = \sum_{t=1}^{n} \lambda_t v_{ti} v_{tj} = (\boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}')_{ij} = \boldsymbol{K}_{ij} = K(x_i, x_j).$$

This proves that $K(\boldsymbol{x}, \boldsymbol{z})$ is a *valid* kernel function that corresponds to the mapping $\boldsymbol{\phi}$. Therefore, the only requirement to derive the mapping $\boldsymbol{\phi}$ is that the eigenvalues of $\boldsymbol{K}$ are non-negatives since if we had a negative eigenvalue λ_s associated with the eigenvector $\boldsymbol{v}_s$, the point

$$\boldsymbol{z} = \sum_{i=1}^{n} \boldsymbol{v}_{si} \phi(\boldsymbol{x}_i) = \sqrt{\boldsymbol{\Lambda}}\boldsymbol{V}'\boldsymbol{v}_s.$$

in the feature space would have norm squared

$$||\boldsymbol{z}||^2 = \boldsymbol{z} \cdot \boldsymbol{z} = \boldsymbol{v}_s'\boldsymbol{V}\sqrt{\boldsymbol{\Lambda}}\sqrt{\boldsymbol{\Lambda}}\boldsymbol{V}'\boldsymbol{v}_s = \boldsymbol{v}_s'\boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}'\boldsymbol{v}_s = \boldsymbol{v}_s'\boldsymbol{K}\boldsymbol{v}_s = \lambda_s < 0,$$

which contradicts the geometry of the space [20].

4.2 Polynomial Kernel

The above section has shown that kernel functions can be used to map a vector space in other spaces in which the target classification problem becomes linearly separable (or in general easier). Another advantage is the possibility to map the initial feature space in a richer space which includes a high number of dimensions (possibly infinite): this may result in a better description of the objects and higher accuracy. For example, the polynomial kernel maps the initial features in a space which contains both the original features and all the possible feature conjunctions. For example, given the components x_1 and x_2, the new space will contain $x_1 x_2$. This is interesting for text categorization as the polynomial kernel automatically derives the feature `hard rock` or `hard disk` from the individual features `hard`, `rock` and `disk`. The conjunctive features may help to disambiguate between *Music Store* and *Computer Store* categories.

The great advantage of using kernel functions is that we do not need to keep the vectors of the new space in the computer memory to evaluate the inner product. For example, suppose that the initial feature space has a cardinality of 100,000 features, i.e., a typical size of the vocabulary in a text categorization problem, only the number of word pairs would be 10^{10}, which cannot be managed by many learning algorithms. The polynomial kernel can be used to evaluate the scalar product between pairs of vectors of such huge space by only using the initial space and vocabulary, as it is shown by the following passages:

$$
\begin{aligned}
(\boldsymbol{x}\cdot\boldsymbol{z})^2 = \big(\sum_{i=1}^{n} x_i z_i\big)^2 \quad &= \big(\sum_{i=1}^{n} x_i z_i\big)\big(\sum_{j=1}^{n} x_i z_i\big) \\
= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j &= \sum_{i,j\in\{1,..,n\}} (x_i x_j)(z_i z_j) \\
= \sum_{k=1}^{m} X_k Z_k \quad &= \boldsymbol{X}\cdot\boldsymbol{Z},
\end{aligned}
$$

where:

- $\boldsymbol{x}$ and $\boldsymbol{z}$ are two vectors of the initial space,
- $\boldsymbol{X}$ and $\boldsymbol{Z}$ are the vectors of the final space and
- $X_k = x_i x_j$, $Z_k = z_i z_j$ with $k = (i-1)\times n + j$ and $m = n^2$.

We note that

- the mapping between the two space is $\phi(\boldsymbol{x}) = (x_i x_j)$ for $j = 1,..,n$ and for $i = 1,..,n$;
- to evaluate $\boldsymbol{X}\cdot\boldsymbol{Z}$, we just compute the square of the scalar product in the initial space, i.e. $(\boldsymbol{x}\cdot\boldsymbol{z})^2$; and
- the final space contains conjunctions and also the features of the initial space ($x_i x_i$ is equivalent to x_i).

Additionally, since $x_i x_j = x_j x_i$, the conjunctions receive the double of the weight of single features. The number of distinct features are: n for $i = 1$ and $j = 1,..,n$; $(n-1)$ for $i = 2$ and $j = 2,..,n$; ..; and 1 for $i = n$ and $j = n$. It follows that the total number of terms is

$$
n + (n-1) + (n-2) + .. + 1 = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}
$$

Another way to compute such number it to consider that, to build all the monomials, the first variable can be chosen out of $n+1$ possibilities (n symbols to form conjunctions and the empty symbol for the single feature) whereas for the second variable only n chances are available (no empty symbol at this time). This way, we obtain all permutations of each monomial of two variables. To compute the number of distinct features, we can divide the number of monomials, i.e. $(n+1)n$, by the number of permutations of two variables, i.e. $2! = 2$. The final quantity can be expressed with the binomial coefficient $\binom{n+1}{2}$.

Given the above observation, we can generalize the kernel from degree 2 to a degree d by computing $(\boldsymbol{x}\cdot\boldsymbol{z})^d$. The results are all monomials of degree d or equivalently all the conjunctions constituted up to d features. The distinct features will be $\binom{n+d-1}{d}$ since we can choose either the empty symbol up to $d-1$ times or n variables.

