Dottorato di Ricerca in Informatica
Università di Bologna, Padova

Ciclo XXI
Settore scientifico disciplinare: INF/01

Kernel Methods for Tree Structured Data

Giovanni Da San Martino

Coordinatore Dottorato:	Relatore:
Prof. S. Martini	Prof. A. Sperduti

Esame finale anno 2009

Abstract

Machine learning comprises a series of techniques for automatic extraction of meaningful information from large collections of noisy data. In many real world applications, data is naturally represented in structured form. Since traditional methods in machine learning deal with vectorial information, they require an a priori form of preprocessing. Among all the learning techniques for dealing with structured data, kernel methods are recognized to have a strong theoretical background and to be effective approaches. They do not require an explicit vectorial representation of the data in terms of features, but rely on a measure of similarity between any pair of objects of a domain, the kernel function. Designing fast and good kernel functions is a challenging problem. In the case of tree structured data two issues become relevant: kernel for trees should not be sparse and should be fast to compute. The sparsity problem arises when, given a dataset and a kernel function, most structures of the dataset are completely dissimilar to one another. In those cases the classifier has too few information for making correct predictions on unseen data. In fact, it tends to produce a discriminating function behaving as the nearest neighbour rule. Sparsity is likely to arise for some standard tree kernel functions, such as the subtree and subset tree kernel, when they are applied to datasets with node labels belonging to a large domain. A second drawback of using tree kernels is the time complexity required both in learning and classification phases. Such a complexity can sometimes prevents the kernel application in scenarios involving large amount of data.

This thesis proposes three contributions for resolving the above issues of kernel for trees. A first contribution aims at creating kernel functions which adapt to the statistical properties of the dataset, thus reducing its sparsity with respect to traditional tree kernel functions. Specifically, we propose to encode the input trees by an algorithm able to project the data onto a lower dimensional space with the property that similar structures are mapped similarly. By building kernel functions on the lower dimensional representation, we are able to perform inexact matchings between different inputs in the original space.

A second contribution is the proposal of a novel kernel function based on the convolution kernel framework. Convolution kernel measures the similarity of two objects in terms of the similarities of their subparts. Most convolution kernels are based on counting the number of shared substructures, partially discarding information about their position in the original structure. The kernel function we propose is, instead, especially focused on this aspect.

A third contribution is devoted at reducing the computational burden related to the calculation of a kernel function between a tree and a forest of trees, which is a typical operation in the classification phase and, for some algorithms, also in the learning phase. We propose a general methodology applicable to convolution kernels. Moreover, we show an instantiation of our technique when kernels such as the subtree and subset tree kernels are employed. In those cases, Direct Acyclic Graphs can be used to compactly represent shared substructures in different trees, thus reducing the computational burden and storage requirements.

Acknowledgements

I would like to express my gratitude to my tutor, Professor Alessandro Sperduti, for its guidance and support.
I also thank my family for its support over these years.
Finally I would like to thank everybody that has been on my side and helped me whenever I needed, and whenever I did not.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **ix**

1 Introduction **1**

1.1 What is Machine Learning 2

1.2 Issues in Structured Data Representation 3

1.3 Kernel Methods for Structured Data 4

1.4 Thesis Motivations 6

1.5 Outline of the Thesis and Original Contributions 8

1.6 Origin of the Chapters 9

I Basics **11**

2 Background **12**

2.1 Definitions and Notation 12

2.2 Machine Learning 16

2.3 Machine Learning For Structured Data 19

2.3.1 Self Organizing Maps 19

2.3.2 Kernel Methods 25

2.3.3 Kernel Functions 29
2.3.4 Evaluating Kernel Functions 33

3 State of the Art on Tree Kernel Functions 37
3.1 Convolution Kernels 38
3.1.1 Subtree Kernel 40
3.1.2 Subset Tree Kernel 42
3.1.3 Approximate Kernels for Trees 45
3.1.4 Partial Tree Kernel 46
3.1.5 Elastic Tree Kernel 47
3.1.6 Grammar-Driven Tree Kernel 50
3.1.7 Semantic Syntactic Tree Kernels 51
3.2 Other Approaches for the Design of Kernels for Tree Structured Data 52
3.2.1 Spectrum Tree Kernel 52
3.2.2 Tree Fisher Kernel 53

II Original Contributions 55

4 A Tree Kernel For Non Discrete Domains 56
4.1 Activation Mask Kernel 57
4.2 Related Work 61
4.3 Experiments and Discussion 62

5 A Novel Kernel for Trees: Convolution Route Kernel 79
5.1 Generalized Route Kernel 80
5.2 An instantiation of the Generalized Route Kernel 82
5.2.1 Implementation 84
5.2.2 Relationship with other Kernels 88
5.3 Experiments and Discussion 88

5.3.1 Experiments on INEX 2005 . . . 89
5.3.2 Experiments on INEX 2006 . . . 91
5.3.3 Experiments on LOGML . . . 94
5.3.4 Discussion . . . 96

6 Efficient Score Computation by Compacting the Model **97**
6.1 General Considerations . . . 98
6.2 Compacting a Forest of Trees . . . 99
6.2.1 From a Forest to a Directed Acyclic Graph . . . 100
6.2.2 Efficient Score Computation . . . 103
6.2.3 The DAG Kernel Perceptron . . . 107
6.2.4 Voted Kernel Perceptron . . . 109
6.2.5 Kernel Combinations . . . 110
6.2.6 Experiments . . . 112

7 Conclusions **123**

A Experimental Settings **127**
A.1 INEX 2005 . . . 127
A.2 INEX 2006 . . . 130
A.3 Penn Treebank II . . . 132
A.4 LOGML . . . 135

References **137**

List of Figures

2.1 An example of a labelled directed graph. 13

2.2 A positional Tree. The number over an arc represents the position of the node with respect to its parent. 14

2.3 A tree (left) and some of its subtrees (right). 14

2.4 A tree (left) and all of its proper subtrees (right). 15

2.5 A tree (left) and all of its subset trees (right). 15

3.1 A subtree (left) and one of its elastic matchings with the tree on the right. 49

3.2 Some examples of q-grams, with $q = 4$. P_i identifies the structure of the path, the string the sequence of labels as encountered by visiting the subtree. 52

4.1 Example of representation in feature space of three trees according to the Activation Mask Kernel for $\epsilon = 1$. On the left part of the image three simple trees and on the right part their activation masks referring to a 5×4 map. The height of each element of the map corresponds to the value of the activation. 60

4.2 Comparison between classification error of the different techniques on the INEX 2005 test dataset. Maps on the x-axis are sorted by SOM-SD classification error. The error values of the AM-kernel are related to the ϵ value selected on validation (which is reported in correspondence of the map error value). 71
4.3 Classification error of the SST and AM-kernel on various datasets with different levels of sparsity. 76
4.4 Comparison between classification error (using 3-fold cross-validation) of the different techniques on the LOGML test dataset. Maps on the x-axis are sorted by SOM-SD classification error. The error values of the AM-kernel are related to the ϵ value selected on validation (which is reported in correspondence of the map error value). 78

5.1 An example of a route connecting nodes labelled with **a** and **e**. The nodes connected by dashed edges are the ones comprising the path between the two nodes. The route is formed by the sequence $2, 3$ since node **b** is the second child of **a** and node **e** is the third child of **b**. . . 80
5.2 A tree (left) and its set of features according to the route kernel defined in eq. (5.12). 84

6.1 Example of how to represent a forest as a minimal DAG with no loss of information. Nodes in the minimal DAG are annotated with a label and the frequency in the forest of the subtree rooted at that node.101
6.2 The algorithm to transform a tree-forest into a minimal DAG. 102
6.3 The algorithm to insert a weighted ADAG in a larger ADAG. 104
6.4 The DAG-Perceptron algorithm. 107
6.5 Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using a polynomial kernel with degree 3 (Poly3) over the training set with 992,819 examples. 114

6.6 Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using the SST tree kernel (Tk) with different values for the λ parameter, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, over the training set with 992,819 examples. For each method, only the fastest and the slower executions are reported. 115

6.7 Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using a linear combination of a polynomial kernel with degree 3 (Poly3) with the SST tree kernel (Tk), i.e. $(1-\gamma) * Poly3 + \gamma * Tk$, over the training set with 992,819 examples. Different values for λ and γ have been considered, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and $\gamma \in \{0.2, 0.3, 0.4, 0.5, 0.6\}$. For each method, only the fastest and the slower executions are reported. 116

6.8 Evolution of the number of tree nodes stored in memory and belonging to the model developed by the Standard Perceptron and the Voted DAG Perceptron during training on the training set with 992,819 examples. Both methods use the SST tree kernel (Tk) with different values for λ, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. For each method, only the executions with the largest and the lower number of stored nodes are reported. 117

6.9 Evolution of the number of tree nodes stored in memory and belonging to the model developed by the Standard Perceptron and the Voted DAG Perceptron during training on the training set with 992,819 examples. Both methods use a linear combination of a polynomial kernel with degree 3 (Poly3) with the SST tree kernel (Tk), i.e. $(1-\gamma) * Poly3 + \gamma * Tk$. Different values for λ and γ have been considered, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and $\gamma \in \{0.2, 0.3, 0.4, 0.5, 0.6\}$. For each method, only the executions with the largest and the lower number of stored nodes are reported. 119

6.10 Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using the SST tree kernel with different values for the λ parameter, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, over the union of the INEX 2005 training and test sets. 120

6.11 Evolution of the number of tree nodes stored in memory and belonging to the model developed by the Standard Perceptron and the Voted DAG Perceptron during training on the union of the INEX 2005 training and test sets. Both methods use the SST tree kernel (Tk) with different values for λ, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$. For the Standard Perceptron only the execution with lower number of nodes is reported. For the Voted Dag Perceptron only the execution with the largest number of nodes is reported. 121

A.1 Parse tree of the sentence "Mary brought a cat to school" along with the PAF trees for Arg0, Arg1 and ArgM. 135

Chapter 1

Introduction

Since the advent of modern computers the amount of available information has been increasing more than our capacity of analysing it. The development of automatic tools for data analysis is still an active area of research. Machine learning comprises a series of techniques for automatic extraction of meaningful information from large collections of noisy data.

Traditional methods in machine learning deal with vectorial information even if, in many real world applications, data are naturally represented in structured form (graphs for instance): XML data, molecular structures in chemical informatics, parse trees in natural language processing and protein sequences in bioinformatics.

In order to apply machine learning techniques designed for vectorial data to structured data, a pre-processing phase is required in order to encode structured information into vectorial form. A pre-processing is always task specific and needs to be suitably designed for any new task. The pre-processing can result in the loss of relevant or necessary information for the given task.

Recent developments in machine learning have produced methods capable of processing graph structured information directly. Among these, kernel methods are becoming more and more popular. Being on one hand theoretically well founded in statistical learning theory, they have on the other hand shown good empirical results in many applications. Kernel methods introduce a novel way of handling structured data since they do not require an explicit representation of each input, they instead require the definition of a kernel, i.e. a similarity function between all pairs of objects of a domain. The definition of kernel functions for structures is a challenging task because of the need to balance the trade-off between accuracy (how well its values represent the true similarity between objects) and its computational complexity.

Kernel methods have proven to be successfully for many real world problems. However, the kernels currently defined in literature have some drawbacks. Investigating ways of overcoming current kernel drawbacks and looking for more effective kernels is the main motivation for this thesis.

1.1 What is Machine Learning

One of the main motivations for the development of machine learning techniques is to automatically extract meaningful information from large collections of data: recognizing human speech, detect fraudulent credit card transactions, perform automatic medical diagnosis. For a more detailed example consider the case of recognizing tumours in magnetic resonance images (MRI). An MRI exam produces many series of images (up to 6) each one containing many images (more than 30). All of them have to be carefully analysed in order to discover the presence of a tumour. This is a long process and therefore the daily number of examined patients is limited. This problem features two of the issues typically faced by machine learning: data may be noisy and no algorithmic solution is known. In the MRI example a radiologist, in most cases, is able to recognize a tumour but he is not able to formalize the problem and define an exact and generally applicable procedure for solving it. The only way to communicate its knowledge to a student, for example, is to give him a series of images along with their classification (whether they contain a tumour or not) and guide him in its inference process giving feedbacks on his hypotheses. The student improves his capacity of recognizing tumours with experience. The aim of machine learning is precisely to provide automatic tools to mimic the human ability to improve its behaviour with experience. Another situation in which the use of a machine learning approach to problem solving is appropriate is when the results are subjective. Algorithms able to adapt to user preferences are usually based on machine learning techniques. For example the notion of interesting web site depends on the user, so an intelligent search engine should learn user preferences and return the results based on them.

1.2 Issues in Structured Data Representation

This section discusses the problems of representing tree structured data in Machine Learning algorithms. Tree data structures are employed to model objects from several domains. In natural language processing, parse trees are modelled as ordered labelled trees. In pattern recognition, an image can be represented by a tree whose vertices are associated with image components, retaining information concerning the structure of the image. In automated reasoning, many problems are solved by searching and the search space is often represented as a tree whose vertices are associated with search states and edges represent inference steps. Also semistructured data such as HTML and XML documents can be modelled by labelled ordered trees.

In order to apply to structured data a learning algorithm not specifically designed for that format the user must first transform the data into a vectorial form. This task is problem dependent, may be computational demanding, and is prone to loss of relevant information. In [8] it is described an example of encoding a dataset of chemical structures into a vectorial form. Each structure is represented by numerical descriptors called topological indices, which code specific morphological properties of the molecule. The topological indices must be defined by a domain expert, and that can be an expensive procedure (given that an expert is available). Moreover an error by the expert may greatly affect the subsequent learner accuracy. Among all indices a subset more appropriate for the given task may be chosen. This selection procedure may have to be repeated if the dataset changes. In many real world problems the preprocessing phase affects heavily the accuracy of the learner. Moreover, to maintain all the structured information, the dimensionality of the resultant vectors may be quite high. This may be a significant drawback considering the fact that many Machine Learning techniques are not able to effectively scale with the dimensionality of the input and therefore their predictive power decreases with increase in the dimensionality of the input. This problem is known as "Curse of Dimensionality" [6]. To get an idea of the reason for this performance degradation, it is sufficient to consider a space $\mathcal{X}$ of dimension d. Suppose that $\mathcal{X}$ is composed by a set of points uniformly distributed. If the number of dimensions of $\mathcal{X}$ increases, the number of points necessary to keep the same density must increase exponentially. In other words, the more the dimensions of the input, the more the probability that the data are sparse. A sparse dataset gives in general too few information to build a good classifier. A flat representation for structured data is thus appropriate when knowledge about the domain can be effectively used to select a set of features. When such knowledge is not available, instead of manually trying different encodings, it is desirable to make use of techniques able to directly handle structured data.

1.3 Kernel Methods for Structured Data

The issue of data representation is faced by kernel methods [7,13,48] from a different perspective. Kernel methods avoid to explicitly represent the data into vectorial form since the only information they require is about the similarity of each pair of data items. By definition, kernel methods look for linear relations in the feature space. Input items are compared via dot products of their representation in the feature space. The feature space is a vectorial description of the data according to a predefined set of features. However kernel methods may avoid to directly access the feature space since it can be shown that it is possible to replace the dot product with a kernel function, a symmetric positive semidefinite function which computes the similarity of a pair of items directly in their original space. The advantage of using kernel functions is that huge, even infinite, feature spaces can be used with a computational complexity not dependent on the size of the feature space but on the complexity of the kernel function. It can be demonstrated that kernel methods, even if they can implicitly make use of very large feature spaces, they do not suffer of the curse of dimensionality since Statistical Learning Theory [65] shows that the generalization capability of a kernel method ultimately depends on the number of misclassified examples in the learning phase. So far we have showed that it can be avoided to access directly the feature space representation of the input examples. The classification of a new example is performed by consider the sign of the application of the kernel function between the example and the classifier (see eq. (2.15)). It can be shown that, if the kernel method satisfies the assumptions of the Representer theorem (see Section 2.3.2), the classifier can be represented as a weighted sum of the training instances (see section 2.2 for details). Thus the classification of an example is performed via a weighted sum of kernel evaluations between the example and a subset of the training instances. Since the representation of the data in feature space is only accessed implicitly when a kernel function between two examples is computed, kernel methods can be applied to any type of input by providing an appropriate kernel function.

Kernel functions have some interesting features:

- the space of kernel functions is closed under operations such as addition and linear combination. It is then very easy to combine data from different sources. For example, when classifying web pages, it would be possible to integrate information from text, images and links by combining the respective kernels.
- If we consider a finite dataset composed by n examples, we can represent the kernel function by a matrix whose size is always $n \times n$, independently from the size of each individual example. This property can be useful when a small dataset of large size examples has to be analyzed.

Kernel methods have proved to be a state of the art technique for many real world problems. They are described in detail in Section 2.3.2. However designing good kernel functions, i.e. fast to compute and expressive (see Section 2.3.4 for a definition) is an open problem. In the following these two important issues are discussed. Generally speaking, the main goal of this research is to find methodologies to overcome them.

1.4 Thesis Motivations

Kernel function evaluations heavily affect the computational burden of kernel methods. It is therefore important to keep their complexity as low as possible. Unfortunately it has been demonstrated that completely expressive kernels for graphs are NP-Hard to compute [54]: for example a kernel $k(G_1, G_2)$ that takes into account the similarity of all possible subgraphs of the two graphs G_1 and G_2 is equivalent to testing whether G_1 and G_2 are isomorphic (a problem known to be NP-Hard). Kernel functions must be a compromise between accuracy of the results and computational complexity of the procedure. In the following we try to make clear what we mean with the term expressiveness. One of the most popular kernel for trees is the subtree kernel (see section 3.1.1 for a description). It counts the number of exactly matching subtrees of the inputs. While the restriction to exact match allows the evaluation of the kernel function to be carried out in $n \log n$ time (where n is the number of nodes), it prevents the application on settings in which the labels of the nodes take values from the domain of real numbers, since hardly there will be any matching subtree. When the number of pairs of inputs having non zero similarity is very low, the kernel has low expressiveness and is said to be sparse. It is a pathological situation since those kernels are likely to not give enough information to the classifier, which will behave like a nearest neighbour rule [30, 61], i.e. it will not be able to generalize well on unseen data. Even in cases in which the labels of the nodes may only have values from a discrete domain, the subtree kernel may be sparse. For example, we collected some statistics from a dataset of XML data (see section A.1) and noticed that the subtree kernel would have resulted in a 0 kernel value, i.e. inputs totally dissimilar, for the 54.71% of the kernel evaluations. Relaxing the constraint which allows matchings only between identical subtrees does not help because the resulting computational complexity of kernel evaluations would make the use of kernel methods infeasible. The development of techniques and kernel functions with an acceptable accuracy/complexity trade-off is an open problem and it is one of the main targets for this research.

A second motivation for the development of novel non sparse and expressive kernel functions comes from the analysis of the literature on kernel for trees. Most of them fall under convolution kernel framework, which expresses a kernel on a pair of structures as a combination of kernels on their constituent substructures. However, all those kernels focus on the presence of the substructures and partially discard information about the position of the substructures in the original structure. This observation led us to investigate whether this type of information can be useful.

There are important computational issues not only in the computation of kernel functions, but also in the classification phase. As mentioned earlier, the hypothesis h returned by a kernel method can be expressed as a linear combination of the inputs. To be more precise, h can be expressed as a linear combination of the wrongly classified inputs. In order to use h the whole set of wrongly classified inputs must be kept in memory. While saving in memory a great amount of plain data may be feasible, saving great amounts of structured data, due to the typical increase in size, may severely limit the applicability of the technique. Just as an example, we collected some statistics from an XML dataset [36] finding out that the total number of nodes of the misclassified inputs tended to increase linearly with the size of the training set. It is worth pointing out that the reduction of the computational resources of a kernel method is not only a computational issue, but it also affects the accuracy of the classifier. In fact, in machine learning it is a well known fact that the accuracy of the classifier improves with the size of the training set.

1.5 Outline of the Thesis and Original Contributions

This section describes the contents of the thesis highlighting its original contributions.

The thesis is divided into two parts. The first part outlines background concepts and gives a survey of the state of the art of kernel for trees.

Chapter 2 introduces the notation and basic concepts used throughout the remaining chapters. Section 2.1 gives basic definitions about the structures used in the following chapters. Section 2.2 introduces the Machine Learning framework. Section 2.3 gives an overview of two approaches for handling tree structured data, the Self Organizing Map for Structured Data and kernel methods. The latter comprises a series of techniques which avoid to explicitly represent the data, since they rely on information about the similarity of objects in a domain. This type of information is given by the kernel functions. Sections 2.3.3 and 2.3.4 describe kernel function properties and discuss the contributions in literature for assessing their quality.

Chapter 3 gives an overview of the kernel functions for tree structured data. Section 3.1 introduces the convolution kernel framework and describes the kernel functions based on it. Section 3.2 gives a overview of other approaches for building kernel functions.

The second part of the thesis is devoted to the presentation of the original contribution.

A drawback of the standard tree kernels is that in the case of large structures and many symbols, the feature space implicitly defined by these kernels is very sparse. Chapter 4 proposes a novel family of kernels based on the activation of a Self Organizing Map for Structured Data, a clustering algorithm which maps tree structured information in such a way that similar trees are mapped onto nearby areas to form clusters. Specifically, we make use of this property to design kernel functions able to perform inexact subtree matching thus reducing the sparsity of the original kernel while trying to keep its structural information. Section 4.1 describes the novel family of kernels based on Self Organizing Map for Structured Data activations, the Activation Mask Kernel. Section 4.2 discusses the relationships of the new kernel with other kernels defined in literature. Section 4.3 present experiments performed to verify the effectiveness of our approach.

A second contribution was motivated by the observation that convolution tree kernels match substructures without taking into account their "relative positioning" with respect to one another. In chapter 5 a novel family of kernels is defined which explicitly focus on this type of information. Section 5.1 gives a formal definition of the novel kernel and Section 5.2 describes an instance of the general form. Section 5.3 describes the experiments performed in order to establish the effectiveness of the kernel.

While Support Vector Machines has a high generalization capability, a drawback of their use is the time required both in learning and classification phases. As a third contribution, in chapter 6 we present a methodology for reducing that computational burden for convolution tree kernels by a suitable encoding of the structures which avoid the re-computation of kernels between the same substructures belonging to different examples. Section 6.1 describes a general methodology applicable to convolution kernels. Section 6.2 describes the application of our idea to the subtree and subset tree kernels and shows experiments proving its effectiveness.

1.6 Origin of the Chapters

The material presented in chapter 4 is based on the following articles [2]. Chapter 5 is based on unpublished work. Chapter 6 is based on the following articles [3, 4].

Part I

Basics

Chapter 2

Background

This chapter introduces basic definitions and concepts necessary for understanding the works presented in chapters 4,5 and 6. Section 2.1 presents the notation and definitions related to trees. Sections 2.2 introduces the machine learning framework. Section 2.3 discusses those techniques for learning on structured data that are used in the following chapters. Since the focus of this work is on kernel methods, sections 2.3.3 introduce basic properties of one of the fundamental components of kernel methods, kernel functions. The chapter ends with section 2.3.4 by discussing ways to evaluate kernel functions.

2.1 Definitions and Notation

This section recalls basic definitions and notation that will be used in the following chapters. We start with some definitions on data structures.

A graph is a pair of sets $G = (V_G, E_G)$, where $V_G = \{v_1, v_n\}$ is an ordered set of nodes and $E_G = \{e_{ij} = (v_i, v_j), \ldots, e_{kl} = (v_k, v_l)\}$ a set of pairs of nodes, the edges. The subscript G will be omitted whenever it is clear from the context which graph we are referring to. An undirected graph is a graph for which $e_{ij} \in E \Leftrightarrow e_{ji} \in E$. A labelled graph is a graph for which a label is attached to each node. Labels will be represented by means of a function $l(v)$ or, when referring to a specific node v_i, by $l_i = l(v_i)$. A path $p(v_i, v_j) = v_i, \ldots, v_j$ in the graph G is a sequence of nodes for which

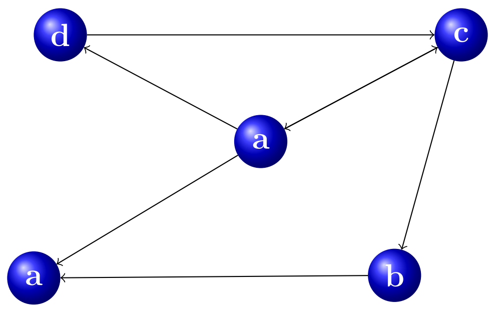

Figure 2.1: An example of a labelled directed graph.

there exists an edge connecting any adjacent nodes, i.e. $(p_i, p_{i+1}) \in E, 1 \leq i \leq l$, where p_i is the i-th node in the path and l is the length of the path (the number of nodes comprising p). Two nodes are connected if there exists a path connecting them. A graph is connected if every pair of distinct vertices in the graph is connected. A graph is said to have a cycle if there exists a path connecting a node with itself, i.e. $\exists\ p = p_1, \ldots, p_l.\ p_1 = p_l$.

Figure 2.1 gives an example of a labelled directed graph. Note that the graph is not connected since there is no path connecting nodes labelled with b and c.

A tree is a directed and connected graph without cycles for which every node has at most one incoming edge. A rooted tree is a tree for which there exists a node with no incoming edges (the root). In order to simplify the notation, we will use $v \in G$ has a shortcut for $v \in V_G$. A leaf is a node with no outgoing edges. If there is a link e_{ij}, node v_i is the parent of v_j and node v_j is a child of v_i. If v_j, v_k are children of v_i, then v_j and v_k are siblings. A node v_j is a descendant of v_i if there exists a path from v_i to v_j (in this case v_i is an ascendant of v_j). An ordered tree is one in which the children of each node are ordered according to some relation.

A positional tree is a tree for which each child node has associated an index representing its position with respect to its siblings. Note that the set of positional trees include the set of ordered trees. Figure 2.2 highlights the differences of an ordered tree with respect to a positional tree: edge labels represent the position of a node. In the following node positions for ordered trees will be omitted. The out-degree of a node is the highest positional index associated to a child of the

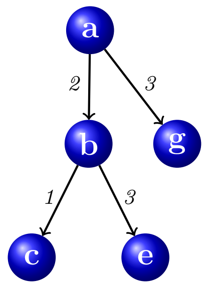

Figure 2.2: A positional Tree. The number over an arc represents the position of the node with respect to its parent.

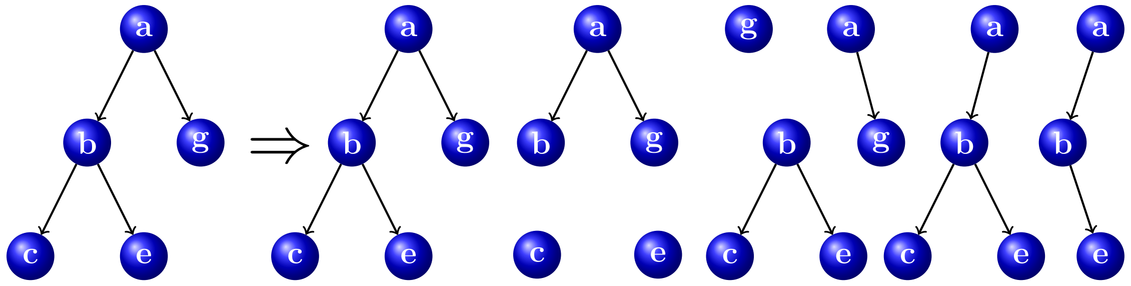

Figure 2.3: A tree (left) and some of its subtrees (right).

node. The maximum out-degree of a tree is the highest index of all the nodes of the tree. The out-degree of a node for an ordered tree corresponds to the number of its children. The depth of a node v_i with respect to one of its ascendants v_j is defined as the number of nodes comprising the path from v_j to v_i. When not specified, the node with respect to the depth is computed, is the root.

A tree can be decomposed in many types of substructures.

Subtree A subtree t is a subset of nodes in the tree T, with corresponding edges, which forms a tree. A subtree rooted at node v_i will be indicated with t_i, while a subtree rooted at a generic node v will be indicated by $t(v)$. When t is used in a context where a node is expected, t refers to the root node of the subtree t. The set of subtrees of a tree will be indicated by N_T. When clear from the context N_T may refer to specific type of subtrees. Figure 2.3 gives an example of a tree together with its subtrees. Various types of subtrees can be defined for a tree T.

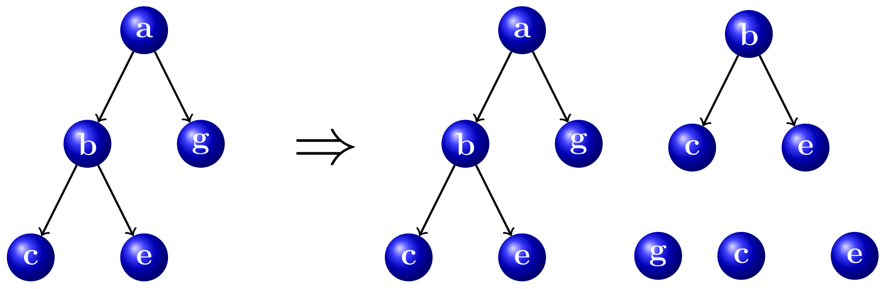

Figure 2.4: A tree (left) and all of its proper subtrees (right).

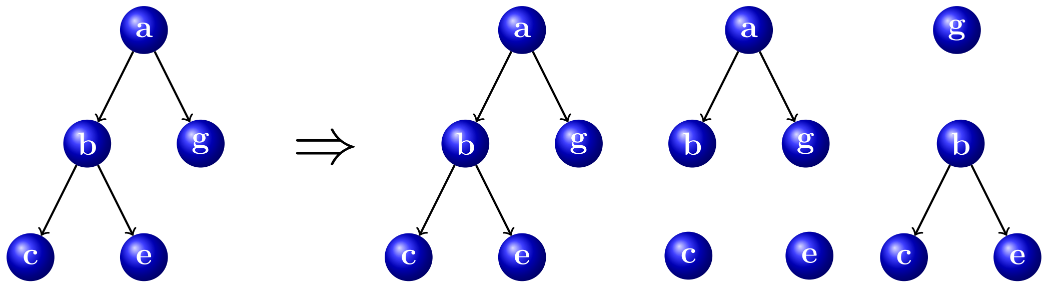

Figure 2.5: A tree (left) and all of its subset trees (right).

Proper Subtree A proper subtree t_i comprises node v_i along with all of its descendants (see figure 2.4 for an example of a tree along with all its proper subtrees).

Subset Tree A subset tree is a subtree for which the following constraint is satisfied: either all of the children of a node belong to the subset tree or none of them. The reason for adding such a constraint can be understood by considering the fact that subset trees were defined for measuring the similarity of parse trees in natural language applications. In that context a node along with all of its children represent a grammar production. Figure 2.5 gives an example of a tree along with some of its subset trees.

2.2 Machine Learning

The machine learning framework encompasses all algorithms capable of improving their behaviour with experience. According to the definition of Mitchell [44], a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, measured according to P, increases with experience E.

Two different scenarios can be distinguished in machine learning: supervised and unsupervised learning. In the supervised scenario a set of pairs, the training set, $S = \{(x_i, y_i) : i = 1, \ldots, n\}$ is provided to the learner. $x_i \in \mathcal{X}$ is the input example ($\mathcal{X}$ denotes the domain of the x_i, X is the set of all x_i appearing in S), $y_i \in Y$ is the label of x_i. Each (x, y) is generated according to an unknown distribution $P(x, y)$. S is assumed to be independent and identically distributed according to $P(x, y)$. The domain of Y determines the type of problem (the following list considers only problems of interest for the present work):

- If $y_i \in \{0, 1\}$ it is a two-class classification problem. It is the simplest case. Most machine learning classification algorithms belong to this class.

- If $y_i \in \{0, \ldots, n\}$ it is a multi-class classification problem. The prediction of an instance is selected among $n + 1$ classes.

- If $y_i \in \mathbb{R}$ it is a regression problem. Regression can be viewed as the problem of fitting a curve representing the target function.

- If $y_i \in \{0, 1\}^m$ it is a multi-label classification problem: the classification of x_i is a vector where each dimension represent the classification with respect to the corresponding label.

The task in supervised learning is to estimate a function $h : \mathcal{X} \rightarrow Y$, representing the relationship between x and y values, having at disposal only the set of examples S. The function h (also called hypothesis) belongs to a set H.

The best h, represented as h^*, minimizes the expected risk:

$$R(h) = \int L(h(x), y) dP(x, y) \tag{2.1}$$

where L is a loss function measuring the classification error of h. L can be, for instance, the total number of misclassified examples (binary loss).

