Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXVI

Settore Concorsuale di afferenza: 01/B1
Settore Scientifico disciplinare: INF01

# Learning with Kernels on Graphs:
## DAG-based kernels, data streams and RNA function prediction.

Presentata da: Nicolò Navarin

Coordinatore Dottorato:
Maurizio Gabbrielli

Relatore:
Alessandro Sperduti

Esame finale anno 2014

# Abstract

In many application domains such as chemoinformatics, computer vision or natural language processing, data can be naturally represented as graphs. Machine learning allows for computers to learn a concept from a set of examples. When the application of analytical solutions for a given problem is computationally unfeasible or we do not know how to analytically solve a problem, machine learning techniques could be a viable way to solve the problem. Classical machine learning techniques are defined for data represented in a vectorial form. Recently some of them have been extended to deal directly with structured data. Among those techniques, kernel methods have shown promising results both from the computational complexity and the predictive performance point of view. Moreover, these methods offer strong theoretical guarantees on the quality of the solution. Kernel methods allow to avoid an explicit mapping in a vectorial form relying on kernel functions, which informally are functions calculating a similarity measure between two entities. However, the definition of good kernels for graphs is a challenging problem because of the difficulty to find a good tradeoff between computational complexity and expressiveness. Another problem we face is learning on data streams, where a potentially unbounded sequence of data is generated by some sources. We considered the case where the learning algorithms have to respect a bound on memory occupation.

There are three main contributions in this thesis.

The first contribution is the definition of a new family of kernels for graphs. The idea is to decompose a graph into a multiset of simpler structures, i.e. Directed

Acyclic Graphs (DAGs). Then the graph kernel is defined as a function of kernels for DAGs. We analyzed two kernels from this family, achieving state-of-the-art results from both the computational and the classification point of view on real-world datasets.

The second contribution consists in making the application of learning algorithms for streams of graphs feasible. Moreover, when memory constraints are present, the adopted budget management policy is critically influencing the overall algorithm performance. We defined a principled way for the management of the budget, based on *LossyCounting*, which is an algorithm originally designed for computing approximated frequency counts over event streams. Our proposal extends *LossyCounting* in order to approximate the solution of a learning algorithm.

The third contribution is the application of machine learning techniques for structured data to a bioinformatics problem, namely non-coding RNA function prediction. In this setting, the secondary structure is thought to carry relevant information. However, existing methods considering the secondary structure have prohibitively high computational complexity, thus limiting their application to very small datasets. Indeed, the tool that is considered the state-of-the-art considers only sequence information. We propose a new representation for RNA sequences. Moreover, we adapted the definition of existing graph kernels, and defined a new one, on such representation. The resulting kernels are able to consider the secondary structure and are fast enough to be applied to large datasets. Our proposed approach outperforms the state-of-the-art.

# Acknowledgements

Foremost, I would like to thank my advisor Prof. Alessandro Sperduti for his guidance and support. Besides my advisor, I would like to thank Dr. Giovanni Da San Martino who helped me at different stages of my PhD journey.
Next, I thank Dr. Fabrizio Costa for giving me the opportunity to visit the Bioinformatics group at the University of Freiburg.
Also, I need to thank my PhD colleagues that made this experience more pleasant and unique.
Thanks to all the people that supported me during these difficult years.
Last but not least, I would like to thank my family for allowing me to realize my own potential. My mother gave me the opportunity to follow university and then the doctoral program. My father helped me to get through the tough times.

# Contents

# Part I

# Introduction and basic concepts

# Chapter 1

# Introduction

> The first ultraintelligent machine is the last invention that man need ever make.
>
> I. J. Good, 1965

## 1.1 Why structured data?

The interest in structured data arises because in many application domains data can be naturally represented in a structured form. For example, XML documents are naturally represented as trees; in NLP each sentence may be represented with its parse tree; in computer vision each image can be represented by its segmentation graph (e.g. [9] or [106]), and in Chemoinformatics chemical compounds can be easily represented as graphs, where each atom is a vertex and the edges represent the bonds between atoms [1]. Figure 1.1 shows some examples of data arising from these applications.

Other application domains where data is naturally represented as graphs include computational biology, social networking, web link analysis or computer networks[1]. The topic of graph data processing is not new. Over the last thirty years there have been continuous efforts in developing new methods for processing graph data. Recently, because of the growing amount of structured data and the need to extract

information from it, there has been the interest, if not the necessity, to apply machine learning on graph data. Nowadays, because of technical advances such as graph kernels [85] and graph mining techniques [153], it is possible to apply these techniques in reasonable time on large datasets, obtaining good results.

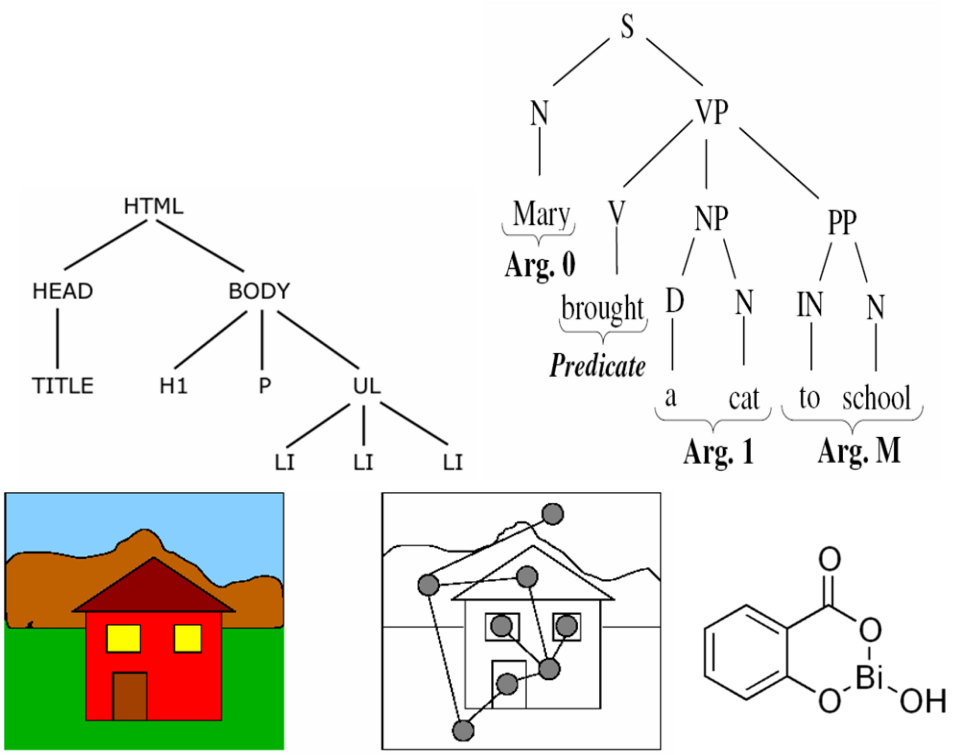


**Figure 1.1**: Examples of data that can be naturally represented in structured form. From left to right, top to bottom: an XML document, a parse tree from NLP, an image and the corresponding segmentation graph and a chemical compound.

## 1.2 Learning on structured data

The amount of data generated in different areas by computer systems is growing at an extraordinary pace, mainly due to the advent of technologies related to the web, ubiquitous services and embedded systems that aim at monitoring the environment in which they are immersed. According to a report from IDC titled "The Digital

Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the far East" [64], only 0.5% of the 643 exabytes of useful data have been analyzed. Examples of this data includes images, social media, sensors (including those in our smartphones and other devices, as well as those that may be implanted into the body), medical devices or biological data, e.g. data generated from DNA sequencers. In this context, to analyze the data means to extract useful information from it. This problem is commonly referred to as *Data mining*.

For some application domains where huge amounts of data have to be processed, an algorithmic solution may be computationally unfeasible. In other cases, it may be very difficult (or impossible) to state the problem itself in a non-ambiguous way, and thus also the development of an algorithmic solution is very difficult. Finally, it is possible that we know how to formalize the problem but we do not know how to solve it in an algorithmic way. This is the case when looking for the effectiveness of drugs against a certain disease. In these cases, it is convenient to adopt a machine learning approach.

*Machine learning* [100] is the branch of Artificial Intelligence that lets computers learn from experience. In particular, in our scenario to learn from experience means to learn a concept from examples. A typical learning task is to classify entities. In this scenario, we are given a set of labeled examples and the goal is to learn the concept underpinning the labeling function and to accurately predict the label for unseen examples.
While classical machine learning techniques have been defined for data represented in a vectorial form, more recently many techniques have been extended to deal directly with structured data [141, 46].

In this thesis, we deal with graph classification. There is a second task commonly referred as graph classification consisting in the prediction of the class labels for single nodes in a graph. This latter task is usually faced with different techniques, such as *label propagation* [29], that are not considered in this thesis.

## 1.3 Learning on graph streams

In some application domains, data is generated at a constant rate by sources that can potentially emit an unbounded sequence of data elements, i.e. data streams. Because of that, processing of data streams requires special care from a computational point of view, since only bounded time and memory resources can be used. However, it was early recognized that data streams tend to evolve with time, giving rise to the well-known concept drift phenomenon (see Section 2.7.1). Due to computational complexity issues, streams of structured data have not been much studied up to now. Only few works involving streams of trees have recently appeared [5] [72], while not much work has been done for streams of graphs.

This is a major drawback, since in many important application areas data can naturally be represented as streams in structured form. For example, modern medicine critically depends on the discovery of new drugs, i.e. chemical compounds. Chemical compounds can be naturally described via their molecular graphs. The discovery of new effective drugs relies on the systematic assessment of many different properties of all possible organic molecules, the so-called chemical space, which is estimated at $10^{20} - 10^{200}$ structures [137]. Only a very small portion of this space (few millions) constitutes the real chemical space, i.e. compounds that have been synthesized, while recent works have explored the possibility to explicitly enumerate a bit larger portion of this space (e.g. in [59] around 26.4 million compounds are covered). Having to deal with such numbers, it is clear that exploration of the chemical space can only be performed by adopting a stream-based approach, where the same compound can occur at different times associated with different but still related classification/prediction tasks. Another example of application involving a graph stream is malware detection. Malware detection consists in recognizing malicious software executed without user's awareness. The number of codes to be analyzed can be enormous and the problem is difficult since some malwares are able to modify their own code in order to avoid detection. In [56] the detection of malwares is performed by representing executables codes as graph nodes and control flow instructions and

API calls as edges. Concept drift can be present since malwares continuously try to exploit more and more sophisticated approaches to deceive the classifier. Also classification of images is a task which may involve streams of graphs. In fact, more and more images are available online. A graph can be built from an image with the aim of representing (spatial) relationships between the objects into the image (see Section 5.1.2 for an example). The task of finding images that a user likes is then an example of a stream of graphs with concept drift. Finally, in [7] a Fault Diagnosis System for Sensor Networks is proposed. The idea is to apply a cognitive algorithm to a stream of graphs representing spatial and temporal relationships among the sensors in order to distinguish between changes in the environment and sensor faults. In this context, concept drift occurs naturally as the environment changes. In the aforementioned paper, a simple thresholding system has been used, but the potential for improvements is large if more sophisticated algorithms are adopted.