A still more general kernel can be derived by introducing a constant in the scalar product computation. Hereafter, we show the case for a degree equal to two:

$$(\boldsymbol{x}\cdot\boldsymbol{z}+c)^2=\big(\sum_{i=1}^{n}x_iz_i+c\big)^2=\big(\sum_{i=1}^{n}x_iz_i+c\big)\big(\sum_{j=1}^{n}x_iz_i+c\big)=$$
$$=\sum_{i=1}^{n}\sum_{j=1}^{n}x_ix_jz_iz_j+2c\sum_{i=1}^{n}x_iz_i+c^2=$$
$$=\sum_{i,j\in\{1,..,n\}}(x_ix_j)(z_iz_j)+\sum_{i=1}^{n}\big(\sqrt{2c}x_i\big)\big(\sqrt{2c}z_i\big)+c^2$$

Note that the second summation introduces n individual features (i.e. x_i) whose weights are controlled by the parameter c which also determines the strength of the degree 0. Thus, we add (n+1) new features to the $\binom{n+1}{2}$ features of the previous kernel of degree 2. If we consider a generic degree d, i.e. the kernel $(\boldsymbol{x}\cdot\boldsymbol{z}+c)^d$, we will obtain $\binom{n+d-1}{d}+n+d-1=\binom{n+d}{d}$ distinct features (which have at least distinct weights). These are all monomials up to and including the degree d.

4.3 String Kernel

Kernel functions can be also applied to discrete space. As a first example, we show their potentiality on the space of finite strings.

Let Σ be a finite alphabet. A string is a finite sequence of characters from Σ, including the empty sequence. We denote by $|s|$ the length of the string $s=s_1,..,s_{|s|}$, where s_i are symbols, and by st the string obtained by concatenating the strings s and t. The string $s[i:j]$ is the substring $s_i,..,s_j$ of s. We say that u is a subsequence of s, if there exist indices $\boldsymbol{I}=(i_1,...,i_{|u|})$, with $1\le i_1<...<i_{|u|}\le|s|$, such that $u_j=s_{i_j}$, for $j=1,...,|u|$, or $u=s[\boldsymbol{I}]$ for short. The length $l(\boldsymbol{I})$ of the subsequence in s is $i_{|u|}-i_i+1$. We denote by Σ^* the set of all string

$$\Sigma^*=\bigcup_{n=0}^{\infty}\Sigma^n$$

We now define the feature space, $F=\{u_1,u_2..\}=\Sigma^*$, i.e. the space of all possible substrings. We map a string s in $\mathbb{R}^\infty$ space as follows:

$$\phi_u(s)=\sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]}\lambda^{l(\boldsymbol{I})} \tag{26}$$

for some $\lambda\le 1$. These features measure the number of occurrences of subsequences in the string s, weighting them according to their lengths. Hence, the inner product of the feature vectors for two strings s and t gives a sum over all common subsequences weighted according to their frequency of occurrences and lengths, i.e.

$$K(s,t)=\sum_{u\in\Sigma^*}\phi_u(s)\cdot\phi_u(t)=\sum_{u\in\Sigma^*}\sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]}\lambda^{l(\boldsymbol{I})}\sum_{\boldsymbol{J}:u=t[\boldsymbol{J}]}\lambda^{l(\boldsymbol{J})}=$$

$$=\sum_{u\in\Sigma^*}\sum_{\boldsymbol{I}:u=s[\boldsymbol{I}]}\sum_{\boldsymbol{J}:u=t[\boldsymbol{J}]}\lambda^{l(\boldsymbol{I})+l(\boldsymbol{J})} \tag{27}$$

The above equation defines a class of similarity functions known as string kernels or sequence kernels. These functions are very effective for extracting features from streams. For example, in case of text categorization, they allow the learning algorithm to quantify the matching between two different words, phrases, sentences or whole documents. Given two strings, *Bank* and *Rank*:

- B, a, n, k, Ba, Ban, Bank, an, ank, nk, Bn, Bnk, Bk and ak are substrings of $Bank$.
- R, a, n, k, Ra, Ran, Rank, an, ank, nk, Rn, Rnk, Rk and ak are substrings of $Rank$.

Such substrings are features in the Σ^* that have non-null weights. These are evaluated by means of Eq. 26, e.g. $\phi_{\mathtt{B}}(\mathtt{Bank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(1-1+1)} = \lambda$, $\phi_{\mathtt{k}}(\mathtt{Bank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(4-4+1)} = \lambda$, $\phi_{\mathtt{an}}(\mathtt{Bank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(3-2+1)} = \lambda^2$ and $\phi_{\mathtt{Bk}}(\mathtt{Bank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(4-1+1)} = \lambda^4$.

Since Eq. 27 requires that the substrings in *Bank* and *Rank* match, we need to evaluate Eq. 26 only for the common substrings, i.e.:

- $\phi_{\mathtt{a}}(\mathtt{Bank}) = \phi_{\mathtt{a}}(\mathtt{Rank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(2-2+1)} = \lambda$,
- $\phi_{\mathtt{n}}(\mathtt{Bank}) = \phi_{\mathtt{n}}(\mathtt{Rank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(3-3+1)} = \lambda$,
- $\phi_{\mathtt{k}}(\mathtt{Bank}) = \phi_{\mathtt{k}}(\mathtt{Rank}) = \lambda^{(i_1-i_1+1)} = \lambda^{(4-4+1)} = \lambda$,
- $\phi_{\mathtt{an}}(\mathtt{Bank}) = \phi_{\mathtt{an}}(\mathtt{Rank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(3-2+1)} = \lambda^2$,
- $\phi_{\mathtt{ank}}(\mathtt{Bank}) = \phi_{\mathtt{ank}}(\mathtt{Rank}) = \lambda^{(i_3-i_1+1)} = \lambda^{(4-2+1)} = \lambda^3$,
- $\phi_{\mathtt{nk}}(\mathtt{Bank}) = \phi_{\mathtt{nk}}(\mathtt{Rank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(4-3+1)} = \lambda^2$,
- $\phi_{\mathtt{ak}}(\mathtt{Bank}) = \phi_{\mathtt{ak}}(\mathtt{Rank}) = \lambda^{(i_2-i_1+1)} = \lambda^{(4-2+1)} = \lambda^3$.

It follows that $K(\mathtt{Bank}, \mathtt{Rank}) = (\lambda, \lambda, \lambda, \lambda^2, \lambda^3, \lambda^2, \lambda^3) \cdot (\lambda, \lambda, \lambda, \lambda^2, \lambda^3, \lambda^2, \lambda^3)$ $= 3\lambda^2 + 2\lambda^4 + 2\lambda^6$.