Since the distribution $P(x, y)$ is unknown, it is not possible to directly use equation (2.1) for selecting the best h. A reasonable approach is to minimize the loss function with respect to the available data.

$$R_e(h) = \frac{1}{n} \sum_{(x,y) \in S} L(h(x), y) \tag{2.2}$$

The set of h such that $R(h) = 0$ is called Version Space [44]. This technique alone, however, does not lead to an optimal h since there can be infinite functions for which $\forall (x_i, y_i) \in S : h(x_i) = y_i$. The ability of a function to correctly classify unseen data is referred to as generalization capability. It is clearly of particular interest to express the generalization capability of an algorithm without referring to a specific instance of the problem (a specific set of data). Statistical Learning Theory is devoted to this problem. Among its results there is a characterization of the classes of functions with respect to the Vapnik-Chervonenkis (VC) dimension, a measure of the complexity of the class. The VC dimension of a family of functions H is defined as the cardinality of the largest subset of points of the domain that can be labelled arbitrarily by choosing a function $h \in H$. Loosely speaking the VC dimension grows with the ability of a set of functions to correctly classify any training set. The following theorem shows that the generalization ability of a family of functions decreases when increasing the VC dimension.

Theorem 2.1 *Let v be the VC dimension of the family of functions H. Then $\forall\ \delta > 0, h \in H$ dependent from a set of parameters Θ, the upper bound*

$$R(h(\Theta)) \leq R_e(h(\Theta)) + \Omega\left(\frac{VC(h(\Theta))}{n}\right), \tag{2.3}$$

where R_e is the empirical risk and n is the size of the training set, holds with probability at least $1 - \delta$ for $n > VC(h(\Theta))$. $\Omega\left(\frac{VC(h(\Theta))}{n}\right)$ is a monotonic increasing function and it is called the confidence interval.

Note that the generalization ability of an algorithm increases by having at disposal a larger amount of data. The confidence interval is also related to the VC dimension: if a function with low complexity is able to correctly classify the training set, then it is likely to have a low expected risk. The minimization of both terms of eq. (2.3) is important. When a function is able to correctly classify the training set but has a large error on the rest of the distribution, then the function is told to overfit the data. When a function has a low confidence interval but has not enough expressive power (it is not able to correctly classify the training set), it is told to underfit the data.

Since the minimization of the empirical risk alone does not guarantee to obtain high accuracy on the whole distribution, in order to obtain a useful solution, the learning process needs to incorporate a bias, based on a priori knowledge of the problem, for restricting the set of functions from which the selection of the best h is performed. Note that, since the choice of the bias is made before seeing the training set, the resulting class of functions, may not contain h^*. On the other side, given a bias, it is possible to build a training set such that any algorithm will perform arbitrarily bad. A priori knowledge may make take the form of

- a restriction of the family H from which h^* will be selected,
- a penalization for complex functions (Regularization). An example of a regularizer is a penalization term which influence the selection towards smooth functions.
- The selection of a functional class according to the structural risk minimization principle [10]. Let $H_1 \subseteq H_2 \subseteq \ldots \subseteq H_k$ be a sequence of family of functions with $VC(H_i) < VC(H_{i+1}), 1 \leq i < k$. Among those functions minimizing the empirical risk for each H_i, the structural risk minimization principle chooses the one minimizing also a bound of the form of eq. (2.3).

In the unsupervised learning scenario there is no label information available. The learner is provided with only a set of instances $X = \{x_i : i = 1, \ldots, n\}$. The task here is to find regularities in the set X. The most classical unsupervised technique is clustering, which has the aim of finding a partition of a dataset such that any object of a partition has higher similarity with objects in the same partition than with objects of different partitions.

Machine learning algorithms can be further classified into batch (or off-line) and on-line algorithms. For batch algorithms the distribution $P(x, y)$ generating the data is fixed, while for on-line algorithms it may vary in time. Batch methods have at disposal the whole training set and the learning and classification phase are distinct: once training has finished, the learner has no possibility to modify its behaviour, i.e. to adapt to new examples. In on-line methods data arrives sequentially and learning takes place together with classification. On-line algorithms must be less computational intensive because the two phases, learning and classification, must be executed together.

2.3 Machine Learning For Structured Data

The aim of this section is to describe some of the learning algorithms, applicable to structured data, that will be used in the following chapters.

2.3.1 Self Organizing Maps

The aim of the Self Organizing Maps (SOM) learning algorithm is to learn a *feature map*

$$\mathcal{M} : \boldsymbol{\mathcal{I}} \rightarrow \boldsymbol{\mathcal{A}} \tag{2.4}$$

which given a vector in the spatially continuous input space $\boldsymbol{\mathcal{I}}$ returns a point in the spatially *discrete* output display space $\boldsymbol{\mathcal{A}}$. This is obtained in the SOM by associating each point in $\boldsymbol{\mathcal{A}}$ to a different neuron. Moreover, the output space $\boldsymbol{\mathcal{A}}$ is typically obtained by arranging this set of neurons as the computation nodes of a one- or two-dimensional lattice. Given an input vector $\boldsymbol{x}_v$, the SOM returns the coordinates within $\boldsymbol{\mathcal{A}}$ of the neuron with the closest weight vector. Thus, the set of neurons induce a partition of the input space $\boldsymbol{\mathcal{I}}$. In typical applications $\boldsymbol{\mathcal{I}} \equiv I\!R^m$, where $m \gg 2$, and $\boldsymbol{\mathcal{A}}$ is given by a two dimensional lattice of neurons. In this setting, high dimensional input vectors are projected into the two dimensional coordinates of the lattice, with the aim of preserving, as much as possible, the topological relationships among the input vectors, i.e., input vectors which are close to each other should be projected to neurons which are close to each other on the lattice. The SOM is thus performing data reduction via a vector quantization approach.

In a more generic case, when the input space is a structured domain with labels in $\boldsymbol{\mathcal{U}}$, we redefine equation (2.4) to be:

$$\mathcal{M}^{\#} : \boldsymbol{\mathcal{U}}^{\#[i,o]} \rightarrow \boldsymbol{\mathcal{A}} \tag{2.5}$$

This can be realized through the use of the following recursive definition:

$$\mathcal{M}^{\#}(G) = \begin{cases} nil_{\boldsymbol{\mathcal{A}}} & \text{if } G = \xi \\ \mathcal{M}_{node}\left(\boldsymbol{u}_s, \mathcal{M}^{\#}(G^{(1)}), \ldots, \mathcal{M}^{\#}(G^{(o)})\right) & \text{otherwise} \end{cases} \tag{2.6}$$

where $s = source(G)$, $G^{(1)}, \ldots, G^{(o)}$ are the (eventually void) subgraphs pointed by the outgoing edges leaving from s, $nil_{\boldsymbol{\mathcal{A}}}$ is a special coordinate vector into the discrete output space $\boldsymbol{\mathcal{A}}$, and

$$\mathcal{M}_{node} : \boldsymbol{\mathcal{U}} \times \underbrace{\boldsymbol{\mathcal{A}} \times \cdots \times \boldsymbol{\mathcal{A}}}_{o \text{ times}} \rightarrow \boldsymbol{\mathcal{A}} \tag{2.7}$$

is a SOM, defined on a generic node, which takes in input the label of the node and the “encoding” of the subgraphs $G^{(1)}, \ldots, G^{(o)}$ according to the $\mathcal{M}^{\#}$ map. By “unfolding” the recursive definition in equation (2.6), it turns out that $\mathcal{M}^{\#}(G)$ can be computed by starting to apply $\mathcal{M}_{node}$ to leaf nodes, and proceeding with the application of $\mathcal{M}_{node}$ bottom-up from the frontier to the supersource of the graph G.

Model of $\mathcal{M}_{node}$

In the previous section we saw that the computation of $\mathcal{M}^{\#}$ can be recast as the recursive application of the SOM $\mathcal{M}_{node}$ to the nodes compounding the input structure. Moreover, the recursive scheme for graph G follows the skeleton skel(G) of the graph. In this section, we give implementation details on the SOM $\mathcal{M}_{node}$.

For each node v in V_G, we have a vector $\boldsymbol{u}_v$ of dimension m. Moreover, we realize the display output space $\boldsymbol{\mathcal{A}}$ through a q dimensional lattice of neurons. We assume that each dimension of the q dimensional lattice is quantized into integers, n_i, $i = 1, 2, \ldots, q$, i.e., $\boldsymbol{\mathcal{A}} \equiv [1 \ldots n_1] \times [1 \ldots n_2] \times \cdots \times [1 \ldots n_q]$. The total number of neurons is $\prod_{i=1}^{q} n_i$, and each "point" in the lattice can be represented by a q dimensional coordinate vector $\boldsymbol{c}$. For example, if $q = 2$, and if we have n_1 neurons on the horizontal axis and n_2 neurons on the vertical axis, then the winning neuron is represented by the coordinate vector $\boldsymbol{y} \equiv (y_1, y_2) \in [1 \ldots n_1] \times [1 \ldots n_2]$ of the neuron which is most active in this two dimensional lattice.

With the above assumptions, we have that

$$\mathcal{M}_{node} : I\!R^m \times ([1 \ldots n_1] \times \cdots \times [1 \ldots n_q])^o \rightarrow [1 \ldots n_1] \times \cdots \times [1 \ldots n_q], \qquad (2.8)$$

and the $m + oq$ dimensional input vector $\boldsymbol{x}_v$ to $\mathcal{M}_{node}$, representing the information about a generic node v, is defined as

$$\boldsymbol{x}_v \;=\; \left[\boldsymbol{u}_v \; \boldsymbol{y}_{ch_1[v]} \; \boldsymbol{y}_{ch_2[v]} \; \cdots \; \boldsymbol{y}_{ch_o[v]}\right], \qquad (2.9)$$

where $\boldsymbol{y}_{ch_i[v]}$ is the coordinate vector of the winning neuron for the subgraph pointed by the i-th pointer of v. In addition, we have to specify how $nil_{\boldsymbol{\mathcal{A}}}$ is defined. We can choose, for example, the coordinate $(\underbrace{-1, \ldots, -1}_{q})$.

Of course, each neuron with coordinates vector $\boldsymbol{c}$ in the q dimensional lattice will have an associated vector weight $\boldsymbol{w}_{\boldsymbol{c}} \in I\!R^{m+oq}$.

Notice that, given a DAG $\mathcal{D}$, in order to compute $\mathcal{M}^{\#}(\mathcal{D})$, the SOM $\mathcal{M}_{node}$ must be recursively applied to the nodes of $\mathcal{D}$. One node can be processed only if all the subgraphs pointed by it have already been processed by $\mathcal{M}_{node}$. Thus, the computation can be parallelized on the graph, with the condition that the above constraint is not violated. A data flow model of computation fits completely this scenario. When considering a sequential model of computation, a node update scheduling constituted by any inverted topological order for the nodes of the graph suffices to guarantee the correct computation of $\mathcal{M}^{\#}$.

Finally, it must be observed that, even if the SOM $\mathcal{M}_{node}$ is formally just taking care of single graph nodes, in fact it is also "coding" information about the structures. This does happen because of the structural information conveyed by the $\boldsymbol{y}_{ch_i[v]}$ used as part of the input vectors. Thus, some neurons of the map will be maximally active only for some leaf nodes, others will be maximally active only for some nodes which are roots of graphs, and so on.

Training algorithm for $\mathcal{M}_{node}$

The weights associated with each neuron in the q dimensional lattice $\mathcal{M}_{node}$ can be trained using the following process:

Step 1 (Competitive step). In this step the neuron which is most similar to the input node $\boldsymbol{x}_v$ (defined as in equation (2.9)) is chosen. Specifically, the (winning) neuron, at iteration t, with the closest weight vector is selected as follows:

$$\boldsymbol{y}_{i^*}(t) \quad = \quad \arg\min_{\boldsymbol{C}_i} \|\Lambda(\boldsymbol{x}_v(t) - \boldsymbol{m}_{\boldsymbol{C}_i}(t))\|, \tag{2.10}$$

where Λ is a $(m + cq) \times (m + cq)$ diagonal matrix which is used to balance the importance of the label versus the importance of the pointers. In fact, the elements $\lambda_{1,1}, \cdots, \lambda_{m,m}$ are set to μ, the remaining elements are set to 1-μ. Notice that if $cq = 0$ and $\mu = 1$, then the standard SOM algorithm is obtained.

Step 2 (Cooperative step). The weight vector $\boldsymbol{m}_{\boldsymbol{y}_{i^*}}$, as well as the weight vector of neurons in the topological neighborhood of the winning neuron, are moved closer to the input vector:

$$\boldsymbol{m}_{\boldsymbol{C}_r}(t+1) = \boldsymbol{m}_{\boldsymbol{C}_r}(t) + \eta(t) f(\Delta_{i^*r})(\boldsymbol{x}_v(t) - \boldsymbol{m}_{\boldsymbol{C}_r}(t)), \tag{2.11}$$

where the magnitude of the attraction is governed by the learning rate η and by a neighborhood function $f(\Delta_{i^*r})$. Δ_{i^*r} is the topological distance between

$\boldsymbol{c}_r$ and $\boldsymbol{c}_{i^*}$ in the lattice, i.e., $\Delta_{i^*r} = \|\boldsymbol{c}_r - \boldsymbol{c}_{i^*}\|$, and it controls the amount to which the weights of the neighboring neurons are updated. Typically, the neighborhood function $f(\cdot)$ takes the form of a Gaussian function:

$$f(\Delta_{i^*r}) \quad = \quad \exp\left(-\frac{\Delta_{i^*r}^2}{2\sigma(t)^2}\right) \tag{2.12}$$

where σ is the spread. As the learning proceeds and new input vectors are given to the map, the learning rate gradually decreases to zero according to the specified learning rate function type. Along with the learning rate, the neighborhood radius $\sigma(t)$ decreases as well[1].

Putting this all together, the training algorithm of the SOM-SD can be described as shown by Algorithm 1, where for the sake of notation we denote $\mathcal{M}_{node}$ by $\mathcal{M}$. We will use this concise notation also in the following.

In this version of the algorithm, the coordinates for the (sub)graphs are stored in $\boldsymbol{y}_v$, once for each processing of graph $\mathcal{D}$, and then used when needed[2] for the training of $\mathcal{M}$. Of course, the stored vector is an approximation of the true coordinate vector for the graph rooted in v. However, since the learning rate η converges to zero this approximation can be negligible.

The SOM can be considered as an instance of a general framework for processing of structured data [27]. Various extensions of the SOM has been described in literature. The Contextual Self-Organizing Map (CSOM-SD) model family is able to capture contextual information about the input structure, i.e. information about the ancestor of a node [24, 25]. The Graph SOM-SD model allows the processing of undirected graphs, and non-positional graphs where the order of edges is not relevant [26].

The heuristic nature of the SOM-SD can not formally guarantee to preserve the topology of the items in the input space. In order to overcome this limitation,

[1]Generally, the neighborhood radius in SOMs never decreases to zero. Otherwise, if the neighborhood size becomes zero, the algorithm reduces to vector quantization (VQ).

[2]Notice that the use of an inverted topological order guarantees that the updating of the coordinate vectors $\boldsymbol{x}_v$ is done before the use of $\boldsymbol{x}_v$ for training.

Algorithm 1: Stochastic Training Algorithm for SOM-SD

```
input: Set of training DAGs $T = \{\mathcal{D}_i\}_{i=1,\ldots,N}$, $o$ maximum outdegree of DAGs in $T$, map $\mathcal{M}$, $Niter$
        number of training iterations, $\mu$ structural parameter, $\eta(0)$, $\sigma$, network size;

begin

initialize the weights for $\mathcal{M}$ with random values from within $\boldsymbol{\mathcal{U}}$;

for $t = 1$ to $Niter$

    shuffle DAGs in $T$;

    for $j = 1$ to $N$

        List($\mathcal{D}_j$) $\leftarrow$ an inverted topological order for vert($\mathcal{D}_j$);
            for $v \leftarrow$first(List($\mathcal{D}_j$)) to last(List($\mathcal{D}_j$)) do
                $\boldsymbol{y}_v \leftarrow \arg\min_{[a,b]} \left(\mu \, \|\boldsymbol{u}_v - \boldsymbol{m}^{(l)}_{[a,b]}\| + (1-\mu) \, \|\boldsymbol{y}_{\mathrm{ch}[v]} - \boldsymbol{m}^{(r)}_{[a,b]}\|\right)$;
                foreach $\boldsymbol{m}_{[c,d]} \in \mathcal{M}$ do
                    $\boldsymbol{m}^{(l)}_{[c,d]} \leftarrow \boldsymbol{m}^{(l)}_{[c,d]} + \alpha(t) f(\Delta_{[c,d]}, \boldsymbol{y}_v) \, (\boldsymbol{m}^{(l)}_{[c,d]} - \boldsymbol{u}_v)$;
                    $\boldsymbol{m}^{(r)}_{[c,d]} \leftarrow \boldsymbol{m}^{(r)}_{[c,d]} + \alpha(t) f(\Delta_{[c,d]}, \boldsymbol{y}_v) \, (\boldsymbol{m}^{(r)}_{[c,d]} - \boldsymbol{y}_{\mathrm{ch}[v]})$;

return $\mathcal{M}$;

end
```

Gianniotis and Tino [20] have proposed a model based approach for constructing topographic maps of tree structured data. The model is formulated as a constrained mixture of hidden Markov tree models. The maps are formulated in a principled framework of probability theory and thus are more theoretically grounded than SOM-SD.

2.3.2 Kernel Methods

The class of kernel methods comprises all those algorithms that do not require an explicit representation of the examples but only information about the similarities among them. The information is given by the kernel functions (for a definition see Section 2.3.3). Any kernel method can be decomposed into two modules:

- a problem specific kernel function.
- A general purpose learning algorithm.

Since the solver interfaces with the problem only by means of the kernel function, it can be used with any kernel function, and vice versa. The modularity of the approach allows to study the two aspects of learning, i.e. representation and optimization, independently.

Kernel methods look for linear relations in the feature space. In the following, for simplicity, the task of classification is considered. The problem is generally expressed as a constrained optimization problem where the objective function usually take the form of eq. (2.3). If the kernel function employed is symmetric positive semidefinite the problem is convex and thus has a global minimum. Note that a global minimum of the cost function exists for any choice of the parameters and kernel function. Thus the global minimum does not correspond to the optimal solution for the problem.

Wahba's representer theorem [67] states that the solution of certain optimization problems involving an empirical risk term and a quadratic regularizer can be written in terms of an expansion of the training examples. Thus, given a dataset

$S = \{(x_i, y_i) : i = 1, \ldots, n\}$ and a kernel function K, the solution w of the problem can be expressed as:

$$w = \sum_{i}^{n} \alpha_i y_i \phi(x_i). \tag{2.13}$$

Before showing how to classify an example, the score function must be introduced:

$$S(x) = \langle w, x \rangle = \sum_{i}^{n} \alpha_i y_i \phi(x_i) \phi(x) = \sum_{i}^{n} \alpha_i y_i K(x_i, x). \tag{2.14}$$

Note that the score function can be expressed as a weighted linear combination of kernel function evaluations between examples in the dataset and x. The classification $c(x)$ of an example with respect to w and kernel K is the sign of the score function:

$$c(x) = sign\left(S(x)\right) = sign\left(\sum_{i}^{n} \alpha_i y_i K(x_i, x)\right), \tag{2.15}$$

The two modules comprising kernel methods, i.e. representation and problem optimization, are discussed in detail in the rest of the chapter. The following two sections describe the kernel methods used in the following: the perceptron and the Support Vector Machines, respectively. Note that the two algorithms are for binary classification problems, but they can be applied to an n-class problem by adopting the one-against-all methodology: first n binary classifiers, each devoted to recognize a single class, are trained. Then, the prediction for the n-class problem is given by the class whose associated classifier gets the highest confidence (score).

Perceptron

In the original formulation the perceptron [56] was meant to classify data encoded by real vectors with a linear decision function (a hyperplane).

Every element of the dataset is represented by a feature vector. A prototype vector w is randomly initialized. Then the classification of each example x_i is compared to the one made by the prototype, computed according to the following formula:

$$f(x) = sign(w \cdot x_i + b)$$

If the perceptron is classifying uncorrectly the example then a new prototype w' is generated from w:

$$w' = w + \alpha y_i x_i$$

where α is a constant ($\alpha > 0$), $y_i \in \{-1, 1\}$ is the class of x_i. The algorithm has been demonstrated to converge to the optimal hyperplane provided that the data are linearly separable [51].

Using the kernel trick it is possible to extend the perceptron to generate a non-linear decision function and/or to treat structured data by using kernels (see for example [37]).

The on-line kernel-perceptron algorithm, adapted to tree-kernels, requires to maintain an implicit representation of the vector w in the feature space. Specifically, this corresponds to keep in memory the set of the already seen examples for which the perceptron prediction was erroneous.

Thus we can consider the set of examples $M = \{(x_i, y_i) \in S : \alpha_i \in \{-1, +1\}\}$ as the *model* of the perceptron and slightly redefine the kernel-perceptron algorithm as in the following. Let $M = \emptyset$ be an initial empty model, a new example x_i is added to the model M whenever its score

$$S(x_i) = \sum_{(x_j, y_j) \in M} y_j K(x_i, x_j)$$

has different sign from its classification y_i. Thus the update and the insertion of the new example follow the rule:

$$\textbf{if } (y_i S(x_i) \leq 0) \textbf{ then } M \leftarrow M \cup \{(x_i, y_i)\}$$

For many applications (see page 35), the cardinality of M, and consequently the memory required for its storage, grows up linearly with the number of tree presentations. Moreover the efficiency in the evaluation of the function $S(x)$ decreases super-linearly. Clearly, this seems not satisfactory for on-line applications.

The perceptron is a simple and relatively fast algorithm. Its main drawback is that it does not provide bounds on the generalization error.

An interesting and effective variant of the simple Perceptron algorithm is the *voted Perceptron* proposed in [18]. This algorithm is motivated by a theory related to converting an on-line learning algorithm into a batch one. Basically, it uses a deterministic version of a simple "leave-one-out" method whose randomized version was proposed in [29]. Specifically, the idea of the voted perceptron is to combine the predictions of the hypotheses visited by the Perceptron algorithm while its training takes place. These hypotheses can be combined in different ways, for example each hypothesis can be given a weight equal to the number of times the same hypothesis has 'survived', i.e. the number of iterations until the next mistake has been made.

Support Vector Machines

Support Vector Machines (SVMs) are based on the Structural Risk Minimization principle for which bounds on the generalization error have been proven [65]. SVM is a binary classifier which projects the examples in a feature space and then looks for an hyperplane separating positive and negative examples. Among the hyperplanes separating the data, it is chosen the one maximizing the margin, i.e. the minimum distance between the hyperplane and the closest example. It is possible to show that the VC dimension of a linear classifier can be upper bounded in terms of the margin [65]. If the training set is linearly separable the separating hyperplane maximizing the margin is unique and corresponds to the solution of the following problem:

$$\begin{array}{l} \arg\min_{w,b} \frac{||w||^2}{2} \\ \text{subject to } \forall(x_i, y_i) \in S . y_i(w \cdot \phi(x_i)) + b \geq 1 \end{array} \tag{2.16}$$

where w and b define the hyperplane in the feature space. Considering that the margin is inversely proportional to the norm of w, minimizing $||w||$ corresponds to finding the less complex function satisfying the constraints in 2.16, i.e. the simplest function correctly classifying each example. The representer theorem [57] states that the solution f of the problem 2.16 can be expressed as:

$$\forall x \in \mathcal{X} . f(x) = \sum_{x_i \in S} \alpha_i k(x_i, x).$$

The examples for which the corresponding α is not 0 are called support vectors.

When the training set is not linearly separable, a function f separating the two classes may be very complex. If the non linear separability is due to noise, perfect classification of the training set is not desirable since it may overfit the data and thus reduce the expected risk. In this case a tradeoff between function complexity and minimization of training error (constraints satisfaction) should be pursued. The problem 2.16 then becomes:

$$\begin{array}{ll} \arg\min_{w,b,\xi} & \frac{||w||^2}{2} + c\sum_{i=1}^{n} \xi_i \\ \text{subject to} & \forall (x_i, y_i) \in S.\ y_i(w \cdot \phi(x_i)) + b \geq 1 - \xi_i \\ & \xi_i \geq 0 \quad i = 1, \ldots, n \end{array} \tag{2.17}$$

The constraints in 2.17 are relaxed with respect to the correspondent constraints in 2.16. The parameter c determines the balance between minimization of training error and minimization of expected risk. The parameter is problem dependent and its best value has to be found empirically.

2.3.3 Kernel Functions

A way for assessing the similarity of objects of a domain is to describe them by a set of features and then count the number of common features. For reasons that will be clear in the following, the space of the features is assumed to be a metric space. A metric space X is a vector space in which a distance $d : X \times X \rightarrow \mathbb{R}^+$ is defined such that $\forall\ x, x', x'' \in X$ the following properties hold:

- $d(x, x') \geq 0$
- $d(x, x') = 0 \Leftrightarrow x = x'$
- $d(x, x') = d(x', x)$
- $d(x, x') \leq d(x, x'') + d(x'', x')$.

The representation in feature space is obtained by the application of an appropriate function ϕ, $x \rightarrow \phi(x) = \{\phi_i(x) | i \geq 1\}$. The elements $\phi_i(x)$ are called the features of x (according to the mapping ϕ). Note that if ϕ is a non linear function, the relative positioning of the objects in the feature space can change with respect to the original space. In this sense the use of appropriate kernel functions may (and it is supposed to) simplify the problem.

The similarity between two objects x, x' can be computed by the dot product of their representation in feature space i.e. $\langle x, x' \rangle = \sum_i^m = \phi_i(x)\phi_i(x')$, where $m = |\phi(x)|$. A kernel is a function measuring the similarity of any pair of objects of a domain, $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which corresponds to a closed form for the dot product of the projection of the examples in feature space.

The Gram matrix G^K related to a kernel K with respect to a set S of examples is defined as

$$G^K_{i,j} = K(x_i, x_j). \tag{2.18}$$

A kernel function is valid if and only if it is symmetric semidefinite positive, i.e. if any of its Gram matrices are symmetric positive semidefinite. A matrix is symmetric if $\forall i, j K(x_i, x_j) = K(x_j, x_i)$ and it is positive semidefinite if $\forall c_1, \dots, c_n \in \mathbb{R}. \sum_{i,j} c_i K(x_i, x_j) c_j = \mathbf{c}^T G^K \mathbf{c} \geq 0$, where $\mathbf{c}^T$ is the tranpose of $\mathbf{c}$. Equivalently a matrix is positive semidefinite if all of its eigenvectors are nonnegative. A kernel function can be expressed as a dot product in a feature space ϕ such that $K(x_i, x_j) = \sum_i \phi_n(x_i)\phi_n(x_j)$.

In the following, when it is clear from the context we will use the term kernel in place of valid kernel.

Given two examples x_i and x_j, the relationship between the distance $d(x_i, x_j)$ in feature space and the kernel $K(x_i, x_j)$ is

$$d(x_i, x_j) = \sqrt{K(x_i, x_i) + K(x_j, x_j) - 2K(x_i, x_j)}.$$

When two examples are mostly dissimilar, the application of a kernel K to them returns 0. When the kernel is normalized (see eq. (2.19)) the maximum value of K is 1.

The class of kernel functions is closed under the operations described in proposition 2.1.

Proposition 2.1 *Let $K_1, K_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be two kernel functions, $X = \{x_1, \ldots, x_n\}$ a set of examples from the domain χ. Then*

1. *$K(x, x') = K_1(x, x') + K_2(x, x')$ is a valid kernel [59] (additive property).*
2. *$K(x, x') = K_1(x, x')K_2(x, x')$ is a valid kernel (multiplicative property). Note that the property holds when K_2 is a positive constant, i.e. multiplying the kernel by a positive constant gives a valid kernel.*
3. *$K(x, x') = f(x)f(x')$, where f is any function defined on the domain χ.*
4. *$K(x, x') = K_4(\phi(x), \phi(x'))$. An application of this property is shown together with the definition of the polynomial kernel (eq. (2.21)).*
5. *$K(x, x') = K_1 \oplus K_3 ((x, u)(x', u')) = K_1(x, x') + K_3(u, u')$, where $K_3 : U \times U \to \mathbb{R}$ is a valid kernel defined on the domain U, is a valid kernel (direct sum property).*
6. *$K(x, x') = K_1 \otimes K_3 ((x, u)(x', u')) = K_1(x, x')K_3(u, u')$, where $K_3 : U \times U \to \mathbb{R}$ is a valid kernel defined on U, is a valid kernel (tensor product property).*

The proofs of the properties (or references to them) can be found in [28, 59]. These properties show that novel kernels can be defined by combining existing kernels. It is possible to combine kernels taking into account different aspects of the data, for example a kernel for web pages can be constructed by the combination of a kernel defined on the set of words in the page and a kernel defined on the incoming and outgoing links.

When only the orientation of the ϕ matters, the representation in feature space and thus the kernel values can be normalized by the following operation:

$$K'(x, x') = \frac{K(x, x')}{\sqrt{K(x, x)K(x', x')}} \tag{2.19}$$

Normalization can be useful when the probability of generating a feature depends on the size of the original data. For example a representation that counts the number of times each label appears in a tree is influenced by the size of the tree: larger trees have higher chances to have many common features with any small tree.

In the following we give some examples of popular kernels for vectorial data. Probably the simplest approach for dealing with structured data is to first transform it to vectorial form and then apply kernel functions defined for vectorial data. This section briefly reviews kernels for vectorial data. In the following we may generically refer to kernels defined in this section as standard kernels.

In the following x, y represent vectors belonging to a space $\mathbb{R}^m$. The simplest kernel known in literature is the linear kernel:

$$K(x, y) = \langle x, y \rangle \,. \tag{2.20}$$

Note that feature space of the linear kernel coincides with the input space.

Another widely used kernel for structured data is the polynomial kernel:

$$K(x, y) = (\langle x, y \rangle + e)^d, \quad e \in \mathbb{R}, \quad d \in \mathbb{N}. \tag{2.21}$$

The feature space associated with the polynomial kernel is composed by products of elements of the original vectors. d is the maximal order of the resulting monomials. When $e = 0$, the feature space is composed by all possible products of groups of d features. Thus the feature space has dimension:

$$|\phi(x)| = \binom{n+d-1}{d} = \frac{(n+d-1)!}{(n-1)!d!}.$$

For example given a vector $x = (x_1, x_2, x_3)$ its representation in feature space for $e = 0$ and $d = 2$ is $\phi(x) = (x_1^2, x_2^2, x_3^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3)$. Note that $|x| + 2$ operations are required for evaluating eq. (2.21), while an explicit evaluation by means of the feature vectors $\phi(x)$ would have required $\binom{n+d-1}{d}$ operations. The dot product of eq. (2.21) can be replaced by any kernel function (see page 31). The resulting operation allows to create new features as a combination of the original ones.

The last function we describe is the gaussian kernel,

$$K(x, y) = exp\left(-\frac{||x-y||^2}{2\sigma^2}\right), \quad \sigma \in \mathbb{R}. \tag{2.22}$$

The gaussian kernel has the particularity to have a feature space of infinite size. Note that in this case the use of a kernel function not only saves computational time, but it is the only way for computing the corresponding dot product in feature space. The kernel value is maximum when $x = y, K(x, x) = 1$ and it is monotonic decreasing when the distance between x and y increases. The parameter σ affects the resulting feature space as follows:

- for very high values of σ, all examples become almost parallel and thus all examples are almost identical.
- Very low values of σ produce feature vectors all orthogonal to each other.

In [30] the effects of feature spaces such as those obtained for the gaussian kernel for extreme values of σ, on a kernel method called ν-Support Vector Machine [58], are described in detail.

Kernel functions described in this section are suitable for dealing with vectorial data. The present work is focused on tree structured data. The kernels described in literature for this type of data are discussed in section 3.

2.3.4 Evaluating Kernel Functions

It is clear from previous discussions (cfr section 2.3.2) that designing "good" kernel functions is a major concern for obtaining a successful application. But what is a good kernel function? This section summarizes the contributions in literature for helping answering this question. All the following discussion refers to the classification problem.