Learning on graph streams is an important problem, and as such it will be one of the focuses of this thesis.

## 1.4 Kernel methods

Different approaches in the application of machine learning to structured data have been explored. The simplest one is to define a mapping from structured data to fixed-size vectors in order to straightforwardly apply classical machine learning algorithms [93, 90]. Those vectors need to encapsulate structural information about the graph. There are several ways to implement this mapping, some of them suited for specific applications. In general, the more information we keep in the vectorial representation, the more computationally demanding the mapping is. Other approaches are based on graph mining [115]. Basically those methods apply pattern matching techniques to graph data, with the need to solve the subgraph isomorphism problem that is an NP-complete problem.

With the application of kernel methods to graph data, a new promising approach emerged, that mixed the speed benefits of the former with the classification

performance of the latter. This approach allows to avoid the explicit mapping in a vectorial form and to define learning algorithms directly on the original structured data. For many tasks, *kernel methods* shows good results outperforming other methods [22, 60, 104, 34, 75, 37, 122, 105, 111, 85, 9, 15, 35, 82, 124, 53].

Moreover, kernel methods offer strong theoretical guarantees and a convenient hard separation between the learning algorithm and the kernel function that applies directly to the examples and maps them in an high dimensional Euclidean space. The kernel function is, informally, a function that represents the similarity between two objects.

The key for the successful application of kernel methods to graph data is the definition of kernel functions for graphs. Early works [68, 112] defined kernels for graphs with an acceptable expressiveness, i.e. kernels that represent a "meaningful" similarity measure between graphs. However these kernels were computationally demanding. More recent works defined efficient kernels, some of them with near-linear time complexity [78, 122], but there is a major drawback concerning these kernels. Indeed, it is difficult to find a good tradeoff between computational complexity and expressiveness because, generally speaking, the faster the kernel is, the lower its generalization performance will be.

Recently, the application of graph kernels is being extended to new domains where representing examples as graphs is not straightforward, but this approach allows to explicitly store the information needed from the task. For example, in bioinformatics and molecular biology a significant research topic is the discrimination and detection of functional RNA sequences. The peculiarity of this problem is that in the sequence there is a lot of hidden information. In particular, it is thought that the specific form of the secondary structure is an important feature for detecting RNA sequences (see Chapter 6). Recently proposed kernels for RNA sequences [116] try to extract this information from the sequence and incorporate it in the kernel calculation, but the resulting algorithm is not applicable to big datasets. In this thesis we will propose a novel approach for this problem.

## 1.5 Contributions

The contributions of this thesis can be grouped in three branches.

The first contribution is the definition of a new family of kernels for graphs. The idea is to define a new similarity measure based on the subtrees of a graph. We analyze two kernels from this family, achieving state-of-the-art results from both the computational and the classification point of view.

The second contribution consists in the definition of learning algorithms for streams of graphs. Namely, we extend a family of online learning algorithms proposed in the literature to structured data. It is worth to notice that the algorithms we present are applicable only in conjunction with kernels that allow for an explicit feature space representation, such as the ones defined in the first contribution.

The third contribution is the application of machine learning techniques for structured data to a specific problem from bioinformatics, namely RNA function prediction. The application of machine learning to this field is not straightforward because several problems have to be faced. Eventually, we explored some feasible solutions and obtained successful results outperforming the state-of-the-art.

## 1.6 Outline

This thesis is organized as follows. Part I provides a comprehensive review of the state-of-the-art in the field. We start introducing kernel methods in Chapter 2. In the same chapter, from Section 2.7 we present the state-of-the-art on online learning algorithms. Then in Chapter 3 we start talking about learning on structured data and specifically on graphs, giving a comprehensive review of the state-of-the-art in kernel functions for structured data.

Part II groups the original contributions of this thesis. In Chapter 4 we propose a novel family of graph kernels and we study extensively two members of this family. Moreover, we thoroughly discuss how to make the computation very efficient, and in Section 4.5 how to apply feature selection techniques to the final learned model.

Part of the work in this chapter has been published in [43] and [44].
These findings are the basics for another study presented in Chapter 5, where the goal is to develop fast online learning algorithms for streams of graphs. We start with an analysis of different possible formulations, proposing a modification of existing learning algorithms that can deal with the explicit feature space of some kernels. Moreover, in Section 5.2 we introduce a brand-new approach to model reduction based on the LossyCounting approach [95], particularly suited for online learning algorithms. This chapter is based on the work published in [45].

Finally, in Chapter 6 we discuss about a particularly interesting problem in bioinformatics, namely non-coding RNA function prediction, and we propose a novel solution based on kernels for graphs. The work in this chapter is a joint work with Fabrizio Costa.

# Chapter 2

# Learning with kernels

> It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.
>
> Charles Darwin

## 2.1 Machine Learning

Learning from experience is one central aspect in what humans call intelligence. Indeed, learning from experience is what enables humans to adapt to various situations, and one of the core aspects of intelligent behavior. Learning from experience is a process that everyone adopts every day, during the extraction of physical laws from experimental data or the use of experience for decision making.

Nowadays we don't have enough information about how humans learn from experience, so we don't know how to make a computer learn even nearly as well as people do, but many algorithms have been developed that exhibit useful types of learning in some specific tasks. This science that lets computers learn from experience is called *machine learning*[136].

There are various applications in which machine learning turns to be the most effective approach. A non-exhaustive list includes speech recognition, handwritten

character recognition, image recognition (and face recognition). The characteristic shared by these problems is that it is very difficult (or impossible) to state the problem itself in a non-ambiguous way, and thus looking for an algorithmic solution for these problems is very difficult.

Moreover, the amount of data collected day by day exceeds the human capability to extract the information hidden in it, so it becomes more and more important to automate the process of learning from data, even so in problems where it may exist an algorithmic solution, but it is too much computationally expensive. The problem of extracting knowledge from data is called *data mining*.

More formally we say that a computer program is able to learn if its performance improves with experience.

**Definition 2.1.** *[100] 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 $T$, as measured by $P$, improves with experience $E$ .*

A textbook example is a computer program that learns how to play chess, that might improve its performance measured as its ability to win, at the class of tasks involving playing chess games, through the experience obtained playing against itself. So in general we have to identify these three features in order to have a well-posed learning problem: the class of tasks, the measure of performance and the source of experience.

*Machine learning* (ML) [99] aims to design and develop algorithms that allow computers to evolve their behaviors based on empirical data, such as from sensor data or databases.

In general, learning involves acquiring general concepts based on some specific training examples. This is not surprising because humans continuously learn general concepts based on examples: think about the concepts of "bird" or "car". Each one can be viewed as a boolean function defined over a larger set, e.g. a function defined over all animals whose value is true for birds and false for all other animals. There may be some examples that are border line, for which the membership to a class

is not trivial. This kind of examples will be the most important ones in kernel methods, as we will see later.

This decision function is what machine learning methods try to approximate.

We can make a taxonomy of different learning problems depending on the nature of the examples and the type of concept we want to learn. A first distinction is between supervised and unsupervised learning. In the former we have a label associated with each instance, while in the latter we don't. In this thesis we will focus on the supervised learning scenario, but some of the techniques can be applied as-is to the unsupervised one. For details about unsupervised learning, refer to [100]. In supervised learning, we are given a set of tuples called *training set* of the form $S = \{(x_i, y_i) : i = 1, \ldots, n\}$, where $x_i \in \mathcal{X}$ is the i-th available instance and $y_i \in Y$ is the associated label. The tuples $(x_i, y_i)$ are called the examples.

The examples in $S$ are assumed to be generated according to some unknown probability distribution $P$. Depending on the domain of $Y$ we can further define various types of classification problems:

- If $y_i \in \{\pm 1\}$ we have a binary classification problem.
- If $y_i \in \{0, \ldots, n\}$ we have a multi-class single-label classification problem.
- If $y_i \in \{\pm 1\}^m$ we have a binary multi-label classification problem.
- If $y_i \in \mathbb{R}$ we have a regression problem, that is the problem of approximating a real-valued target function.

Binary classification is the better understood and studied task. Since many classification methods have been developed specifically for binary classification, multi-class classification often requires the combined use of multiple binary classifiers. In this thesis we deal with both types of classification problems.

The goal of supervised learning is to find the function $c : \mathcal{X} \to Y$ that best represents the relationship between $x_i$s and $y_i$s. Since the only information we know about this function are the examples in $S$, the best we can do is try to approximate this function as tightly as possible. The function we want to learn is called the

(optimal) hypothesis $h$, that comes from an hypothesis space $H$ that is fixed a-priori (and it *must* be fixed in order to perform learning). An hypothesis is often referred to as a model, that is an abstraction that tries to explain the reality (in our case tries to explain the evaluation of the unknown $c$ function).

The best from all possible hypothesis $h$, that we will refer to as $h^*$, is the one that minimizes the *risk*:

$$R(h) = \int_{\mathcal{X} \times Y} L(h(x), y) dP(x, y)$$

where L is a *loss* function that measures the classification error of $h$, $\mathcal{X}$ is the space of all possible instances and $Y$ the space of the labels.

It's not possible to directly use this formulation for the selection of the best $h$ because the probability distribution $P(x, y)$ is unknown. An alternative approach consists in minimizing the error with respect to the data we have, thus defining the *empirical risk*:

$$R_e(h) = \frac{1}{n} \sum_{(x,y) \in S} L(h(x), y) \tag{2.1}$$

The techniques that select the best model (according to specific criteria) from the set of all possible models (from $H$) are referred to as model selection techniques.

In the following we will use the concept of Vapnik-Chervonenkis dimension (*VC-dimension*) [139] of an hypothesis space $H$. The VC-dimension is a measure of the capacity of an hypothesis space $H$, defined as the cardinality of the largest set of points that $H$ can shatter. To shatter a set of points means that, for all possible assignments of labels to those points, there exists a $h \in H$ such that $h$ makes no errors when evaluating that set of data points (that is $R_e(h) = 0$).

It is worth to distinguish between two different supervised machine-learning frameworks. In *batch* learning, we assume that there exists a probability distribution over the examples, and that we have access to a training set drawn i.i.d. from this distribution. A batch learning algorithm uses the training set to generate a single output hypothesis. We expect a batch learning algorithm to generalize, in the sense that its output hypothesis should accurately predict the labels of previously unseen examples, which are sampled from the same distribution.