From this example, we note that short non-discontinuous strings receive the highest contribution, e.g. $\phi_{\mathtt{B}}(\mathtt{Bank}) = \lambda > \phi_{\mathtt{an}}(\mathtt{Bank}) = \lambda^2$. This may appear counterintuitive as longer string should be more important to characterize two textual snippets. Such inconsistency disappears if we consider that when a large string is matched, the same will happen for all its substrings. For example, the contribution coming from $Bank$, in the matching between the "$Bank\ of\ America$" and "$Bank\ of\ Italy$" strings, includes the match of B, a, n, k, Ba, Ban,..., an so on.

Moreover, it should be noted that Eq. 27 is rather expensive from a computational viewpoint. A method for its fast computation trough a recursive function was proposed in [38].

First, a kernel over the space of strings of length n, Σ^n is computed, i.e.

$$K_n(s,t) = \sum_{u \in \Sigma^n} \phi_u(s) \cdot \phi_u(t) = \sum_{u \in \Sigma^n} \sum_{\boldsymbol{I}: u = s[\boldsymbol{I}]} \sum_{\boldsymbol{J}: u = t[\boldsymbol{J}]} \lambda^{l(\boldsymbol{I}) + l(\boldsymbol{J})}.$$

Second, a slightly different version of the above function is considered, i.e.

$$K'_i(s,t) = \sum_{u \in \Sigma^n} \phi_u(s) \cdot \phi_u(t) = \sum_{u \in \Sigma^i} \sum_{\boldsymbol{I}: u = s[\boldsymbol{I}]} \sum_{\boldsymbol{J}: u = t[\boldsymbol{J}]} \lambda^{|s| + |t| - i_1 - j_1 + 2},$$

for $i = 1, .., n-1$. $K'_i(s,t)$ is different than $K_n(s,t)$ since, to assign weights, the distances from the initial character of the substrings to the end of the string, i.e. $|s| - i_1 + 1$ and $|t| - j_1 + 1$, are used in place of the distances between the first and last characters of the substrings, i.e. $l(\boldsymbol{I})$ and $l(\boldsymbol{J})$.

It can be proved that $K_n(s,t)$ is evaluated by the following recursive relations:

- $K'_0(s,t) = 1$, for all s,t,
- $K'_i(s,t) = 0$, if min ($|s|$,$|t|$)$< i$,
- $K_i(s,t) = 0$, if min ($|s|$,$|t|$)$< i$,
- $K'_i(sx,t) = \lambda K'_i(s,t) + \sum_{j:t_j=x} K'_{i-1}(s, t[1:j-1])\lambda^{|t|-j+2}, i = 1, ..,n-1$,
- $K_n(sx,t) = K_n(s,t) + \sum_{j:t_j=x} K'_{n-1}(s, t[1:j-1])\lambda^2$.

The general idea is that $K'_{i-1}(s,t)$ can be used to compute $K_n(s,t)$ when we increase the size of the input strings of one character, e.g. $K_n(sx,t)$. Indeed, K'_i and K_i compute the same quantity when the last character of the substring $u \in \Sigma^i$, i.e. x, coincides with the last character of the string, i.e. the string can be written as sx. Since $K'_i(sx,t)$ can be reduced to $K'_i(s,t)$, the recursion relation is valid. The computation time of such process is proportional to $n \times |s| \times |t|$, i.e. an efficient evaluation.

4.4 Lexical Kernel

The most used Information Retrieval (IR) paradigm is based on the assumptions that (a) the semantic of a document can be represented by the semantic of its words and (b) to express the similarity between document pairs, it is enough to only consider the contribution from matching terms. In this view, two words that are strongly related, e.g. synonyms, do not contribute with their *relatedness* to the document similarity.

More advanced IR models attempt to take the above problem into account by introducing term similarities. Complex and interesting term similarities can be implemented using external (to the target corpus) thesaurus, like for example the Wordnet hierarchy [26]. For example, the terms *mammal* and *invertebrate* are under the term *animal* in such hierarchy. In turns, the terms *dog* and *cat*, are under the term *mammal*. The length of the path that connects two terms in such hierarchy intuitively provides a sort of similarity metrics. Once a term relatedness is designed, document similarities, which are the core functions of most Text Categorization algorithms, can be designed as well.

Given a term similarity function σ and two documents d_1 and $d_2 \in D$ (the document set), we define their similarity as:

$$K(d_1, d_2) = \sum_{f_1 \in d_1, f_2 \in d_2} (w_1 w_2) \times \sigma(f_1, f_2) \tag{28}$$

where w_1 and w_2 are the weights of the words (features) f_1 and f_2 in the documents d_1 and d_2, respectively. Interestingly such similarity can be a valid kernel function and, therefore, used in SVMs. To prove this we need to verify the Mercer's conditions, i.e. that the associated kernel matrix (see Proposition 1) is positive semi-definite. We can apply single value decomposition and check the eigenvalues. In case we find that some

of them are negative, we can still use the lexical kernel by squaring its associated matrix. Indeed, the kernel $K(d_1, d_2)$ can be written as $\boldsymbol{P} = \boldsymbol{M}' \cdot \boldsymbol{M}$, where $\boldsymbol{M}$ is the matrix defined by $\sigma(f_1, f_2)$ and $\boldsymbol{M}'$ is its transposed. Since $\boldsymbol{P}$ is surely positive semi-definite (it is a square), $K(d_1, d_2) = \boldsymbol{P}$ satisfies the Mercer's conditions.

The lexical kernel has been successfully applied to improve document categorization [8] when few documents are available for training. Indeed, the possibility to match different words using a σ similarity allows SVMs to recover important semantic information.

5 Tree Kernel Spaces

The polynomial and the string kernels have shown that, starting from an initial feature set, they can automatically provide a very high number of interesting features. These are a first example of the usefulness of kernel methods. Other interesting kernel approaches aim to automatically generate large number of features from structures. For example, tree kernels are able to extract many types of tree fragments from a target tree. One of their purposes is to model syntactic information in a target learning problem. In particular, tree kernels seem well suited to model syntax in natural language applications, e.g. for the extraction of semantic predicative structures like *bought(Mary, a cat, in Rome)* [54].