Being valid is a necessary requirement for a kernel function. However not all valid kernels all equally good for a task. For example a kernel such that $\forall x_i, x_j \in \mathcal{X}, i \neq j. K(x_i, x_j) = 0$ is a valid kernel but will have a poor generalization capability because no information is available for points in feature space except for those in the training set. We formalize the concept by introducing the sparsity index

$$Sparsity(K, S) = \frac{|\{(i,j) \in S | K(i,j) = 0\}|}{|S|^2}. \tag{2.23}$$

The sparsity index computes the proportion of example pairs in the instance set S whose kernel value is 0.

On the contrary, given a set of examples x_i along with their classification y_i (assume for simplicity to be solving a two-class classification problem), it is easy to design a valid optimal kernel: $K(x_i, x_j) = y_i y_j$.

Cristianini et al. [14] have defined a measure for assessing the appropriateness of a kernel function in a supervised setting called kernel alignment. Let $S = \{x_1, x_2, \ldots, x_n\}$ be the set of instances compounding a training set and K_1, K_2 two kernel functions defined on S. The empirical alignment between K_1 and K_2 is defined as the Frobenius inner product between the corresponding normalized Gram matrices(see eq. (2.18)):

$$A(K_1, K_2, S) = \frac{\left\langle G^{K_1}, G^{K_2} \right\rangle_F}{\sqrt{\left\langle G^{K_1}, G^{K_1} \right\rangle_F \left\langle G^{K_2}, G^{K_2} \right\rangle_F}}, \tag{2.24}$$

where $\left\langle G^{K_1}, G^{K_2} \right\rangle_F = \sum_{i,j=1}^{n} G^{K_1}_{i,j} G^{K_2}_{i,j}$.

Values of A range from -1 to 1. The higher the value of $A(K_1, K_2, S)$, the higher the similarity between K_1 and K_2 with respect to S.

The value A can be used to measure how appropriate a kernel K is for a given two-class classification task by aligning K with a matrix Y defined as: $Y_{i,j} = y_i y_j$, where $y_i = \{-1, +1\}$ is the class associated to an instance x_i. In the case of a multiclass classification task Y can be defined as $Y_{i,j} = 1$ if $y_i = y_j$ and $Y_{i,j} = 0$ if $y_i \neq y_j$. Note that the codomain of A for the multiclass case ranges from 0 to 1. In [33] the notion of alignment is extended to the case of classification of unbalanced datasets and the problem of regression.

Gärtner [19] defined three properties that should be fulfilled by a good kernel: completeness, correctness, appropriateness. Let $c : \mathcal{X} \rightarrow \Omega$ be a function that assigns to every example of a domain its class. Functions c are grouped into a concept class C.

A kernel that is able to incorporate all necessary knowledge for solving a problem is said to be complete. A kernel is complete if no pair of different examples have the same representation (in terms of the kernel): $K(x_i, \cdot) = K(x_j, \cdot) \Rightarrow x_i = x_j$. Any kernel for structured data that discards information about the structure is hardly complete. When information about class membership is available, the constraint $x_i = x_j$ can be relaxed thus turning the definition into: $\forall c \in C.K(x_i, \cdot) = K(x_j, \cdot) \Rightarrow c(x_i) = c(x_j)$.

A kernel is said to be correct with respect to a concept class C and an hypothesis space, if for every concept can be found an hypothesis that correctly classifies all examples. In the case of the Support Vector Machines, the hypothesis space is composed of all linear functions in feature space. Thus the definition of correctness become: $\forall c \in C, \exists \alpha_i \in \mathbb{R}, x_i \in \mathcal{X}, \theta \in \mathbb{R}$ such that $\forall x \in \mathcal{X}. \sum_i \alpha_i K(x_i, x) \geq \theta \Leftrightarrow c(x)$.

Appropriateness refers to the extent to which examples that are close to each other in class membership are also close to each other in feature space. A kernel is appropriate for learning concepts in a given concept class by a learning algorithm if polynomial bounds on its generalization error can be derived for some algorithms using this kernel. A complete and correct kernel separated the concept well, i.e. it is able to achieve high accuracy on the given data. An appropriate kernel is able to generalize well to unseen data.

All previous discussions take into account the quality of a kernel. When background knowledge cannot drive the choice of substructures, an effective kernel may be obtained by arbitrarily enlarging the feature space. However using a exponential number of features without having a polynomial kernel function for implicitly computing the dot products between examples may lead to intractable algorithms. The reasoning is valid not only for extreme cases. Even quadratic kernel complexity, in some practical situations, can lead to unacceptable running time. The reason is that, for most complex (and usually more accurate) kernel methods, kernel function evaluations heavily affect the total running time of the algorithm. In order to give an illustrative example of this claim, we computed some statistics on an execution of the Support Vector Machines algorithm (see section 2.3.2) by using the svm-light software [32, 45, 47] on the dataset from INEX 2005 Competition (a description of the dataset can be found in section A.1). The problem is a 11-class classification task but a two-class problem was derived by giving label +1 to the examples of the first class and −1 to the examples of all other classes. The statistics were collected by the gprof utility [22]. The kernel function used is the subset tree kernel (see section 3.1.2). The standard c parameter of the Support Vector Machines took the values $\{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$. All other parameters were given default values. The algorithm spends on average the 93.27% of its execution time by performing kernel evaluations. While the value depends on the problem, the solver and the kernel function, it gives a qualitative idea of the dependence of the execution time from the kernel evaluation procedure. Thus kernel function complexity should be kept as low as possible, especially when dealing with large datasets.

In practical applications, the right trade-off between expressive power and computational complexity should be selected according to the current task.

Chapter 3

State of the Art on Tree Kernel Functions

Kernel methods make use of kernel functions to measure the similarity of the items in feature space. Kernel functions are the only type of information specific to the task that this class of learning algorithms may use. Thus they play a crucial role for the final outcome of the kernel method.

The methodologies for designing kernels for trees include:

- The use of the convolution kernel framework. Convolution kernels are based on the idea that a complex object can be described in terms of its constituent parts, for example a string can be described in terms of its substrings and a tree in terms of its subtrees. Thus a convolution kernel measures the similarity of two objects in terms of the similarities of their subparts.

- Explicit extraction of features from trees. This method builds a vectorial representation of the data and then applies kernels for vectorial data to the representation obtained. For an example see section 3.2.1.

- Kernels based on generative models. These kernels measure the similarity of two items as a function of the parameters and states reached by a generative model for the data. An example of this class of kernels is the Fisher kernel (see section 3.2.2).

In the following, we will discuss the techniques listed previously. Due to their popularity, convolution kernels have received much attention from the research community. Section 3.1 describes the convolution kernel framework and reviews kernels derived from it. The description of other types of kernels are grouped together in section 3.2.

3.1 Convolution Kernels

Convolution kernels for structures were first introduced by Haussler in [28]. They are a general methodology for computing kernels on complex discrete objects.

By splitting the original object into parts and assuming to have at disposal a positive semidefinite kernel on the parts, Haussler [28] describes a way for combining the kernels on the parts that preserves positive semidefiniteness. In particular, convolution kernels express a kernel on a discrete object by a sum of kernels of their constituent parts.

Definition 3.1 (Convolution Kernel) *Let $\chi, \chi_1, \chi_2, \ldots, \chi_D$ be $D+1$ non empty separable metric spaces, $x \in \chi$ a structure and $\vec{x} = (x^1, x^2, \ldots, x^D)$ the parts of x. A relation $R : \chi_1 \times \chi_2 \times \ldots \times \chi_D \times \chi$ where $R(\vec{x}, x)$ is true if and only if $x^1, x^2, \ldots, x^D$ are the parts of x. Moreover let $R^*(x)$ be the set of all the subparts of x. Then the convolution kernel can be expressed as:*

$$k(x_i, x_j) = \sum_{\vec{x_i} \in R^*(x_i)} \sum_{\vec{x_j} \in R^*(x_j)} \prod_{d=1}^{D} k_d(x_i^d, x_j^d), \tag{3.1}$$

where the k_d are kernels defined on the substructures. In the following we will refer to the k_d as local kernels. In [28] it is demonstrated that, if the k_d are positive semidefinite, the kernel in eq. (3.1) is also positive semidefinite.

Recently Shin et al. [60] have introduced a more general form than the one of eq. (3.1) called the mapping kernel. Basically the idea of the mapping kernel is to restrict the set of substructure pairs over which the local kernel is computed. Formally speaking, the mapping kernel is defined as follows: let each $x \in \chi$ be associated with a finite subset χ'_x, where χ'_x is the set of substructures associated with x. Similarly to [28], it is assumed to have at disposal a positive semidefinite kernel $k : \chi^{'} \times \chi^{'} \to \mathbb{R}$. Then the mapping kernel is defined as follows:

$$K(x_i, x_j) = \sum_{(x'_i, x'_j) \in \mathcal{M}_{x_i, x_j}} k(x'_i, x'_j), \tag{3.2}$$

where $\mathcal{M}$ is part of a mapping system $\mathbb{M}$ defined as follows:

$$\mathbb{M} = \left(\chi, \left\{\chi^{'}_{x} | x \in \chi\right\}, \left\{\mathcal{M}_{x_i,x_j} \subseteq \chi^{'}_{x_i} \times \chi^{'}_{x_j} | (x_i, x_j) \in \chi \times \chi\right\}\right). \tag{3.3}$$

$\mathbb{M}$ is a triplet composed by the domain of the examples, the space of the substructures of the examples, and a function $\mathcal{M}$ specifying for which pairs of substructures the local kernel has to be computed.

$\mathcal{M}$ is assumed to be finite and symmetric, i.e. $\forall x_i, x_j \in \chi. |\mathcal{M}_{x_i,x_j}| < \infty$ and $(x'_j, x'_i) \in \mathcal{M}_{x_j,x_i}$ if $(x'_i, x'_j) \in \mathcal{M}_{x_i,x_j}$. In [60] it is proved that the kernel K of eq. (3.2) is positive semidefinite *if and only if* the mapping system $\mathcal{M}$ is *transitive*. A mapping systems is transitive if and only if $\forall x_1, x_2, x_3 \in \chi. (x'_1, x'_2) \in \mathcal{M}_{x_1,x_2} \wedge (x'_2, x'_3) \in \mathcal{M}_{x_2,x_3} \Rightarrow (x'_1, x'_3) \in \mathcal{M}_{x_1,x_3}$. The main advantage of the mapping kernel is to give a necessary and sufficient condition for building kernels on complex objects which is typically much easier to prove than positive semidefiniteness.

Since there will references to the mapping kernel in the following, an example is given in order to clarify its definition. Consider the linear kernel between two vectors x and u, $K(x, u) = \sum_i^m x_i u_i$. It can be seen as a special case of mapping kernel where:

- an m-dimensional vector is described by a set of pairs $(x_i, i) \in \chi \times \mathbb{N}$.
- χ_x is the set of pairs composed by the components of the vector x and their position in the vector.
- $((x', i), (u', j)) \in \mathcal{M}_{x,u}$ if $i = j$.

For example the 3-dimensional vector $x = (1, 4, 5)$ is described by $\big((1, 1), (4, 2), (5, 3)\big)$.

The kernel is defined as:

$$K(x,y) = \sum_i \sum_j \delta(i,j) x_i u_j = \sum_i x_i u_i,$$

where $\delta(i,j)$ is the dirac function returning 1 whether the two arguments are equal, 0 otherwise.

Convolution kernels have been successfully applied to a variety of problems involving structured data (for references see the following sections), thus appearing to be a viable way of dealing with structured data. The following sections are devoted to review convolution kernels for tree structured data.

3.1.1 Subtree Kernel

Viswanathan et al. [66] describe a string kernel which is also applicable to trees. The features of the kernel when applied to strings or trees are, respectively, the substrings or the proper subtrees of the input string (tree). Examples of proper subtrees can be found in Figure 2.4.

The kernel returns a weighted sum of the number of common substrings (proper subtrees). While there is a quadratic number of substrings of a string, the assumption which the proposed procedure is based on is that the number of matching substrings is small compared to the size of the feature space. The kernel for strings is defined as follows:

$$K(x_i, x_j) = \sum_{s \in x_i} \sum_{u \in x_j} [\![s = u]\!] w_s = \sum_{s \in \mathcal{A}^*} h_s(x) h_s(y) w_s,$$

where s, u are substrings of the strings x_i, x_j, w_s is the weight associated to the substring s, $\mathcal{A}^*$ is the set of non empty strings of the alphabet $\mathcal{A}$, $h_s(x)$ counts the frequency of the substring s in x, and $[\![condition]\!]$ is a function returning 1 whether *condition* is true, i.e. the two substrings are identical, 0 otherwise.

In order to efficiently compute the set of common substrings
Viswanathan et al. [66] propose to represent the set of substrings by a suffix tree [64].

By keeping a look-up table for associating weights to substrings it can be shown that the kernel can be computed in linear time in the size of the strings.

The kernel can be used for comparing ordered trees by first encoding them into a string format. The encoding of a proper subtree starting at node v is given by the function $tag(v)$, which can be computed according to the following procedure:

- if v is an unlabelled leaf then $tag(v) = [\,]$;
- if v is a labelled leaf then $tag(v) = [\,l(v)\,]$;
- if v is an unlabelled node with children $v_1, v_2, \cdots, v_c$, $tag(v) = [\,tag(v_1)\,tag(v_2)\,\cdots\,tag(v_c)\,]$, where the tags of children are sorted in lexicographical order;
- if v is labelled then $tag(v) = [\,l(v)\,tag(v_1)\,tag(v_2)\,\cdots\,tag(v_c)\,]$, where again tags are considered sorted in lexicographical order.

For example, the tree on the left of Figure 2.4 can be represented by $[a[b[c][e]][g]]$.

In theorem 1 of [66] it is proven that:

1. the tag of the root node, $tag(root)$, is an unique identifier of the tree and can be constructed in $(\lambda+2)(n \log_2 n)$ time and $O(n)$ space, where n is the number of nodes and λ is the maximum length of a label;
2. all those strings $tag(v)$ starting with [and ending with the corresponding balanced] are proper subtrees;
3. arbitrary substrings of $tag(root)$ correspond to subset trees of the input tree.

In [43] it is pointed out that not all subset trees can be generated if the trees are represented as strings by the encoding proposed in this section. Figure 2.5 gives an example of a subset tree, the string $[a[b][g]]$, which cannot be represented by a substring of $[a[b[c][e]][g]]$. While for any tree x, the set of its substrings (obtained by the encoding described in this section) include the set of its proper subtrees, it is possible to build a kernel counting the number of proper subtrees by setting $w_s = 0$ for all those substrings not starting and ending with balanced brackets.

3.1.2 Subset Tree Kernel

As seen in section 3.1.1, the technique proposed in [66] can not measure a similarity function based on common subset trees. A kernel for trees, based on counting matching subset trees, has been proposed in [12].

Let's consider a finite set of trees in which m different subset trees are present. In this case each feature, i.e. subset tree, can be indexed by an integer between 1 and m. Then $h_s(T)$ is the number of times the subset tree indexed with s occurs in tree T. We represent each tree T as a feature vector $\phi(T) = [h_1(T), h_2(T), \ldots, h_m(T)]$. The inner product between two trees under the representation $\phi(T) = [h_1(T), h_2(T), \ldots\ h_m(T)]$ is:

$$K(T_1, T_2) = \phi(T_1) \cdot \phi(T_2) = \sum_{s=1}^{m} h_s(T_1) h_s(T_2).$$

Thus the subset tree kernel (SST) defines a similarity measure between trees which is proportional to the number of shared subset trees.

The subset tree can be efficiently calculated by a recursive procedure as follows. Let the function $I_i(n)$ be equal to 1 if the subset tree indexed by i is rooted at node v, 0 otherwise. Then $h_i(T_1) = \sum_{t_1 \in N_{T_1}} I_i(v_1)$. The subset tree kernel can be written as follows:

$$\begin{aligned} K(T_1, T_2) &= \sum_{s=1}^{m} h_s(T_1) h_s(T_2) = && (3.4) \\ &= \sum_{s=1}^{m} \sum_{t_1 \in N_{T_1}} I_s(t_1) \sum_{t_2 \in N_{T_2}} I_s(t_2) && (3.5) \\ &= \sum_{t_1 \in N_{T_1}} \sum_{t_2 \in N_{T_2}} \sum_{s=1}^{m} I_s(t_1) I_s(t_2) && (3.6) \\ &= \sum_{t_1 \in N_{T_1}} \sum_{t_2 \in N_{T_2}} C(t_1, t_2) \end{aligned}$$

where $C(t_1, t_2) = \sum_{s=1}^{m} h_s(t_1) h_s(t_2)$. Let a *production at* node t be the subset tree constituted by t and only its direct children, then $C(t_1, t_2)$ can be recursively computed according to the following rules:

1. if the productions at t_1 and t_2 are different then $C(t_1, t_2) = 0$;
2. if the productions at t_1 and t_2 are the same, and t_1 and t_2 have only leaf children (i.e. they are pre-terminals symbols) then
$$C(t_1, t_2) = 1 \tag{3.7}$$
3. if the productions at t_1 and t_2 are the same, and t_1 and t_2 are not pre-terminals, then
$$C(t_1, t_2) = \prod_{j=1}^{nc(t_1)} (1 + C(ch_j[t_1], ch_j[t_2])) \tag{3.8}$$
where $nc(t_1)$ is the number of children of t_1 and $ch_j[t]$ is the j-th child of node t.

It can be shown that by substituting eq. (3.8) with the following, we obtain the subtree kernel (ST) as defined in section 3.1.1:
$$C(t_1, t_2) = \prod_{j=1}^{nc(t_1)} (C(ch_j[t_1], ch_j[t_2])). \tag{3.9}$$
In the following we will generically refer to the kernels defined in this section as the parse tree kernels.

Note that C can be seen as the local kernel for the convolution kernel defined by eq. (3.1), where χ is the space of trees and χ_x the set of subtrees of tree x.

Computing the subtree and subset tree kernels ultimately consists in filling a matrix of size $|N_{T_1}| \times |N_{T_2}|$ with the appropriate C values and then summing all C up. From this last observation it may be concluded that the worst case time computational complexity of the kernel is $O(|N_{T_1}| \times |N_{T_2}|)$. However, as discussed by the same authors, a more accurate analysis shows that the actual complexity of the above procedure depends on the number of matching productions. Thus the computation of $C(t_1, t_2)$ can be avoided for all (t_1, t_2) whose production at root node are different. This observation has resulted in the Fast Tree Kernel algorithm [47], which has the same worst case complexity but in practical applications may provide a relevant speed up.

Subset tree kernel values depend on the number of nodes in the trees. Consider a small tree t. It should have highest kernel value when compared with itself (it should be most similar to itself). However a large tree with higher kernel value can be easily constructed by replicating subset trees of t into the larger tree. In order to overcome this situation, it is possible to normalize the kernel according to the formula 2.19.

The subset tree kernel is known to be peaked, i.e. there tend to be a significant difference between the kernel value of a structure with itself, than the kernel value with any other structure. This is due to the influence of larger substructures, since they not only contribute to the kernel with a high value of C but they also contribute to it with all of their substructures. In order to reduce this phenomenon, the influence of larger subtrees is thus downweighted. This is obtained by modifying the kernel as follows:

$$K(T_1, T_2) = \sum_{s=1}^{m} \lambda^{size(s)} h_s(T_1) h_s(T_2), \tag{3.10}$$

where $0 < \lambda \leq 1$ is a weighting parameter and $size(s)$ is the number of nodes of the subtree s. The addition of λ to eq. (3.10) can be accounted for in the recursive formula by the following modifications to eq. (3.8) and (3.9):

$$C(t_1, t_2) = \lambda, \tag{3.11}$$

$$C(t_1, t_2) = \prod_{j=1}^{nc(t_1)} (\lambda + C(ch_j[t_1], ch_j[t_2])). \tag{3.12}$$

Convolutions kernels are recognized to have some drawbacks:

- a worst case quadratic complexity makes their application to very large datasets infeasible (especially in conjunction with demanding learning algorithms such as the SVM), since the sparsity assumption is less easily satisfiable and the quadratic cost may become prohibitive.

- They can't be used with continuous labels: the probability that a substructure of a tree would match with any substructure of every other tree would be extremely low, and thus the resultant feature space would tend to be sparse. The same reasoning applies also in the case of labels taken from a discrete but numerous set.

Most of the works in literature on tree kernels are based on the article of Collins et al. presented in this section and, basically, describe extensions and adaptation of it. In fact some authors [34] use the term tree kernel to identify all those kernels based on eq. (3.4) and the corresponding recursive C formula. Mainly two fields of research can be distinguished. They are motivated either by:

1. reduce the kernel computational complexity;

2. adding expressiveness to the kernel.

In the following sections convolution kernels methods aimed at solving one of the problems listed above, are discussed. Section 3.1.3 introduces a fast tree kernel. Another kernel aimed at a low computational complexity is the one presented in section 3.2.1. Sections 3.1.4, 3.1.5, 3.1.6, and 3.1.7 present expressive kernel functions.

3.1.3 Approximate Kernels for Trees

In cases when computational complexity issues don't allow the application of kernels with quadratic time complexity, more efficient kernels, i.e. computable in linear time, with a sufficient expressive power have to be designed. In [55] an approximated tree kernel with worst case linear time complexity is described. The field of application is the detection of attacks by the analysis of net logs. Net logs are composed by very deep trees and thus the worst case quadratic complexity of the subset tree kernel makes its application to the task infeasible. The speed up is obtained by selecting a restricted set of sparse and discriminative features:

$$K(x_i, x_j) = \sum_{s} \gamma(s) \sum_{\substack{t_i \in x_i \\ l_i = l(root(s))}} \sum_{\substack{t_j \in x_j \\ l_j = l(root(s))}} C(t_i, t_j),$$

where t_i is a node of x_i and $\gamma(s) = \{0, 1\}$ is a function telling whether the current subtree has to be selected.

The goodness of a kernel can be measured by its alignment to the matrix yy^T (see section 2.3.4 for a definition of alignment), where y is a vector collecting the class of the examples:

$$\left\langle yy^T, K \right\rangle_{\mathcal{F}} = \sum_{y_i = y_j} K(x_i, x_j) - \sum_{y_i \neq y_j} K(x_i, x_j).$$

In order for the kernel to be fast, a small number of features should be selected. The final problem to be solved can be formulated as follows:

$$\begin{array}{ll} \max_{\gamma \in \{0,1\}^{|S|}} \displaystyle\sum_{i,j=1}^{n} y_i y_j \sum_{s \in S} \gamma(s) \sum_{\substack{t_i \in x_i \\ l_i = root(s)}} \sum_{\substack{t_j \in x_j \\ l_j = root(s)}} C(t_i, t_j), \\ \text{subject to} \quad \displaystyle\sum_{s \in S} \gamma(s) = N \end{array}$$

where $|S|$ is the size of the feature space, n the size of the training set and N is the number of different substructures we want to use for the subsequent kernel computations. Their experiments suggest that the method can achieve an accuracy comparable with the tree kernel while being more efficient.

Among the extensions proposed by the authors, it is worth pointing out the possibility to bound the expected runtime by a user-defined quantity b (this of course to the detriment of the accuracy). The bound can be obtained by adding a constraint of the following form to the problem:

$$\sum_{s \in S} \gamma(s) \sigma(s) \leq b,$$

where $\sigma(s)$ is the frequency of the substructure s in the whole set.

3.1.4 Partial Tree Kernel

One way for adding expressiveness to the parse kernel described in section 3.1.2 is to modify its recursive definition in order to enlarge the corresponding feature space.

Moschitti [46] proposed the Partial Tree kernel (PT) which can carry out partial matching between subtrees. Note that the definition of partial tree corresponds to the definition of subtree (see section 2.1). Figure 2.3 gives an example of a tree together with some of its partial trees. Note that the feature space of the partial tree kernel is much larger than the one of the subset tree kernel.

The partial tree kernel is obtained from eq. (3.4) by replacing cases 2 (eq. (3.7)) and 3 (eq. (3.8)) of C definition with:

$$C(n_1, n_2) = 1 + \sum_{J_1, J_2, |J_1|=|J_2|} \prod_{i=1}^{|J_1|} C(ch_{n_1}[J_{1i}], ch_{n_2}[J_{2i}]), \tag{3.13}$$

where $J_{11}, J_{12}, J_{13}, \ldots$ $J_{21}, J_{22}, J_{23}, \ldots$ are index sequences associated with the ordered child sequences ch_{n_1} and ch_{n_2} respectively, J_{1i} and J_{2i} point to the i-th children in the two sequences, $|J_1|$ returns the length of the sequence J_1. The subset tree kernel can be obtained back from eq. (3.13) by considering only the contribution of the longest child sequence from node pairs with same children. The partial tree kernel can be evaluated in $O(\rho^3|N_{T_1}||N_{T_2}|)$, where ρ is the maximum out-degree of the two trees.

3.1.5 Elastic Tree Kernel

In [35] is described a kernel for structured data modelled by labelled ordered trees, called elastic tree kernel. The kernel extends the one in [11] in allowing matching between nodes with different labels and matching between substructures built by combining subtrees with their descendants. Consider two trees T_1 and T_2 and two subtrees, t_1 and t_2, rooted at nodes $v_1 \in T_1$ and $v_2 \in T_2$, respectively. In addition to the matchings of the subset tree kernel, the following are also permitted:

- t_1 and t_2 may match even if they don't have the same number of children. The only constraint is to preserve the order of the children, i.e. matching child 2 of t_1 with child 4 of t_2 *and* child 3 of t_1 with child 1 of t_2 is not allowed. Then the C formula of eq. (3.8) becomes

$$C(v_1, v_2) = S_{v_1,v_2}(nc(v_1), nc(v_2)), \tag{3.14}$$

where $nc(v)$ is the number of children of v. Since all matches preserve left-to-right ordering S can be defined as:

$$\begin{aligned} S_{v_1,v_2}(i,j) = \; & S_{v_1,v_2}(i-1,j) + S_{v_1,v_2}(i,j-1) - S_{v_1,v_2}(i-1,j-1) + \\ & + S_{v_1,v_2}(i-1,j-1)C(ch_i[v_1], ch_j[v_2]), \end{aligned} \tag{3.15}$$

and $S_{v_1,v_2}(i,0) = S_{v_1,v_2}(0,j) = 1$. Basically the idea is that the number of matchings considering l children can be expressed in terms of the number of matchings considering $l-1$ children. Note the formulation in eq. (3.15) can be efficiently computed by means of dynamic programming.

- t_1 and t_2 may match even if their labels are not identical. Different labels are penalized by assigning to them a lower matching value. This is obtained by multiplying each match by a value determined by a function $P_{mut}(l_1|l_2) \in [0,1]$, where $P_{mut}(l_1|l_2)$ is the cost for transforming label l_1 into l_2. For identical labels $P_{mut}(l_1, l_1) = 1$ and thus they contribute to the kernel value in the same way they contribute for the subset tree kernel. The C function then becomes

$$C(v_1, v_2) = \sum_{a \in \mathcal{A}} P_{mut}(l_1|a) P_{mut}(l_2|a) S_{v_1,v_2}(nc(v_1), nc(v_2)), \tag{3.16}$$

where $\mathcal{A}$ is the space of labels. Thus eq. (3.16) takes into account all possible mutations of the labels of the nodes being computing the C value.

- t_1 and t_2 can be elastic trees. An elastic is a subset of nodes for which the relative positions in the original tree are preserved: if a node v is a descendant of a node m in T_1, then v must be a descendant of m in the elastic subtree t_1; if v is, for example, to the left of m, then it must be to the left of m in t_1. Figure 3.1 shows an example of a subtree (on the left) and one of its elastic matching with the tree on the right. Since all descendants of a node can be part of an elastic subtree, all of them have to be considered. This leads to the substitution of the C term in eq. (3.15) with:

$$C_a(v_1, v_2) = \sum_{v_a \in D(v_1)} \sum_{v_b \in D(v_2)} C(v_a, v_b), \tag{3.17}$$

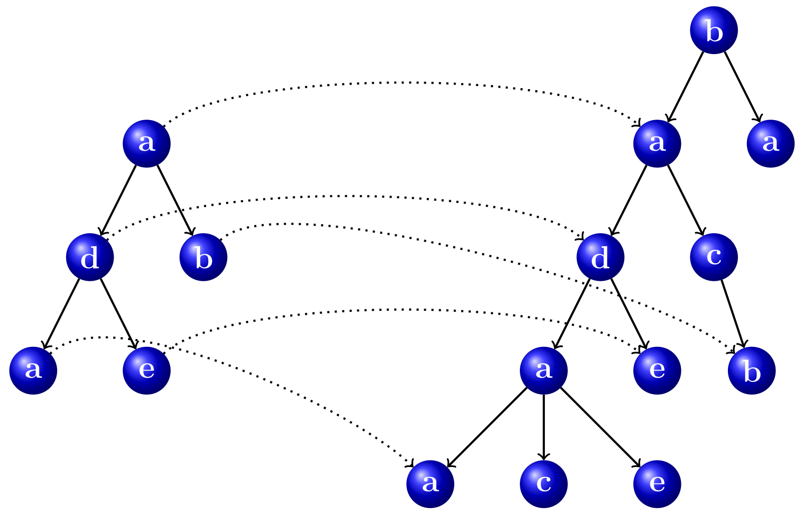

Figure 3.1: A subtree (left) and one of its elastic matchings with the tree on the right.

where $D(v)$ is the set composed by v and all of its descendants of v and the C function is the one defined in eq. (3.14). C_a can be computed efficiently by the following formula:

$$\begin{aligned} C_a(v_1, v_2) = & \sum_{j=1}^{nc(v_2)} C_a(v_1, ch_j[v_2]) + \sum_{j=1}^{nc(v_1)} C_a(v_2, ch_j[v_1]) - \\ & - \sum_{j=1}^{nc(v_2)} \sum_{i=1}^{nc(v_1)} C_a(ch_i[v_1], ch_j[v_2]) + C(v_1, v_2), \end{aligned} \tag{3.18}$$

Despite the fact that the feature space associated with the elastic tree kernel is much larger of the one associated with the parse tree kernel, the authors show that the computational complexities of the two algorithms are the same.

In [34] it is demonstrated that the elastic tree kernel cannot be generalized for the application to labelled unordered trees. First the authors show that the problem of computing the elastic tree kernel for unordered trees has the same difficulty of the problem of counting the number of times one tree T appears into another tree T', formally formulated as: $K^{|T|}(T, T')$. Then they show that $K^{|T|}(T, T')$ can be reduced to a problem known to be NP-HARD: #Perfect Matchings(G). #Perfect Matchings(G) of the bipartite graph $G = X \cup Y$, such that $|X| = |Y|$, counts the number of perfect matchings in G. A matching is a set edges (x, y) such that $x \in X$ and $y \in Y$ and every x and y appear only once in all matches. The matching is perfect if all nodes of the graph are considered.

Any algorithm for computing the elastic tree kernel for unordered trees is NP-HARD since it can be ultimately reduced to #Perfect Matchings(G).

A generalization of the elastic tree kernel is presented in [39]. It is based on the idea that tree edit distance can be defined as an optimization problem of a common representation, called tree mapping. The authors propose four kernel functions and demonstrate that three of them are positive semidefinite. Among these, they prove that the elastic tree kernel is positive semidefinite (the proof was not present in [35]).

3.1.6 Grammar-Driven Tree Kernel

A different technique for adding expressiveness is to allow a limited level of soft matching between node labels.

In [69] a grammar-driven convolution tree kernel which introduces more linguistic knowledge into the subset tree kernel. The kernel enlarges the feature space of [11] by allowing approximate substructure and tree node matchings according to a given grammar. Approximate substructure matching means that children of a node can be ignored provided that the following constraints are satisfied:

- the remaining nodes form a valid grammatical rule.
- there must be at least two children in the reduced rule and the first child must be kept in the reduced rule. These constraints try to retain the underlying semantic meaning of their corresponding original rules.

For example the two parse trees corresponding to the sentences "*buy a red car*" and "*buy a car*" wouldn't match for the parse tree kernel, while the grammar-driven tree kernel allows them to match by reducing the former sentence to
"*buy a red car*" $\rightarrow$ "*buy a car*".

The second modification allows a restricted level of soft matching between labels. Two different node labels may match if they belong to the same equivalence node set. The equivalent node sets are defined a priori. A matching between two different labels is penalized by a factor λ_{l_1,l_2} dependant on the two labels.

The authors do not provide an efficient implementation of the kernel, but claim that in their experiments the kernel had a complexity of $O(|T_1||T_2|)$.

The grammar-driven tree kernel is a specification of the partial tree kernel [46]. Indeed the partial tree kernel generates a much larger feature space but it also matches non linguistically motivated structures. This may potentially compromise the performance since some of the over-generated features may possibly be noisy due to the lack of linguistic interpretation and constraint.