On the other hand in the *online* learning framework, we typically make no statistical assumptions regarding the origin of the data. An online learning algorithm receives a sequence of examples and processes these examples one-by-one. On each online-learning round, the algorithm receives an instance and predicts its label using an internal hypothesis, which is kept in memory. Then, the algorithm receives the correct label corresponding to the instance, and uses the new instance-label pair to update and improve its internal hypothesis. The concept of generalization is much more difficult to define for online learning algorithms with respect to the batch ones [27]. Indeed, as we will see in Section 2.7, in online learning the underlying concept may change, meaning that the probability distribution over the examples is continuously changing.

The sequence of internal hypotheses constructed by the online algorithm from round to round is referred as the online hypothesis sequence. Typically, online learning have stronger requirements in terms of computational complexity and memory occupation then the batch framework.

## 2.2 Kernel methods

In many business and scientific applications the use of machine learning methods helped to speed up and reduce the cost of certain processes.

In these research fields, recently a class of learning algorithms has received much attention because of the solid foundation in learning theory and the empirical results that outperforms any other learning method in many benchmarks as well as real-world applications. These are *kernel methods*, whose most popular example is the Support Vector Machine [138], that will be explained in detail in Section 2.4.2.

Kernel methods are all the learning algorithms that can represent the solution in terms of the input examples. As mentioned in Section 2.2, kernel methods comprehends all those algorithms that do not work on an explicit representation of the examples, but need only some information about their pairwise similarity. This function for computing similarity has to be a kernel function (see Section 2.3).

Every kernel method can be decomposed in two components:

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

The following sections are organized as follows. We start presenting the most important concepts and algorithms belonging to the kernel methods family in Sections 2.3 and 2.4. Section 2.5 discusses the drawbacks of the kernel approach. In Section 2.6 we briefly present state-of-the-art algorithms for SVM training in the batch scenario.

Then we will move to online learning (Section 2.7), stochastic gradient descent (Section 2.7.3) and the related algorithms. Finally in Section 2.7.4 we present the Budgeted online learning algorithms.

## 2.3 Kernel functions

In this section we formally define what a kernel function is following the notation in [73], and we will show some examples of kernel functions defined on vectors.
Given a set $X$ and a function $K : X \times X \to \mathbb{R}$, we say that $K$ is a *kernel* on $X \times X$ if $K$ is:

- symmetric, i.e. if for any $x$ and $y \in X$ $K(x, y) = K(y, x)$ and
- positive-semidefinite, i.e. if for any $N \geq 1$ and any $x_1, \ldots, x_N \in X$, the matrix $K$ defined as $K_{i,j} = K(x_i, x_j)$ is positive-semidefinite, that is $\sum_{i,j} c_i c_j K_{i,j} \geq 0$ for all $c_1, \ldots, c_N \in \mathbb{R}$ or equivalently if all its eigenvalues are non-negative.

It is easy to see that if each $x \in X$ can be represented as $\phi(x) = \{\phi_n(x)\}_{n \geq 1}$ such that the value returned by $K$ is the ordinary dot product $K(x, y) = \langle \phi(x), \phi(y) \rangle = \sum_n \phi_n(x)\phi_n(y)$ then $K$ is a kernel. If $X$ is a countable set, the converse is also always true , that is a given kernel $K$ can be represented as $K(x, y) = \langle \phi(x), \phi(y) \rangle$ for some choice of $\phi$. The vector space induced by $\phi$ is called the *feature space*. Note that it follows from the definition of positive-semidefiniteness that the zero extension of a

kernel is a valid kernel, that is, if $S \subseteq X$ and $K$ is a kernel on $S \times S$ then $K$ may be extended to be a kernel on $X \times X$ by defining $K(x, y) = 0$ if $x$ or $y$ are not in $S$. It is easy to show that kernels are closed under summation, i.e. a sum of kernels is a valid kernel.

## 2.4 Kernel machines

Defined what a kernel function is, we can describe how kernel methods work in more detail. Kernel methods search for linear relations in the *feature space*. In these methods, the learning algorithm is formulated as an optimization problem that, if the adopted function is a kernel and thus symmetric positive semidefinite, is convex and has a global minimum.

Let us consider for sake of simplicity a binary classification problem (see Section 2.1). Let $X$ be a set of examples, and suppose that these examples are not linearly separable, that is there does not exist an hyperplane in the input space that can correctly separate positive and negative examples.

What happens with kernel methods is that examples are nonlinearly projected into a high-dimensional space (defined by the $\phi$ function associated to the kernel function), where they are supposed to be more sparsely distributed, that is the distance among examples, and thus the distance between positive and negative examples, is larger. In this space, examples are more likely to be linearly separable, so we can search and hopefully find a linear separator. This separator, if back-projected into the input space, corresponds to a nonlinear separator between the two classes.

The *representer theorem* states that the solution of certain optimization problems that involves empirical risk and a quadratic regularizer, can be expressed as a combination of the examples in the training set [144].

In particular, kernel methods are defined as convex optimization problems on some feature space. If the vectors $\phi$ appear only inside dot products, they can be calculated by the corresponding kernel function.

As a consequence, the optimization problem and its solution are defined over the input space, and the algorithm works only implicitly in the feature space via the kernel function. This technique is referred as the kernel trick.

Figure 2.1 shows an example of the application of the kernel trick for the classification of points in a 2-dimensional space.

We will see in the following sections some examples of kernel methods.

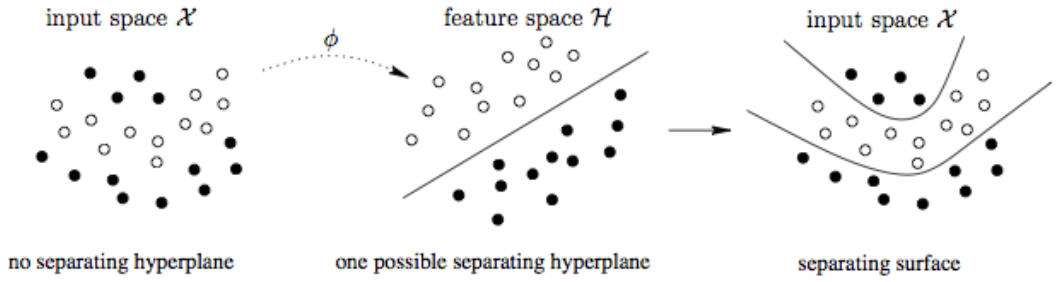


**Figure 2.1**: Example of classification using the kernel tick.

### 2.4.1 The Perceptron algorithm

One of the oldest algorithms used in machine learning (from early 60s) is an online algorithm for learning a linear decision function (an hyperplane) on real-valued vectorial data, called the Perceptron Algorithm [14].

A prototype vector $w$ is (randomly) initialized, and used as the decision function. An example $x_i$ is predicted as positive iff $w \cdot x_i + b > 0$ and as negative otherwise. The prediction is then compared with the real class of the example. On a mistake, a new prototype vector $w'$ is generated as $w' = w + \alpha y_i x_i$, where $0 < \alpha \leq 1$ is a constant that influences the learning rate, and $y_i \in \{+1, -1\}$ is the class of the example. This is a very simple algorithm, that guarantees to find a linear separator between positive and negative examples, if it exists.

The original algorithm has been extended in order to be applied with kernel functions, where the examples $x_i$ are substituted with $\phi(x_i)$ in the formulation.

This new formulation lies on the observation that $w$, after have seen a number $t$ of examples, is no more than a weighted sum of the examples where we have made mistakes so far, that is:

$$w_t = \alpha y_{i_1} x_{i_1} + \ldots + \alpha y_{i_{t-1}} x_{i_{t+1}}$$

where $x_{i_1}, \ldots, x_{i_{t-1}}$ is the set of misclassified examples.

So to compute $\phi(w) \cdot \phi(x)$ we can just do:

$$\phi(w) \cdot \phi(x) = \alpha y_{i_1} K(x_{i_1}, x) + \alpha y_{i_{t-1}} K(x_{i_{t+1}}, x)$$

where $\phi$ is the function that maps examples in the feature space and $K$ the corresponding kernel function.

The examples used to define the hypothesis (i.e. the misclassified examples) are referred to as *support vectors*. Perceptron finds *one* separator for the examples in $\mathcal{X}$. The perceptron is a fast and simple algorithm that can be applied to online learning tasks. Its main drawback is that it does not provide bounds on generalization error.

### 2.4.2 The Support Vector Machine

One of the most important algorithms in kernel methods is the Support Vector Machine(SVM) [138].

SVM is based on the principle of *structural risk minimization*[139] that is an inductive principle for model selection used for learning from finite training data sets. It describes a general model of capacity control and provides a tradeoff between hypothesis space complexity (the VC dimension of approximating functions, see Section 2.4) and the quality of fitting the training data (empirical error, defined in Equation 2.1). The procedure is outlined below.

Given a class of hypotheses (in this case being the hyperplanes in the feature space defined from the adopted kernel function), divide it into a hierarchy of nested subsets in order of increasing complexity. Perform empirical risk minimization on each subset (this is essentially parameter selection). Select the model in the series

whose sum of empirical risk and VC confidence (i.e. a value that directly depends on VC dimension) is minimal.

Let us now focus on how SVM works. In a first phase, the examples are projected in a feature space; then we search for a hyperplane that separate positive and negative examples maximizing the *margin*, that is the minimum distance between the hyperplane and the nearest example. We want to maximize the margin because VC dimension of a linear classifier can be expressed as a function of the margin. If the training set is linearly separable in the feature space, then the hyperplane that maximizes the margin, referred as optimum hyperplane, is unique and correspond to the solution of the following problem:

$$arg\min_{w,b} \frac{||w||^2}{2} \text{s.t. } \forall (x_i, y_i) \in S. y_i(w \cdot \phi(x_i)) + b \geq 1 \tag{2.2}$$

where $w$ and $b$ define the hyperplane in the *feature space*. We will refer to this formulation as *primal*, as it is expressed in the feature space. The margin is inversely proportional to the norm of $w$, so minimizing $||w||^2$ corresponds to selecting the simpler hypothesis from the ones that satisfy the constraints. The *representer theorem* states that the solution $f$ of the problem 2.2 can be reformulated as:

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

The examples for which $\alpha \neq 0$ are called support vectors. We will refer to this formulation as *dual* since it is expressed in terms of the corresponding dual optimization problem.

In many cases we may not want to classify correctly all training examples to avoid the so-called overfitting problem, maybe for the presence of noise in the data, maybe for the high complexity of the hypothesis in the case training set is not linearly separable. More precisely, overfitting takes place when the case of the selected hypothesis has poor generalization capabilities, that is it has poor classification performance on unseen data. Intuitively, it is as if the classifier has learned by heart the training examples, so it is not able to classify correctly the new data.