Indeed, previous work shows that defining linguistic theories for the modeling of natural languages (e.g. [35]) is a complex problem, far away from a sound and complete solution, e.g. the links between syntax and semantic are not completely understood yet. This makes the design of syntactic features for the automatic learning of semantic structures complex and consequently both remarkable deep knowledge about the target linguistic phenomena and research effort are required.

Kernel methods, which do not require any noticeable feature design effort, can provide the same accuracy of manually designed features and sometime they can suggest new solutions to the designer to improve the model of the target linguistic phenomenon.

The kernels that we consider in next sections represent trees in terms of their substructures (fragments). Their are based on the general notion of convolution kernels hereafter reported.

Definition 11. *General Convolution Kernels*
Let $X, X_1, .., X_m$ be separable metric spaces, $x \in X$ a structure and $\boldsymbol{x} = x_1, ..., x_m$ its parts, where $x_i \in X_i \quad \forall i = 1, .., m$. Let R be a relation on the set $X \times X_1 \times .. \times X_m$ such that $R(\boldsymbol{x}, x)$ holds if $\boldsymbol{x}$ are the parts of x. We indicate with $R^{-1}(x)$ the set $\{\boldsymbol{x} : R(\boldsymbol{x}, x)\}$. Given two objects x and $y \in X$, their similarity $K(x, y)$ is defined as:

$$K(x, y) = \sum_{\boldsymbol{x} \in R^{-1}(x)} \sum_{\boldsymbol{y} \in R^{-1}(y)} \prod_{i=1}^{m} K_i(x_i, y_i) \tag{29}$$

Subparts or fragments define a feature space which, in turn, is mapped into a vector space, e.g. $\mathbb{R}^n$. In case of tree kernels, the similarity between trees is given by the number of common tree fragments. These functions detect if a common tree subpart

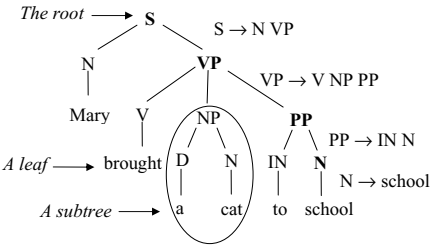

Fig. 17. A syntactic parse tree

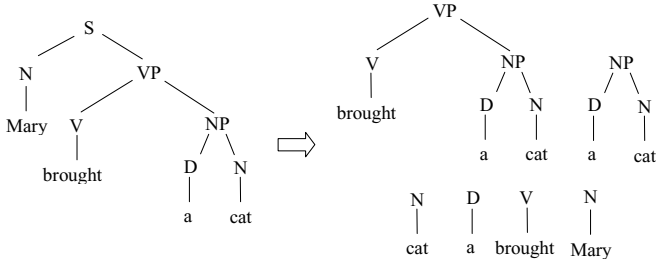

Fig. 18. A syntactic parse tree with its SubTrees (STs)

belongs to the feature space that we intend to generate. For such purpose, the fragment type needs to be described. We consider three important characterizations: the SubTrees (STs), the SubSet Trees (SSTs) and the Partial Trees (PTs).

5.1 SubTree, SubSet Tree and Partial Tree Kernels

Trees are directed, connected acyclic graphs with a special node called root. Their recursive definition is the following: (1) the root node, connected with one or more nodes (called children), is a tree and (2) a child can be a tree, i.e. a SubTree, or a node without children, i.e. a leaf.

In case of syntactic parse trees each node with its children is associated with a grammar production rule, where the symbol at left-hand side corresponds to the parent and the symbols at right-hand side are associated with the children. The terminal symbols of the grammar are always associated with the leaves of the tree. For example, Figure 17 illustrates the syntactic parse of the sentence `"Mary brought a cat to school".`

We define a **SubTree** (ST) as any node of a tree along with all its descendants. For example, the line in Figure 17 circles the SubTree rooted in the `NP` node. A **SubSet Tree** (SST) is a more general structure which not necessarily includes all its descendants. The only restriction is that an SST must be generated by applying the same grammatical rule set that generated the original tree, as pointed out in [19]. Thus, the difference with the SubTrees is that the SST's leaves can be associated with non-terminal symbols. For example, `[S [N VP]]` is an SST of the tree in Figure 17 and it has the two non-terminal symbols `N` and `VP` as leaves.

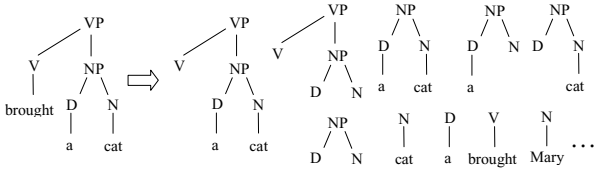

Fig. 19. A tree with some of its SubSet Trees (SSTs)

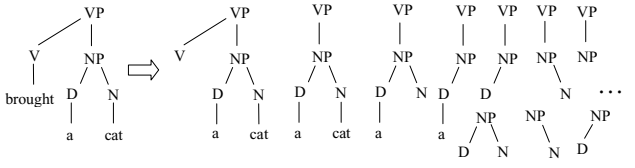

Fig. 20. A tree with some of its Partial Trees (PTs)

If we relax the constraint over the SSTs, we obtain a more general form of substructures that we defined as **Partial Trees** (PTs). These can be generated by the application of partial production rules of the original grammar. For example, `[S [N VP]]`, `[S [N]]` and `[S [VP]]` are valid PTs of the tree in Figure 17.

Given a syntactic tree, we may represent it by means of the set of all its STs, SSTs or PTs. For example, Figure 18 shows the parse tree of the sentence `"Mary brought a cat"` together with its 6 STs. The number of SSTs is always higher. For example, Figure 19 shows 10 SSTs (out of all 17) of the SubTree of Figure 18 rooted in `VP`. Figure 20 shows that the number of PTs derived from the same tree is even higher (i.e. 30 PTs). These different substructure numbers provide an intuitive quantification of the different information level of the diverse tree-based representations.