3.1.7 Semantic Syntactic Tree Kernels

Moschitti et al. [9] introduce a family of kernels, specifically designed for being used in text categorization tasks, called Semantic Syntactic Tree Kernels. The kernels introduce an embedded semantic term kernel and a leaf weighting component. They allow partial matches between tree fragments, where a partial match between two subtrees occurs when they differ only by their terminal symbols. The partial match between terminal nodes is performed according to a predefined kernel k_S. The tree fragment kernel is defined as:

$$K(f_1, f_2) = comp(f_1, f_2) \prod_{i=1}^{nt(f_1)} k_S(f_1(i), f_2(i)), \tag{3.19}$$

where the function $comp(f_1, f_2)$ equals 1 whether f_1 and f_2 differs only in the terminal nodes, 0 otherwise, $nt(f_1)$ is the number of terminal nodes of the two tree fragments. The semantic syntactic tree kernel is obtained by modifying eq. (3.7) as follows:

$$C(v_1, v_2) = \lambda k_S(v_1, v_2). \tag{3.20}$$

The authors discuss two types of k_S kernels. The first type is based on a taxonomy for computing the term similarity. The second type is based on latent semantic indexing and computes the similarity by means of co-occurrence analysis of terms in documents and vice versa.

3.2 Other Approaches for the Design of Kernels for Tree Structured Data

This section describes the approaches taken for the design of kernel functions for trees which are not derived from the convolution kernel framework.

3.2.1 Spectrum Tree Kernel

In [38] a kernel based on counting common tree q-grams is described. Tree q-grams are subtrees isomorphic to paths with q nodes. The kernel is an adaptation of a string kernel [41] to tree structured data. Figure 3.2 gives an example of some q-grams. The q-grams are represented by pairs $(P_i, l_1, \ldots, l_q)$, where P_i represent the type of subtree (by fixing the parameter q there are $q-1$ different type of isomorphic patterns not counting the labels).

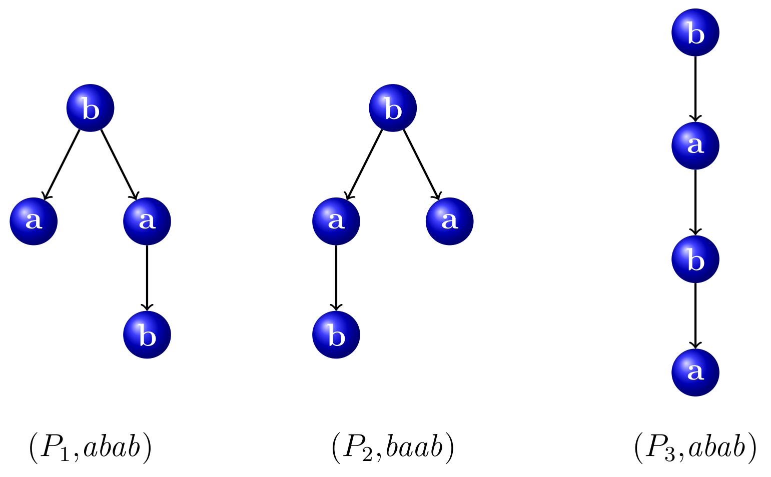

Figure 3.2: Some examples of q-grams, with $q = 4$. P_i identifies the structure of the path, the string the sequence of labels as encountered by visiting the subtree.

A subtree P_i matches a tree T at a node n if there exists a one-to-one mapping f from the nodes of P into the nodes of T satisfying the following constraints:

- f maps the root of P to n

- The ordering of the children is preserved by the mapping. Suppose f maps x to y and x has children $x_1, \dots, x_k$, y has children $y_1, \dots, y_m$, $m \geq k$, then the ordering of the children is preserved if there exists a monotone function g such that $f(x_i) = y_{g(i)}$ and $i_1 < i_2 \Rightarrow g(i_1) < g(i_2)$.

- $\forall x \in P, l(x) = l(f(x))$.

In order to enumerate efficiently all q-grams, the LabelGram algorithm [52] has been used. It runs in $O(qg^2|T|)$, where g is the maximum out-degree of the tree T and $|T|$ is the number of nodes of T. Being $G_q(T)$ the vector collecting information about all q-grams in T, the kernel is defined as follows:

$$K(T_1, T_2) = \langle G_q(T_1), G_q(T_2) \rangle . \tag{3.21}$$

The computational time of computing eq. (3.21) depends on the number of common q-grams. In their experiments the authors show that the computational time is “almost linear”.

3.2.2 Tree Fisher Kernel

The Fisher Kernel has been introduced in [31], it is derived from a generative model using the gradient of the log likelihood with respect to the parameters of the generative model as the features. This procedure defines a metric directly from the generative model, capturing the differences in the generative process of a pairs of objects. The Fisher kernel assumes that the data is generated from a parametric probability distribution: $P(x|\vec{\theta})$, where $\vec{\theta} = (\theta_1, \dots, \theta_n)$ is a set of parameters of the model. The idea is to form a representation of the data in terms of those parameters which are sufficient statistics of x. This is achieved by means of the Fisher Score U_x:

$$U_x = \nabla_\theta \log P(x|\theta). \tag{3.22}$$

The Fisher kernel is defined as:

$$K(x, z) = U_x^T I^{-1} U_x, \tag{3.23}$$

where I^{-1} is the Fisher Information matrix:

$$I^{-1} = E_{P(x|\theta)} U_x U_x^T. \tag{3.24}$$

The Fisher Information is the expected value of the inner product of the representation U_x over $P(x|\theta)$. In practice, often the information matrix is set to the identity matrix. It is worth noting that when deriving a kernel from a generative model, the value of kernel between two objects depends also on the other objects used for constructing the generative model, i.e the training set. In this sense the Fisher kernel adapts to the data instead of being a priori defined.

Nicotra et al. [50] describes an application of the Fisher kernel to structured data. Hidden Tree Markov Models [16] are used as the generative models for trees. Their algorithm is applied to rooted positional k-ary trees with a label associated to each node.

Part II

Original Contributions

Chapter 4

A Tree Kernel For Non Discrete Domains

Kernel methods offer a novel approach to the treatment of structured data which has proven to be effective in many applications. In particular, convolution tree kernels have received much attention from the research community. Despite their effectiveness, most of them may produce sparse feature spaces when applied to datasets with node labels belonging to a large domain. In order to identify this situation we have defined the sparsity index in eq. (2.23). When a kernel is sparse with respect to a dataset, all kernel values $K(x_i, x_j)$ for different objects are smaller than the kernel value of the object with itself, $K(x_i, x_i)$. In this situation, the convolution kernel approach can never be trained efficiently, and it will behave like a nearest neighbour rule [61]. The learner thus will be accurate on the training data, but unable to generalize well on unseen data. This intuition is formalized in [30] for the case of normalized kernels and the ν-SVM [58], which is a variant of the SVM (see section 2.3.2). The authors prove that if the kernel function takes a value smaller than a certain δ for any pair of different examples then each example is a support vector for the ν-SVM. The value δ depends on the size of the training set and on the c parameter of the ν-SVM.

Reducing the sparsity of the Subtree and Subset tree kernels by allowing soft matching between node labels is not feasible. By modifying the recursive formulation of the two kernels, i.e. eq. (3.7) and eq. (3.8), for allowing soft matching would result in a significant increase of the computational burden. In fact any subtree of the first tree would match to any subtree of the second tree.

We investigate on whether the ability of a Self-Organizing Maps for Structured Data to produce a compressed representation of structural information may be used to generate kernels less sparse than the current tree kernels which, at the same time, preserve the structural properties of the inputs. Specifically, we propose a family of kernels, called Activation Mask Kernels, defined on top of SOM-SD exploiting both its compression and "topology" preserving capabilities. While for concreteness we developed our idea on the SOM-SD algorithm, any topology preserving low dimensional representation of the data, which produces a map as output, could have been used: for example the CSOM-SD or the graph SOM.

The experimental results obtained on a classification task involving a relatively large corpus of XML formatted data, provide evidence that, when sparsity on the data is present, the proposed kernels are able to improve the overall categorization performance over each individual method, i.e. either SVM using tree kernels or SOM-SDs equipped with a 1-NN classification rule. This demonstrates that, neither tree kernels nor SOM-SDs are always able to retain all the relevant information for classification. The approach proposed in this chapter can thus be considered as a first step in the direction of defining approaches able to fully exploit the structural information needed to solve learning tasks defined on structured domains.

The chapter is organized as follows: the Activation Mask Kernel is proposed in Section 4.1, the relationship with similar techniques are discussed in section 4.2, experimental findings are discussed in Section 4.3.

4.1 Activation Mask Kernel

In this section, we show how novel tree kernels can be defined on the basis of a SOM-SD. The basic idea is to represent each vertex of a tree upon its activation on a SOM-SD map and then define a kernel on this space. Specifically, with no loss of generality, we assume to have set an enumeration of the neurons based on their position in the map, e.g. the one obtained by a bottom-up left-to-right visit of the map. According to this enumeration each neuron is associated with a unique index $m \in 1, \ldots, a \times b$, where a and b are the horizontal and vertical dimensions of the map.

Let $ne_\epsilon[m]$ denote the set of indices of neurons (from a SOM-SD) in the ϵ-neighbourhood of the neuron with index m, i.e. $\{m' | \Delta_{m'm} \leq \epsilon\}$, where Δ is the topological distance defined on the 2-dimensional map. An interesting measure of similarity between two subtrees which takes into account the topology induced by the SOM-SD can be defined as the cardinality of the intersection of the ϵ-neighbours of the neurons mostly activated by these subtrees. We define the set of neurons shared by the two ϵ-neighbours related to structures t_1 and t_2 as

$$I_\epsilon(t_1, t_2) = ne_\epsilon[y_{t_1}] \ \cap \ ne_\epsilon[y_{t_2}], \tag{4.1}$$

where, we recall, y_{t_i} is the index of the winning neuron for the root node of the subtree t_i. A similarity measure between two trees T_1 and T_2 can be defined by the function:

$$K_\epsilon^{(I)}(T_1, T_2) = \sum_{t_1 \in N_{T_1}} \sum_{t_2 \in N_{T_2}} |I_\epsilon(t_1, t_2)|. \tag{4.2}$$

Alternative functions which emphasize the alignment between the activation profiles of two subtrees can be considered instead of the strict intersection. For example, it is possible to weight differently matching regions depending on the distance from the activated neurons:

$$K_\epsilon(T_1, T_2) = \sum_{\substack{t_1 \in T_1, \\ t_2 \in T_2, \\ m \in I_\epsilon(t_1, t_2)}} Q_\epsilon(m, y_{t_1}) Q_\epsilon(m, y_{t_2}),$$

where $Q_\epsilon(m, m')$ is inversely proportional to the distance $\Delta_{mm'}$ between map neurons m and m' and $Q_\epsilon(m, m') = 0$ when the neurons are not in the ϵ-neighborhood of each other, i.e. $\Delta_{mm'} > \epsilon$. As an example, $Q_\epsilon(m, m')$ can be defined as

$$Q_\epsilon(m, m') = \begin{cases} \epsilon - \eta \Delta_{mm'} & \text{if } \Delta_{mm'} \leq \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

where $0 \leq \eta \leq 1$ is a parameter determining how much the distance influences the neighbourhood activation.

Since this kernel is built on activation masks of a SOM-SD, we shall refer to this approach as the *activation mask kernel* (AM-kernel).

Figure 4.1 gives an example of construction of the feature space representation of three trees according to the AM-kernel. On the left part of the image three simple trees selected from the INEX 2005 dataset (see section 4.3) and on the right part their activation masks referring to a 5×4 map. The height of each element of the map corresponds to the value of the activation. Note that the tree on top and the tree on the centre are more similar to each other than to the tree on the bottom and this is reflected in the activation masks.

The similarity function $K_\epsilon(T_1, T_2)$ is a kernel for any choice of $Q_\epsilon(\boldsymbol{m}, \boldsymbol{m}')$. A way to demonstrate this is to show that there exists a function ϕ such that for every T_1, T_2, we have $\phi(T_1) \cdot \phi(T_2) = K_\epsilon(T_1, T_2)$, i.e. K_ϵ can be expressed as a dot product in the feature space induced by ϕ. Specifically, let us define a feature space which has the same dimension as the map produced by the SOM-SD, i.e. Let $a \times b$, thus obtaining $\phi(T) \in \mathbb{R}^{a \times b}$. Now, given a tree T, we define the mask $M \in \mathbb{R}^{a \times b}$ where every element of M is associated to a neuron of the map. Let M be initially set to the null vector. The feature vector is then constructed by computing the best-matching neuron y_t for each subtree $t \in T$ when presented to the SOM-SD. Then, the entries of M associated to neighbours within radius ϵ of y_t are updated according to $M_m = M_m + Q_\epsilon(m, y_t)$; finally, the feature vector $\phi(T)$ will be defined as $\phi(T) = [M_1, \ldots, M_{a \times b}]$. At this point it is easy to check that for a given tree T, $M_m(T) = \sum_{t \in T} Q_\epsilon(m, y_t)$ where t runs over all possible subtrees of T, and we can check that the kernel is obtained by performing the dot product in feature space,

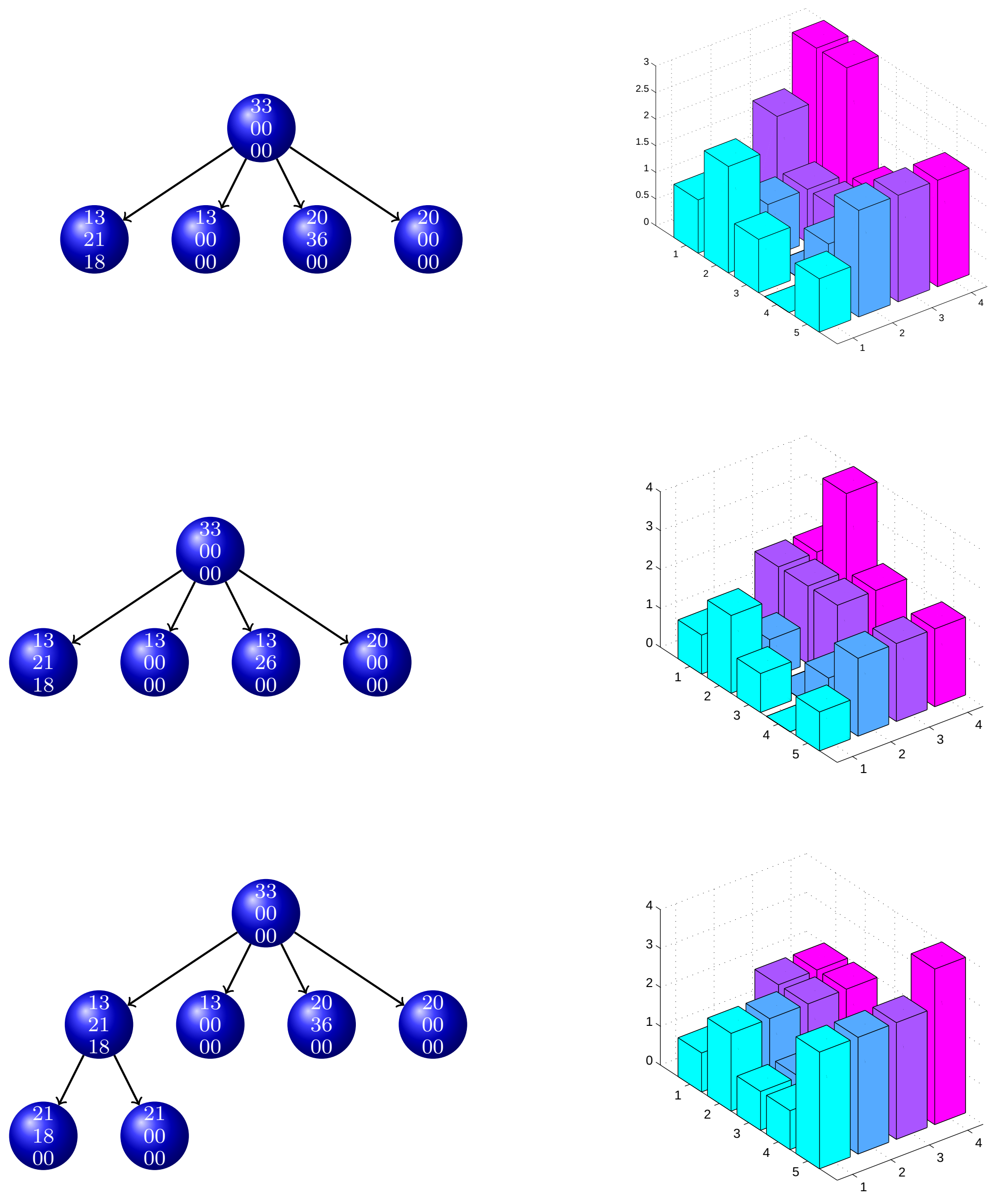

Figure 4.1: Example of representation in feature space of three trees according to the Activation Mask Kernel for $\epsilon = 1$. On the left part of the image three simple trees and on the right part their activation masks referring to a 5×4 map. The height of each element of the map corresponds to the value of the activation.

i.e.

$$\begin{aligned}
M(T_1)\cdot M(T_2) &= \sum_m M_m(T_1)M_m(T_2)\\
&= \sum_m \sum_{t_1\in N_{T_1}} Q_\epsilon(m,y_{t_1}) \sum_{t_2\in N_{T_2}} Q_\epsilon(m,y_{t_2})\\
&= \sum_{t_1\in N_{T_1},t_2\in N_{T_2}} \sum_m \big(Q_\epsilon(m,y_{t_1})\cdot Q_\epsilon(m,yt_2)\big)\\
&= \sum_{t_1,t_2} \sum_{m\in I_\epsilon(t_1,t_2)} \big(Q_\epsilon(m,y_{t_1})\cdot Q_\epsilon(m,y_{t_2})\big)\\
&= K_\epsilon(T_1,T_2),
\end{aligned}$$

where the third derivation is justified by the fact that $Q_\epsilon(m,y_t)=0$ whenever m is not in the ϵ-neighbourhood of y_t.

The computational complexity of a kernel evaluation of the proposed approach is governed by eq. (4.1) and eq. (4.2). Specifically it is dominated by the selection of the winning neurons y, which has to be performed for each vertex of each tree involved. Thus the whole process has complexity $O(a\cdot b\cdot(|T_1|+|T_2|))$. Note that the proposed approach requires the initial training of a SOM-SD, which however is performed only once, thus not affecting the overall computational complexity of kernel evaluations.

4.2 Related Work

The novelty of the proposed approach consists in creating a novel set of features from the current dataset. It differentiates from feature selection approaches [23] and feature weighting approaches [68] in the adaptive nature of the feature creation process.

Moschitti et al. [9] describe a kernel which allows a limited degree of soft matching. However in their approach only leaf node labels can match while not being identical. Our approach allows soft matchings between entire structures.

The elastic tree kernel [35] also allows matching between node with different labels and subtrees with "partially" different structure. Although the definition of the elastic tree kernel allows soft matching between labels, in the experiments presented only exact matchings were considered. The similarity function between labels in [35] is defined as the sum of the similarities of each of the two labels with respect to each possible label of the domain. Clearly the application of such approach is severely limited by the size of the label domain. The Activation Mask Kernel creates a fixed set of features and then performs exact matching on those features. By fixing the feature space we generally restrict its size with respect to complex kernels such as the elastic tree kernel, and thus potentially avoid overfitting. The drawback of our approach consists in the fact that the novel features must keep enough information for the learning task. Section 4.3 presents some empirical results and discusses the settings in which the use of SOM-SD enhances the learning accuracy.

4.3 Experiments and Discussion

Experiments have been performed to evaluate the performances of the new AM-kernel. We used the INEX 2005 dataset, a relatively large set of XML formatted documents which were made available as part of the 2005 INEX Competition [15]. The dataset is described in detail in section A.1.

As a baseline we considered the SVM with ST and SST kernels applied to the dataset. The obtained results, together with the values of the sparsity index (eq. (2.23)) and alignment (eq. (2.24)) for each kernel, all computed on the test set, are shown in Table A.2. The best accuracy on test set has been obtained by the SST kernel with an error rate of 11.21%.

The experiments started by training a number of maps with the SOM-SD software[1]. Due to SOM-SD training times (e.g., about 12 hours for a single large map (110×80) on a AMD Athlon(tm) 64 X2 Dual Core Processor 3800+), and the number of parameters involved, a comprehensive sampling of the parameter space was not feasible. Thus, we decided to run preliminary experiments involving the validation set to sort out the most relevant parameters with respect to the definition of the

[1] *http://www.uow.edu.au/∼markus/apods/software.html*

proposed kernels. The selected parameters were the map dimension, the number of training iterations, and the value of μ_1 (we set μ_2 as $1 - \mu_1$). For these parameters the following values were used:

- map dimension: $110 \times 80, 77 \times 56, 55 \times 40$;
- number of training iterations: $32, 64, 128$;
- μ_1: $0.05, 0.25, 0.45, 0.65, 0.85$.

For what concerns the other SOM-SD (hyper) parameters, the following values were chosen: $\alpha = 1$, neighborhood radius=18, type of α decrease=sigmoidal, map topology=hexagonal. By combining the above parameters, 45 different maps were built with the aim of spanning as much as possible the space of SOM-SD parameters and therefore getting insights on the dependency of the final results from the maps. Since the SOM-SD is an unsupervised technique, the union of the training and validation sets has been used for creating the maps.

After the training phase each map was evaluated on the test set using a k-nn procedure with $k = 1$. Table 4.1 reports the classification performance of each map.

map	map size	training iterations	μ	test error (%)
1	110 × 80	128	0.05	12.264
2	110 × 80	128	0.25	14.259
3	110 × 80	128	0.45	11.370
4	110 × 80	128	0.65	10.455
5	110 × 80	128	0.85	**8.647**
6	110 × 80	32	0.05	12.617
7	110 × 80	32	0.25	16.587
8	110 × 80	32	0.45	10.912
9	110 × 80	32	0.65	15.423
10	110 × 80	32	0.85	11.661
11	110 × 80	64	0.05	13.282
12	110 × 80	64	0.25	11.723
13	110 × 80	64	0.45	14.238
14	110 × 80	64	0.65	11.245
15	110 × 80	64	0.85	8.855
16	55 × 40	128	0.05	21.638
17	55 × 40	128	0.25	29.079
18	55 × 40	128	0.45	28.081
19	55 × 40	128	0.65	21.326
20	55 × 40	128	0.85	22.511
21	55 × 40	32	0.05	35.169
22	55 × 40	32	0.25	32.488
23	55 × 40	32	0.45	27.770
24	55 × 40	32	0.65	22.137
25	55 × 40	32	0.85	25.629
26	55 × 40	64	0.05	31.844
27	55 × 40	64	0.25	27.541
28	55 × 40	64	0.45	27.749

29	55 × 40	64	0.65	20.121
30	55 × 40	64	0.85	19.144
31	77 × 56	128	0.05	21.451
32	77 × 56	128	0.25	24.215
33	77 × 56	128	0.45	23.488
34	77 × 56	128	0.65	16.296
35	77 × 56	128	0.85	9.956
36	77 × 56	32	0.05	16.234
37	77 × 56	32	0.25	22.282
38	77 × 56	32	0.45	19.310
39	77 × 56	32	0.65	18.624
40	77 × 56	32	0.85	17.585
41	77 × 56	64	0.05	17.169
42	77 × 56	64	0.25	22.864
43	77 × 56	64	0.45	21.721
44	77 × 56	64	0.65	9.457
45	77 × 56	64	0.85	15.735

Table 4.1: Classification error of the SOM-SD maps on the INEX 2005 dataset. Lowest error is in bold.

The resulting classification error ranges from significantly above the baseline (35.17%) to a very much lower values of the classification error (8.65%). The mean classification error is 18.94% with a standard deviation of 6.98. This means that the results are indeed very sensitive to the parameter choice. In the following we analyse the dependence of the classification error from each parameter:

map size The mean classification error of 110 × 80 maps is 12.24% with standard deviation 2.23, the mean error of 77 × 56 maps is 18.43% with standard deviation 4.52 and the mean classification error of 55 × 40 maps is 26.15% with standard deviation 4.87. For this problem, clearly bigger maps are to be preferred. This is due either to the fact that smaller maps do not have enough neurons for representing effectively all statistically interesting type of structures for the task or that different structures are positioned too close, i.e. the topology of the input space can not be preserved.

Training iterations Table 4.2 reports the mean classification error of the maps with respect to the number of training iterations. Statistics are divided according to map size since the values significantly depend on that. Both 55×40 and 110×80 maps decrease mean error by increasing the number of training iterations, so it seems a viable suggestion to use as many training iterations as possible.

	Mean Classification Error (%) w.r.t to Map Size			
	110×80	77×56	55×40	All maps
iter=32	13.44 (2.45)	18.81 (2.26)	28.64 (5.23)	20.30 (7.71)
iter=64	11.87 (2.07)	17.39 (5.35)	25.28 (5.44)	18.18 (6.74)
iter=128	11.40 (2.09)	19.08 (5.97)	24.53 (3.74)	18.34 (6.60)

Table 4.2: Mean classification error of the SOM-SD maps with respect to number of training iterations (between brackets the standard deviation). Statistics are divided according to map size. The last column reports the mean classification error of all the maps.

Parameter μ_1 The parameter μ_1 determines the influence of the label component when computing the similarity between a neuron and a structure encoded as a neuron. μ_2 determines the influence of the children component. We recall that in all experiments μ_2 has been set to $1 - \mu_1$. μ parameters are clearly task dependant. In our case, table 4.3 shows that a high value of μ_1 leads to lowest classification error on average.

Experiments proceeded by testing the activation mask kernels (AM) defined in Section 4.1. For each map and for different values of ϵ (see eq. (4.1)) a kernel was defined. For each kernel, the c parameter of the SVM was selected on the

	Mean Classification Error (%) w.r.t to Map Size			
	110×80	77×56	55×40	All maps
μ_1=0.05	12.72 (0.52)	18.28 (2.78)	29.55 (7.05)	20.19 (3.32)
μ_1=0.25	14.19 (2.43)	23.12 (0.99)	29.70 (2.53)	22.34 (0.86)
μ_1=0.45	12.17 (1.80)	21.51 (2.10)	27.87 (0.19)	20.52 (1.03)
μ_1=0.65	12.37 (7.21)	14.79 (4.76)	21.19 (1.01)	16.12 (3.12)
μ_1=0.85	9.72 (1.68)	14.43 (3.98)	22.43 (3.24)	15.52 (1.17)

Table 4.3: Mean classification error of the SOM-SD maps with respect to the parameter μ_1 (between brackets the standard deviation). Statistics are divided according to map size. The last column reports the mean classification error of all the maps.

validation set from the following values: $0.001, 0.01, 0.1, 1, 10, 100, 1000$. Finally, with the selected value, an SVM was trained on the union of the training and validation sets and then evaluated on the test set. In all the experiments described in the following a time out of 24 hours on each executable run was set. This was done in order to obtain the results in a fair amount of time. Indeed, with this limitation, the overall learning phase with the SVM lasted for more than 3 months.

Map	Activation Mask kernel Classification error						Improvement (%) w.r.t	
	$\epsilon = 0$	$\epsilon = 1$	$\epsilon = 2$	$\epsilon = 3$	$\epsilon = 4$	$\epsilon = 5$	SST	SOM-SD
1	7.046	**6.215**	6.153	6.340	6.672	6.693	44.6	49.3
2	7.358	6.444	**6.381**	6.589	6.485	6.755	43.1	55.2
3	6.610	5.383	**5.238**	5.654	6.132	6.194	53.3	53.9
4	6.319	5.841	5.716	**6.256**	6.568	6.568	44.2	40.2
5	6.464	5.965	6.277	**6.693**	7.254	6.340	40.3	22.6
6	6.028	5.280	<u>**5.196**</u>	5.695	6.236	6.630	53.6	58.8
7	6.610	5.945	5.259	5.425	5.487	**5.508**	50.9	66.8
8	6.506	5.737	5.737	5.903	**5.737**	5.986	48.8	47.4
9	5.965	5.737	**5.924**	6.153	6.194	6.049	47.2	61.6
10	6.028	5.695	**5.945**	6.173	6.194	6.402	47.0	49.0
11	6.485	6.090	**6.028**	6.402	6.485	6.340	46.2	54.6
12	6.818	5.633	**5.342**	5.674	5.425	5.882	52.3	54.4
13	6.444	5.321	**5.404**	5.300	5.550	5.591	51.8	62.0
14	6.111	5.217	5.280	5.425	**5.758**	5.924	48.6	48.8
15	6.672	5.529	**5.882**	6.049	6.153	6.589	47.5	33.6
16	7.462	**6.984**	7.524	7.691	7.545	7.566	37.7	67.7
17	7.732	7.358	**6.838**	7.088	7.005	7.233	39.0	76.5
18	7.878	7.982	**8.273**	8.252	7.753	8.252	26.2	70.5
19	7.026	7.192	6.901	**7.524**	7.275	7.005	32.9	64.7
20	7.358	7.067	7.566	**7.649**	7.462	7.732	31.8	66.0
21	8.501	8.127	8.293	**8.293**	8.584	9.146	26.0	76.4
22	8.834	**9.083**	8.626	8.938	9.104	8.938	19.0	72.0
23	8.397	8.397	8.127	8.293	8.065	**8.190**	26.9	70.5
24	**8.605**	8.792	8.418	8.709	8.584	8.481	23.2	61.1
25	8.273	**8.481**	8.481	8.917	8.709	8.481	24.3	66.9
26	**8.896**	8.543	8.626	8.377	8.356	8.501	20.6	72.1
27	7.628	**7.296**	6.859	7.233	7.379	7.400	34.9	73.5

28	6.880	7.129	6.963	7.337	7.483	**7.441**	33.6	73.2
29	7.649	**7.129**	7.919	7.732	7.608	7.940	36.4	64.6
30	**8.460**	7.899	8.148	8.148	7.857	8.190	24.5	55.8
31	7.753	**7.483**	7.774	8.044	8.896	7.899	33.2	65.1
32	7.171	7.213	**7.483**	8.106	8.086	7.566	33.2	69.1
33	7.067	6.693	**6.547**	6.610	6.527	6.859	41.6	72.1
34	6.381	6.444	6.028	**6.527**	6.901	6.942	41.8	59.9
35	6.444	5.571	**5.716**	6.194	6.360	7.067	49.0	42.6
36	6.319	**5.716**	5.737	6.901	6.028	6.069	49.0	64.8
37	7.587	7.213	**6.818**	6.901	7.192	6.818	39.2	69.4
38	6.256	**6.256**	6.007	6.236	6.527	6.880	44.2	67.6
39	7.400	**6.631**	6.776	7.254	7.795	7.524	40.9	64.4
40	6.735	**6.360**	6.090	6.776	6.693	6.797	43.3	63.8
41	**6.340**	6.132	6.527	6.568	7.504	7.192	43.4	63.1
42	7.026	6.402	6.714	**7.441**	6.776	6.880	33.6	67.5
43	7.026	6.901	7.026	**7.171**	7.192	7.628	36.0	67.0
44	7.129	6.319	**6.194**	6.402	6.818	6.672	44.7	34.5
45	6.277	**6.111**	6.485	6.506	6.444	7.171	45.5	61.2
Mean	7.110	6.687	6.694	6.968	7.041	7.109	39.5	60.6
Std	0.814	1.030	1.044	0.996	0.956	0.893	0.012	0.094

Table 4.4: Classification error of the AM kernel on the INEX 2005 dataset. The ϵ that would be selected in validation for each map is in bold. The map that would be selected in validation is underlined. The improvement, for example with respect to SST, is computed according to the following formula: $100 \cdot \frac{err_{SST} - err_{AM}}{err_{SST}}$, where err is the classification error of an algorithm.

The classification error of each activation mask kernel is reported in Table 4.4. In the last two columns of the table, we have reported the error improvement (in %)

obtained by the best performing kernel on the validation set (in bold) when varying the ϵ value with respect to SOM-SD and SST performance, respectively. The improvement, for example with respect to SST, is computed according to the following formula: $100 \cdot \frac{err_{SST} - err_{AM}}{err_{SST}}$, where err is the classification error of an algorithm. The classification error on the test set for each value of ϵ ranges from 9.15% to 5.20% with a mean value of 6.93% and standard deviation 0.97. By selecting the ϵ value on the validation set the classification error ranges from 9.08% to 5.20% mean classification error 6.79% and standard deviation 1.06.