It is possible to define a tradeoff between the mistakes on the training set and the complexity of the hypothesis:

$$arg \min_{w,b,\epsilon} \frac{||w||^2}{2} + C \sum_{i=1}^{n} \epsilon_i$$

$$\text{s.t. } \forall (x_i, y_i) \in S. y_i(w \cdot \phi(x_i)) + b \geq 1 - \epsilon_i,$$

$$\epsilon_i \geq 0, i = 1, \ldots, n.$$

The $C$ parameter of the SVM influences this tradeoff. The optimal value of $C$ depends on the problem.

The dual version of the soft-margin support vector machine can be expressed as follows:

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

subject to:

$$0 \leq \alpha_i \leq C, \quad \text{for } i = 1, 2, \ldots, n, \sum_{i=1}^{n} y_i \alpha_i = 0$$

where the variables $\alpha_i$ are Lagrange multipliers. This is the standard formulation for the SVM.

## 2.5 The curse of dimensionality and dimension reduction

When dealing with high-dimensional data, various phenomena that do not occur in low-dimensional settings arise. This problem is known as the curse of dimensionality [58].

The curse of dimensionality arises from the fact that when the dimensionality of the space where learning is performed increases, the volume of this space increases so fast that the available data becomes sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data you need to support the result often grows

exponentially with the dimensionality. Moreover, organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data however all objects appear to be sparse and dissimilar in many ways which prevents common data organization strategies from being efficient.

This problem is also faced by kernel methods because of the implicit mapping in the high dimensional feature space. To address this problem in machine learning, techniques for *dimensionality reduction* are used, that are techniques for reducing the number of variables under consideration, i.e. for reducing the dimensionality of the space where learning is performed.

Feature selection is an example of a dimensionality reduction technique, which tries to find an informative subset of the original variables using statistical measures.

Feature selection has been successfully applied to various machine learning algorithms. The advantages of using these techniques are the reduction of the noise in input data discarding some useless or not-correlated information (for the task), making learning from the data simpler.

In kernel methods, dimensionality reduction consists in reducing the dimensionality of the feature space. As stated in Section 2.2 usually kernel methods perform the mapping in the feature space only implicitly, so the application of dimensionality reduction techniques in the feature space is not trivial. The study of how to extend the application of these techniques to kernel methods is an interesting research line, because various existing kernel functions may benefit from it. In Section 4.5.1 we will present feature selection techniques applicable to the problem of learning from graph data in more detail.

## 2.6 SVM training

### 2.6.1 Gradient Descent

Gradient descent (GD) is a mathematical technique for finding the minimum of a function. This method uses the fact that the gradient $\nabla f$ of a function points in

the direction of greatest increase. This means that $-\nabla f$ points in the direction of greatest decrease. GD can be applied to solve the SVM optimization problem stated in Section 2.4.2. For example, if we consider the primal SVM formulation, a natural iterative algorithm to find the minimum of equation 2.2, referred for clarity as $f_{SVM}$, is to update an estimate $w_t$ using $w_{t+1} = w_t - \alpha \nabla f_{SVM}(w_t)$, where $0 < \alpha < 1$ is a learning rate (necessary for convergence) and $\nabla f_{SVM}(w_t)$ is the gradient of $f_{SVM}(w_t)$ computed on the whole training set.

### 2.6.2 The SMO algorithm

Sequential minimal optimization is an algorithm for efficiently solving the large optimization problem which arises during the training of support vector machines, invented by Microsoft research [110]. SMO breaks the main problem into a set of the smallest possible quadratic programming problems, which are then analytically solved. The algorithm proceeds as follows:

1. Find a Lagrange multiplier $\alpha_1$ that violates the Karush–Kuhn–Tucker (KKT) conditions for the optimization problem.

2. Pick a second multiplier $\alpha_2$ and optimize the pair $(\alpha_1, \alpha_2)$.

3. Repeat steps 1 and 2 until convergence.

When all the Lagrange multipliers satisfy the KKT conditions (within a user-defined tolerance), the problem has been solved. Although this algorithm is guaranteed to converge, heuristics are used to choose the pair of multipliers to accelerate the rate of convergence. The complexity of this algorithm is quadratic in the dataset size, while previous methods were cubic.

## 2.7 Online learning algorithms

When the amount of data to process becomes very large, kernel methods have a bottleneck related to the computational complexity of the learning algorithms. The

problem of processing huge amounts of data is receiving increasing attention. According to a report from IDC titled "The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the far East" [64], only 0.5% of the 643 exabytes of useful data have been analyzed. Examples of this data include images, social media, sensors (including those in our smartphones and other devices, as well as those that may be implanted into the body) or medical devices. Indeed, the most popular learning algorithm, the SVM, scales quadratically in the dataset size. This complexity may be impractical when dealing with datasets of several thousands of examples. The simplest way to handle large datasets is to randomly discard data, but there are statistical benefits to process more data [19].

Recent research focused on large-scale learning, e.g. [154, 28], but the majority of the methods focus on linear SVMs not allowing for the application of the kernelized formulation. Other researches slightly modify the optimization problem to make the parallelization possible [155]. Typically, the complexity of these methods remains quadratic.

A promising way to deal with large-scale datasets that has been recently proposed is to use online learning algorithms, that are learning algorithms that process one example at a time. Indeed, it is possible to apply online learning algorithms on big batch datasets, allowing for multiple (sequential) passes over the data. Typically, each complete pass over the data is referred as an *epoch*.

This approach is very flexible because it allows an user to adjust the tradeoff between computational time and the quality of the solution.

Moreover, even if online algorithms are in general simpler with respect to batch ones, for most of them the solution (given that some properties are respected) converges to the global optimum. These algorithms belong to the general family of Stochastic Gradient Descent algorithms[120], that will be presented in Section 2.7.3. Unlike other approaches, in the non-kernelized version SGD has constant update time and constant space occupation.

Algorithms belonging to the SGD family can be easily kernelized to be applied to nonlinear classification problems. However, it becomes necessary to store the set of

support vectors and in general the number of SVs grows linearly with the size of the dataset. In the large-scale scenario we deal with huge datasets, thus this approach leads to the problem of exceeding the available memory and to the linear growth of the training and test times [5].

In order to face these issues, in [146] it is proposed a budgeted version of SGD, referred as BSGD (see Section 2.7.4).

### 2.7.1 Stream data mining

Data streams are becoming more and more frequent in many application domains thanks to the advent of new technologies, mainly related to web and ubiquitous services [11, 62]. In a data stream, data elements are generated at a rapid rate and with no predetermined bound on their number. For this reason, processing should be performed very quickly (typically in linear time) and using bounded memory resources. Data streams can be viewed as ordered infinite sequences of instances, flowing at variable rates and typically produced by a non-stationary source. They can be generated for example by sensor networks, user clicks, web server connections or emails.

Moreover, every real-world classification system may take advantage of continuously adapting the model with a feedback from the new data.

Unfortunately, conventional knowledge discovery tools cannot manage this overwhelming volume of streaming data. The study of the issues related to the new nature of incoming data represents the starting point to understand the main factors influencing the data streams problem.

Batch machine learning techniques typically require data to be entirely stored in memory, and to work with multiple passes on this data. In data streams scenario, the huge amount of data cannot be stored in memory or on disk. Thus, it's crucial to design mining algorithms using efficient techniques to bound the time and space necessary to extract a model.

Moreover, in streaming contexts a phenomenon called *concept drift* frequently occurs, and makes mining data streams even more complex. Indeed, in the data

streams world the underlying concept that we are trying to learn is not stable, but constantly evolving over time [86]. For example, if we want to predict weekly merchandise sales in an online shop we can develop a predictive model that works satisfactorily. The problem is that the behavior of the customers may change over time. The model may use inputs such as the amount of money spent on advertising, promotions being run, and other metrics that may affect sales. The model is likely to become less and less accurate over time - this is concept drift. In the merchandise sales application, one reason for concept drift may be seasonality, which means that shopping behavior changes seasonally. Perhaps there will be higher sales in the winter holiday season than during the summer, for example. This problem in the classification context requires special techniques.

The main challenge of stream data mining is to accurately capture the continuous changing decision concepts and scale up to large volume of stream data. The nature of data streams requires the use of algorithms that involve at most one pass over the data and try to keep track of time-evolving features (concept drift). Summarizing, the characteristics required from online learning algorithms are:

- example processing in linear time;
- no need for storing examples after they have been processed (single pass over data);

### Incremental (online) Learning

Incremental learning has recently attracted growing attention from both academia and industry. From the computational intelligence point of view, there are at least two main reasons why incremental learning is important. First, from data mining perspective, many of today's data-intensive computing applications require learning algorithms to be capable of incremental learning from large-scale dynamic stream data, and to build up the knowledge base over time to benefit future learning and decision-making processes. Second, from the machine intelligence perspective, biological intelligent systems are able to learn information incrementally throughout

their lifetimes, accumulate experience, develop spatial-temporal associations, and coordinate sensor-motor pathways to accomplish goals. We mainly focus on the first issue. Historically, inductive machine learning has focused on non-incremental learning tasks, i.e., where the training set can be constructed a priori and learning stops once this set has been duly processed, because it is the simplest learning scenario. There are, however, a number of areas, such as agents, where learning tasks are incremental. In recent years, the attention has been posed on this type of learning (see e.g. [71]) and some work has been done for defining incremental learning systems, for example in [74] is proposed a general incremental learning framework that is able to learn from stream data for classification purpose. The traditional online learning problem can be summarized as follows. Suppose a, possibly infinite, data stream in the form of pairs $(x_1, y_1), \ldots, (x_t, y_t), \ldots$, is given. Here $x_t \in \mathbb{X}$ is the input example and $y_t = \{-1, +1\}$ its classification. Notice that, in the traditional online learning scenario, the label $y_t$ is available to the learning algorithm after the class of $x_t$ has been predicted. The goal is to find a function $h : \mathbb{X} \to \{-1, +1\}$ which minimizes the error, measured with respect to a given loss function, on the stream. There are additional constraints on the learning algorithm about its speed: it must be able to process the data at least at the rate it gets available to the learning algorithm. Moreover, the amount of memory available to represent $h()$ is limited by memory constraints or by the user.

### Concept drift

When moving from the classical batch learning scenario to the online setting, the learning problem becomes more challenging. When dealing with large amounts of data produced from data streams, the problem of extracting knowledge from the data becomes more difficult because the data distribution and the underlying concept may be subject to continuous changes. An effective stream data mining algorithm should therefore be capable of dealing with such changing concepts and producing accurate models. In practice, this issue can be addressed by detecting changes in the data streams and continuously updating the prediction models according to the most

recently arrived data.