5.2 The Kernel Functions

The main idea of the tree kernels is to compute the number of the common substructures between two trees T_1 and T_2 without explicitly considering the whole fragment space. For this purpose, we slightly modified the kernel function proposed in [19] by introducing a parameters σ, which enables the ST or the SST evaluation. For the PT kernel function, we designed a new algorithm.

The ST and SST Computation. Given a tree fragment space $\{f_1, f_2, ..\} = \mathcal{F}$, we defined the indicator function $I_i(n)$, which is equal to 1 if the target f_i is rooted at node n and 0 otherwise. It follows that:

$$K(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2) \tag{30}$$

where N_{T_1} and N_{T_2} are the sets of the T_1's and T_2's nodes, respectively and $\Delta(n_1, n_2) = \sum_{i=1}^{|\mathcal{F}|} I_i(n_1) I_i(n_2)$. This latter is equal to the number of common fragments rooted at the n_1 and n_2 nodes. We can compute Δ as follows:

1. if the productions at n_1 and n_2 are different then $\Delta(n_1, n_2) = 0$;
2. if the productions at n_1 and n_2 are the same, and n_1 and n_2 have only leaf children (i.e. they are pre-terminal symbols) then $\Delta(n_1, n_2) = 1$;
3. if the productions at n_1 and n_2 are the same, and n_1 and n_2 are not pre-terminals then

$$\Delta(n_1, n_2) = \prod_{j=1}^{nc(n_1)} (\sigma + \Delta(c_{n_1}^j, c_{n_2}^j)) \tag{31}$$

where $\sigma \in \{0, 1\}$, $nc(n_1)$ is the number of the children of n_1 and c_n^j is the j-th child of the node n. Note that, as the productions are the same $nc(n_1) = nc(n_2)$.

When $\sigma = 0$, $\Delta(n_1, n_2)$ is equal 1 only if $\forall j \quad \Delta(c_{n_1}^j, c_{n_2}^j) = 1$, i.e. all the productions associated with the children are identical. By recursively applying this property, it follows that the SubTrees in n_1 and n_2 are identical. Thus, Eq. 30 evaluates the SubTree (ST) kernel. When $\sigma = 1$, $\Delta(n_1, n_2)$ evaluates the number of SSTs common to n_1 and n_2 as proved in [19].

To include the leaves as fragments it is enough to add, to the recursive rule set above, the condition:

0. if n_1 and n_2 are leaves and their associated symbols are equal then $\Delta(n_1, n_2) = 1$

We will refer to such extended kernels as ST+bow (*bag-of-words*) and SST+bow. Moreover, we use the decay factor λ as follows[7]:$\Delta(n_x, n_z) = \lambda$ and $\Delta(n_x, n_z) = \lambda \prod_{j=1}^{nc(n_x)} (\sigma + \Delta(c_{n_1}^j, c_{n_2}^j))$.

The Δ computation complexity is $O(|N_{T_1}| \times |N_{T_2}|)$ time as proved in [19]. We will refer to this basic implementation as the Quadratic Tree Kernel (QTK).

The PT Kernel Function. The evaluation of the Partial Trees is more complex since two nodes n_1 and n_2 with different child sets (i.e. associated with different productions) can share one or more children, consequently they have one or more common substructures, e.g. `[S [DT JJ N]]` and `[S [DT N N]]` have the `[S [N]]` (2 times) and the `[S [DT N]]` in common.

To evaluate all possible substructures common to two trees, we can (1) select a child subset from both trees, (2) extract the portion of the syntactic rule that contains such subset, (3) apply Eq. 31 to the extracted partial productions and (4) sum the contributions of all children subsets.

Such subsets correspond to all possible common (non-continuous) node subsequences and can be computed efficiently by means of sequence kernels [38]. Let $\boldsymbol{J}_1 = (J_{11}, .., J_{1r})$ and $\boldsymbol{J}_2 = (J_{21}, .., J_{2r})$ be the index sequences associate with the ordered child sequences of n_1 and n_2, respectively, then the number of PTs is evaluated by the following Δ function:

$$\Delta(n_1, n_2) = 1 + \sum_{\boldsymbol{J}_1, \boldsymbol{J}_2, l(\boldsymbol{J}_1) = l(\boldsymbol{J}_2)} \prod_{i=1}^{l(\boldsymbol{J}_1)} \Delta(c_{n_1}^{\boldsymbol{J}_{1i}}, c_{n_2}^{\boldsymbol{J}_{2i}}), \tag{32}$$

[7] To have a similarity score between 0 and 1, we also apply the normalization in the kernel space, i.e. $K_{normed}(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) \times K(T_2, T_2)}}$.

where $l(\boldsymbol{J}_1)$ indicates the length of the target child sequence whereas $\boldsymbol{J}_{1i}$ and $\boldsymbol{J}_{2i}$ are the i^{th} children in the two sequences. We note that:

1. Eq. 32 is a convolution kernel [34] (see Definition 11).
2. Given a sequence of common children, $\boldsymbol{J}$, the product in Eq. 32 evaluates the number of common PTs rooted in n_1 and n_2. In these PTs, the children of n_1 and n_2 are all and only those in $\boldsymbol{J}$.
3. By summing the products associated with each sequence we evaluate all possible PTs (the root is included).
4. Tree kernels based on sequences were proposed in [72; 21] but they do not evaluate all tree substructures, i.e. they are not convolution kernels.
5. We can scale down the contribution from the longer sequences by adding two decay factors λ and μ:

$$\Delta(n_1, n_2) = \mu\Big(\lambda + \sum_{\boldsymbol{J}_1, \boldsymbol{J}_2, l(\boldsymbol{J}_1)=l(\boldsymbol{J}_2)} \lambda^{d(\boldsymbol{J}_1)+d(\boldsymbol{J}_2)} \prod_{i=1}^{l(\boldsymbol{J}_1)} \Delta(c_{n_1}^{\boldsymbol{J}_{1i}}, c_{n_2}^{\boldsymbol{J}_{2i}})\Big)$$

where $d(\boldsymbol{J}_1) = \boldsymbol{J}_{1l(\boldsymbol{J}_1)} - \boldsymbol{J}_{11} + 1$ and $d(\boldsymbol{J}_2) = \boldsymbol{J}_{2l(\boldsymbol{J}_2)} - \boldsymbol{J}_{21} + 1$.