According to these experiments, the method used for selecting the parameters is reliable. In fact, if for each map we select the best performance obtained on the test set and we subtract this value from the performance obtained by the value of ϵ selected by the validation set, the mean value obtained over the set of maps is 0.25 (with standard deviation 0.256). Moreover, selecting both the map and the ϵ in validation, would have led us to obtain the lowest classification error, i.e. 5.20%.

In these experiments, the use of the Activation Mask kernel always improved the classification performance. The mean improvement of the AM-kernel with respect to the SOM-SD is 60.5% with standard deviation 0.12. In some cases the error is reduced up to the 76.5% with respect to SOM-SD and up to the 53.6% with respect to the SVM with SST kernel. The mean improvement with respect to the SST is 39.4% with standard deviation 0.09. The cumulative low standard deviation suggests that the improvement is quite independent with respect to the chosen map. In order to further discuss the dependence of the AM-kernel accuracy from the related map, a graphical comparison among the classification error on the test set of the methodologies involved in the experiment has been made in Figure 4.2. The error values of the AM-kernel are related to the ϵ value selected on validation. The plot suggests that the map accuracy influence the error of the AM-kernel. Nevertheless starting from any map the error obtained by the AM-kernel is significantly lower than the SST and SOM-SD ones.

In the following we analyse the dependence of the classification error from each parameter of the SOM-SD and from the ϵ of the AM-kernel:

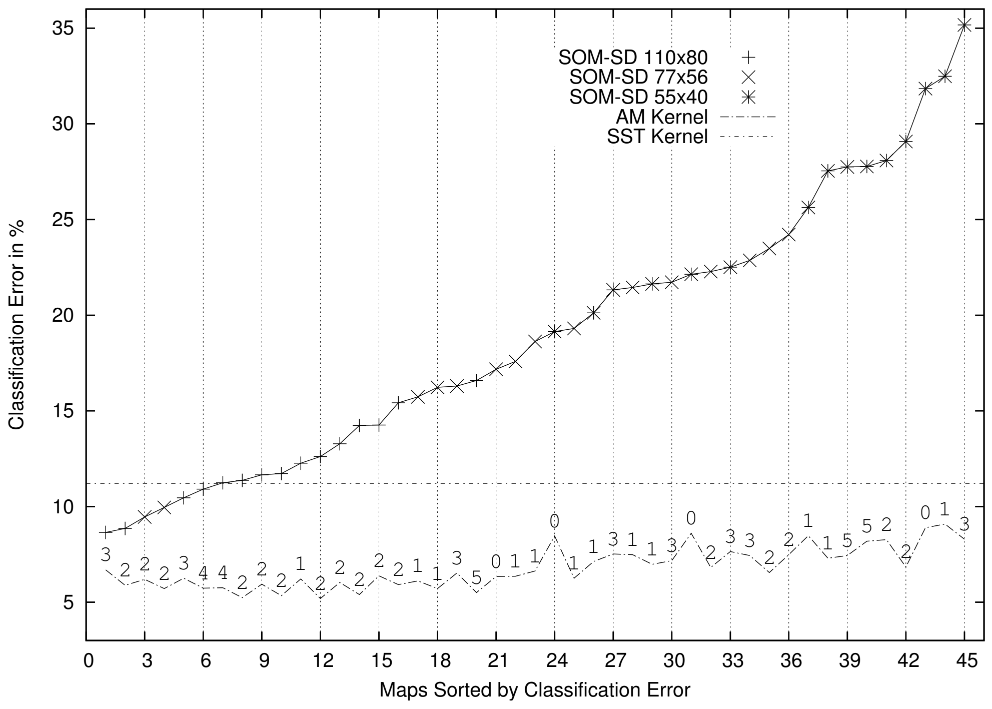

Figure 4.2: Comparison between classification error of the different techniques on the INEX 2005 test dataset. Maps on the x-axis are sorted by SOM-SD classification error. The error values of the AM-kernel are related to the ϵ value selected on validation (which is reported in correspondence of the map error value).

map size The mean classification error of 110×80 maps is 5.83% with standard deviation 0.44, the mean error of 77×56 maps is 6.59% and the mean classification error of 55×40 maps is 7.94%. As for the SOM-SD, larger maps result in lowest classification error. Note however that, for each map size, the standard deviation is significantly lower than the standard deviation of the SOM-SD. The gap between the classification error related to different map sizes is significantly lower than SOM-SD one.

Training iterations Table 4.5 reports the mean classification error of the maps with respect to the number of training iterations. Statistics are again divided according to map size. Differently from the SOM-SD, the AM-kernel classification accuracy of 110×80 and 77×56 maps do not benefit from a higher number of training iterations. That is reasonable since, if the map is large enough, the neurons with time specialize more and more tending, in the end, to represent singular structures. In other words, it is more likely that different structures are represented by the same set of neurons (getting a higher matching value) during the first learning iterations than at the end of the learning process. Since our goal was to let match different structures for reducing sparsity, not making use of too many training iterations seems a viable suggestion. Only for 55×40 maps a higher number of iterations helps in reducing the classification error. This may be due to the fact that, since the map is relatively small, very different structures can be forced to be encoded by nearby prototypes (influencing each other representation) and thus more training iterations are needed for differentiating those structures.

	Mean Classification Error (%) w.r.t to Map Size			
	110×80	77×56	55×40	All maps
iter=32	5.66 (0.33)	6.36 (0.54)	8.53 (1.07)	6.85 (1.50)
iter=64	5.68 (0.50)	6.65 (0.89)	7.84 (0.80)	6.73 (1.08)
iter=128	6.16 (0.74)	6.75 (0.51)	7.45 (0.57)	6.79 (0.65)

Table 4.5: Mean classification error of the AM-kernel with respect to number of training iterations (between brackets the standard deviation). Statistics are divided according to map size. The last column reports the mean classification error of all the maps.

Parameter μ_1 Table 4.6 shows the behaviour of the AM-kernel with respect to μ_1 values. It seems that there is no evident correlation of the accuracy and the parameter μ_1. By comparing corresponding elements of table 4.6 and table 4.3 it can be noticed that the error is always reduced and the lower standard deviation of the AM-kernel results suggests that AM-kernel results are quite robust.

Parameter ϵ We finally analyse the dependence of the results from the neighbour-

	Mean Classification Error (%) w.r.t to Map Size			
	110×80	77×56	55×40	All maps
μ_1=0.05	5.81 (0.54)	6.51 (0.90)	8.06 (0.98)	6.79 (0.81)
μ_1=0.25	5.74 (0.56)	7.25 (0.37)	7.74 (1.19)	6.91 (0.71)
μ_1=0.45	5.46 (0.25)	6.66 (0.47)	7.97 (0.46)	6.70 (0.39)
μ_1=0.65	5.98 (0.25)	6.45 (0.23)	7.75 (0.76)	6.73 (0.42)
μ_1=0.85	6.17 (0.45)	6.06 (0.32)	8.20 (0.47)	6.81 (0.42)

Table 4.6: Mean classification error of the AM-kernel with respect to the parameter μ_1 (between brackets the standard deviation). Statistics are divided according to map size. The last column reports the mean classification error of all the maps.

hood size of the AM-kernel. Data is presented in table 4.7. The classification error, for all map size, initially decreases by increasing ϵ, reaches a minimum when $\epsilon = 1$ or $\epsilon = 2$ and then increases again. This seems to suggest that, at least for this task, propagating information to a restricted number of neighbouring nodes is beneficial. When, on the contrary, the ϵ value is too high, the influence of a node extends too far letting match dissimilar structures which are not supposed to.

	Mean Classification Error (%) w.r.t to Map Size			
	110×80	77×56	55×40	All maps
ϵ=0	6.50 (0.39)	6.86 (0.50)	7.97 (0.64)	7.11 (0.51)
ϵ=1	5.74 (0.36)	6.50 (0.54)	7.83 (0.71)	6.69 (0.54)
ϵ=2	5.72 (0.41)	6.53 (0.60)	7.84 (0.67)	6.69 (0.56)
ϵ=3	5.98 (0.44)	6.91 (0.59)	8.01 (0.60)	6.97 (0.54)
ϵ=4	6.16 (0.50)	7.05 (0.76)	7.92 (0.62)	7.04 (0.63)
ϵ=5	6.23 (0.40)	7.06 (0.46)	8.03 (0.63)	7.11 (0.49)

Table 4.7: Mean classification error of the SOM-SD maps with respect to the parameter ϵ. Statistics are divided according to map size. The last column reports the mean classification error of all the maps.

In order to explain the obtained results, we collected statistics about sparsity on the test with respect to AM neighbourhood size. The values obtained are listed in table 4.8. The sparsity index for all maps with $\epsilon = 0$ is basically equivalent to the

	Mean Sparsity Index (%) w.r.t to Map Size			
	110×80	77×56	55×40	All maps
ϵ=0	0.55 (0.0001)	0.55 (0.0005)	0.54 (0.0018)	0.55 (0.0008)
ϵ=1	0.54 (0.0097)	0.47 (0.0356)	0.28 (0.0536)	0.43 (0.0330)
ϵ=2	0.40 (0.0593)	0.15 (0.0181)	0.11 (0.0072)	0.22 (0.0282)
ϵ=3	0.16 (0.0361)	0.10 (0.0084)	0.09 (0.0093)	0.12 (0.0179)
ϵ=4	0.11 (0.0134)	0.09 (0.0113)	0.08 (0.0153)	0.09 (0.0134)
ϵ=5	0.09 (0.0132)	0.07 (0.0241)	0.07 (0.0222)	0.08 (0.0199)

Table 4.8: Mean sparsity index of the AM-Kernel maps with respect to the parameter ϵ on the INEX 2005 dataset (between brackets the standard deviation). Statistics are divided according to map size. The last column reports the mean sparsity index of all the maps.

baseline: 0.54% (see table A.2). This means that apparently no different structures are represented by the same neuron in the map. Note that having the same set of matching structures does not imply that kernel functions must be equal since different kernel functions may weight differently each match. The reason for adding the ϵ parameter was precisely to reduce sparsity, i.e. to allow matchings between structures represented similarly by the SOM-SD. By increasing the neighbourhood size ϵ, the sparsity reduces. This is more evident for smaller maps since it is more likely that structures are represented nearby. However, if we compare the corresponding elements of table 4.7 and table 4.8 it is evident that a reduction in sparsity does not implies systematically a reduction of the classification error. Generally speaking, from $\epsilon = 0$ to $\epsilon = 2$ both the sparsity and the classification error tend to be reduced; from $\epsilon = 3$ to $\epsilon = 5$ the sparsity still decreases but the classification error increases. In other words low sparsity does not guarantee high accuracy. High ϵ values may over-represent a structure on the map thus making it similar to structures which are considered different for the current task.

In order to further sustain our claim that the AM kernel is especially useful for tasks in which traditional kernels are sparse, we ran the same set of experiments on a similar, but non sparse dataset involving XML documents which has been used for the 2006 INEX Competition (see section A.2). In this case the training, validation and test sets consist of 4237, 1816 and 6054 documents, respectively. Each document belongs to 1 out of 18 classes. By applying the same methodology as in the previous experiment, the following results were obtained. The sparsity of the SST kernel was 0.0025 and its classification error was 59.31%. In this case, the mean sparsity of the AM kernels, computed over 45 different maps, ranged from 0.0026 (with standard deviation 0.0000051) to 0.0003 (with standard deviation 0.0003034) when considering the same set of values for the ϵ parameter. The SOM-SD classification error ranged from 67.66% to 60.77% with a mean value of 63.98%. The test error of the AM kernel varied from 64.24% to 58.24% with a mean value of 61.579%.

In order to make an empirical analysis of the relationship between sparsity and classification error, we run a number of experiments on a set of artificial datasets with different values of sparsity. We considered the two-class problem of discriminating the examples of the INEX 2006 dataset belonging to class 8 from the examples belonging to any other class. From this dataset we created 7 more datasets with the following values of the sparsity index with respect to SST kernel: 0.0025, 0.07, 0.15, 0.21, 0.40, 0.57, 0.75. The datasets were obtained by concatenating to each label a uniformly generated random number. In this way identical labels have a chance to be transformed into different labels, thus adding sparsity. The number of digits composing the random numbers is constant and thus no different labels can become equal. By varying the range of the random numbers the desired level of sparsity can be obtained. The test has been performed on the following map: size$= 110 \times 80$, training iterations$= 64$, $\mu_1 = 0.45$. Figure 4.3 compares the classification error on the test set of the SST and AM-kernel on each dataset. Best parameters of the SST, i.e. λ, and AM-kernel (ϵ) are selected on validation among the same values used for the previous experiments. The experiments suggests that the AM-kernel, under

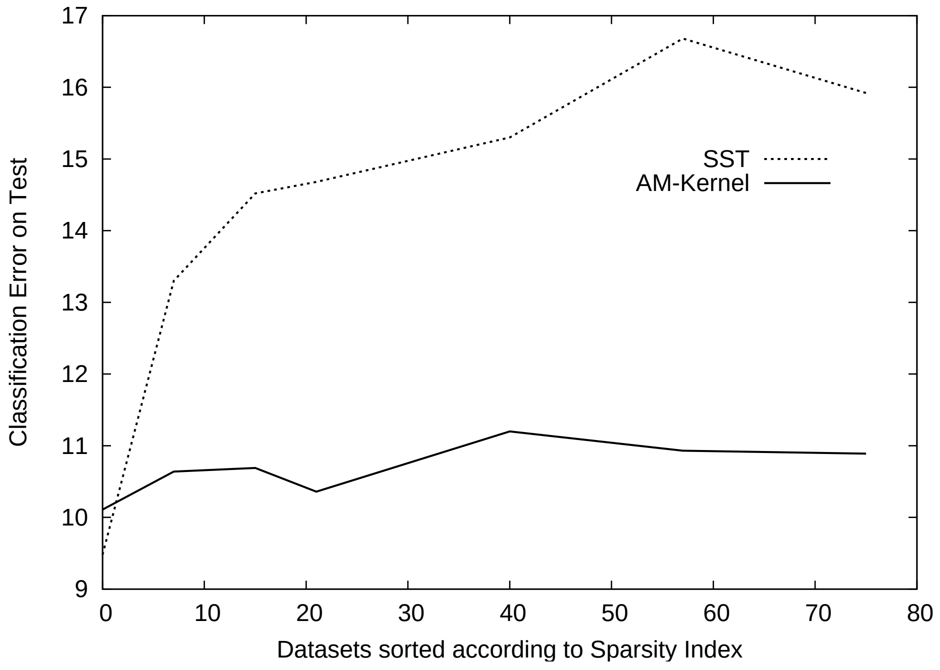

Figure 4.3: Classification error of the SST and AM-kernel on various datasets with different levels of sparsity.

certain limits, is robust to the increase of sparsity.

It has not been formally demonstrated that the SOM-SD algorithm is able to always produce the lower dimensional representation which best represent the topology of the input space. While the demonstration is beyond the scope of this chapter, we empirically investigated the usefulness of the SOM-SD learning algorithm by running the same set of experiments on the INEX 2005 dataset starting from random, i.e. non trained, maps. Since the number of training iterations was fixed to 0, 15 maps were created. Classification error on the test set of the AM-kernel (results on the validation set are very similar) ranges from 90.36% to 28.21% with a mean value of 51.55% and standard deviation 17.68. Results are most evidently correlated with the parameter μ_1: higher values of μ_1 give lowest classification error. This is not surprising since being the map random the structural information contained in the neurons is useless or misleading, thus by giving more importance to label information best results are obtained. The results of the last experiment clearly show the usefulness of the SOM-SD learning algorithm.

In order to assess whether the obtained results were dependent on the specific datasets involving XML documents, we decided to perform additional experiments involving a different dataset. We selected a dataset which is typically used for data mining research, i.e. the LOGML dataset (see section A.4). It consists of user sessions of the Rensselaer Polytechnic Institute Computer Science Department website, processed in order to obtain tree representations (see section A.4 for details). It is a two-class classification problem. 3 datasets are available. They comprises 8074, 7409 and 7628 examples, respectively. The datasets are very sparse: the mean of the 3 sparsity index values is 0.9595. The classification error of the SST and the AM-Kernel was computed by performing a 3-fold cross-validation considering, in each round, one of the dataset as the test set. The results are obtained by using 10 different maps, obtained by using the following parameters:

- map dimension: 110×80;
- number of training iterations: $32, 64$;
- μ: $0.05, 0.25, 0.45, 0.65, 0.85$.

In Figure 4.4 a comparison between the classification error of the different techniques on the LOGML test set is plotted, again following the same style of presentation as in Figure 4.2. It can be noted that the results obtained for this dataset show less variance since they are obtained by a 3-fold cross-validation approach. Also for this dataset, the AM kernel was able to get a significant improvement over SOM-SD, and over SST, although the improvement in this case is smaller.

As a last remark we want to stress the fact that the AM-kernel can be defined on top of any topology preserving low dimensional representation of the data, not necessarily it must be combined with the SOM-SD. For example, when information about the ascendant nodes is important for the task, maps could be built by means of the CSOM-SD. The AM-kernel can be defined for more complicated types of structures such as graphs by using the graph SOM-SD.

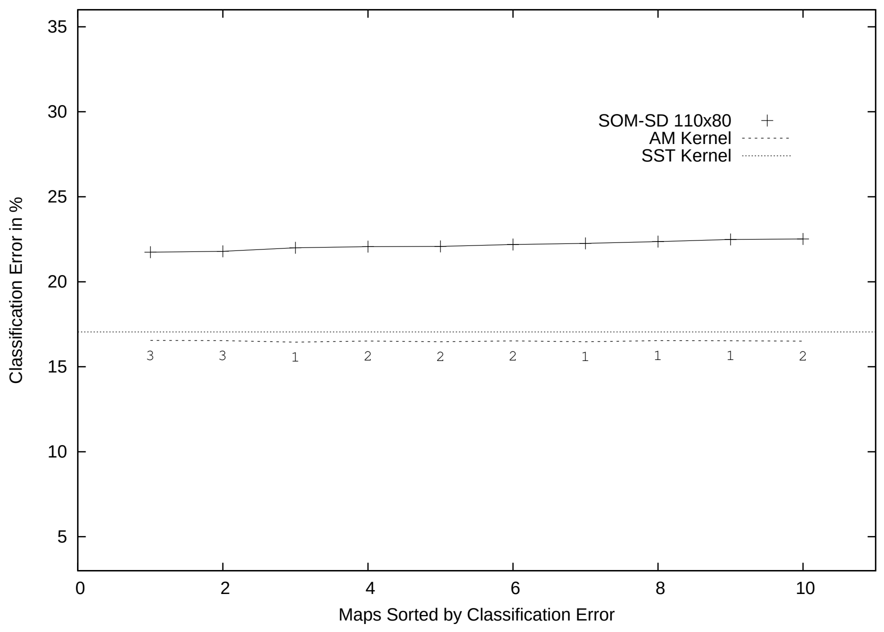

Figure 4.4: Comparison between classification error (using 3-fold cross-validation) of the different techniques on the LOGML test dataset. Maps on the x-axis are sorted by SOM-SD classification error. The error values of the AM-kernel are related to the ϵ value selected on validation (which is reported in correspondence of the map error value).

Chapter 5

A Novel Kernel for Trees: Convolution Route Kernel

As the proportion of publications listed in chapter 3 suggests, convolution kernels, among the class of tree kernels, have received great attention from the research community. Convolution kernels describe complex objects in terms of their constituent parts. However, such an approach tends to discard explicit information about the "relative positioning" of the elements with respect to one another. Even kernels based on counting the number of common paths such as the Spectrum Tree Kernel (see Definition 3.2.1) do not take into account the relative positioning of the nodes. The aim of the present chapter is to investigate the usefulness of this type of information for building expressive non sparse tree kernels.

The chapter is organised as follows: the novel family of kernels, called generalized route kernels, is defined in Section 5.1. Section 5.2 describes an instance of the generalized route kernel and provides an efficient algorithm for its computation. Section 5.3 describes experiments performed on two collections of XML formatted documents showing that the proposed kernel improves on state-of-the-art tree kernels, thus confirming the viability of the approach.

5.1 Generalized Route Kernel

This section formally describes the proposed Generalized Route Kernel. In particular, it will be gradually introduced by starting from a simple formulation, and then adding pieces with the aim of progressively enriching the feature space. In the end, we will obtain a kernel which is able to match set of routes according to the similarity of the labels of the correspondent nodes.

Definition 5.1 (Route) *Let T be a (positional) tree, $v_1, v_2 \in T$ any two nodes in the tree, and $p(v_1, v_2) = [v_1, \ldots, v_2]$ the (shortest) path connecting v_1 and v_2 through the edges of T (not considering edge direction). Then the route from v_1 to v_2 in T, denoted by $\pi(v_1, v_2)$, is the sequence of indices of edges connecting the consecutive nodes in the path $p(v_1, v_2)$. This indices are taken positive (or negative) whenever edges are traversed away from (resp. towards) the root.*

Figure 5.1 gives an example of a tree and a route computed between nodes **a** and **e**. The nodes connected by dashed edges represent the shortest path connecting nodes **a** and **e**, i.e. $p(a, e)$. The route connecting nodes **a** and **e** is represented by the sequence $[2, 3]$, since node **b** is the second child of **a** and node **e** is the third child of **b**. The route connecting nodes **g** and **b** is $[-3, 2]$.

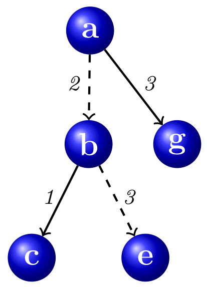

Figure 5.1: An example of a route connecting nodes labelled with **a** and **e**. The nodes connected by dashed edges are the ones comprising the path between the two nodes. The route is formed by the sequence 2, 3 since node **b** is the second child of **a** and node **e** is the third child of **b**.

Some other functions must be introduced before getting to the definition of the route kernel. The parent of a node is identified by the function $pa(v)$. If a node, i.e. the root, has no parent then $pa()$ is undefined. The function $chpos(v)$ returns the position of v with respect to its parent, i.e. $chpos(v) = i$ if v is the i-th child of $pa(v)$. When $pa(v)$ is undefined, $chpos(v)$ is also undefined. We finally introduce the function $p^l(v_i, v_j)$ which returns the sequence of node labels of the nodes in $p(v_i, v_j)$ respecting the order of the nodes in the path, i.e. if $p(v_i, v_j) = v_i, v_h, v_s, v_j$, then $p^l(v_i, v_j) = (l_i, l_h, l_s, l_j)$. For example, given the path connecting the nodes labelled with a and g in Figure 5.1, $p^l = (a, b, e)$.

The route $\pi(v_i, v_j)$ can be recursively defined as:

$$\pi(v_i, v_j) = \begin{cases} \pi\left(v_i, pa(v_j)\right).chpos(v_j) & \text{if} \quad v_i \neq v_j \\ \epsilon & \text{if} \quad v_i = v_j \end{cases} \tag{5.1}$$

where v_i, v_j are nodes of a tree T, and the "." operator creates a sequence from two list of objects, and ϵ is a symbol for the empty sequence. We use a function δ for comparing generic objects

$$\delta(x, x') = \begin{cases} 1 & \text{if} \quad x = x' \\ 0 & \text{if} \quad x \neq x'. \end{cases} \tag{5.2}$$

A first kernel can be defined by comparing the set of all routes:

$$K_1(T_1, T_2) = \sum_{v_i, v_j \in T_1} \sum_{v_l, v_m \in T_2} k_1(\pi(v_i, v_j), \pi(v_l, v_m)), \tag{5.3}$$

where k_1 is a kernel defined on the routes. For example by setting $k_1(x, x') = \delta(x, x')$, K_1 would become a kernel counting the number of common routes of the two trees. It is straightforward to show that if k_1 is a valid kernel then K_1 is a valid kernel. Let $\mathcal{M}$ be the set of trees, χ'_T the set of paths in T, $\mathcal{M}_{T_1,T_2} = \chi'_{T_1} \times \chi'_{T_2}$ the Cartesian product of the two sets of paths of T_1 and T_2, then K_1 is an instance of the mapping kernel (see Section 3.1). $\mathcal{M}_{T_1,T_2}$ is clearly a transitive function and thus K_1 is positive semidefinite.

In order to add expressiveness to the kernel we combine k_1 with a kernel k_2 defined on sequences of node labels:

$$k_2(p^l(v_i, v_j), p^l(v_l, v_m)). \tag{5.4}$$

The combined local kernel $k_3 = k_1 \otimes k_2$ can be defined based on the product between k_1 and k_2:

$$\begin{aligned}
&k_1(\pi(v_i, v_j), \pi(v_l, v_m))k_2(p^l(v_i, v_j), p^l(v_l, v_m)) &= \\
&k_1 \otimes k_2 \left(\left(\pi(v_i, v_j), p^l(v_i, v_j) \right), \left(\pi(v_l, v_m), p^l(v_l, v_m) \right) \right) &= \\
&k_3' \left(\left(\pi(v_i, v_j), p^l(v_i, v_j) \right), \left(\pi(v_l, v_m), p^l(v_l, v_m) \right) \right) &= \\
&k_3 \left(p(v_i, v_j), p(v_l, v_m) \right).
\end{aligned} \tag{5.5}$$

Note that the operator $\otimes$ preserves positive semidefiniteness (see page 31), and thus k_3' is a valid kernel. k_3 is obtained from k_3' noticing that a pair of nodes (v_i, v_j) uniquely determines the route $\pi(v_i, v_j)$ and the path $p(v_i, v_j)$, and thus k_3 can be defined to be equivalent to k_3' although using a simplified notation.

A further extension to the kernel can be obtained by letting match group of features as in eq.(2.21). This would allow to count as features the simultaneous presence of groups of nodes at particular positions:

$$K_{gr}(T_1, T_2) = \left(\sum_{v_i, v_j \in T_1} \sum_{v_l, v_m \in T_2} k_3\left((v_i, v_j), (v_l, v_m) \right) + e \right)^d, \tag{5.6}$$

where $e \in \mathbb{R}$, $d \in \mathbb{N}$.

5.2 An instantiation of the Generalized Route Kernel

In this section, we discuss an instance of the generalized route kernel and give for it an efficient implementation. Our purpose, in this section, is to obtain a kernel function matching identical pairs of type (π, l), where l is the label of the last node in the path associated to the route π. The following modifications to eq. (5.6) are needed. Since k_3 is defined in terms of k_1 and k_2, we start by modifying these two functions:

$$k_1(\pi(v_i, v_j), \pi(v_l, v_m)) = \delta(\pi(v_i, v_j), \pi(v_l, v_m))\lambda^{|\pi(v_i,v_j)|}, \tag{5.7}$$

where $\lambda \in \mathbb{R}$ is a user defined parameter and $|\pi(v_i, v_j)|$ is the length of the route $\pi(v_i, v_j)$, which corresponds to the length of the corresponding path. k_1 in eq. (5.7) let match only identical routes. Note that the value of each match is weighted according to a value λ dependent on the length of the route. The basic idea motivating the introduction of the parameter λ is to downweight the influence of larger routes in the same way as described for the ST and SST kernels (see section 3.1.2). The function k_2 is modified as follows:

$$k_2(p^l(v_i, v_j), p^l(v_l, v_m)) = \delta(l_j, l_m). \tag{5.8}$$

k_2 tests if the labels of the last nodes in the two paths are identical. Note that the use of the kernel k_2 could have been avoided by imposing the following condition on $\mathcal{M}_{T_1,T_2}$:

$$((v_i, v_j), (v_l, v_m)) \in \mathcal{M}_{T_1,T_2} \Leftrightarrow l_j = l_m. \tag{5.9}$$

In the following, we also experiment an alternative definition for the kernel k_2 based on the production rooted at the last node of the path:

$$k_{prod}(p(v_i, v_j), p(v_l, v_m)) = \delta(prod(v_j), prod(v_m)), \tag{5.10}$$

where $prod(v)$ is the subtree rooted at node v and composed by all the children of v.

We further restrict the set of feasible routes by imposing the following condition to the sets $\chi^{'}_{_T}$:

$$\chi^{'}_{_T} = \left\{ p(v_i, v_j) | v_i, v_j \in T \wedge v_j \in \overset{v_i}{\triangle} \right\}, \tag{5.11}$$

where $v_j \in \overset{v_i}{\triangle}$ means that v_j is a descendant of v_i. In other words routes are allowed only between a node and its descendants or the node itself.

The final form of the route kernel is thus the following:

$$K_{route}(T_1, T_2) = \left(\sum_{(p(v_i,v_j),p(v_l,v_m)) \in \mathcal{M}_{T_1,T_2}} \delta(v_j, v_m)\delta(\pi(v_i, v_j), \pi(v_l, v_m)) + e \right)^d, \tag{5.12}$$

Since matches are allowed only with identical routes, a node v at depth o in the tree has associated o non null features: one feature related to the path composed only by the same node, one feature related to the path $p(pa(v), v)$, one feature related to the path $p(pa(pa(v)), v) = p(pa^2(v), v)$, and so on until the feature related to the path connecting the root of the tree to the node:$p(pa^o(v), v)$. The total number of non null features for a tree with $|T|$ nodes is less or equal than $\sum_{i=1}^{|T|} depth(v_i) = avgdepth(T) \cdot |T|$, where $depth(v)$ is the depth of node v and $avgdepth(T)$ is the average depth of a node in T. Note that the total number of non null features is equal to $avgdepth(T) \cdot |T|$ when the labels of the nodes in T are all different. Figure 5.2 gives an example of the set of features associated with a simple tree.

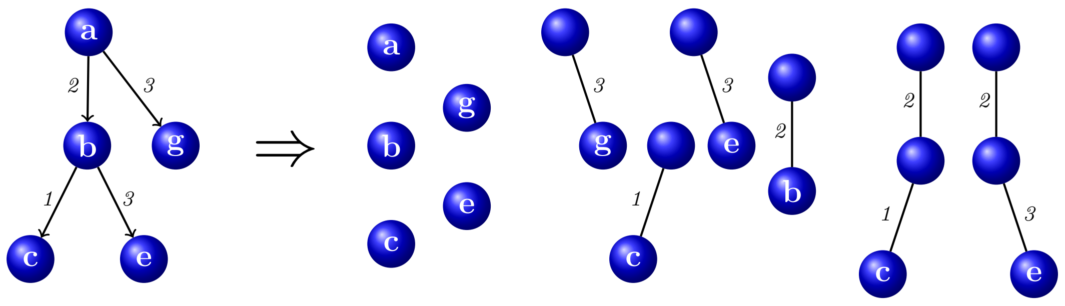

Figure 5.2: A tree (left) and its set of features according to the route kernel defined in eq. (5.12).

5.2.1 Implementation

We now turn our attention to an efficient implementation of the kernel proposed in eq. (5.11) and eq. (5.12). Without loss of generality it is assumed that parameters e and d are set to 0 and 1, respectively. Since no routes of different length can match, the definition of $\mathcal{M}$ given by eq. (5.9) and eq. (5.11), can be rewritten as

$$\mathcal{M}_{T_1,T_2} = \bigcup_{s=0}^{o} \mathcal{M}^{(s)}_{T_1,T_2}, \tag{5.13}$$

where o is the smallest of the maximum depth of the trees T_1 and T_2. $\mathcal{M}^{(s)}_{T_1,T_2}$ is defined as:

$$((v_i, v_j), (v_l, v_m)) \in \mathcal{M}^{(s)}_{T_1,T_2} \Leftrightarrow l_j = l_m \wedge |\pi(v_i, v_j)| = |\pi(v_l, v_m)| = s. \tag{5.14}$$

Clearly $\forall i \neq j, T_1, T_2 . \mathcal{M}^{(i)}_{T_1,T_2} \cap \mathcal{M}^{(j)}_{T_1,T_2} = \emptyset$. Eq. (5.12) can be rewritten as

$$K_r(T_1, T_2) = \left(\sum_{((v_i,v_j),(v_l,v_m)) \in \mathcal{M}_{T_1,T_2}} \delta(\pi(v_i, v_j), \pi(v_l, v_m)) \right) = \\ \sum_{s=0}^{o} \left(\sum_{((v_i,v_j),(v_l,v_m)) \in \mathcal{M}^{(s)}_{T_1,T_2}} \delta\left(\pi(v_i, v_j), \pi(v_l, v_m)\right) \right) = \\ \sum_{s=0}^{o} \left(\sum_{((pa^s(v_j),v_j),(pa^s(v_m),v_m)) \in \mathcal{M}^{s}_{T_1,T_2}} \delta\left(\pi(pa^s(v_j), v_j), \pi(pa^s(v_m), v_m)\right) \right) = \sum_{s=0}^{o} \left(\sum_{\substack{v_j \in T_1 \\ v_m \in T_2}} C^{(s)}(v_j, v_m) \right), \tag{5.15}$$

where $C^{(s)}(v_i, v_j)$ can be computed according to the following rules:

1. if $s = 0$ then $C^{(s)}(v_i, v_j) = \delta(v_i, v_j)$;

2. if $s > 0$ then $C^{(s)}(v_i, v_j) = C^{(s-1)}(v_i, v_j)\delta(chpos(pa(v_i)^s), chpos(pa(v_j)^s))$.