We can characterize concept drifting in two macro classes:

- loose concept drifting;
- rigorous concept drifting;

the main difference between them being the speed in which changes in concepts happen. In the former the concept slowly changes over the time, while in the latter at a certain time $t$ the concept changes.

### 2.7.2 Formalization and Feasible approaches

The most commonly used formalization for stream data mining with concept drifting is presented in [88] and models a stream as an infinite sequence of elements $e_1, \ldots, e_j, \ldots$. We can divide a data stream into batches $b_1, b_2, \ldots, b_n, \ldots$ where $b_i = e_{i_1}, \ldots, e_{i_{n_i}}$. For each batch $b_i$, we assume data as identically distributed with regard to a distribution $P_i(e)$. Depending on the amount and type of concept drift, $P_i(e)$ will differ from $P_{i+1}(e)$.

#### Data accumulation policy

As summarized in [74], the family of methods that adopts a data accumulation policy, simply develops a new hypothesis whenever a chunk of data is received. Thus the hypothesis $h_t$ is based on all the available data accumulated up to time $t$, i.e. $\{b_1, b_2, \ldots, b_t\}$ and the previously trained hypothesis $h_{t-1}$ is discarded. This is a too straightforward and simplistic approach for many scenarios: in fact the existing learned hypothesis $h_{t-1}$ is not used for the learning of the new one $h_t$. When concept drift happens, the learner is not able to adapt quickly to the new data. We would like to point out that for some memory-based approaches such as the locally weighted linear regression method, a certain level of previous experience can be accumulated to avoid the "catastrophic forgetting" problem. Nevertheless, this group of methods generally requires the storage of all accumulated data items. Therefore, it may not

be feasible in many data-intensive real applications (as when we deal with examples in structured form) due to the constraints of limited memory and computational resources.

#### Classifier ensemble for stream data mining

Existing research in the area has proposed a set of ensemble frameworks for stream data mining. Under this framework, one can build classifiers on rather small data chunks without missing major patterns.

One of the first approaches that deals with streams of structured data, in particular with trees, is proposed in [72]. The idea is not to keep in memory all examples that come from the data stream. Instead, only a fixed number of data chunks are kept in memory (e.g. the last $n$ chunks). A model is generated for each data chunk. When a chunk becomes outdated, it is not deleted but it is aggregated with another old data chunk in a DAG that is a compressed lossy representation of them, generating one single model. With this policy, we have a bounded, fixed number of models, and thus of classifiers, that are combined in a (linear) committee.

When a new data chunk arrives, there is a re-weighting phase that computes a new weight for each model based on its accuracy on a part (25%) of the current chunk. This weighting phase keeps only the informative parts of the data, thus this makes possible to deal with concept drifting. Old concepts will have a very low weight.

### 2.7.3 Online Stochastic gradient descent algorithms

Online algorithms to approximately solve the SVM optimization problem have been proposed in order to incrementally update the model working on a single example at a time. For example LASVM [16] proposes a tradeoff between optimality and scalability modifying the SMO algorithm to incrementally upgrade the model. Gradient descent methods are an appealing alternative to the quadratic programming methods. In stochastic gradient descent (SGD) the SVM training is transformed

in an unconstrained problem. The algorithm scan the data one example at a time and the model is updated using (sub-)gradient descent over an instantaneous objective function. This class of algorithms, due to its iterative nature, is often run in epochs, performing multiple passes over the input data for improved accuracy. The primal version of the PA, sketched in Algorithm 1, represents the solution as a sparse vector $w$. In machine learning $w$ is often referred to as the *model*. SGD differs from batch gradient descent in the way the gradient is estimated[18]. In the batch gradient descent algorithm, each iteration involves a burdening computation of the average of the gradients of the loss function $\nabla_w L(x_n, w)$ over the entire training set, see Section 2.6.1. The elementary online stochastic gradient descent algorithm is obtained by dropping the averaging operation in the batch gradient descent algorithm. Instead of averaging the gradient of the loss over the complete training set, each iteration of the online gradient descent consists of choosing an example $x_t$ at random, and updating the parameters $w_t$ according to the following formula:

$$w_{t+1} = w_t - \alpha \nabla_w L(x_t, w_t)$$

where $\alpha$ is the learning rate and L() is a loss function and $\nabla_w L(x_t, w_t)$ indicates the gradient, that is the vector of partial derivatives, of $L(x_t, w_t)$ with regard to $w$ .

It is worth noticing that most online algorithms actually are stochastic gradient descent algorithms. For example, the perceptron algorithm presented in Section 2.4.1 updates the model, when an error occurs, with the following update rule:

$$w_{t+1} = w_t + 2\alpha y_t w_t^T x_t$$

is a stochastic gradient descent with the following loss function

$$L_{perceptron}(x, w) = (sign(w^T x) - y) w^T x. \tag{2.3}$$

The resulting SGD rule is:

$$w_{t+1} = w_t - \alpha (sign(w_t^T x_t) - y_t) w_t^T x_t$$

Since the desired class is either +1 or -1, the weights are not modified when the pattern x is correctly classified. Therefore this parameter update rule is equivalent

**Algorithm 1** Primal Stochastic gradient descent online learning.

1: Initialize $w$: $w_0 = [0, 0, \ldots, 0]$
2: **for each round** $t$ **do**
3: Receive an instance $x_t$ from the stream
4: Compute the score of $x_t$: $S(x_t, w_t) = w_t^T x_t$
5: Receive the correct classification of $x_t$: $y_t$
6: **if** $L((x_t, y_t), M_t) > 0$ **then**
7: update the hypothesis: $w_{t+1} = w_t - \alpha \nabla_w L(x_t, w_t)$
8: **end if**
9: **end for**

to the perceptron rule. Algorithm 1 summarizes the primal SGD procedure. When a new example $x_t$ arrives(line 3) the algorithm computes its score (line 4) by means of a dot product with the current model $w_t$. The prediction is then $sign(S(x_t, w_t))$. After the prediction has been made, the correct labeling $y_t$ is received. If a prediction mistake occurred and the loss function value is greater than zero (line 6), the algorithm needs to update the model accordingly (gradient descent step). The new model $w_t + 1$ is computed accordingly to the particular gradient descent rule (line 7). It is worth to notice that $L(x_t, w_t)$ is the gradient of the instantaneous loss function $L$ defined only on the latest example. In the original SGD formulation, $L((x_t, y_t), w_t) = max(0, 1 - y_t S(x_t, w_t))$ is the *hinge loss* function.

Note that an equivalent dual version of SGD algorithm exists. For sake of simplicity, we will introduce a similar dual algorithm in the next section, and the dual Budget Stochastic Gradient Descent in Section 2.7.4.

### Online Passive-Aggressive

The Passive-Aggressive (PA) [39], among the different online learning algorithms, presents state-of-the-art performances, especially when budget constraints are present [149]. In Section 5.2 we propose an extension to this algorithm that deals with structured data. There are two versions of the algorithm, primal and dual, which differ in the way the solution $h()$ is represented. We will assume in the following that the model vector $w$ is sparse. In other words, we are not going to store the whole vector $w$, but only the elements that differs from zero. In the following we will use $|w|$ as the

number of non null elements in $w$. This assumption allows us not to work directly in the input space (the space in which examples live), but to use a mapping function $\phi$ from the input space to another bigger vectorial space, referred as *feature space*. For more details about the possible mappings, see Section 2.3.

Let us define the score of an example as:

$$S(x_t, w_t) = w_t \cdot \phi(x_t). \tag{2.4}$$

Note that $h(x)$ corresponds to the sign of $S(x)$. The algorithm, sketched in Algorithm 2, proceeds as in the stochastic gradient descent presented in Section 2.7.3: the vector $w$ is initialized as a null vector (line 1) and it is updated whenever the sign of the score $S(x_t)$ of an example $x_t$ is different from $y_t$ (line 6). The update rule of the PA finds a tradeoff between two competing goals: preserving $w$ as much as possible and changing it in such a way that $x_t$ is correctly classified. In the algorithm the weight assigned to the new example entering in the support set is referred as $\tau_t$. In [39] it is shown that the optimal update rule is:

$$\tau_t = \min\left(C, \frac{\max(0, 1 - S(x_t, w_t))}{\|x_t\|^2}\right), \tag{2.5}$$

where $C$ is the tradeoff parameter between the two competing goals above. It can be shown that Passive-Aggressive belongs to the family of stochastic Gradient Descent algorithms.

**Algorithm 2** Primal Passive-Aggressive online learning.

1: Initialize $w$: $w_0 = (0, \ldots, 0)$
2: **for each round** $t$ **do**
3: Receive an instance $x_t$ from the stream
4: Compute the score of $x_t$: $S(x_t, w_t) = w_t \cdot \phi(x_t)$
5: Receive the correct classification of $x_t$: $y_t$
6: **if** $y_t S(x_t) \leq 1$ **then**
7: update the hypothesis: $w_{t+1} = w_t + \tau_t y_t \phi(x_t)$
8: **end if**
9: **end for**

Under mild conditions [41], to every $\phi()$ corresponds a kernel function $K(x_t, x_u)$, defined on the input space such that $\forall x_t, x_u \in \mathbb{X}$, $K(x_t, x_u) = \phi(x_t) \cdot \phi(x_u)$, as

explained in Section 2.2. Notice that $w = \sum_{i \in M} y_i \tau_i \phi(x_i)$, where $M$ is the set of examples for which the update step (line 7 of Algorithm 2) has been performed. Then Algorithm 2 has a correspondent dual version, presented in Algorithm 3, in which the $\tau_t$ value is computed as

$$\tau_t = \min\left(C, \frac{\max(0, 1 - S(x_t, M_t))}{K(x_t, x_t)}\right) \tag{2.6}$$

and the score of equation (2.4) becomes $S(x_t, M_t) = \sum_{x_i \in M_t} y_i \tau_i K(x_t, x_i)$.

**Algorithm 3** Dual Passive-Aggressive online learning.

1: Initialize $M$: $M_0 = \{\}$
2: **for each round** $t$ **do**
3: Receive an instance $x_t$ from the stream
4: Compute the score of $x_t$: $S(x_t, M_t) = \sum_{(x_i, y_i, \tau_i) \in M_t} y_i \tau_i K(x_t, x_i)$
5: Receive the correct classification of $x_t$: $y_t$
6: **if** $y_t S(x_t) \leq 1$ **then**
7: compute $\tau_t = \min\left(C, \frac{\max(0, 1 - S(x_t, M_t))}{K(x_t, x_t)}\right)$
8: update the hypothesis: $M_{t+1} = M_t \cup \{(x_t, y_t, \tau_t)\}$
9: **end if**
10: **end for**

Here $M$ is the, initially empty set of tuples corresponding to support examples which the update rule modifies as $M = M \cup \{x_t, y_t, \tau_t\}$, where $x_t$ is the example, $y_t$ its label and $\tau_t$ its weight computed accordingly to Equation 2.6. It can be shown that the primal and dual algorithms compute the same solution. However, the dual algorithm does not have to explicitly represent $w$, since it is only accessed implicitly through the corresponding kernel function.