Finally, as the sequence kernels and the Eq. 31 can be efficiently evaluated, the same can be done for Eq. 32. The computational complexity of PTK is $O(p\rho^2|N_{T_1}| \times |N_{T_2}|)$, where p is the largest subsequence of children that we want to consider and ρ is the maximal outdegree observed in the two trees. However, as shown in [40], the average running time tends to be linear for natural language syntactic trees.

6 Conclusions and Advanced Topics

In this chapter we have shown the basic approaches of traditional machine learning such as *Decision Trees* and *Naive Bayes* and we have introduced the basic concepts of the statistical learning theory such as the characterization of learning via the PAC theory and VC-dimension. We have also presented, the Perceptron algorithm to introduce a simplified theory of Support Vector Machines (SVMs) and kernel methods. Regarding the latter, we have shown some of their potentials, e.g. the Polynomial, String, Lexical and Tree kernels by alluding to their application for Natural Language Processing (NLP).

The interested reader, who would like to acquire much more practical knowledge on the use of SVMs and kernel methods can refer to the following publications clustered by topics (mostly from NLP): *Text Categorization* [9; 56; 10; 6; 5; 11; 12; 7; 13]; *Corefernce Resolution* [66; 65]; *Question Answering* [51; 13; 14; 55; 49; 50]; *Shallow Semantic Parsing* [54; 32; 45; 3; 30; 46; 31; 48; 42; 47; 57; 22; 44]; *Concept segmentation and labeling of text and speech* [23; 24; 59; 36; 37; 33]; *Relational Learning* [68; 69; 52; 67; 70; 71; 58; 43; 39; 27]; *SVM optimization* [40; 1; 41; 2; 53; 60; 61; 63; 62]; *Mapping Natural Language to SQL* [28; 29]; *Protein Classification* [17; 18]; *Audio classification* [4]; and *Electronic Device Failure detection* [25].

The articles above are available at `http://disi.unitn.it/moschitti/Publications.htm` whereas complementary training material can be found at

http://disi.unitn.it/moschitti/teaching.html. Additionally, SVM software comprising several structural kernels can be downloaded from http://disi.unitn.it/moschitti/Tree-Kernel.htm.

Acknowledgement

The work for this tutorial has been partially founded by the European Coordinate Action, EternalS, Trustworthy Eternal Systems via Evolving Software, Data and Knowledge (project number FP7 247758).

I would like to thank Roberto Basili for his contribution to an early draft of this chapter.

References

1. Aiolli, F., Martino, G.D.S., Moschitti, A., Sperduti, A.: Fast On-line Kernel Learning for Trees. In: Proceedings Sixth International Conference on Data Mining, ICDM 2006. IEEE, Los Alamitos (2006)
2. Aiolli, F., Martino, G.D.S., Moschitti, A., Sperduti, A.: Efficient Kernel-based Learning for Trees. In: IEEE Symposium on Computational Intelligence and Data Mining, pp. 308–316. IEEE, Stati Uniti d'America (2007)
3. Ana-Maria, G., Moschitti, A.: Towards Free-text Semantic Parsing: A Unified Framework Based on FrameNet, VerbNet and PropBank. In: The Workshop on Learning Structured Information for Natural Language Applications. EACL (2006)
4. Annesi, P., Basili, R., Gitto, R., Moschitti, A., Petitti, R.: Audio Feature Engineering for Automatic Music Genre Classification. In: RIAO, Paris, France, pp. 702–711 (2007)
5. Basili, R., Cammisa, M., Moschitti, A.: A Semantic Kernel to Exploit Linguistic Knowledge. In: Bandini, S., Manzoni, S. (eds.) AI*IA 2005. LNCS (LNAI), vol. 3673, pp. 290–302. Springer, Heidelberg (2005)
6. Basili, R., Cammisa, M., Moschitti, A.: Effective use of WordNet Semantics via Kernel-based Learning. In: Proceedings of the Ninth Conference on Computational Natural Language Learning, pp. 1–8. The Association for Computational Linguistics (June 2005)
7. Basili, R., Cammisa, M., Moschitti, A.: A semantic Kernel to Classify Texts with very few Training Examples. Informatica, an International Journal of Computing and Informatics 1, 1–10 (2006)
8. Basili, R., Cammisa, M., Moschitti, A.: Effective use of wordnet semantics via kernel-based learning. In: Proceedings of Ninth Conference on Computational Natural Language Learning, Ann Arbor, Michigan USA, June 29-30 (2005)
9. Basili, R., Moschitti, A.: NLP-driven IR: Evaluating Performance over a Text Classification Task. In: International Joint Conference of Artificial Intelligence (2001)
10. Basili, R., Moschitti, A.: Automatic Text Categorization: from Information Retrieval to Support Vector Learning. Aracne Publisher (2005)
11. Basili, R., Moschitti, A., Pazienza, M.T.: Extensive Evaluation of Efficient NLP-driven Text Classification. Applied Artificial Intelligence (2006)
12. Bloehdorn, S., Basili, R., Cammisa, M., Moschitti, A.: Semantic Kernels for Text Classification based on Topological Measures of Feature Similarity. In: Sixth International Conference on Data Mining, ICDM 2006, pp. 808–812. IEEE, Los Alamitos (2006)