Eq. (5.15) suggests a strategy for computing the route kernel: by starting from the set of common labels of the two trees, identical routes of increasing length can be looked for. Clearly, since a common route of length s can have match only if its subroute of length $s-1$ has a match, the proposed strategy can be stopped as soon as no more routes of current length are found.

The algorithm we propose (algoritm 2) assumes to treat trees with arbitrary but finite out-degree. It starts by computing the number of nodes with the same labels.

It then proceeds by comparing the routes of length s, with s going to from 1 to the minimum of the maximum of the depths of the two trees.

In the following the behaviour of the algorithm is analyzed in detail. Lines from 1 to 4 initialize internal variables and create a sorted list of nodes for each tree. The procedure costs $O(|T| \log |T|)$. Lines from 5 to 12 compute the number of matchings due to the routes of length 0, i.e. the number of common labels. The computational complexity of the procedure is $O(\rho|T|)$. Line 14 creates an array which will contain information about the matchings between routes at current level. Its cost is $O(maxout)$. Each while iteration (lines 15 to 42) costs $O(n)$, since L lists may not contain more than $O(n)$ elements and for each list $O(1)$ operations are performed. Line 15 has a computational complexity of $O(maxout)$. However, it can be skipped by making use of a variable and an array of the same size of F. The variable, say t, is initially set to 0 and it is incremented every time the while loop is entered. Whenever F is written to (lines 17 and 25), the current value of t is recorded in the auxiliary array, say F'. When values from F are read (lines 22 and 32), they are considered valid if the corresponding value of F' is equal to t, otherwise the read operation returns a value of 0. Each while iteration counts the number of common routes of length s. Initially $s = 1$; If at least one common route of length s is found (there is a non empty list L), then, in the next while iteration, routes of length $s+1$ are looked for. Clearly, there are no more than the length of the longest path in the smallest tree, i.e. min(maxdepth(T_1),maxdepth(T_2)), of such paths. By summing up the cost of all lines, the total cost of the algorithm in the worst case becomes: $O(|T| * avgdepth(T) + |T| \log |T|)$. Note that the average depth of a node in a tree can be at most $O(n)$, thus the complexity of the algorithm can be at most $O(n^2)$.

Algorithm 2: Pseudo-code for computing the kernel $k_{route}(T_1, T_2)$.

```
   Input: $T_1$ and $T_2$, two trees. $\lambda$, a user defined parameter.
   Output: $k_{route}(T_1, T_2)$
1  k = 0; $maxout$ = max {outdegree($T_1$),outdegree($T_2$)};
2  create a sorted list $List_1$ of nodes $v \in T_1$ according to their labels;
3  create a sorted list $List_2$ of nodes $v \in T_2$ according to their labels;
4  for $j = 1$ to 2 do
5      $v_j$= $\mathbf{0}$;                                   // vector of dimension $\rho$
6      foreach $v \in List_j$ do
7          $v.label$= $l(v)$;
8          $v_j$ += $v.label$;
9      end
10 end
11 k += $\lambda v_1^{\intercal} v_2$;
12 $\lambda_2 = \lambda \cdot \lambda$;
13 create array F[] of dimension $maxout$;
14 while $|List_1| > 0 \wedge |List_2| > 0$ do
15     for $i = 1$ to $maxout$ do $v_{1,i}$= $\mathbf{0}$, F[1,i] = 0;
       ;                        // previous step (and line 39) can be avoided, see text.
16     foreach $node \in List_1$ do
17         F[$chpos(v)$]= 1;
18         $v_{1,chpos(v)}$ += $v.label$;
19     end
20     for $i = 1$ to $maxout$ do $v_{2,i}$= $\mathbf{0}$, F[2,i] = 0;
21     foreach $v \in List_2$ do
22         if $F[1,chpos(v)]= 0$ then
23             remove $v$ from $List_2$;
24             else
25                 F[2,i] = 1;
26                 $v_{2,chpos(v)}$ += $v.label$;
27                 substitute $v$ with $w$ s.t. $w \equiv pa(v)$ and $w.label$= $v.label$;
28             end
29         end
30     end
31     foreach $v \in List_1$ do
32         if $F[2,chpos(v)]= 0$ then
33             remove $v$ from $List_1$;
34             else
35                 substitute $v$ with $w$ s.t. $w \equiv pa(v)$ and $w.label$= $v.label$;
36             end
37         end
38     end
39     for $i = 1$ to $maxout$ do k += $\lambda_2 v_{1,i}^{\intercal} v_{2,i}$;
40     $\lambda_2 = \lambda_2 \cdot \lambda$;
41 end
42 return $k$
```

5.2.2 Relationship with other Kernels

It is known that, with respect to the feature space, $ST \subseteq SST \subseteq PT$. Given a tree T, ST associates to T at most a linear number of non-null features. SST and PT, with respect to T, have at most an exponential number of non-null features. The number of non-null features of the Route kernel is at most $avgdepth(T)|T|$. The features space of the Route kernel is not directly comparable with the one of the PT kernel. However if all labels of a tree were identical, the feature space of the Route kernel would properly be included into the feature space of the PT kernel. This is not the common case: two matching nodes for the Route kernel having different ascendants would not match for the PT kernel. The analysis of the Gram matrices of the various kernels show that ST and SST are more sparse than the route kernel, which in turn is more sparse than the Partial tree kernel. In fact, the sparsity index of the PT kernel on the three datasets is 0. The sparsity of the route kernel is 0.46% for INEX 2005 and 0 for all the other datasets. Regarding computational complexity, ST is faster than SST, which in turn is faster than PT. The Route kernel has a lower computational complexity than the PT kernel and, in the worst case, the same quadratic complexity of the SST. Note that, when $avgdepth(T)$ is $O(1)$, the computational complexity of the Route kernel is equivalent to the one of the ST kernel.

5.3 Experiments and Discussion

Experiments were performed to test the effectiveness of the proposed kernel with respect to ST, SST, the polynomial version of SST (obtained by exponentiating the kernel value in the same way as described for the route kernel in eq. (5.6)) and the PT kernel. The implementation of these kernels is available as part of the svm-light software[1]. Our approach has been tested on the INEX 2005 dataset (section 5.3.1),

[1]http://dit.unitn.it/~moschitt/Tree-Kernel.htm

the INEX 2006 dataset (section 5.3.2), the LOGML dataset (section 5.3.3). In some cases the training procedure was stopped due to excessive training times. Specifically we set a 24 hours time out for each single learning procedure. The time out was necessary because of the number of parameters involved.

5.3.1 Experiments on INEX 2005

In order to verify the effectiveness of the proposed kernel, a number of experiments were run on the INEX 2005 dataset. For each setting of the hyper-parameters, SVM-based multi-class classification was performed by using the one-against-all methodology.

Experiments were carried out with the local kernel defined on node labels, eq. (5.8) and with the local kernel defined on productions, eq. (5.10). In both scenarios a subspace of the parameters of the route kernel were evaluated. Specifically, experiments were performed with both normalized and not normalized route kernel. The e parameter in eq. (5.12) was set to 0 in all experiments. The parameter d in eq. (5.12) was set to $1, 2, 3$. For each combination of the previous parameters, the λ and c parameter of the SVM were selected in validation among the values: $\lambda = \{0.05, 0.1, 0.25, 0.50, 0.75, 1.0, 2.0\}$ and $c = \{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$. The lowest classification error of the subtree and subset tree kernel is 11.21% (see appendix A.1). The polynomial version of the SST reached a 10.67% classification error on the test set. A first set of experiments were performed by making use of the kernel defined on node labels, eq. (5.8). Table 5.1 summarizes the results obtained. The classification error ranges from 3.39% to 3.02%. The lowest classification error, 3.02%, is obtained by using the polynomial version with degree 3 of the normalized kernel.

We proceeded by testing the route kernel with local kernel defined on productions, as described by eq. (5.10). Table 5.2 summarizes the results obtained. The classification error ranges from 3.58% to 3.35%. The lowest classification error is reached by the polynomial version with degree 2 of the not normalized kernel.

kernel	d	best λ	best c	error % (valid)	error % (test)
not normalized	1	0.25	1	3.10	3.14
not normalized	2	1	0.001	3.38	3.10
not normalized	3	0.75	0.001	3.66	3.39
normalized	1	0.25	10	3.17	3.17
normalized	2	0.25	10	3.10	3.06
normalized	3	0.5	10	3.17	3.02

Table 5.1: Accuracy of the route kernel with local kernel defined on node labels on the INEX 2005 dataset. The columns represent, respectively, the type of kernel, the exponent d in eq. (5.12), the λ and c values selected on validation, the classification error on validation, the classification error on the test set.

kernel	d	best λ	best c	error % (valid)	error % (test)
not normalized	1	0.25	1	3.31	3.52
not normalized	2	0.25	0.1	3.66	3.35
not normalized	3	0.75	0.001	3.80	3.41
normalized	1	0.50	100	3.52	3.39
normalized	2	0.1	10	3.52	3.54
normalized	3	0.1	10	3.94	3.58

Table 5.2: Accuracy of the route kernel with local kernel defined on node productions on the INEX 2005 dataset. The columns represent, respectively, the type of kernel, the exponent d in eq. (5.12), the λ and c values selected on validation, the classification error on validation, the classification error on the test set.

Table 5.3 compares the classification error of the route kernel with respect to ST, SST, the polynomial version of SST and the PT kernel. Note that the Partial tree kernel has lowest classification error, 2.96%, while the route kernel with local kernel defined on labels places second with 3.06% classification error.

kernel	valid error %	test error %
ST	13.15	11.27
SST	12.79	11.21
Polynomial SST	12.09	10.67
Partial Tree	2.96	2.96
Route, $k_2 \equiv$ eq. (5.8)	3.10	3.06
Route, $k_2 \equiv$ eq. (5.10)	3.31	3.52

Table 5.3: Comparison between the classification error of ST, SST, the polynomial SST, the Partial Tree kernel and the Route kernel, respectively. Data refer to the INEX 2005 dataset. The columns represent, respectively, the type of kernel, the lowest classification error on validation, the corresponding classification error on the test set.

5.3.2 Experiments on INEX 2006

A second round of experiments has been performed on the INEX 2006 dataset to further test the kernel. For each setting of the hyper-parameters, SVM-based multiclass classification was performed by using the one-against-all methodology in a similar manner as for the experiments on the INEX 2005 dataset.

Both local kernels were experimented with the same set of parameters used for the INEX 2005 dataset (see section 5.3.1). The lowest classification error obtained on the test set by the subset tree kernel is 60.14%. The polynomial version of the SST reached a 59.30% classification error on the test set. Table 5.4 summarizes the results with the local kernel defined on node labels. Classification error values range from 77.14% to 59.94%.

kernel	d	best λ	best c	error % (valid)	error % (test)
not normalized	1	0.05	1000	62.05	64.35
not normalized	2	0.25	1	58.72	59.94
not normalized	3	0.05	0.1	60.55	62.24
normalized	1	0.05	100	63.88	63.73
normalized	2	0.25	1000	62.46	70.53
normalized	3	0.25	10	62.14	77.14

Table 5.4: Accuracy of the route kernel with local kernel defined on node labels on the INEX 2006 dataset. The columns represent, respectively, the type of kernel, the exponent d in eq. (5.12), the λ and c values selected on validation, the classification error on validation, the classification error on the test set.

The lowest classification error is obtained by the polynomial version with degree 2 of the not normalized kernel. The improvement of the route kernel with respect to SST, computed as in Section5.3.1, goes from -30.08% to -1.07%. The improvement with respect to the AM-kernel ranges from -26.72% to 1.52%. The local kernel, in the best case, can only slightly improve on the AM-kernel and it always performs worst than SST.

Experiments proceeded by testing the route kernel with local kernel defined on node productions. Table 5.5 summarizes the results obtained.

Classification error values range from 63.05% to 57.12%. The lowest classification error is obtained by the polynomial version with degree 2 of the normalized kernel.

Table 5.6 compares the classification error of the route kernel with respect to ST, SST, the polynomial version of SST and the PT kernel. Note that the lowest classification error is obtained by the route kernel with local kernel defined on node productions, 58.09%. The partial tree kernel reaches a 58.17%. Other kernels have a classification error going from 1.47% worse than the route kernel to 9.89%. The INEX 2006 is a harder task than INEX 2005. Nonetheless the route kernel, in conjunction with the local kernel defined on productions, is able to improve on the classification error of all the other techniques we compared to.

kernel	d	best λ	best c	error % (valid)	error % (test)
not normalized	1	0.50	1	60.94	62.62
not normalized	2	0.05	1	56.33	57.12
not normalized	3	0.05	0.1	57.39	57.71
normalized	1	0.5	1	61.44	63.05
normalized	2	0.3	10	55.55	58.09
normalized	3	0,3	100	58.22	58.97

Table 5.5: Accuracy of the route kernel with local kernel defined on node productions on the INEX 2006 dataset. The columns represent, respectively, the type of kernel, the exponent d in eq. (5.12), the λ and c values selected on validation, the classification error on validation, the classification error on the test set.

kernel	valid error %	test error %
ST	68.32	67.98
SST	56.55	59.56
Polynomial SST	55.55	59.88
Partial Tree	57.83	58.87
Route, $k_2 \equiv$ eq. (5.8)	58.72	59.94
Route, $k_2 \equiv$ eq. (5.10)	55.55	58.09

Table 5.6: Comparison between the classification error of ST, SST, the polynomial SST, the Partial Tree kernel and the Route kernel, respectively. Data refer to the INEX 2006 dataset. The columns represent, respectively, the type of kernel, the lowest classification error on validation, the corresponding classification error on the test set.

5.3.3 Experiments on LOGML

The LOGML dataset consists of user sessions of the Rensselaer Polytechnic Institute Computer Science Department website. It is a binary classification problem. 3 datasets are available. They comprises 8074, 7409 and 7628 examples, respectively. Because LOGML is divided into 3 datasets, it was natural to compute the classification error of the kernels by performing a 3-fold cross-validation considering, in each round, one of the dataset as the test set. The set of parameters involved is the same as the one for the INEX 2005 experiments (see section 5.3.1).

kernel	d	best λ	best c	cross validation error %
not normalized	1	0.75	0.1	16.73
not normalized	2	0.1	1	16.84
not normalized	3	0.1	1	17.45
normalized	1	0.75	1	16.20
normalized	2	0.5	1	16.36
normalized	3	0.1	1	16.82

Table 5.7: Classification error of the route kernel with local kernel defined on node labels on the LOGML dataset. The columns represent, respectively, the type of kernel, the exponent d in eq. (5.12), the best λ and c values, the corresponding cross validation error.

Table 5.7 summarizes the result obtained by the route kernel with local kernel defined on node labels. The lowest error is obtained by the normalized version of the kernel setting $\lambda = 0.75$ and $c = 1$.

Table 5.8 summarizes the result obtained by the route kernel with local kernel defined on node productions. The lowest error is obtained by the normalized version of the kernel setting $\lambda = 0.1$ and $c = 1$.

Finally, table 5.9 compares the route kernel to the ST, SST, the polynomial version of SST and the PT kernels. The values listed are the mean of the classification error on the three folds. Note that the Route kernel with local kernel defined on node labels has the lowest mean classification error, 16.20%.

kernel	d	best λ	best c	cross validation error %
not normalized	1	0.1	1	16.98
not normalized	2	0.1	1	18.15
not normalized	3	0.05	1000	19.76
normalized	1	0.1	1	16.79
normalized	2	0.05	1	17.97
normalized	3	0.05	100	20.09

Table 5.8: Classification error of the route kernel with local kernel defined on node productions on the LOGML dataset. The columns represent, respectively, the type of kernel, the exponent d in eq. (5.12), the best λ and c values, the corresponding cross validation error.

kernel	cross validation error %
ST	16.72
SST	16.84
Polynomial SST	16.82
Partial Tree	16.40
Route, $k_2 \equiv$ eq. (5.8)	16.20
Route, $k_2 \equiv$ eq. (5.10)	16.79

Table 5.9: Comparison between the classification error of ST, SST, the polynomial SST, the Partial Tree kernel and the Route kernel, respectively. Data refer to the LOGML dataset.

5.3.4 Discussion

Experimental results obtained on the three non-trivial datasets, some of which involving up to 18 different classes, have shown that the feature space induced by the proposed kernel is rich enough to give state of the art performances with respect to the most widely used tree kernels. The proposed kernel is thus able to reach quite good results while keeping a reasonable computational complexity. In addition to that, the proposed algorithm for computing the kernel can be easily adapted to parallel computation. In fact, each tread of computation could take responsibility for computing the contribution to the kernel given by the matchings between routes that end up on specific nodes (of the two trees) with identical label. The same approach can not be applied to the other kernels because of the strong dependencies among nodes.

Chapter 6

Efficient Score Computation by Compacting the Model

While Support Vector Machines has a high generalization capability, several authors have pointed out that a drawback of this approach is the time required both in learning and classification phases [5, 17, 49, 62]. Typically kernel methods spend most of their time during execution evaluating kernel functions. This stays true also for the classification phase. Kernel methods express a solution as a combination of a subset of training examples, the support vectors. Since the computation of the score depends on the number of support vectors, when this number is a large fraction of the training set, the total time required may become excessive for some applications. This is especially true when dealing with large amount of data.

Several approaches have been pursed to tackle this problem.

Nguyen and Ho [49] describe an iterative process to replace two support vectors with a new one representing both of them. The replacement operation involves the maximization of a one-variable function in the range $[0, 1]$. Anguita et al. [5] replace the set of support vectors with a single one, called archetype. However the archetype is defined in the feature space and thus it is necessary to solve a further quadratic optimization problem to find an approximation in input space. The approximated model, according to the authors "maintain the ability to classify the data with a moderate increase of the error rate". Downs et al. [17] proposes a numerical algorithm for eliminating the linearly dependent support vectors and update accordingly their correspondent base vectors. Kudo et al. [40] propose to represent the model with a vector summing the feature space projections of all the examples in the model. Since this approach is not convenient when the feature space is very large, the authors describe a way to approximate the model.

Tipping [62] proposed the Relevance Vector Machine, an algorithm based on Bayesian theory with a functional form similar to SVM which tries to minimize the number of support vectors produced during learning. However the Relevance Vector Machine requires $O(N^3)$ time and $O(N^2)$ memory to train. Relevance Vector Machines not only bound the user to a particular learning algorithm, but also can not be used for on-line settings or large datasets.

All previous cited papers reduce the computational burden of the classification phase by finding an approximation of the model. In most cases the approximation is found by solving an optimization problem. The computational complexity or type of learning algorithm prevent their use in on-line settings. In this chapter we describe a way to speed up the computation of the score with no approximation and without the need to solve an optimization problem. The approach we describe can be applied to any convolution kernel and with any kernel based algorithm. The basic idea is to avoid the re-computation of kernels between the same substructures belonging to different examples.

6.1 General Considerations

We start by recalling the definition of generalized convolution kernel given in section 3.1:

$$k(x, x_i) = \sum_{(x', x'_i) \in \mathcal{M}_{x,x_i}} k(x', x'_i) = \sum_{x' \in \chi'_x} \sum_{x'_i \in \chi'_{x_i}} [\![(x', x'_i) \in \mathcal{M}_{x,x_i}]\!] k(x', x'_i),$$

where $[\![condition]\!]$ is defined as:

$$[\![condition]\!] = \begin{cases} 1 & \text{if } \ condition \ \text{ is true} \\ 0 & \text{otherwise.} \end{cases}$$

The score function, instantiated for the case of convolution kernels with respect to a model M, can be rewritten as:

$$\begin{aligned}
S(x) = & \sum_{x_i \in M} \alpha_{x_i} \sum_{x' \in \chi'_x} \sum_{x'_i \in \chi'_{x_i}} [\![(x', x'_i) \in \mathcal{M}_{x,x_i}]\!] k(x', x'_i) & = \\
= & \sum_{x' \in \chi'_x} \sum_{\substack{x'_i \in \chi'_{x_i} \\ x_i \in M}} [\![(x', x'_i) \in \mathcal{M}_{x,x_i}]\!] \alpha_{x_i} k(x', x'_i) & = \\
= & \sum_{x' \in \chi'_x} \sum_{\substack{x'_i \cdot (x', x'_i) \in \mathcal{M}_{x,x_i} \\ x_i \in M}} \alpha_{x_i} k(x', x'_i) & = \\
= & \sum_{x' \in \chi'_x} f(x'_i, M)\, k(x', x'_i).
\end{aligned} \tag{6.1}$$

where $f(x'_i, M) = \sum_{\substack{x'_i \cdot (x', x'_i) \in \mathcal{M}_{x,x_i} \\ x_i \in M}} \alpha_{x_i}$. $f(x'_i, M)$ sums the α's related to the same substructure appearing in different structures of the model. It is clear that the knowledge of the $f(x'_i, M)$ values would avoid to recompute many times the same kernel function (for the same substructures in different structures), thus reducing the overall time required for computing the score. As it will be shown in the next sections, the savings could be remarkable especially when dealing with large amount of data. It is worth noting that our purpose is not to store k values, thus what we are going to describe is not a cache-like approach. The reason for that is to avoid to have to recompute some kernel values. In fact, to our knowledge, no finite-size cache (unless of course it can save all kernel values) can guarantee to have no cache miss.

6.2 Compacting a Forest of Trees

In the following sections it will be shown how to instantiate the results of section 6.1 to the case of tree structured data. Equation (6.1) suggests a general methodology for reducing the computational resources needed for computing the score function. In order to compute $f(x'_i, M)$ efficiently a suitable data structured for encoding the substructures of the structures in the model has to defined. In the following we discuss this problem for learning scenarios involving convolution kernel functions for which the set χ'_x is composed by the set of subtrees of x. We start by observing that a scoring function is defined on the basis of a set of trees (a forest). It is quite reasonable to assume that different trees in the forest share common subtrees, otherwise learning would hardly be effective. If this assumption is true, considering each tree independently from the others would imply that many partial kernel calculations, i.e. the ones corresponding to the shared subtrees, are recomputed from scratch. We suggest to represent the forest as a unique structured model, namely an annotated minimal Directed Acyclic Graph (DAG) representing shared subtrees only once while keeping track of the frequency of each subtree.

Once the minimal DAG has been computed, the evaluation of the scoring function for a new tree can be performed by a single match between the tree and the minimal DAG. As we will see, this can lead to a great gain in computation.

6.2.1 From a Forest to a Directed Acyclic Graph

Given a tree forest F, if there exists a set of trees $\mathcal{T} \subseteq F$ which share a common subtree $\hat{T}$, then we can think to explicitly represent $\hat{T}$ only once. Following this idea, we define a procedure that merges all the trees in a forest F into a single *minimal* DAG, i.e., a DAG with a *minimal* number of vertices.

More formally, a minimal DAG is represented as an *annotated* DAG, where each node is annotated with a pair (*label*, *frequency*). The *label* field represents the label associated with the node, while the *frequency* field is used to count how many repetitions of the same subtree rooted in that node are present in the tree forest. The exact role of the frequency field will become clearer in the following.

The advantage of having a minimal DAG is the considerable reduction in space complexity to represent a forest with no loss of information. This space reduction will eventually lead also to a time complexity reduction. In some cases, this reduction can even be exponential. In Figure 6.1 an example of a forest and its minimal DAG representation, is given. In the following we also give the pseudocode and implementation details, of a procedure able to efficiently compute shared subtrees and to exploit this information to represent a forest as an annotated DAG.

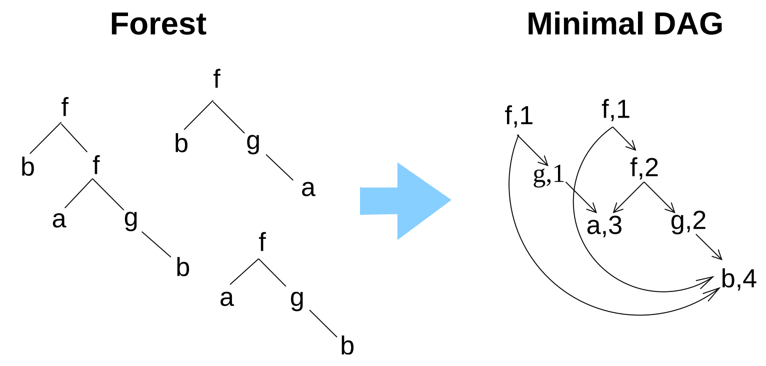

Figure 6.1: Example of how to represent a forest as a minimal DAG with no loss of information. Nodes in the minimal DAG are annotated with a label and the frequency in the forest of the subtree rooted at that node.

Minimal DAG creation

Figure 6.2 describes the algorithm for creating a minimal DAG from forest of trees. The procedure `InvTopologOrder`(T_j) used in the algorithm returns a total order of vertices of T_j which is compatible with the (inverted) partial order defined by the arcs of T_j. Thus, the first vertices of the list will be vertices with zero outdegree, followed by vertices which have only children with zero outdegree, and so on. Using this order guarantees the (unique) existence of vertices $c_i \in \mu D$ s.t. $dag\_rooted\_at(c_i) \equiv dag\_rooted\_at(ch_i[v])$. In fact, for each i, the vertex $ch_i[v]$ is processed before vertex v and is either inserted in μD or recognized as a duplicated of a vertex already present in μD.

It should be noted that the function $dag\_rooted\_at(\cdot)$ can be implemented quite efficiently by an indexing mechanism, where a unique code is defined for a void child, and a unique code for the root of each different DAG is generated by recursively considering the label of the root and the (unique) codes computed for its children.

```
MinimalDAG

Input: A tree forest $F = \{T_1, \ldots, T_k\}$
/* $l \equiv label$, $f \equiv frequency$, $dag \equiv dag\_rooted\_at$ */
Initialize: $\mu D \leftarrow$ void DAG;

 for $j \leftarrow 1$ to $N$ do

 $vertex\_list \leftarrow$ InvTopologOrder($T_j$);

 while $vertex\_list \neq \emptyset$ do

         $v = pop(vertex\_list)$;
         if $\exists u \in \mu D$ s.t. $dag(u) \equiv dag(v)$
                 then $f(u) \leftarrow f(u) + f(v)$
                 else
                   add to $\mu D$ a node $w$ where
                       $l(w) = l(v)$ and $f(w) = f(v)$
                   forall children $ch_i[v]$ of $v$
                       add arc $(w, c_i)$ to $\mu D$ where
                         $c_i \in Nodes(\mu D)$ and
                         $dag(c_i) \equiv dag(ch_i[v])$

 return $\mu D$
```

Figure 6.2: The algorithm to transform a tree-forest into a minimal DAG.

In our implementation we have realized an indexing mechanism by using Adelson-Velsky Landis (AVL) trees [1]. Let t be a vertex of a tree T and l the length of the longest path in T starting from t and reaching a vertex of T with 0 out-degree. Then an AVL tree for each possible value of l is defined, i.e. $AVL^{(l)}$. When a vertex $s \in T$ with 0 out-degree is processed, there is an attempt to insert it in $AVL^{(0)}$ using as key the label associated with s. If the key is already present, it means that a vertex s' with 0 out-degree and same label has already been inserted in $AVL^{(0)}$. In that case, s is marked, the frequency for s' is incremented by 1, and the pointer to s' is associated with it, so that, when the parents of s are processed, their pointers to

s are substituted by the pointer to s'. When all the vertices with 0 out-degree are processed, vertices with $l = 1$ are considered and the same process is repeated with the following two differences: *i)* the children of q are checked and for each marked child, its pointer is substituted by the associated pointer; *ii)* the key used for the insertion in $AVL^{(1)}$ is given by the concatenation of the label associated with q with the ordered sequence of (revised) pointers to its children. If the insertion of q fails, i.e., an "equivalent" vertex is already present, the same operations described for s are executed. The treatment of vertices with $l > 1$ is the same described for the case $l = 1$. Both insertion and lookup into an AVL tree take $O(\log(n))$, where n is the number of items contained into the AVL tree.

Notice that using a different AVL tree for each value of l allows us to reduce the number of vertices inserted in the AVL, thus reducing the searching time for the key.

Adding a tree to a minimal DAG

The algorithm used to insert a new tree into the model is depicted in Figure 6.3. Note that it is very similar to the generation of a minimum DAG with the difference being that in this case the frequency associated with the model is updated with the frequency of the subtree to be added weighted by the quantity α.

6.2.2 Efficient Score Computation

We have shown how to transform a tree forest into an annotated DAG with no loss of information. Now, we show how (a variant of) the minimal DAG can be exploited for the efficient computation of a general scoring function involving tree structured data.

Going back to the definition of the tree kernels given in Section 3.1.2, it is useful to notice that the core of the computation of these kernels stands on the computation of the $C(t_i, t_j)$ and that these values could be computed just once for shared substructures and re-used when needed. Thus, the basic idea of our approach

```
                              TreeIns

Input: An ADAG $\mu D$ and a tree $(T, \alpha)$ to be inserted
/* $l \equiv label$, $f \equiv frequency$, $dag \equiv dag\_rooted\_at$ */

 $vertex\_list \leftarrow$ InvTopologOrder($T$);

 while $vertex\_list \neq \emptyset$ do

        $v = pop(vertex\_list)$;

        if $\exists u \in \mu D$ s.t. $dag(u) \equiv dag(v)$

              then $f(u) \leftarrow f(u) + \alpha \cdot f(v)$

              else

                 add to $\mu D$ a node $w$ where

                    $l(w) = l(v)$ and $f(w) = \alpha \cdot f(v)$

                 forall children $ch_i[v]$ of $v$

                    add arc $(w, c_i)$ to $\mu D$ where

                       $c_i \in Nodes(\mu D)$ and

                       $dag(c_i) \equiv dag(ch_i[v])$

 return $\mu D$
```

Figure 6.3: The algorithm to insert a weighted ADAG in a larger ADAG.

is to build a weighted annotated DAG (the *model*), very similar to the minimal DAG, for the training forest. Clearly, the model should also contain information about the coefficients of the linear combination of the scoring function.

Such a model can be built incrementally and the frequencies associated to the nodes are computed by simply cumulating the frequencies already associated to the nodes of the current minimal DAG with the frequencies of the (sub)trees belonging to the tree which is currently added to the model. Note that, this model is computed only once and then kept in memory in a way that it will be possible to perform the computation of the scoring function for new trees efficiently. The algorithm used to insert a new tree into the model is presented in Figure 6.3.

Now, let $M_{(F,\vec{\alpha})}$ be the model obtained as described above after inserting all the trees, with the associated α values, belonging to the forest F, and T any tree. Notice that the insertion order of the trees is irrelevant, since any insertion order leads to the same model. We define the following quantity representing the degree of *match* between the tree and the model:

$$S_{\mu DAG}(M_{(F,\vec{\alpha})}, T) = \sum_{t_i \in M_{(F,\vec{\alpha})}} \sum_{t_k \in T} f_i C(t_i, t_k), \tag{6.2}$$

where f_i is the weighted frequency associated to the root node of the subtree t_i in $M_{(F,\vec{\alpha})}$.

Now, we can show that computing this quantity is equivalent to the computation of the scoring function

Theorem 6.1 *Let $M_0 = \emptyset$ be the void initial minimal DAG. Consider a forest F and the models obtained by sequentially inserting each tree $T_i \in F$ with the associated α_i, i.e. $M_i = TreeIns(M_{i-1}, (T_i, \alpha_i))$, where $i = 1, \dots, |F|$ and $M_{|F|} \equiv M_{(F,\vec{\alpha})}$. Let $S_{\mu DAG}(M_{(F,\vec{\alpha})}, T)$ be defined as in (6.2) and f_i the weighted frequencies in $M_{(F,\vec{\alpha})}$, then the following holds:*

$$S(T) = S_{\mu DAG}(M_{(F,\vec{\alpha})}, T).$$

Proof: Let us consider the set of all the possible subtrees S_k indexed by $k = 1, ..., m_S$. First of all, we can check easily that, if the algorithm in Figure 6.3 is used to insert n trees into the model, starting from the void model, then we have:

$$f_k = \sum_{i=1}^{n} \alpha_i \beta_k(T_i) \tag{6.3}$$

where $\beta_k(T)$ is the number of times a given subtree S_k appears into a tree T.