### 2.7.4 Budget online stochastic gradient descent algorithms

Respecting a memory budget is one of the constraints of online learning algorithms, as explained in Section 2.7. Moreover, if the algorithm complexity depends on the memory occupation like in the dual algorithms presented in Section 2.7.3, an effective method to control the speed of online learning algorithms is to control the amount of memory the algorithm is allowed to use. Assigning a fixed memory budget in this way ensures that the algorithm do not run out of memory, and

burden the computational complexity of the algorithm since for most of them the computational complexity can be expressed as a function of the budget. In this section we will review some of the most important algorithms in literature for this scenario.

### Budget stochastic gradient descent

The paper [146] proposes a modification of SGD algorithm limiting the model size to a budget $\mathcal{B}$ expressed as the number of Support Vectors representing the model. A budget maintenance step is performed whenever the number of SVs exceeds the budget, namely $|M| > \mathcal{B}$. This step reduces the size of $M$ by one. The result of this step is a degradation of the classifier. The budget maintenance strategy greatly influences this degradation. In Section 2.8 some of the principal strategies are presented, including random, oldest ones and the examples having lowest $\tau$ value.

**Algorithm 4** Dual Stochastic gradient descent online learning on a budget.

1: Initialize $M$: $M_0 = \{\}$
2: **for each round** $t$ **do**
3: Receive an instance $x_t$ from the stream
4: Compute the score of $x_t$: $S(x_t, M_t) = \sum_{(x_i, y_i, \tau_i) \in M} y_i \tau_i K(x_t, x_i)$
5: Receive the correct classification of $x_t$: $y_t$
6: **if** $L((x_t, y_t), M_t) > 0$ **then**
7: **while** $size(M_t) + size(x_t) > \mathcal{B}$ **do**
8: select an example $j$ and remove it from $M_t$
9: **end while**
10: compute $\tau_t = \alpha L((x_t, y_t), M_t)$
11: update the hypothesis: $M_{t+1} = M_t \cup \{(x_t, y_t, \tau_t)\}$
12: **end if**
13: **end for**

Let's proceed briefly explaining the algorithm. When a new example $x_t$ arrives(line 3) the algorithm computes its score (line 4) applying a kernel function with all the support vectors. Here the $\tau$ is the weight associated to each support vector (calculated in line 10). The prediction is then $sign(S(x_t))$. After the prediction has been made, the correct labeling $y_t$ is received. If a prediction mistake occurred and the loss function value is greater than zero (line 6), the algorithm needs

to update the model accordingly (gradient descent step). At this point, if the budget $\mathcal{B}$ is full, the budget maintenance step have to be performed. In this case, according to the particular policy, a support vector is removed. This step lead to a degradation of the model, that can be bounded for some budget maintenance policies. Finally, the example $x_t$ can be inserted into the model $M$. A new tuple is generated (line 11) containing the example, its label, and the weight $\tau$ computed accordingly to the particular gradient descent rule (line 10). It is worth to notice that $L(x_t, M_t)$ is the gradient of the instantaneous loss function $L$ defined only on the latest example. In the original BSGD formulation, $L((x_t, y_t), M_t) = max(0, 1 - y_t S(x_t, M_t))$ is the *hinge loss* function. Different works proposed modifications of the way the budget is maintained, each one giving funny names to the resulting algorithms. Most of them are briefly explained in Section 2.8. For sake of simplicity, in this chapter we will separate the learning algorithms with the budget policies.

In the next sections, we will present several learning algorithms that can be viewed as slight modifications of Algorithm 4. For this reason, where possible we will present the algorithms pointing out the differences with this one.

### Budget perceptron

The simplest budget online algorithm is, not surprisingly, the budget perceptron. This modification of the original perceptron algorithm was first proposed in [40]. We can easily obtain the algorithm instantiating the loss function$L(x_t, y_t, M_t)$ calculation in lines 6 and 10 of the BSGD Algorithm 4 to the perceptron rule $L_{perceptron}(x_t, y_t, M_t) = (sign(S(x_t, M_t) - y)y_t S(x_t, M_t)$. It is worth to notice that this rule is the same rule presented in equation 2.3 but for the dual version of the algorithm.

### Budget online Passive-Aggressive

In [149] it is proposed to modify the dual version of the passive aggressive algorithm presented in Algorithm 3 introducing the same budget constraint as in BSGD. Recalling from Section 2.7.3, the main difference between the perceptron and the passive-aggressive algorithm is that in the former the model is updated only if a

classification mistake occurs, while in the latter the model is updated even when a *margin error* occurs. In other words, we update the model even when an example is correctly classified but it is too close to the separating hyperplane (for an example, see Figure 2.2).

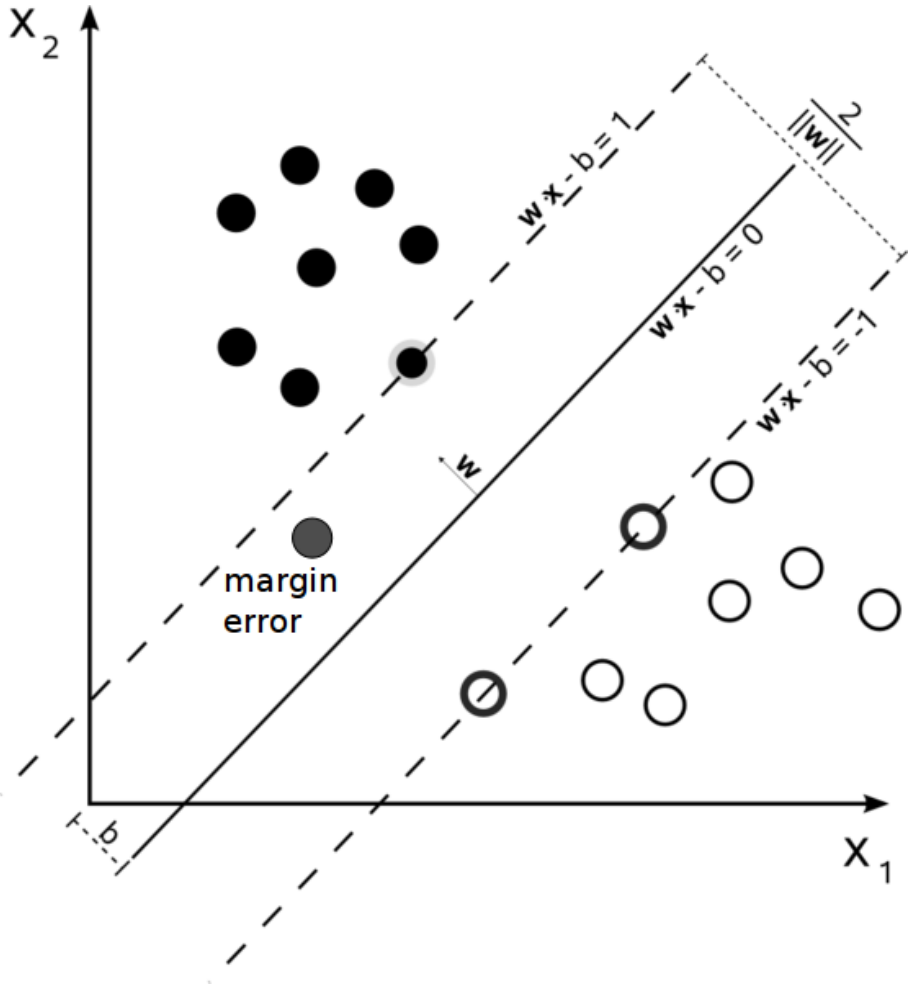


**Figure 2.2**: Example of a margin error.

For this algorithm, the loss function is computed according to Equation 2.5, and looks like:

$$L(x_t, y_t, M_t)_t = y_t - sign(S(x_t, M_t)) \min\left(C, \frac{\max(0, 1 - |S(x_t, M_t)|)}{K(x_t, x_t)}\right). \quad (2.7)$$

So for this formulation, the $\alpha$ parameter is fixed to one since the tradeoff is managed by the parameter $C$.

The resulting algorithm is given in Algorithm 5.

**Algorithm 5** Dual Passive-Aggressive online learning on a budget.

1: Initialize $M$: $M_0 = \{\}$
2: **for each round** $t$ **do**
3: Receive an instance $x_t$ from the stream
4: Compute the score of $x_t$: $S(x_t, M_t) = \sum_{(x_i, y_i, \tau_i) \in M} y_i \tau_i K(x_t, x_i)$
5: Receive the correct classification of $x_t$: $y_t$
6: **if** $y_t S(x_t) \leq 1$ **then**
7: **while** $size(M_t) + size(x_t) > \mathcal{B}$ **do**
8: select an example $j$ and remove it from $M_t$
9: **end while**
10: compute $\tau_t = \min\left(C, \frac{\max(0, 1 - S(x_t))}{K(x_t, x_t)}\right)$
11: update the hypothesis: $M_{t+1} = M_t \cup \{(x_t, y_t, \tau_t)\}$
12: **end if**
13: **end for**

The differences between Algorithm 5 and Algorithm 4 are in line 6, where the further looks for margin errors instead of classification errors, and in line

### 2.7.5 Feature selection

When dealing with fixed-size vectors as inputs, it is common to apply techniques of *feature selection* as a preprocessing step. Feature selection refers to the process of selecting a subset of relevant features for use in model construction, discarding the non-informative ones. Feature selection techniques provide several benefits when constructing predictive models:

- improved model interpretability;
- shorter training times;
- enhanced generalization by reducing overfitting.

In the context of kernel methods for structured data, feature selection is not commonly used since kernel functions avoid to explicitly accessing the feature space. However in Section 4.5.1 we propose a way to apply feature selection to a specific family of graph kernels.

In the literature, several feature selection techniques have been proposed. The paper [30] shows the impact of different feature selection techniques on SVM training. We will briefly review the two main approaches. A first approach for feature selection consists on trying to estimate a-priori the impact a feature is likely to have in the final model. A typical approach is to compute a statistical measure for estimating the relevance of each feature w.r.t. the target concept, and to discard the less-correlated features.