13. Bloehdorn, S., Moschitti, A.: Combined Syntactic and Semanitc Kernels for Text Classification. In: Amati, G., Carpineto, C., Romano, G. (eds.) ECiR 2007. LNCS, vol. 4425, pp. 307–318. Springer, Heidelberg (2007)
14. Bloehdorn, S., Moschitti, A.: Exploiting Structure and Semantics for Expressive Text Kernels. In: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, pp. 861–864. ACM, New York (2007)
15. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the vapnik-chervonenkis dimension. Journal of the Association for Computing Machinery 36(4), 929–965 (1989)
16. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Mining and Knowledge Discovery 2(2), 121–167 (1998)
17. Cilia, E., Moschitti, A.: Advanced Tree-based Kernels for Protein Classification. In: Basili, R., Pazienza, M.T. (eds.) AI*IA 2007. LNCS (LNAI), vol. 4733, pp. 218–229. Springer, Heidelberg (2007)
18. Cilia, E., Moschitti, A., Ammendola, S., Basili, R.: Structured kernels for automatic detection of protein active sites. In: Mining and Learning with Graphs Workshop (MLG) (2006)
19. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: ACL 2002 (2002)
20. Cristianini, N., Shawe-Taylor, J.: An introduction to Support Vector Machines. Cambridge University Press, Cambridge (2000)
21. Culotta, A., Sorensen, J.: Dependency Tree Kernels for Relation Extraction. In: Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL 2004), Main Volume, Barcelona, Spain, pp. 423–429 (July 2004)
22. Diab, M., Moschitti, A., Pighin, D.: Semantic Role Labeling Systems for Arabic Language using Kernel Methods. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 798–806. Association for Computational Linguistics, Columbus (June 2008)
23. Dinarelli, M., Moschitti, A., Riccardi, G.: Re-Ranking Models for Spoken Language Understanding. In: Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pp. 202–210. Association for Computational Linguistics, Athens (March 2009)
24. Dinarelli, M., Moschitti, A., Riccardi, G.: Re-Ranking Models Based-on Small Training Data for Spoken Language Understanding. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, pp. 1076–1085. Association for Computational Linguistics (2009)
25. Dutta, H., Waltz, D., Moschitti, A., Pighin, D., Gross, P., Monteleoni, C., Salleb-Aouissi, A., Boulanger, A., Pooleery, M., Anderson, R.: Estimating the Time Between Failures of Electrical Feeders in the New York Power Grid. In: Next Generation Data Mining Summit, NGDM 2009, Baltimore, MD (2009)
26. Fellbaum, C.: WordNet: An Electronic Lexical Database. MIT Press, Cambridge (1998)
27. Giannone, C., Basili, R., Naggar, P., Moschitti, A.: Supervised Semantic Relation Mining from Linguistically Noisy Text Documents. International Journal on Document Analysis and Recognition 2010, 1–25 (2010)
28. Giordani, A., Moschitti, A.: Semantic Mapping Between Natural Language Questions and SQL Queries via Syntactic Pairing. In: Horacek, H., Métais, E., Muñoz, R., Wolska, M. (eds.) NLDB 2009. LNCS, vol. 5723, pp. 207–221. Springer, Heidelberg (2010)
29. Giordani, A., Moschitti, A.: Syntactic Structural Kernels for Natural Language Interfaces to Databases. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS, vol. 5781, pp. 391–406. Springer, Heidelberg (2009)

30. Giuglea, A., Moschitti, A.: Semantic Role Labeling via FrameNet, VerbNet and PropBank. In: COLING-ACL 2006: 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, pp. 929–936. Association for Computational Linguistics (July 2006)
31. Giuglea, A.M., Moschitti, A.: Shallow Semantic Parsing Based on FrameNet, VerbNet and PropBank. In: ECAI 2006, 17th Conference on Artificial Intelligence, including Prestigious Applications of Intelligent Systems (PAIS 2006), Riva del Garda, Italy, August 29-September 1. IOS, Amsterdam (2006)
32. Giuglea, A.M., Moschitti, A.: Knowledge Discovery using FrameNet, VerbNet and PropBank. In: Meyers, A. (ed.) Workshop on Ontology and Knowledge Discovering at ECML 2004, Pisa, Italy (2004)
33. Hahn, S., Dinarelli, M., Raymond, C., Lefevre, F., Lehnen, P., Mori, R.D., Moschitti, A., Ney, H., Riccardi, G.: Comparing Stochastic Approaches to Spoken Language Understanding in Multiple Languages. IEEE Transaction on Audio, Speech and Language Processing PP (99), 1–15 (2010)
34. Haussler, D.: Convolution Kernels on Discrete Structures. Technical report ucs-crl-99-10, University of California Santa Cruz (1999)
35. Jackendoff, R.: Semantic Structures. Current Studies in Linguistics series. The MIT Press, Cambridge (1990)
36. Johansson, R., Moschitti, A.: Reranking Models in Fine-grained Opinion Analysis. In: Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010), Beijing, China, pp. 519–527 (August 2010)
37. Johansson, R., Moschitti, A.: Syntactic and Semantic Structure for Opinion Expression Detection. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning, Sweden, pp. 67–76 (July 2010)
38. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. In: NIPS, pp. 563–569 (2000)
39. Mehdad, Y., Moschitti, A., Zanzotto, F.: Syntactic/Semantic Structures for Textual Entailment Recognition. In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 1020–1028. Association for Computational Linguistics, Los Angeles (June 2010)
40. Moschitti, A.: Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 318–329. Springer, Heidelberg (2006)
41. Moschitti, A.: Making tree kernels practical for natural language learning. In: EACL 2006: 11th Conference of the European Chapter of the Association for Computational Linguistics. ACL (2006)
42. Moschitti, A.: Syntactic Kernels for Natural Language Learning: the Semantic Role Labeling Case. In: Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, pp. 97–100. ACL (2006)
43. Moschitti, A.: Syntactic and Semantic Kernels for Short Text Pair Categorization. In: Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pp. 576–584. Association for Computational Linguistics, Athens (March 2009)
44. Moschitti, A.: LivingKnowledge: Kernel Methods for Relational Learning and Semantic Modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 15–19. Springer, Heidelberg (2010)
45. Moschitti, A., Giuglea, A.M., Coppola, B., Basili, R.: Hierarchical Semantic Role Labeling. In: Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL 2005), June 30, pp. 201–204. Association for Computational Linguistics (2005)