Now, let $root(S)$ be the root node of a tree S, we have

$$\begin{aligned} K(T_i, T) &= \sum_{t_i \in T_i} \sum_{t \in T} C(t_i, t) \\ &= \sum_{k,j} \beta_k(T_i) \beta_j(T) C(root(S_k), root(S_j)) \end{aligned}$$

where k, j vary over the space of all possible subtrees. The equality above is true because $\beta_i(T) = 0$ whenever the subtree S_i is not present in T. Hence,

$$\begin{aligned}
S(T) &= \sum_{i=1}^{n} \alpha_i K(T_i, T) \\
&= \sum_{i=1}^{n} \alpha_i \sum_{k,j} \beta_k(T_i)\beta_j(T)C(root(S_k), root(S_j)) \\
&= \sum_{k,j} (\sum_{i=1}^{n} \alpha_i \beta_k(T_i))\beta_j(T)C(root(S_k), root(S_j)) \\
&= \sum_{k,j} f_k \bar{f}_j C(root(S_k), root(S_j))
\end{aligned}$$

This last equality is true because of eq. (6.3) and because $\bar{f}_j = \beta_j(T)$ is true by definition in a minimal DAG.

Now, since $f_k = 0$ when the subtree S_k is not a subgraph of M_n and $\bar{f}_j = 0$ when the subtree S_j is not a subgraph of $\mu DAG(T)$, then we obtain

$$\begin{aligned}
S(T) &= \sum_{t_k \in M_n} \sum_{t_j \in \mu DAG(T)} f_k \bar{f}_j C(t_k, t_j) \\
&= S_{\mu DAG}(M_n, \mu DAG(T)).
\end{aligned}$$

Thus the statement is proved when $n = |F|$. $\square$

The result of the above theorem is quite interesting, since it states that any score function involving the considered tree kernels and representable as in eq. (2.14), thus any solution of a kernel method problem involving the considered tree kernels, can be efficiently represented (reduction in space complexity) and computed (reduction in time complexity) by resorting to annotated minimal DAGs. Thus, after learning, it is possible to compact the obtained model in order to obtain a more efficient computation of the scoring function. Thus, for on-line learning, it seems there is the possibility to improve the efficiency both in space and time. This is the subject of the next section, where we show how learning with the kernel perceptron, and its voted variant, can be made efficient.

```
DAG-Perceptron Algorithm

Input: stream of pairs $(T_i, y_i)$, where $y_i \in \{-1, +1\}$
Initialize: Model $M \leftarrow$ void DAG;
Repeat forever
  read $(T_i, y_i)$ from the stream;
  Compute Perceptron score:
      $S(T_i) \leftarrow S_{\mu DAG}(M, T_i)$;
  if $y_i S(T_i) \leq 0$ then
      $M \leftarrow TreeIns(M, (T_i, y_i))$
```

Figure 6.4: The DAG-Perceptron algorithm.

6.2.3 The DAG Kernel Perceptron

In this section we show how the result of the theorem can be exploited to make efficient on-line learning for tree-structured data using the tree kernels described in Section 3.1.2. Specifically, as an example, we define an efficient version of the Perceptron algorithm. We call this version the DAG Kernel Perceptron, since it is based on the minimal DAG described above.

DAG-Based Implementation of the Perceptron

We now describe our DAG-based implementation of the Perceptron algorithm (see section 2.3.2 for a description). The algorithm is presented in Figure 6.4. The model is represented as a annotated minimal DAG. Whenever an input tree is misclassified the model is updated by adding it to the model. In doing that, the weight y_i is added to each node in the annotated DAG corresponding to the input tree nodes. The soundness of the algorithm is trivially guaranteed by the theorem given in Section 6.2.2.

Another complexity issue, which is important when dealing with large amounts of data, is considered in the following, and new strategies are given to further improve the efficiency and effectiveness of our approach.

The computation of the tree kernel is based on the recursive computation of the $C(t_1, t_2)$ values. When considering a model represented as a tree forest, storing all the $C(t_1, t_2)$ values is not a problem since the computation of the total kernel is done by summing the values of the kernels between each tree in the forest and the input tree. Since the same storage space can be reused for different trees in the forest, the storage requirement is dominated by the largest tree in the forest. On the contrary, the DAG requires to keep in memory all of its nodes, which store the $C(t_1, t_2)$ values. Since the number of DAG vertices generally grows with training[1], a significant storage requirement is expected, especially when considering data mining applications, where the number of input items could be huge.

For this reason, it is important to limit storage requirements. In this respect, two observations can be done: *i)* when considering a vertex v belonging to the input tree, it is readily evident that when all the $C(u, v)$ entries, with u belonging to the DAG, are computed, the entries referring to children of v can be removed, since no other tree vertex will refer to them; *ii)* an entry $C(u, v)$ is computed (and thus stored) only if *production(u)=production(v)*, thus the "name" of a vertex u belonging to the DAG, can be defined as the composition of *production(u)* plus a progressive numerical id assigned to the vertices of the DAG bearing the same *production(u)*. Equivalently this means that, given a vertex v in the input tree, the elements of the row $C(\cdot, v)$ can be enumerated progressively, disregarding the 0 valued elements, which correspond to vertices in the DAG that do not bear the same production as v.

[1]In fact, each error leads to the insertion of a new tree in the DAG model. In the most favourable case, the inserted tree is already present in the DAG, and only the frequencies associated to nodes of the DAG which correspond to nodes of the inserted tree need to be updated. In the worst case, neither the whole tree, nor any subtree belonging to it are present in the DAG and all the nodes of the inserted tree need to be added to the DAG.

On the basis of these observations, the following joint strategies can be adopted to reduce the storage requirements: *i)* the input tree is read using a depth-first visit and as soon as a vertex completes the computation of its $C(u, v)$ entries, the storage space for the $C(u, v)$ entries referring to its children is deallocated; *ii)* for each distinct production, a list of matching vertices in the DAG is maintained with the aim of both speeding up the search for a production match, and also to assign a progressive numerical id to the matching vertices as well as the total number of matching vertices to the production; in this way, when a new vertex in the input tree is visited, it is possible to know how much storage space must be dynamically allocated for that vertex. It should be noticed that each list associated with a production can be maintained very efficiently by just: (1) using a counter c recording the current total number of vertices belonging to the list; (2) assigning as id to a new vertex the current value of c; (3) inserting the new vertex at the beginning of each list and incrementing c by 1. All the above operations can be done in constant time. The application of the above strategies reduces the storage need from $O(N_{dag}N_{tree})$, where N_{dag} is the number of nodes in the DAG, and N_{tree} is the number of nodes in the input tree, to $O(P_{max}h_{tree}b_{tree})$, where P_{max} is the length of the longest list of matching vertices associated with productions, h_{tree} is the depth of the input tree, and b_{tree} is the branching factor of the input tree. Of course, when considering more than 1 production, $N_{dag} > P_{max}$ and if there are q productions with the same probability to be associated with a vertex, $P_{max} = \frac{N_{dag}}{q}$. Moreover, usually $N_{tree} \geq h_{tree}b_{tree}$.

6.2.4 Voted Kernel Perceptron

In this section we describe how to use a DAG for computing the score function for the voted perceptron. The algorithm is described in Section 2.3.2. In our setting, trees are presented sequentially to the algorithm, and after e mistakes occurred on the input trees $T_1, \ldots, T_e$, the score function is

$$S_e(T) = \sum_{j=1}^{e} \alpha_j K(T_j, T) \tag{6.4}$$

which, exploiting Theorem 6.1, can be rewritten as

$$S_e(T) = \sum_{t_i \in M_e} \sum_{t_k \in T} f_i^e C(t_i, t_k) \tag{6.5}$$

where M_e is the model obtained by the kernel perceptron after e mistakes, and f_i^e are the corresponding weighted frequencies. Let E be the total number of mistakes after that all the training examples have been visited. Then the score for the average (unnormalized) voted perceptron is defined as

$$S_{voted}(T) \;=\; \sum_{e=1}^{E} c_e S_e(T)$$

where c_e is the the number of iterations between mistake e and mistake $e+1$. Exploiting eq. (6.5), we can rewrite the above equation as

$$\begin{aligned} S_{voted}(T) &= \sum_{e=1}^{E} c_e \sum_{t_i \in M_e} \sum_{t_k \in T} f_i^e C(t_i, t_k) \\ &= \sum_{t_i \in M_E} \sum_{t_k \in T} \left(\sum_{e=1}^{E} c_e f_i^e \right) C(t_i, t_k) \end{aligned} \tag{6.6}$$

where we impose $f_i^e = 0$ if $t_i \notin M_e$. Thus, it is clear that the final voted model is obtained by defining the new weighted frequencies $\tilde{f}_i = \sum_{e=1}^{E} c_e f_i^e$. If we define $\tilde{f}_i^e = \sum_{j=1}^{e} c_j f_i^j$, then it holds that $\tilde{f}_i^{e+1} = \tilde{f}_i^e + c_{e+1} f_i^{e+1}$, which can be used as an on-line rule to compute incrementally $\tilde{f}_i = \tilde{f}_i^E$. This implies a doubling of the storage requirement since for each i we need both a variable to store the current value of f_i^e and a variable to store the current value of $\tilde{f}_i^{e-1}$, however, the time complexity does not increase significantly since the update of the $\tilde{f}$s occurs only when a mistake is made, and the larger the model is, i.e. more variables need to be updated, the less likely a mistake is generated.

6.2.5 Kernel Combinations

Another efficiency issue is related to the possibility to exploit additional numerical features $\boldsymbol{\xi} \equiv [\xi_1, \ldots, \xi_d]$ associated with each tree. In that case, the score can be obtained as a combination of the score obtained by the tree kernel with the score obtained by these numerical features. For example, if the combination is the sum, the score can be computed as

$$S(T_i) = S_{\mu DAG}(M, T_i) + S_\phi(M_\phi, \xi_i),$$

where M_ϕ is the set of feature vectors plus labels corresponding to errors. When using a nonlinear kernel for the computation of the feature score, a proper treatment is due. In fact, let consider the generic computation of the score for the features

$$S_\phi(\boldsymbol{\xi}_i) = \sum_{(\boldsymbol{\xi}_j, y_j) \in M_\phi} y_j K(\boldsymbol{\xi}_i, \boldsymbol{\xi}_j)$$

If d is large, assuming that the computation of the kernel is $O(d)$, the computational complexity for the score is $O(d|M_\phi|)$.

If the $\boldsymbol{\xi}$ vectors are sparse, let say that no more that $k \ll d$ components are nonzero, then a more efficient computation can be performed. In fact, nonzero features of $\boldsymbol{\xi}$ vectors can be organized for fast access by feature id. Assuming that the probability for a feature to be nonzero is $l = \frac{k}{d}$, this means that each feature will be associated with an inverted list with expected length equal to $l|M_]phi|$. Thus, given an input feature, for each j a match in its inverted list should be found, which can be done in no less than $O(\log(l|M_\phi|))$ by exploiting the sorting of the items by vector index. This leads to a total complexity of $O(|M_\phi| k \log(l|M_\phi|))$. However, we can do better than this. In fact, we know that all the items contained into the inverted lists of matching features are used for computing the score. The only problem is to recognize for each j which are the features that match the input. This can be done using the following procedure. We assume that the inverted lists are sorted by decreasing vector index. First of all the inverted lists corresponding to matching input features are recovered and their heads are inserted into a max heap. Then the maximum value is extracted by the heap and its successor in the corresponding inverted list is inserted into the heap. This process is repeated giving origin to a stream of indexes extracted by the heap where equal indexes are clustered together, allowing the computation of the kernel for that index. This procedure has a complexity that is dominated by the insertion into the heap of all the items into the matching inverted lists. Since the heap will never contain more than k items, insertion costs $\log(k)$, while the total number of items is $kl|M_\phi|$. Thus the total complexity is $O(kl|M_\phi|\log(k))$, which is better than the previous one only if $l\log(k) < \log(l|M_\phi|)$, i.e., $k^l < l|M_\phi|$. Noticing that $l \in [0,1]$, it is not difficult to realize that when $k \ll |M_\phi|$ a significant savings in computation can be obtained. For example, assuming binary data structures are used, if $|M_\phi| = 2^{14}$, $l = 10^{-4}$, and $k = 2^5$, we have $l\log_2(k) = 0.0005$ versus $\log_2(l|M_\phi|) = 0.71228762$, with a speedup of more than 1424.

When considering the *voted perceptron* it is trivial to note that when a feature of a tree is inserted into the model, it remains till the end of training, i.e. when all examples have been visited. Consequently, the weight c to be associated to that feature is equal to $N - j$, where N is the total number of visited examples, while j is the index of the tree containing that feature and that was erroneously classified[2]. Thus, no additional information need to be stored in the model, while the weight c_j in the voted perceptron associated to each feature belonging to tree j, will exactly be $c_j = N - j$.

6.2.6 Experiments

The experiments presented in this section aim to show that our approach based on DAGs provides two kinds of benefits to the perceptron algorithms: a much faster computation time and a much lesser memory requirement.

For such purpose, we measured the computation time and the memory allocation for both the traditional Perceptron algorithm and the one based on DAGs. The target learning tasks were those involved in Semantic Role Labelling, i.e. the task to automatically extract a predicate along with its argument from a natural language sentence. This is usually divided in two classification steps: argument boundary detection and argument classification. In the former step, all the nodes of the

[2]Here we assume that examples are presented in increasing index order.

sentence parse tree are classified in *correct* or *incorrect* boundaries. The *correct* label means that the leaves (i.e. words) of the tree rooted in the target node are all and only those constituting an argument. In the latter step, given a correct boundary node (i.e. an argument node), its type, i.e. Arg0, Arg1,..,Arg5, ArgA and ArgM, is determined.

As a referring dataset, we used PropBank along with PennTree bank 2 (see section A.3 for a description).

In our experiments, we concentrated on boundary detection as the number of classifying instances is much larger. Indeed, they include all parse-tree nodes. For these experiments, we used the first 7 sections of PennTree bank for training for a total of 71,523 positive and 921,296 negative examples.

As the DAG performance is affected by node distribution within trees along with their maximum and average out-degree, we have studied such characteristics in our data sets. Table A.6 reports statistics about the data derived from the boundary detection dataset. We note that there are a large number of relatively small trees which however can have a large out-degree. Globally, the amount of nodes that have to be processed is very large, thus, the dataset is suitable to demonstrate the computational efficiency of our approach.

We started the experiments with the aim of comparing the standard perceptron and the dag voted perceptron. Note that computational complexity of the voted perceptron is higher than the standard perceptron, so the comparison is slightly unfair. The reason for using the voted perceptron is that we plan to do more experimentations, in a future work, aimed at increasing the accuracy on the task. The task is very complex from an efficiency point of view since the number of instances of the boundary dataset was about one million.

This dataset is quite demanding for computational expensive approaches like Support Vector Machines: only using a polynomial kernel on standard features (in general much faster than tree kernels), 10 days were required to converge.

Figure 6.5, 6.6 and 6.7 show the execution times for the standard perceptron and the voted dag perceptron when using the polynomial kernel, the subset tree kernel and a linear combination of both, respectively. In the case of the tree kernel and combination of the two kernels, executions with various parameters were run: the λ of the tree kernel has been given the values $\lambda \in \{0.3,\ 0.4,\ 0.5,\ 0.6,\ 0.7,\ 0.8,\ 0.9,\ 1.0\}$, the γ weighting the linear combination between tree kernel and polynomial kernel has been given the values $\gamma \in \{0.2, 0.3, 0.4, 0.5, 0.6\}$. However only the values related to the faster and slower parameter settings are plotted. When using the polynomial kernel, Figure 6.5, the total time spent by the voted dag perceptron and the standard perceptron for classifying the training set are 7503.87 and 9332.64 seconds, respectively. Note that the voted perceptron is 1828, 77 seconds faster than the standard perceptron. The use of the tree kernel, see Figure 6.6, allows the voted

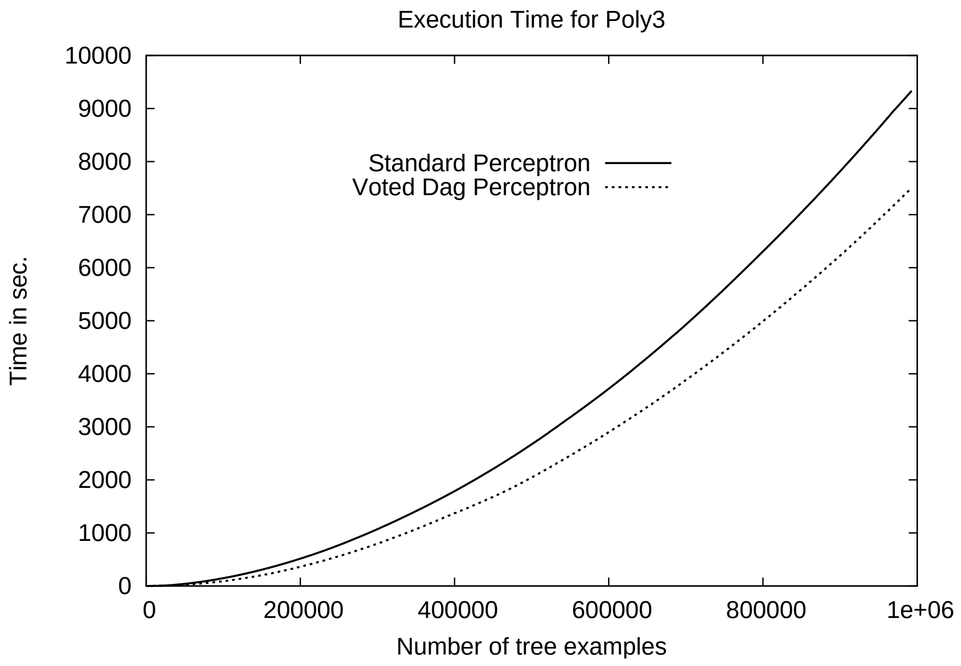

Figure 6.5: Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using a polynomial kernel with degree 3 (Poly3) over the training set with 992,819 examples.

dag perceptron to achieve the highest performance: 8681.10 seconds in the worst case ($\lambda = 1.0$) against 43995.02 seconds in the most favourable case for the standard perceptron ($\lambda = 0.3$), a gap of 9.8 hours, more than 5 times faster. Learning with

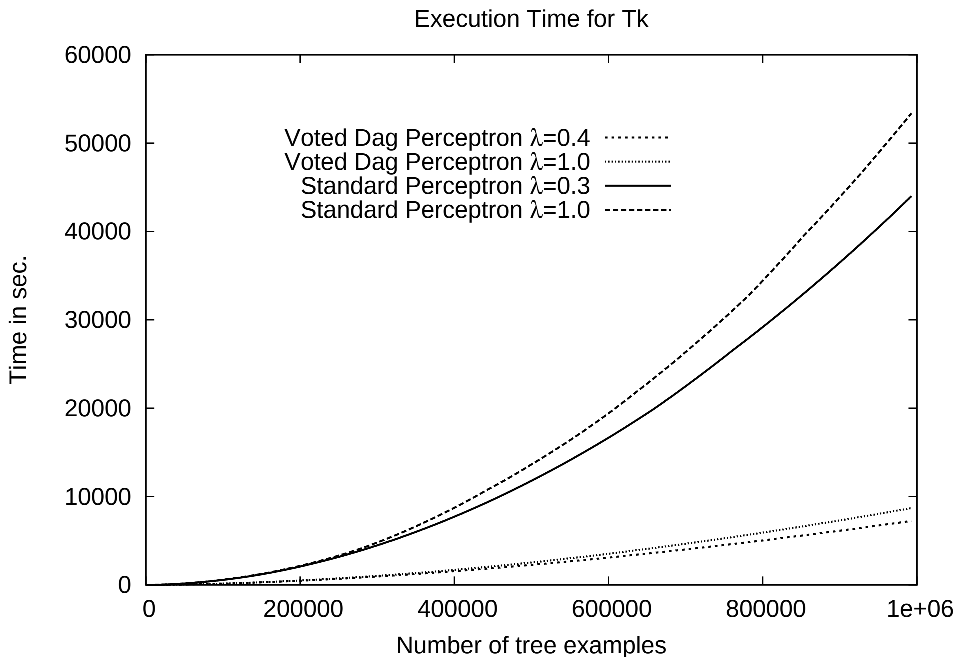

Figure 6.6: Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using the SST tree kernel (Tk) with different values for the λ parameter, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, over the training set with 992,819 examples. For each method, only the fastest and the slower executions are reported.

the combination of kernels (see Figure 6.7), requires in the worst case ($\lambda = 1.0$ and $\gamma = 0.6$) 11798.03 seconds for the voted dag perceptron and 20348.36 seconds for the standard perceptron in the most favourable case ($\lambda = 0.4$ and $\gamma = 0.3$). Note that the voted perceptron is 8550.33 seconds faster (about 2.4 hours). While the time saved in the case of the polynomial kernel alone is not remarkable, it becomes really significant when a tree kernel is involved.

Figure 6.8 and 6.9 show the memory usage of standard perceptron and voted dag perceptron algorithms. When the tree kernel is employed, Figure 6.8, the model created by the voted dag perceptron comprises from 202682 ($\lambda = 0.4$) to 232419 ($\lambda = 1.0$) nodes while the model created by the standard perceptron comprises from 1091652 ($\lambda = 0.4$) to 1475692 ($\lambda = 1.0$) nodes. Note that the amount of memory

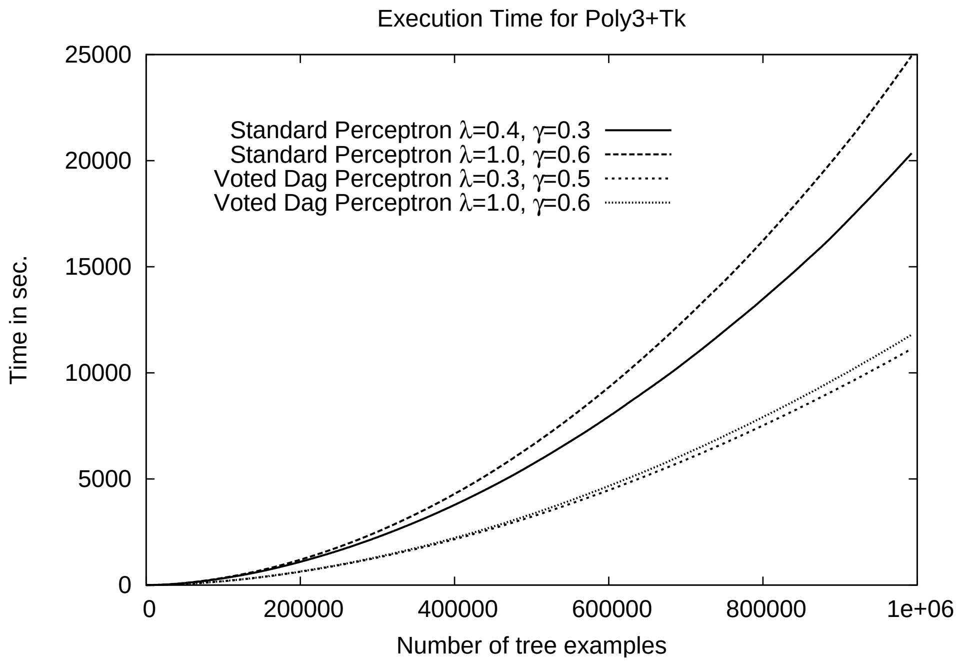

Figure 6.7: Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using a linear combination of a polynomial kernel with degree 3 (Poly3) with the SST tree kernel (Tk), i.e. $(1-\gamma)*Poly3+\gamma*Tk$, over the training set with 992,819 examples. Different values for λ and γ have been considered, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and $\gamma \in \{0.2, 0.3, 0.4, 0.5, 0.6\}$. For each method, only the fastest and the slower executions are reported.

used by the standard perceptron can be 7 times higher than the one employed by the voted dag perceptron. The model created by the voted dag perceptron, when a combination of kernels is employed (Figure 6.9), comprises from 97856 to 103544 nodes, while the model created by the standard perceptron comprises from 328932 to 458814 nodes. he amount of memory used by the standard perceptron can be 4.6 times higher than the one employed by the voted dag perceptron.

A further experimentation has been performed in order to show that the results obtained are not only related to the particular dataset or domain. Thus, for this second round of experiments we chose to get our data from the INEX 2005 dataset (see section A.1 for a description). The training set comprises 4820 examples and is

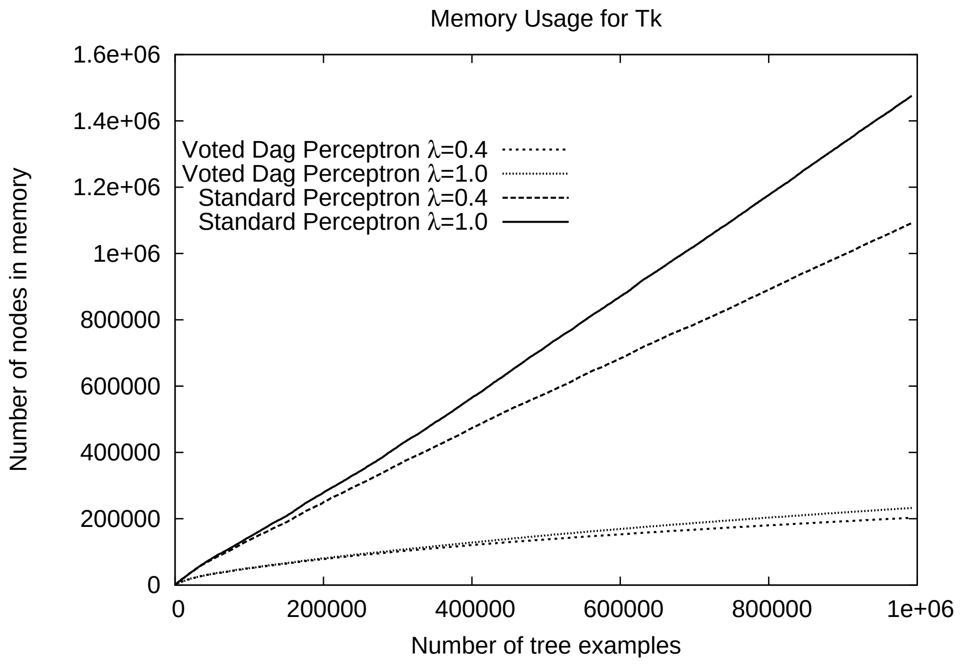

Figure 6.8: Evolution of the number of tree nodes stored in memory and belonging to the model developed by the Standard Perceptron and the Voted DAG Perceptron during training on the training set with 992,819 examples. Both methods use the SST tree kernel (Tk) with different values for λ, i.e. $\lambda \in \{0.3,\ 0.4,\ 0.5,\ 0.6,\ 0.7,\ 0.8,\ 0.9,\ 1.0\}$. For each method, only the executions with the largest and the lower number of stored nodes are reported.

relatively small with respect to the one used previously. The training times are very fast, less than 1 second, thus risking that delays due external factors, such as disk access, may significantly modify the final values obtained. We thus concatenated the training and test set and obtained one file with 9631 examples. The INEX 2005 task is multiclass, we transformed it into a two-class problem by considering as positive the examples of class 3, one of the most numerous, and as negative the examples of any other class. We compared the standard perceptron and the voted dag perceptron with respect to training times and memory usage. Since the dataset is composed of only structured data, only the tree kernel was tested. The lambda values used were $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. Figure 6.10 compares the execution time of the standard perceptron and the voted dag perceptron. For each algorithm only the faster and slower parameter settings are plotted. Even for this dataset the voted dag perceptron outperforms the standard perceptron: the latter takes from 7.05 to 7.60 seconds to converge, while the former requires only from 1.22 to 1.27 seconds. Finally Figure 6.11 compares the amount of memory used by the two algorithms. For the Standard Perceptron only the execution with lower number of nodes is reported. For the Voted Dag Perceptron only the execution with the largest number of nodes is reported. This is due to the fact that the difference between the curves related to the largest and lower number of nodes of the same algorithms are so small compared to the difference between curves related to different algorithms that the curves related to the same algorithm are indiscernible. The lower number of nodes for the standard perceptron is 50800, while the largest number of nodes composing the model for the voted dag perceptron is 1611. The number of nodes kept in memory by the standard perceptron is 31.5 times higher than the number of nodes kept in memory by the voted dag perceptron.

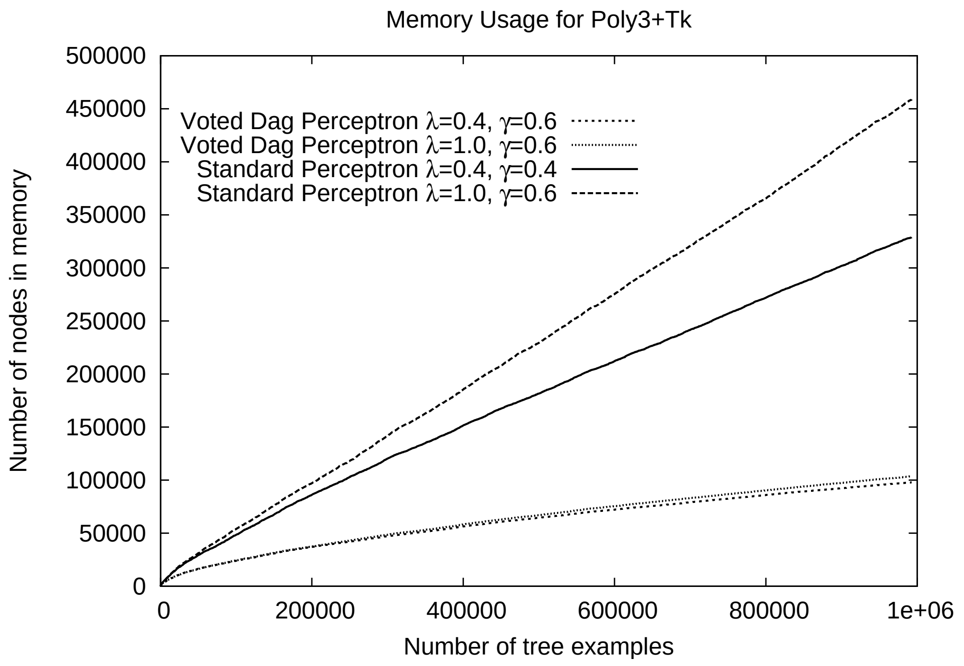

Figure 6.9: Evolution of the number of tree nodes stored in memory and belonging to the model developed by the Standard Perceptron and the Voted DAG Perceptron during training on the training set with 992,819 examples. Both methods use a linear combination of a polynomial kernel with degree 3 (Poly3) with the SST tree kernel (Tk), i.e. $(1-\gamma) * Poly3 + \gamma * Tk$. Different values for λ and γ have been considered, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and $\gamma \in \{0.2, 0.3, 0.4, 0.5, 0.6\}$. For each method, only the executions with the largest and the lower number of stored nodes are reported.

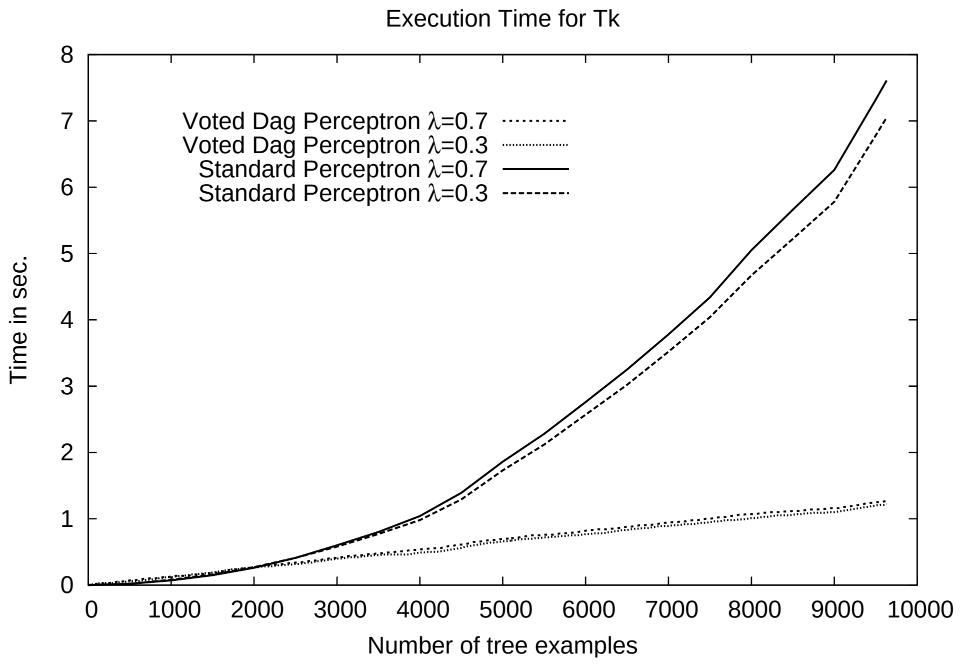

Figure 6.10: Execution time in seconds for the Standard Perceptron and the Voted DAG Perceptron using the SST tree kernel with different values for the λ parameter, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, over the union of the INEX 2005 training and test sets.