The measure we seek should take into account both feature frequency and target information. F-Score is a value that measures the discrimination of two sets of real numbers. The F-score [31] of a feature $i$ is defined for binary classification tasks as follows:

$$Fs(i) = \frac{(AVG_i^+ - AVG_i)^2 + (AVG_i^- - AVG_i)^2}{\dfrac{\sum_{j \in Tr^+} (f_i^j - AVG_i^+)^2}{|Tr^+| - 1} + \dfrac{\sum_{j \in Tr^-} (f_i^j - AVG_i^-)^2}{|Tr^-| - 1}} \tag{2.8}$$

where $AVG_i$ is the average frequency of feature $i$ in the dataset, $AVG_i^+$ ($AVG_i^-$) is the average frequency of feature $i$ in positive (negative) examples, $|Tr^+|$ ($|Tr^-|$) is the number of positive (negative) examples and $f_i^j$ is the frequency of feature $i$ in the $j^{th}$ example of the dataset. It should be noticed that features that get small values of F-score are not very informative with respect to the binary classification task.

Once the F-Score value has been calculated for every feature, it is possible to fix a threshold to cut low and high F-Scores.

In Section 4.5.1 we will present a new formulation of F-Score allowing for an incremental evaluation.

Another method to estimate feature importance is to train the model and to select, after the training phase, the features that mostly contribute to the classification. However, it is necessary to train the model on the complete set of features, giving up the speed benefits of feature selection. To avoid this problem, we can train a simpler (and thus faster) algorithm. In [30] it is proposed to use Random Forest, which is a classification method that also provides feature importance. Obviously,

different feature selection approaches can be combined in order to produce a more powerful feature selection pipeline.

## 2.8 Managing the budget

In online learning, algorithms have to be designed to respect a strict budget constraint. The budget is a limit on the amount of memory an algorithm is allowed to occupy. To design such algorithms is a challenging problem, thus for some algorithms in literature this constraint has been relaxed, e.g. [107].

Nonetheless, imposing a budget constraint is important:

- to avoid that the algorithm runs out of memory, essential in tasks that involve life-long learning;
- depending on the algorithm design, to adjust the accuracy/speed tradeoff.

The applicability of the budget constraint makes sense with kernelized algorithms, whose memory occupation is not bounded. In order to apply these constraints, when the model is represented as a set of examples, we have to select which examples have to stay in the Support Vector set and which do not. On the other hand, the application of a memory budget on *primal* algorithms, for example the *primal* version of the PA presented in Algorithm 2 is not straightforward and in general, when the model is represented as a vector, limiting the size of the vector implies the application of feature selection techniques. For this reason in this section we will focus on those algorithms where the model $M$ is represented as a set of examples.

To maintain a fixed number of support vectors, a budget maintenance step have to be performed every time the space needed from the support vector set exceeds a predefined memory budget $\mathcal{B}$. Without loss of generality, we can assume that if the examples are fixed-size vectors, a fixed number of examples can be accommodated into the budget $\mathcal{B}$. We will refer to this number as $b = |\mathcal{B}|/size(x)$. After the reduction step, the model will comprehend $i < b$ support vectors. This step

inevitably lead to a degradation of the model $M$. We will refer to this degradation as the difference between $M$ before and after the removal step at the $t$-th round as $\Delta_t$. Budget maintenance is a critical design issue, and different strategies may lead to different performances of the resulting algorithm. It can be shown that the gradient error (i.e. the deviation from the optimal gradient descent direction) is directly proportional to the weight degradation.

The main budget maintenance strategies can be grouped in three main categories: removal, projection and merging.

The policy managing the budget affects both the computational and spatial complexity of the algorithm. For example, when projection is used the space and time complexity of the resulting algorithm scales quadratically with the budget and linearly with merging or removal policies. Different budget maintenance strategies have been defined in literature, and are briefly presented in the following.

- Stoptron[107]: this is a very simple algorithm, that updates the model until the budget limit is reached and then, even if errors occurs, stops updating the model.

- Random Perceptron[25]: whenever the perceptron makes a mistake and the budget is full, a random support vector is deleted.

- Forgetron[48]: it is the first budget perceptron with performance guarantees. The idea is to reduce the weight of each example in the support vector set whenever a new support vector is added. Then, only when the budget is full, the oldest vector (that is the one with smallest weight by construction) is deleted.

- Projectron[107]: before adding a new example to the support vectors set, we check if the new hypothesis can be approximated using a smaller number of support vectors. If this is the case, the hypothesis is modified accordingly (introducing a small approximation error).

- BPA-P: in [149] he update rule of the dual version of the PA is extended by adding the following constraint: the new model $M_t$ must be spanned from only $\mathcal{B}$ of the available $\mathcal{B}+1$ examples in $M \cup \{x_t\}$. In other words, a projection step is added to the simple deletion of an example in $M$ whenever the budget is full.

- BPA-NN[149]: this approach is similar to BPA-P but, for reducing the complexity, the removed support vector is projected to only a subset of the support vectors in the models, the nearest ones.

- Twin Vector Machine [148]: the idea is to merge support vectors together in order to respect the budget constraint. Authors calls the merged support vectors *twin vectors* indicating that they merge two support vectors together.

In Section 5.2 we will propose a novel budget maintenance strategy.

# Chapter 3

# Learning on structured data

> The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.
>
> Edsger Dijkstra

Classical machine learning algorithms, e.g. the ones presented in Chapter 2, can directly be applied to all kinds of data that are easily embedded in a vectorial form, as this is the traditional setting of machine learning. For example in kernel methods, different kernel functions have been defined for vectors, like the *polynomial kernel*[118] or the *radial basis kernel*[121].

In this chapter we will present the motivations behind the application on machine learning techniques on structured data in Section 3.1. Then in Section 3.2 we will see why learning on graph streams is an interesting research direction. From Section 3.3 until the end of the chapter we will review the state-of-the-art on kernels for structured data.

## 3.1 Learning on graphs

Classical machine learning techniques are traditionally defined for data represented in a vectorial form. However, there are many potential application domains where

there are not natural representations of the examples in this form. For example, the task of predicting a certain property of chemical compounds (e.g. carcinogenicity, mutagenicity or toxicity) given their chemical structure is one of the tasks where it is very difficult to extract a vectorial representation of data without losing useful information. Such compounds can be easily represented as graphs where vertices represent the atoms and are labeled according to the type of the atom (e.g. Carbon or Oxygen), while edges represent the bonds between atoms and are labeled according to the bond type (e.g. single, double ...). Another example is the problem of protein-protein interactions in bioinformatics, where the protein secondary structure can be easily seen as a graph where nodes correspond to the amino acids and bonds refer to the distance between them. Also other fields may benefit from the application of learning algorithms on structured data: in computer vision, an image (or a video frame) can be viewed as its segmentation graph (see for example [9]). These are only some examples of the possible applications of learning algorithms on graphs (see [1]). Thus, despite the complexity of the problem, the application of learning algorithms on structured data is interesting because there are many applications that can benefit from this approach.

One possible approach is to derive a satisfactory vectorial representation for ad-hoc problems, for example see [111], but this is a painful, time consuming work and it should be done ex-novo for every different task.

Another solution consists is extending learning methods in such a way that they can be applied *directly* to learning problems where the representation of objects as vectors is not trivial. In particular, in this thesis we will consider examples represented as graphs, or as special types of graphs like trees.

In this learning scenario, encapsulating all information in a vectorial form is difficult because, if we want to map the examples into vectors, we have to design such a function. Ideally, we would require this mapping to respect some properties:

- isomorphic graphs must be mapped in the same vector;
- the function has to be computed efficiently (polynomial time);

- non-isomorphic graphs must be mapped in different vectors.

It is worth to notice that these properties are necessary conditions for the successfully application of a learning algorithm on our data.

Since the problem of deciding graph isomorphism is believed not to be efficiently solvable (not to be in P), we probably cannot hope to find a function that satisfies these properties, and without this function we cannot satisfactorily apply standard machine learning algorithms to vectorial representations of graphs.

One can instead try to find a function that violates some of the conditions (the second or the third one) but is still good enough for most learning problems, but this approach is not trivial and introduces various complications. In general, the more information we keep in the vectorial representation, the more computationally demanding the mapping is.

Therefore, it seems that a promising approach to deal with structured data is to avoid to explicitly map them in a vectorial form, and defining learning algorithms directly on these data. We will see that, for efficiency reasons, most of the learning algorithms defined in this way maps examples into vectors, too. The difference is that this map is not ad-hoc defined for the specific task, and so it has not to be defined by a domain-expert. Moreover, some learning algorithms perform this map only implicitly, with benefits from the computational point of view.

In particular, we saw in Section 2.2 that kernel methods have the nice property to separate the learning algorithm from the kernel function that faces with the examples and that maps them into an high-dimensional Euclidean space. We will see in Section 3.3 how to define kernel functions for structured data, in order to apply all kernel learning algorithms to this kind of data.

Recently, machine learning turned its attention on graph-structured data. Learning on graphs is much more challenging than learning on vectors as we saw in the previous section. In spite of this, various algorithms for learning on graphs has been proposed for traditional mining problems, as frequent pattern mining or classification. In this section, we will have an overview on some of these methods (for other methods, see [1]).

### 3.1.1 Notations

A graph is a triplet $G = (V, E, L)$, where $V$ (alternatively $V(G)$) is the set of vertices ($|V|$ is the number of vertices), $E$ the set of edges and $L()$ a function returning the label of a node.

A graph is undirected if $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$, otherwise it is directed. The set of neighbors of a vertex $v_i$ is defined as $\{v_j | (v_i, v_j) \in V\}$. This set may be referred as the set of children if the graph is directed.

A walk in a graph is a sequence of nodes $v_1, \ldots, v_n$ such that $v_i \in V$, $1 \leq i \leq n$ and $(v_i, v_i + 1) \in E$. A path is a walk with no repeated vertices. A cycle is a path for which $v_1 = v_n$; a cycle is even/odd if its number of nodes is even/odd, respectively. A simple cycle (or circuit) is a cycle with no repetitions of vertices or edges allowed, other than the repetition of the starting and ending vertex. A graph is connected if there exists a path connecting each pair of nodes. A directed connected graph is rooted if exactly one node has no incoming edges. A graph is ordered if the set of neighbors of each node is ordered. A tree is a rooted connected directed acyclic graph where each node has at most one incoming edge. A subtree of a tree $T$ is a connected subset of nodes of $T$. A proper subtree is a subtree composed by a node and all of its descendants. The symbol $\overset{v}{\triangle}$ is used to denote the proper subtree of $T$ rooted at $v$. Moreover, we denote as $\overset{v}{\triangle}|_h$ the subtree of $T$ resulting from a breadth-first visit starting at $v$ and limited to $h$ levels of depth. A subset tree is a subtree with the constraint that, for each node, either all or none of its children are included. Given a node $v$ of a tree, $\rho(v)$ represents the out-degree of $v$, i.e. the number of nodes connected to $v$. We will use $\rho$ as the maximum outdegree of a node in either a tree or a graph. The depth $depth(v)$ of a node $v$ is the number of edges in the shortest path between the root of the tree and $v$. If the tree is ordered, $ch_v[j]$ represents the $j$-th child of $v$ and $chs_v[j_1, j_2, \ldots, j_n]$ indicates the set of children of $v$ with indices $j_1, j_2, \ldots, j_n$.