46. Moschitti, A., Pighin, D., Basili, R.: Semantic Role Labeling via Tree Kernel Joint Inference. In: Proceedings of the 10th Conference on Computational Natural Language Learning, pp. 61–68. Association for Computational Linguistics (June 2006)
47. Moschitti, A., Pighin, D., Basili, R.: Tree Kernel Engineering for Proposition Reranking. In: MLG 2006: Proceedings of the International Workshop on Mining and Learning with Graphs (in conjunction with ECML/PKDD 2006), pp. 165–172 (September 2006)
48. Moschitti, A., Pighin, D., Basili, R.: Tree Kernel Engineering in Semantic Role Labeling Systems. In: EACL 2006: 11th Conference of the European Chapter of the Association for Computational Linguistics: Proceedings of the Workshop on Learning Structured Information in Natural Language Applications, pp. 49–56 (2006)
49. Moschitti, A., Quarteroni, S.: Kernels on Linguistic Structures for Answer Extraction. In: 46th Conference of the Association for Computational Linguistics, pp. 113–116. ACL, Columbus (2008)
50. Moschitti, A., Quarteroni, S.: Linguistic Kernels for Answer Re-ranking in Question Answering Systems. Information Processing & Management 2010, 1–36 (2010)
51. Moschitti, A., Quarteroni, S., Basili, R., Manandhar, S.: Exploiting Syntactic and Shallow Semantic Kernels for Question/Answer Classification. In: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, pp. 776–783. Association for Computational Linguistics, USA (2007)
52. Moschitti, A., Zanzotto, F.M.: Experimenting a General Purpose Textual Entailment Learner in AVE. In: Peters, C., Clough, P., Gey, F.C., Karlgren, J., Magnini, B., Oard, D.W., de Rijke, M., Stempfhuber, M. (eds.) CLEF 2006. LNCS, vol. 4730, pp. 510–517. Springer, Heidelberg (2007)
53. Moschitti, A., Zanzotto, F.M.: Fast and effective kernels for relational learning from texts. In: Proceedings of the 24th Annual International Conference on Machine Learning, pp. 649–656. ACM, New York (June 2007)
54. Moschitti, A.: A study on convolution kernel for shallow semantic parsing. In: Proceedings of the 42th Conference on Association for Computational Linguistic (ACL 2004), Barcelona, Spain (2004)
55. Moschitti, A.: Kernel Methods, Syntax and Semantics for Relational Text Categorization. In: Proceeding of ACM 17th Conf. on Information and Knowledge Management (CIKM 2008), Napa Valley, CA, USA (2008)
56. Moschitti, A., Basili, R.: Complex Linguistic Features for Text Classification: a Comprehensive Study. In: McDonald, S., Tait, J.I. (eds.) ECIR 2004. LNCS, vol. 2997, pp. 181–196. Springer, Heidelberg (2004)
57. Moschitti, A., Pighin, D., Basili, R.: Tree Kernels for Semantic Role Labeling. Computational Linguistics, 193–224 (2008)
58. Nguyen, T., Moschitti, A., Riccardi, G.: Convolution Kernels on Constituent, Dependency and Sequential Structures for Relation Extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 1378–1387. Association for Computational Linguistics, Singapore (August 2009)
59. Nguyen, T.V.T., Moschitti, A., Riccardi, G.: Kernel-based Reranking for Named-Entity Extraction. In: Coling 2010: Posters, Beijing, China, pp. 901–909 (August 2010)
60. Pighin, D., Moschitti, A.: Efficient Linearization of Tree Kernel Functions. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009), pp. 30–38. Association for Computational Linguistics (2009)
61. Pighin, D., Moschitti, A.: Reverse Engineering of Tree Kernel Feature Spaces. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, pp. 111–120. Association for Computational Linguistics (2009)

62. Pighin, D., Moschitti, A.: On Reverse Feature Engineering of Syntactic Tree Kernels. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning, pp. 223–233. Association for Computational Linguistics, Uppsala (July 2010)
63. Severyn, A., Moschitti, A.: Large-Scale Support Vector Learning with Structural Kernels. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010, Part III. LNCS, vol. 6323, pp. 229–244. Springer, Heidelberg (2010)
64. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, Heidelberg (1995)
65. Versley, Y., Ponzetto, S.P., Poesio, M., Eidelman, V., Jern, A., Smith, J., Yang, X., Moschitti, A.: BART: A Modular Toolkit for Coreference Resolution. In: ACL (Demo Papers), pp. 9–12 (2008)
66. Vesley, Y., Moschitti, A., Poesio, M.: Coreference Systems based on Kernels Methods. In: International Conference on Computational Linguistics, pp. 961–968. Association for Computational Linguistics (2008)
67. Zanzotto, F.M., Moschitti, A.: Automatic Learning of Textual Entailments with Cross-Pair Similarities. In: The Joint 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL). Association for Computational Linguistics, Sydney (2006)
68. Zanzotto, F.M., Moschitti, A.: Similarity between Pairs of Co-indexed Trees for Textual Entailment Recognition. In: The TextGraphs Workshop at Human Language Technology. Association for Computational Linguistics (2006)
69. Zanzotto, F.M., Moschitti, A., Pennacchiotti, M., Pazienza, M.T.: Learning Textual Entailment from Examples. In: The Second Recognising Textual Entailment Challenge. The Second Recognising Textual Entailment Challenge (2006)
70. Zanzotto, F.M., Pennacchiotti, M., Moschitti, A.: Shallow Semantics in Fast Textual Entailment Rule Learners. In: The Third Recognising Textual Entailment Challenge, pp. 72–77. Association for Computational Linguistics (2007)
71. Zanzotto, F.M., Pennacchiotti, M., Moschitti, A.: A Machine Learning Approach to Recognizing Textual Entailment. Natural Language Engineering 15(4), 551–582 (2009)
72. Zelenko, D., Aone, C., Richardella, A.: Kernel methods for relation extraction. Journal of Machine Learning Research (2003)