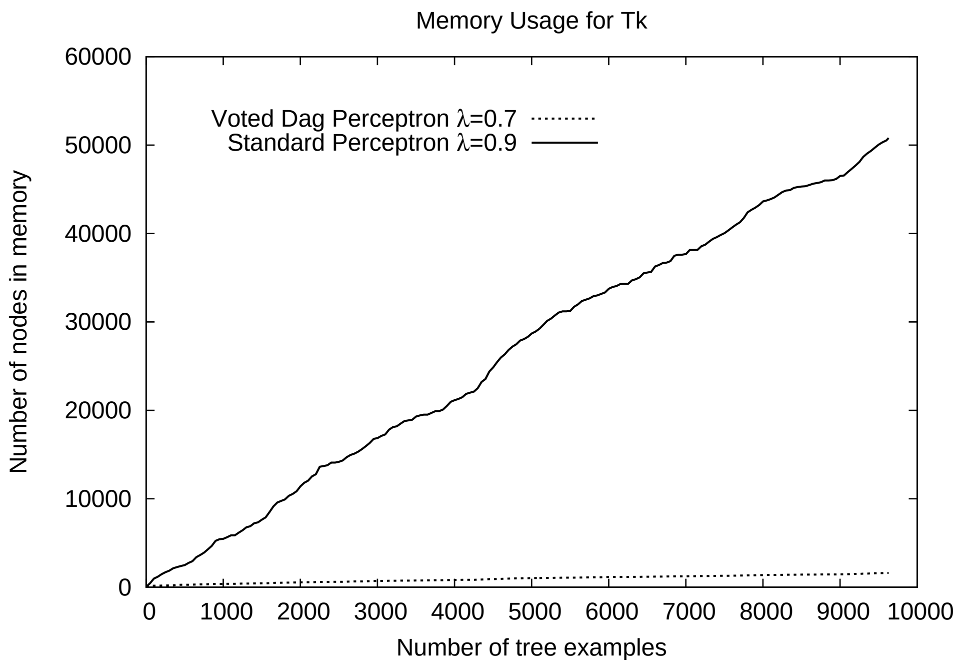

Figure 6.11: Evolution of the number of tree nodes stored in memory and belonging to the model developed by the Standard Perceptron and the Voted DAG Perceptron during training on the union of the INEX 2005 training and test sets. Both methods use the SST tree kernel (Tk) with different values for λ, i.e. $\lambda \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$. For the Standard Perceptron only the execution with lower number of nodes is reported. For the Voted Dag Perceptron only the execution with the largest number of nodes is reported.

Chapter 7

Conclusions

The aim of this thesis was to investigate ways to overcome some of the current drawbacks of kernel methods. Particularly we faced the following issues: designing expressive and non sparse kernel functions and alleviating the computational burden related to the computation of the score in both learning and classification phase. In the following we recall briefly the contributions for solving these problems and propose future developments.

In practical applications involving structured data, using a kernel method may not give an optimal performance because of the sparsity of the adopted kernel. This is particularly true for structured data involving discrete variables. An example of this event for subset and subtree kernels applied to XML documents represented as trees has been discussed. We have suggested that such sparsity can be reduced by learning a similarity function on the trees which can then be exploited to define non sparse kernels. Specifically, we have suggested to learn such a function by exploiting SOM-SD, which is an unsupervised dimensionality reduction method for structured data with the property that similar items in the input space tend to be represented similarly by the map. Then, we have defined a family of kernels for trees on top of the SOM-SD map, and according to the topological information coded into the map. The aim of this approach was to learn, in an unsupervised fashion, a kernel which is neither sparse nor uninformative. Experimental results on a relatively large corpus of XML documents, for which both subset and subtree kernels exhibit the sparsity problem, have shown that the new kernels are able to improve with respect to the performance of SOM-SD and the standard tree kernels. This improvement is quite independent from the map used to define the kernel, thus showing that the proposed approach is quite robust. Experimental results obtained on a similar dataset, for which, however, subset and subtree kernels do not exhibit the sparsity problem, show that there is not a significant improvement in performances. Thus it seems reasonable to state that the proposed approach is particularly suited in situations where standard tree kernels are sparse. A future development for this approach could be related to building more sophisticated kernels on the map activations. A first step could be to define a variant of the AM-kernel which weights the encoding of a substructure according to its similarity to the neuron it is mapped to. A second line for future research for the AM-kernel could be to employ different dimensionality reduction algorithms. The heuristic nature of the SOM-SD can not formally guarantee to preserve the topology of the items in the input space. The Generative probabilistic modelling proposed in [20] uses a more theoretically grounded approach to the problem of projecting data onto a lower dimensional space, and thus may be employed to optimise the dimensionality reduction step in such a way that the accuracy of the AM-kernel is improved.

A central issue when designing kernel functions is in defining expressive non sparse kernels. Considering the class of convolution tree kernels based on decomposing the input tree into its substructures, it is known that it is impossible to define more expressive kernels computable in polynomial time than those already presented in literature. We have observed, however, that all those kernels focus on the mere presence of the substructures and partially discard information about the position of the substructures in the original structure. We have therefore proposed a novel family of non sparse kernels which especially focus on this aspect. The results obtained show that indeed these kernels are very effective and motivate us to look for novel applications in real case scenarios such as problems in chemistry or bioinformatics. We furthermore plan to compare our kernel with more kernel for trees and investigate the relationship with them.

Kernel methods are effective approaches to the modelling of structured objects in learning algorithms. A drawback of such approaches is related to their typically high computational complexity for the computation of the score. To alleviate this problem when kernels such as the subtree and subset tree kernels are used, Direct Acyclic Graphs can be used to compactly represent shared substructures and feature vectors in different trees, thus reducing the computational burden and storage requirements. Results show that substantial computational savings can be obtained for the perceptron algorithm using tree and polynomial kernels over the PropBank dataset. The experiments on this very large dataset show that our model makes the use of kernels for trees practical for applications involving a very large amount of data. The basic idea of using Direct Acyclic Graphs for encoding a forest of trees can be exploited in all the learning algorithms where the decision function is computed as a linear combination of kernel evaluations. Moreover, we have shown that the same idea behind our contribution can be exploited for any convolution kernel, provided that a suitable and efficient way of encoding a set of substructure is defined. As future work we plan to enlarge the number of convolution kernels for which the computation of the score can be made efficient by compacting the model. A further line of research that can be pursued is to develop principled strategies for pruning the model in such a way that the effects on the subsequent score computations are minimized. By representing the model as a single structure, one can analyse and eliminate those substructures, belonging to different examples, that vanish each other contribution to the score computation. For some settings it could be possible to give bounds on the error introduced by the pruning operation.

Appendix A

Experimental Settings

The following appendices are devoted to the description and analysis of the datasets used in the experiments presented in chapters 4, 5 and 6.

A.1 INEX 2005

The INEX 2005 dataset comprises a relatively large set of XML formatted documents which were made available as part of the 2005 INEX Competition [15] (data can be downloaded from *http://xmlmining.lip6.fr*). The dataset is formed by XML documents describing movies from the IMDB site[1]. Specifically, we make use of the corpus (`m-db-s-0`), which consists of $9,640$ documents containing XML tags only, i.e. no further textual information available. All documents have one out of 11 possible target values. 3377 documents comprise the training set, while 1447 documents constitute the validation set. All remaining documents form the test set.

The dataset that has been used for many experiments in the thesis is a modified version of the corpus (`m-db-s-0`), which is described in [63]. As it will be discussed the corpus (`m-db-s-0`) consists of too large structures (with consequent increase in computational complexity) to allow an in-depth exploration of the properties of the algorithms proposed. The dataset produced by the preprocessing proposed in [63], which we are going to describe in the following, has been used to win the INEX 2005

[1]http://www.imdb.com

Competition. Thus, besides reducing the size of the structures, it gives us a strong benchmark to which compare our results.

A tree structure is extracted for each of the documents in the dataset by following the general XML structure within the documents. This resulted in a dataset consisting of 684191 vertices (subtrees) with maximum out-degree 6418. Managing such large structures would have posed computational complexity issues especially in the learning phase, thus practically limiting the number of experiments that could have been performed. For example in chapter 4 the dataset is used for training a SOM-SD map: not counting node label size, map prototypes of a two-dimensional map should be of size $6418 \cdot 2 = 12836$. Managing such large vectors would have dramatically delayed the training process. Thus, while not strictly necessary, a pre-processing step was performed on the dataset in order to reduce its dimensionality. First, repeated sequences of tags within the same level of a structure were collapsed. For example, the structure:

```
<BB>
    <a> </a>
    <b> </b>   is consolidated to   <BB>
    <a> </a>                            <a> </a>
    <b> </b>                            <b> </b>
    <a> </a>                        </B>
    <b> </b>
</BB>
```

A further dimension reduction has been achieved by collapsing simple sub-structures which have the property of a data sequence into a single vertex. For example, the sequential structure `<A><b><c></c></b></A>` can be collapsed to `<A><b&c></b&c></A>`, and even further to `<A&b&c>`. The pre-processing step reduced the maximum out-degree to 32, and the total number of vertices to $124,359$.

A preprocessing step was performed in order to reduce also the size of the node labels. The following steps are particularly suited for the SOM algorithms, but do not affect the behaviour of the kernel methods. A unique ID is associated with each of the possible 197 XML tags. In order to account for nodes which represent collapsed sequences, we attached a three dimensional data label to each node (the longest collapsed sequence is of length 3). The first element of the data label gives the ID of the XML tag it represents, the second element of the data label is the ID number of the first tag of a collapsed sequence of nodes, and consequently, the third element is the ID of the tag of the leaf node of a collapsed sequence. For nodes which do not represent a collapsed structure, the second and third element in the data label is set to zero. Note that by using a progressive number for encoding the labels we are imposing a metric on the labels: while all different labels should be equally dissimilar, it happens that different labels having close IDs turn out to be more similar than labels having far IDs. Note that avoiding to impose a metric would have required to define perpendicular vectors for each pair of different labels. This can be achieved by 197 sized vectors which, again, would have delayed the training process of the SOM-SD.

Summing up, the reason for applying this preprocessing of the data is threefold:

- it reduces the turn around time for the experiments, and hence, allows a more comprehensive exploration of the parameter space;
- it replicates the experimental setting of [63], which produces SOM-SD maps with state of the art performances on this task;
- the resulting dataset, as it will be shown in this section, is sparse, and thus it is the right candidate to support our claims about the Activation Mask Kernel (see chapter 4).

Some statistics about the frequency of the class of the examples have been collected. They are summarized in table A.1. The dataset is unbalanced. Class 4 has the lowest number of examples, 172, while class 8 has the highest, 769.

In order to a baseline, the SVM with Subtree and Subset tree kernels (see section 3.1.2) was applied to the dataset. The values of the parameter λ used are $0.1, 0.2, 0.3, 0.4$, $0.5, 0.6$, 0.7, 0.8, 0.9, 1.0. The parameter c of the SVM is selected

frequency	598	486	701	172	435	231	261	769	333	386	448
class	1	2	3	4	5	6	7	8	9	10	11

Table A.1: Number of examples for each class of the examples of the INEX 2005 training dataset.

on the validation set among the following values: $0.001, 0.01, 0.1, 1, 10, 100, 1000$. The obtained results, together with the values of the sparsity index (eq. (2.23)) for each kernel, all computed on the test set, are shown in Table A.2. The best accuracy on test set has been obtained by the SST kernel with an error rate of 11.21%. This result was obtained by setting λ to 1.1 and setting the c hyper-parameter of SVM to 10. Note that that both ST and SST kernels are sparse on the INEX 2005 dataset.

	Classification Error	Sparsity Index
ST Kernel	11.27%	0.5471
SST Kernel	11.21%	0.5471

Table A.2: Classification error and Sparsity Index of the Tree Kernels on the INEX 2005 dataset.

A.2 INEX 2006

The INEX 2006 dataset is derived from the IEEE corpus composed of 12000 scientific articles from IEEE journals in XML format. It includes XML formatted documents, each from one of 18 different journals, covering both transactional and non-transactional journals and across various topics in computer science. However, there are up to five journals that belong to the same structural (transactional or non-transactional) and semantic (topics) grouping, therefore distinct differences cannot be expected from documents of several journals. Furthermore, the journals are unbalanced in the number of documents they contain in the training dataset, therefore, this learning task is high in complexity, yet contains features that are commonly found in real world problems. The documents used in the training process is the training portion of the dataset, which consists of 6053 documents, and the number of XML tags in each document ranges widely, from 9 to 7024, with a total of 3966123 tags. To represent the structure of a document, a tree could be used where each node in the tree represents the occurrence and location of XML tags. This would result in large trees where the maximum depth is 19 and the maximum out-degree is 1023. Another observation of the INEX 2006 dataset is that there are a total of 165 unique tags, and some tags occur with a high frequency in a number of documents, but not all tags occur in all documents. The documents used in the testing process is the testing portion of the dataset, which consists of 6054 documents, and the proportion of documents in each journal is comparable to the training data, to ensure that the rules learned from training can be applied to the test data and that a similar level of performance can be expected. In the experiments presented in chapters 4 and 5, the dataset has been split into training, validation and test sets. Each set is made of 4237, 1816 and 6054 documents, respectively. Each document belongs to 1 out of 18 classes. Statistics about the number of documents belonging to each class are presented in table A.3. Class 5 has the lowest number of examples, 105, and class 3 the highest, 939. Overall, the dataset is quite unbalanced.

frequency	160	335	939	285	266	351	281	116	320
class	1	2	3	4	5	6	7	8	9
frequency	230	171	476	526	369	105	294	564	265
class	10	11	12	13	14	15	16	17	18

Table A.3: Number of examples for each class of the INEX 2006 training dataset.

The training data has a total of almost 4 million nodes and a maximum outdegree of 1023. The XML documents are represented by trees. Some preprocessing is applied in order to improve the turn around time for the experiments. Specifically only documents headers were extracted and then considered for the learning phase.

In order to a baseline, the SVM with Subtree and Subset tree kernels was applied to the dataset. The values of the parameter λ used are $0.1, 0.2, 0.3, 0.4, 0.5, 0.6$, 0.7, 0.8, 0.9, 1.0. The parameter c of the SVM is selected on the validation set among the following values: $0.001, 0.01, 0.1, 1, 10, 100, 1000$. The obtained results, together with the values of the sparsity index for each kernel, all computed on the test set, are shown in Table A.4. The best accuracy on test set has been obtained by the SST kernel with an error rate of 59.56%. This result was obtained by setting λ to 0.3 and setting the c hyper-parameter of SVM to 1. Note that that both ST and SST kernels are absolutely not sparse on the INEX 2006 dataset.

	Classification Error	Sparsity Index
ST Kernel	67.98%	0.002489
SST Kernel	59.56%	0.002489

Table A.4: Classification error and Sparsity Index of the Tree Kernels on the INEX 2006 dataset.

A.3 Penn Treebank II

The Penn Treebank II corpus [42] consists of material from the Dow-Jones news service. It is composed of about 1 million words tagged for part of speech and about 53,700 sentences annotated with predicative information.

Among others, the PropBank project [36] proposes predicate argument structures to encode shallow semantics from texts. The basic assumption is that such predicative structures are strictly connected to the syntax of the textual sentences. Figure A.1 exemplifies such idea by showing the parse tree of the sentence: `"Mary brought a cat to school"` along with the predicate argument annotation proposed by the PropBank project. Only verbs are considered as predicates whereas arguments are labeled sequentially from Arg0 to Arg5 plus ArgMs including several type of adjuncts.

Previous work has shown that the automatic PropBank argument annotation, i.e. Semantic Role Labeling (SRL), can be carried out by applying machine learning techniques, e.g. [21,53]. These latter represent predicate argument relationships with vectors of features extracted from the syntactic parse tree of the target sentence.

Such *standard* features, firstly proposed in [21], refer to flat information derived from parse trees, i.e. *Phrase Type*, *Predicate Word*, *Head Word*, *Governing Category*, *Position* and *Voice*.

For example, *Phrase Type* is the label of the *argument node*, i.e. the node that dominates all and only the argument words. In Figure A.1 the values of such feature are `N`, `NP` and `PP` for Arg0, Arg1 and ArgM, respectively. The *Parse Tree Path*, instead, represents the path in the parse-tree between a predicate node and one of its argument nodes. It is expressed as a sequence of nonterminal labels linked by direction symbols (up or down), e.g. `V↑VP↓NP` is the path between the predicate and Arg1.

An alternative representation proposed in [45], is based on the application of tree kernels to subtrees encoding the predicate/argument relation. More precisely, each predicate/argument pair is associated with the minimal subtree that includes the word sequences of them both, hereafter called PAF. For example, in Figure A.1, the substructures inside the three frames are the semantic/syntactic structures associated with the three arguments of the verb *to bring*, i.e. S_{Arg0}, S_{Arg1} and S_{ArgM}.

It is worth to note that PAF aims at capturing all the information between a predicate and one of its arguments. PAF is quite intuitive and, to conceive it, the designer requires much less linguistic knowledge about semantic roles than those necessary to manually define effective features. The main drawback of its use is that important structural information, i.e. inter-argument dependencies, is neglected.

To each tree in the dataset is also associated a vector of features extracted from the syntactic parse tree of the target sentence. Thus a single example $e_i \equiv (T_i, v_i, c_i)$ is constituted by a tree T_i , a vector of features v_i , and a class label c_i.

The large PropBank corpus makes the learning via tree kernels quite time consuming. We collected some statistics on the execution time of the Support Vector Machine in table A.5. Numbers refer to execution on an Intel Core 2 Duo E6400 2.13GHz based PC in three different scenarios[2]: *i)* for each example e_i only

[2]The following values for the hyperparameters have been used: *i)* $c = 1$; *ii)* $c = 1,\ \lambda = 0.4$; *iii)* $c = 0.7692$, $\lambda = 0.4,\ \gamma = 0.3$. These values have been selected by using the validation set.

the vector of features v_i is considered, i.e. the associated tree T_i is not used; an example is thus in the form (v_i, c_i); a polynomial kernel of degree 3 is used (row 1 in Table): $K_{Poly3}(v_i, v_j) = (v_i \cdot v_j + 1)^3$; *ii)* for each example e_i only the tree T_i is used and an example is in the form (T_i, c_i); the SST tree kernel is used (row 2 of Table): $K_{SST}(T_i, T_j)$; *iii)* both the input tree T_i and the associated vector of features v_i of an example e_i are considered; the used kernel is given by a linear combination of the two kernels mentioned above (row 3 in Table): $K(e_i, e_j) = (1 - \gamma) * K_{Poly3}(v_i, v_j) + \gamma * K_{SST}(T_i, T_j)$.

Kernel	Training time in seconds
Poly3	76,917
Tk	235,899
Poly3+Tk	390,914

Table A.5: Training times for SVM^{Light} in three different scenarios involving a polinomial kernel of degree 3 (Poly3), the SST tree kernel (Tk), and a linear combination of the previous kernels (Poly3+Tk). The training set involves 992,819 examples.

In the experiments presented in the chapter 6 the first 7 sections of PennTree bank for training were used. They consist of a total of 71,523 positive and 921,296 negative examples. Section 24 was used as validation set for a total of 7,705 positive and 108,104 negative examples, while section 23 was used as test set for a total of 13,159 positive and 171,114 negative examples.

We have also collected statistics about node distribution, maximum and average outdegree. Table A.6 reports statistics about the data derived from the boundary detection dataset.

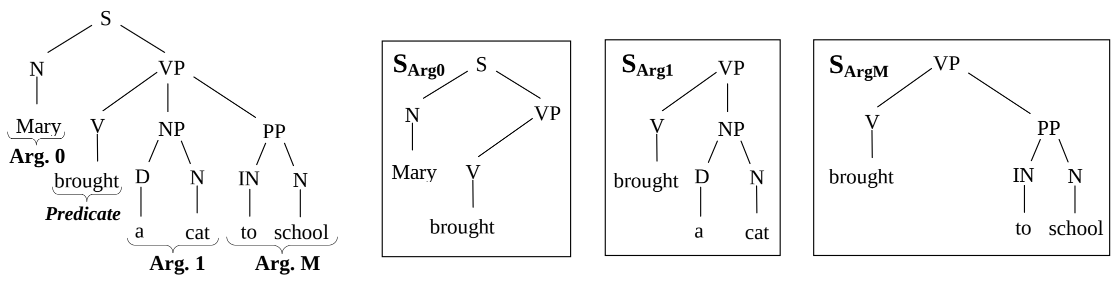

Figure A.1: Parse tree of the sentence "Mary brought a cat to school" along with the PAF trees for Arg0, Arg1 and ArgM.

	Training	Validation	Test
Number of trees	992,819	115,809	184,273
Total number of nodes	14,365,253	1,686,167	2,643,822
Average number nodes in a tree	14.47	14.56	14.35
Average maximum outdegree	2.32	2.33	2.3
Max outdegree	15	15	13

Table A.6: Features of syntactic trees in the boundary detection dataset.

A.4 LOGML

The LOGML dataset consists of user sessions of the Rensselaer Polytechnic Institute Computer Science Department website[3], collected over a period of three weeks. Each user session consists of a graph and contains the websites a user visited on the Computer Science domain. These graphs were transformed to trees by only enabling forward edges starting from the root node. The goal of the classification task is to discriminate between users who come from the edu domain and users from another domain, based upon the users browsing behavior. 3 datasets are available. They comprises 8074, 7409 and 7628 examples, respectively. The maximum outdegree of the trees is 137. The datasets are unbalanced: for each of them about 76% examples belongs to positive class. The datasets are very sparse with respect to the

[3]http://www.cs.rpi.edu

SST kernel: the mean of the 3 sparsity index values is 0.9595.

Because of the availability of the three datasets, it was natural to compute the classification error of the ST and the SST kernels by performing a 3-fold cross-validation considering, in each round, one of the dataset as the test set. Table A.7 shows the cross validation error and the mean sparsity index on the three datasets, for the ST and SST kernels. The best accuracy on test set has been obtained by the ST kernel with an error rate of 16.72%. This result was obtained by setting λ to 1.0 and setting the c hyper-parameter of SVM to 1. Note that that both ST and SST kernels are sparse on the LOGML dataset.

	Cross validation Error	Sparsity Index
ST Kernel	16.72%	0.9635
SST Kernel	16.84%	0.9595

Table A.7: Cross validation error and Sparsity Index of the Tree Kernels on the LOGML dataset.

References

[1] G. M. AdelsonVelskii and Y. M. Landis. An information organization algorithm. *Translation in NASA document n63-11777*, 1963.

[2] F. Aiolli, G. D. S. Martino, A. Sperduti, and M. Hagenbuchner. "kernelized" self organizing maps for structured data. In *ESANN 2007 Conference*, April 24-27 2007.

[3] F. Aiolli, G. D. S. Martino, A. Sperduti, and A. Moschitti. Fast on-line kernel learning for trees. In *ICDM*, volume 0, pages 787–791, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[4] F. Aiolli, G. D. S. Martino, A. Sperduti, and A. Moschitti. Efficient kernel-based learning for trees. In *CIDM*, pages 308–315, 2007.

[5] D. Anguita, S. Ridella, and F. Rivieccio. An Algorithm for Reducing the Number of Support Vectors. In *Proceeding of WIRN04*, 2004.

[6] R. E. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton Universsity Press, 1961.

[7] K. P. Bennett. Support vector machines: Hype or hallelujah. *SIGKDD Explorations*, 2:2000, 2000.

[8] A. M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Application of Cascade Correlation Networks for Structures to Chemistry. *Applied Intelligence*, 12(1):117–147, 2000.

[9] S. Bloehdorn and A. Moschitti. Structure and semantics for expressive text kernels. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 861–864, New York, NY, USA, 2007. ACM.

[10] V. S. Cherkassky and F. Mulier. *Learning from Data: Concepts, Theory, and Methods.* John Wiley & Sons, Inc. New York, NY, USA, 1998.

[11] M. Collins and N. Duffy. Convolution kernels for natural language. In *Advances in Neural Information Processing Systems 14*, pages 625–632. MIT Press, 2001.

[12] M. Collins and N. Duffy. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *ACL02*, 2002.

[13] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press, March 2000.

[14] N. Cristianini, J. Shawe-taylor, A. Elissee, and J. Kandola. On kernel-target alignment. In *Advances in Neural Information Processing Systems 14*, pages 367–373. MIT Press, 2002.

[15] L. Denoyer and P. Gallinari. Report on the xml mining track at inex 2005 and inex 2006: categorization and clustering of xml documents. *SIGIR Forum*, 41(1):79–90, 2007.

[16] M. Diligenti, P. Frasconi, and M. Gori. Hidden tree markov models for document image classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:2003, 2003.

[17] T. Downs, K. E. Gates, and A. Masters. Exact simplification of support vector solutions. *Journal of Machine Learning Research*, 2(293-297):85–87, 2001.

[18] Y. Freund and R. E. Schapire. Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296, 1999.

[19] T. Gartner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.

[20] N. Gianniotis and P. Tino. Visualisation of tree-structured data through generative probabilistic modelling. *IEEE Transactions on Neural Networks*, 2008.

[21] D. Gildea and D. Jurasfky. Automatic labeling of semantic roles. *Computational Linguistic*, 28(3):496–530, 2002.

[22] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[23] I. Guyon. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[24] M. Hagenbuchner, A. Sperduti, and A. C. Tsoi. Contextual processing of graphs using self-organizing maps. In *European symposium on Artificial Neural Networks*, 27 - 29 April 2005.

[25] M. Hagenbuchner, A. Sperduti, and A. C. Tsoi. Contextual self-organizing maps for structured domains. In *Workshop on Relational Machine Learning*, 2005.

[26] M. Hagenbuchner, A. Sperduti, and A. C. Tsoi. Self-organizing maps for cyclic and unbounded graphs. In *ESANN*, pages 203–208, 2008.

[27] B. Hammer, A. Micheli, M. Strickert, and A. Sperduti. A general framework for unsupervised processing of structured data. *Neurocomputing*, 57(5):33–35, 2004.

[28] D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, 1999.

[29] D. P. Helmond and M. K. Warmuth. On weak learning. *Journal of Computer and System Science*, pages 551–573, 1995.

[30] K. Ikeda. Effects of kernel function on ν-support vector machines in extreme cases. *IEEE Transactions on Neural Networks*, 17(1):1–9, 2006.

[31] T. S. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers. In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pages 487–493, Cambridge, MA, USA, 1999. MIT Press.

[32] T. Joachims. *Making large-scale support vector machine learning practical.* MIT Press, Cambridge, MA, USA, 1999.

[33] J. Kandola and J. Shawe-taylor. Refining kernels for regression and uneven classification problems. In *Proc. of the Ninth Int. Workshop on Artificial Intelligence and Statistics*, 2003.

[34] H. Kashima. *Machine Learning Approaches for Structured Data.* PhD thesis, Graduate School of Informatics, Kyoto University, Japan, 2007.

[35] H. Kashima and T. Koyanagi. Kernels for semi-structured data. In *ICML*, pages 291–298, 2002.

[36] P. Kingsbury and M. Palmer. From Treebank to PropBank. In *Proceedings of LREC'02*, Las Palmas, Spain, 2002.

[37] J. Kivinen, A. J. Smola, and R. C. Williamson. Online learning with kernels. *Signal Processing, IEEE Transactions on*, 52(8):2165–2176, 2004.

[38] T. Kuboyama, K. Hirata, H. Kashima, K. F. Aoki-Kinoshita, and H. Yasuda. A spectrum tree kernel. *Information and Media Technologies*, 2(1):292–299, 2007.

[39] T. Kuboyama, K. Shin, and H. Kashima. Flexible tree kernels based on counting the number of tree mappings. In *ECML/PKDD Workshop on Mining and Learning with Graphs*, 2006.

[40] T. Kudo and Y. Matsumoto. Fast methods for kernel-based text analysis. In *ACL*, pages 24–31, 2003.

[41] C. S. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for svm protein classification. In *Pacific Symposium on Biocomputing*, pages 566–575, 2002.

[42] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: the penn treebank. *Comput. Linguist.*, 19(2):313–330, 1993.

[43] S. Menchetti. *Learning Preference and Structured Data: Theory and Applications*. PhD thesis, Dipartimento di Sistemi e Informatica, DSI, Università di Firenze, Italy, December 2005.

[44] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[45] A. Moschitti. A study on convolution kernels for shallow semantic parsing. In *ACL '04: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 335, Morristown, NJ, USA, 2004. Association for Computational Linguistics.

[46] A. Moschitti. Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML*, pages 318–329, 2006.

[47] A. Moschitti. Making tree kernels practical for natural language learning. In *Proceedings of EACL'06*, Trento, Italy, 2006.

[48] K. R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181–201, 2001.

[49] D. Nguyen and T. B. Ho. A bottom-up method for simplifying support vector solutions. *IEEE Transactions on Neural Networks*, 17(3):792–796, 2006.

[50] L. Nicotra, A. Micheli, and A. Starita. Tree fisher kernel. In *Proceedings. 2004 IEEE International Joint Conference on Neural Networks*, pages 1917 – 1922, 2004.

[51] A. B. J. Novikoff. On convergence proofs on perceptrons. *Proceedings of the Symposium on the Mathematical Theory of Automata*, 12:615–622, 1962.

[52] N. Ohkura, K. Hirata, T. Kuboyama, and M. Harao. The -gram distance for ordered unlabeled trees. In *Discovery Science*, pages 189–202, 2005.

[53] S. Pradhan, K. Hacioglu, V. Krugler, W. Ward, J. H. Martin, and D. Jurafsky. Support vector learning for semantic argument classification. *Machine Learning Journal*, 2005.

[54] J. Ramon and T. Gärtner. Expressivity versus efficiency of graph kernels. pages 65–74. ECML/PKDD'03 workshop proceedings, September 2003.

[55] K. Rieck, U. Brefeld, and T. Krüger. Approximate kernels for trees. Technical report, Fraunhofer Publica [http://publica.fraunhofer.de/oai.har] (Germany), 2008.

[56] F. Rosemblatt. A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[57] B. Scholkopf, R. Herbrich, and A. J. Smola. A generalized representer theorem. *Proceedings of the Annual Conference on Computational Learning Theory*, pages 416–426, 2001.

[58] B. Schölkopf, A. J. Smola, R. C. Williamson, and P. L. Bartlett. New support vector algorithms. *Neural Computation*, 12(5):1207–1245, 2000.

[59] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

[60] K. Shin and T. Kuboyama. A generalization of haussler's convolution kernel: mapping kernel. In *ICML*, pages 944–951, 2008.

[61] J. Suzuki and H. Isozaki. Sequence and tree kernels with statistical feature mining. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 1321–1328. MIT Press, Cambridge, MA, 2006.

[62] M. E. Tipping. Sparse bayesian learning and the relevance vector machine. *The Journal of Machine Learning Research*, 1:211–244, 2001.

[63] F. Trentini, M. Hagenbuchner, A. Sperduti, F. Scarselli, and A. C. Tsoi. A self-organising map approach for clustering of xml documents. In *Proceedings of the WCCI*, Vancouver, Canada, July 2006. IEEE Press.

[64] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[65] V. N. Vapnik. *Statistical Learning Theory.* Wiley, New York, 1998.

[66] S. Vishwanathan and A. J. Smola. Fast kernels on strings and trees. In *Proceedings of Neural Information Processing Systems 2002*, 2002.

[67] G. Wahba. *Spline Models for Observational Data*, volume 59 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* SIAM, Philadelphia, 1990.

[68] D. Wettschereck, D. W. Aha, and T. Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11:273–314, 1997.

[69] M. Zhang, W. Che, A. Aw, C. L. Tan, G. Zhou, T. Liu, and S. Li. A grammar-driven convolution tree kernel for semantic role classification. In *ACL*, 2007.