### 3.1.2 Pattern mining on graphs

The problem of pattern mining on graphs can be stated as the problem of, having a group of graphs, determine all patterns (subgraphs) that appear at least in a fixed fraction of the graph dataset. These patterns are referred as frequent patterns.

To solve this problem, we have to face with the isomorphism issue in determining whether one graph is a subgraph of another graph. Moreover, since there may be overlapping among frequent patterns, the anti-monotonicity property of frequent patterns, assumed from most frequent pattern mining algorithms, is violated.

However, for this scenario most of the well-known techniques for frequent pattern mining on vectorial data can be extended to graphs. For example, Apriori[3]-style algorithms search for frequent patterns of increasing dimension in the dataset. However, the complexity of these methods is in general exponential in the size of graphs, so in many real-world datasets these techniques are not feasible.

Some of these techniques are used by algorithms for classification in a preprocessing phase, for the generation of a vectorial representation of graph examples, e.g. in [115]. This is another example of a technique for the generation of ah-hoc vectorial representations for graphs, saw in Chapter 3.

### 3.1.3 Graph classification algorithms

This is one of the tasks we deal with in this thesis. The task is to learn a model from a set of labeled graphs and use it to classify unseen examples. This is an active research area. Among the various and very different techniques that has been proposed, some of them have to be remarked: Kernel-based classification methods, Boosting-based classification and neural networks methods. A detailed description of available techniques concerning kernel methods on structured data will be provided in Chapter 3.3, so we will not explain in details these techniques here.

The second approach, the boosting-based one, addresses also the problem of revealing what graph structures (sub-structures) are relevant for classification. This

is achieved through pattern mining. We can build a binary feature vector corresponding to each graph based on the presence or absence of a certain sub-structure (subgraph) in the graph, and apply an off-the-shelf classifier on these vector representations. Since the entire set of subgraphs is often very large, we have to focus on a small subset of it. This subset is usually selected using frequent pattern mining. However, frequent patterns are not necessary relevant patterns (e.g. in Chemoinformatics patterns C-C or C-C-C are very frequent but have almost no significance in predicting characteristics like activity, toxicity ...). For this reason, boosting is used to automatically select a relevant set of subgraphs as features for classification. An example of an application of this approach is GBoost [115].

Neural networks techniques [49] consist in the application of artificial neural networks directly to graph-structured data.

Among all the techniques that have been applied to the graph classification problem in literature, learning methods based on graph kernels have largely outperformed other methods in terms of classification performance and often in computational complexity. That is the reason why recent research has focused on these methods.

## 3.2 Graph streams

Recently there has been some work in extending stream data mining to graph structures. In many real world tasks involving streams, representing data as graphs is a key for success, e.g. in fault diagnosis systems for sensor networks [7], malware detection [56], image classification or the discovery of new drugs (see Section 5.1.2 for some examples). Moreover, there has been a work on streaming graph classification [2]. This problem is quite difficult because it requires some effective approximations in order to be feasible. These has been achieved in practice via the well-known min-hash technique [21] (or the min-wise independent permutations locality sensitive hashing scheme), that is a way of quickly estimating how similar two objects are. This technique has been extended in order to deal with a stream of graphs, leading to an approximation of a similarity measure between graphs.

It is worth to notice that graph streams usually are stream of edges, i.e. an example may not arrive all at once. This further complicates the problem.

Recently, there is a continuously incrementing number of applications that can be modeled as streams of graphs over a predetermined massive underlying universe of nodes. Some examples are as follows:

- The communication pattern of users in social network in a modest time window can be decomposed into a group of disconnected graphs, which are defined over a massive-domain of user nodes.
- The browsing pattern of a single user (constructed from a proxy log) is typically a small subgraph of the web graph. The browsing pattern over all users can be interpreted as a continuous stream of such graphs.
- The intrusion traffic on a communication network is a stream of localized graphs on the massive IP-network.

Motivated from the increasing number of novel applications requiring these techniques, the classification of graph streams is one of the research areas where we focused. Chapter 5 will present such contributions.

### 3.2.1 Learning on graph data streams

Most online learning algorithms assume that the input data can be described by a set of features, i.e. there exists a function $\phi : \mathbb{X} \rightarrow \mathbb{R}^s$, which maps the input data onto a feature vector of size $s$ where learning is performed[1]. We will loose this condition assuming that $s$ may be very large (possibly infinite) but only a finite number of $\phi(x)$ elements, for every $x$, is not null, i.e. $\phi(x)$ can be effectively represented in sparse format.

In this section, we introduce the problem of learning a classifier from a (possibly infinite) stream of graphs respecting a strict memory constraint. In Chapter 5,

[1]While the codomain of $\phi()$ could be of infinite size, in order to simplify the notation we will use $\mathbb{R}^s$ in the following.

we will present an algorithm for processing graph streams on a fixed budget that performs comparably to the non-budget version, while being much faster.

Similar problems have been faced in literature, e.g. learning on large-scale data streams [52, 108, 131, 33, 63], learning on streams of structured data [8], learning on streams with concept drift [81, 119, 89], and learning on structured streams with concept drift [11, 13, 12]. However, none of the existing approaches considers memory constraints. Some methods provide bounds on the memory occupation, e.g. [12], but they cannot limit a priori the amount of memory the algorithm requires. Consequently, depending on the particular algorithm and on the data stream, there exists the possibility for the system to run out of memory. This makes such approaches unfeasible for huge graph streams.

Recently, the paper [92] proposed an ensemble learning algorithm on a budget for streams of graphs. Each graph is represented through a set of features and a hash function maps each feature into a fixed-size vector, whose size respect the given budget. Different features mapped to the same vector element by the hash function are merged into one. The performances of the learning algorithm vary according to the hash function and the resulting collisions. While the use of an ensemble allows to better cope with concept drift, it increases the computational burden to compute the score for each graph.

Learning from graphs is per se very challenging. In fact, state-of-the-art approaches use kernels for graphs, which usually are computationally demanding since they involve a very large number of structural features. Recent advances in the field, have focused on the definition of efficient kernels for graphs which allow a direct sparse representation of a graph onto the feature space [122, 38, 44]. For further details, see Chapter 3.6.

## 3.3 Kernels for structured data

For the motivations discussed in Section 3.1, in the last years the research on kernel methods focused on the definition of valid kernel functions for structured data.

Initially Kernel methods, as other standard machine learning tools, have been applied to various real-world problems with examples in vectorial form.

A first approach to apply this kind of methods on problems with data represented in other forms was the derivation of an ad-hoc, domain specific vectorial representation for the data and then the application of a classical learning algorithm for vectors on it.

This has been done for example in [111] representing graphs in a vectorial form using various polynomial-time graph invariants. The problem is that this function is very specific, and incorporates much domain-knowledge. Therefore, a new function has to be defined for every application domain.

With kernel methods, we have another option: thanks to the modularity of this approach (see Section 2.2), we can leave the learning algorithm unchanged, represent data in any meaningful way and adapt the kernel function to the chosen representation. This approach does not simplify the general problem but it allows a more systematic solution.

So kernel methods can be applied wherever a valid kernel function is defined, and kernel functions in general can be defined over any type of entity.

Furthermore, positive definite kernel functions themselves can be seen as inner products between the images of examples in a high dimensional Euclidean space. Thus, it appears more natural to extend kernel methods to structured data by directly defining a positive definite kernel function on the data rather then first mapping it into vectors and then mapping it into yet another space. Moreover, mapping the data only once and implicitly, using the kernel function, has computational advantages. Therefore, we should define how kernel methods can be applied to learning problems on structured data. Fortunately, kernel methods can be applied straightforwardly to any kind of data as long as a meaningful (that is, significant for the task) kernel function is known.

Unfortunately, defining a meaningful and fast kernel function on structured data is a very difficult task, because there is a tradeoff between the computational complexity of the kernel and its expressiveness, i.e. how much information from the

original data is considered.

In particular, the computation of large Gram matrices is known to be one of the bottlenecks of kernel methods. When dealing with simple vectors of reasonable size, this problem only materializes with very large databases. For complex structures such as texts or dense graphs, computational limits are hit much earlier. Hence, defining a kernel that can leverage the wealth of information provided by such structures while being still computationally tractable remains a major challenge, and is one of the research lines proposed in this thesis. Between the two ends of this spectrum lie structures such as *strings* or *trees*, which, although challenging from a combinatorial perspective, can be handled with relative efficiency through the convolution kernel framework (see Section 3.4).

We will focus on these types of data for drawing an increasing-complexity (non-exhaustive) taxonomy on kernels for structured data, emphasizing kernels for graph-structured data.

The research on kernel methods for structured data started with the work of Haussler [73] that we will summarize in Section 3.4, in which it is proposed a general framework that allows the definition of kernel functions over structured objects based on the decomposition of the object into simpler parts. In the next sections we will see some of the state-of-the-art kernel functions for structured data that has been defined in literature. This is an active area of research, since a fast and generally meaningful kernel cannot exist. In fact, every kernel function has its own drawbacks, that we will explain in detail.

## 3.4 Convolution kernels

One of the first and most important results in the field of kernels for structured data is Haussler's R-convolution kernel framework. This is a framework for defining positive definite kernel functions over any kind of objects, based on positive semidefinite kernels on some kind of decomposition of the object, that are kernels on sub-structures.

In short, let $\mathcal{X}$ be a space of objects (that are our data points) and let each object $x$ be associated with a finite subset $\mathcal{X}'_x$ of a space $\mathcal{X}'$. Furthermore, assume that a kernel over this domain $k : \mathcal{X}' \times \mathcal{X}' \to \mathbb{R}$ is defined. To define an $R-convolution$ kernel, Haussler [73] assumed a *finite* relation $R \subseteq {\mathcal{X}'}^D \times \mathcal{X}$, and let

$$K(x,y) = \sum_{(x'_1,\ldots,x'_n,x)\in R} \left( \sum_{(y'_1,\ldots,y'_n,y)\in R} \prod_{i=1}^{D} k(x'_i, y'_i) \right).$$

The commonly used formulation with $D = 1$ becomes:

$$K(x,y) = \sum_{(x',x)\in R} \sum_{(y',y)\in R} k(x', y').$$

With regard to positive definiteness of $R-convolution$ kernels, Haussler showed the following theorem.

**Theorem 3.1** (Haussler 1999)**.** *If $k$ is positive definite, $K$ is also positive definite.*

[129] rewritten this definition, and showed that any $R-convolution$ kernel can be derived from kernels determined by the following formula:

$$K(x,y) = \sum_{(x',y')\in {\mathcal{X}'}_x \times {\mathcal{X}'}_y} k'(x', y')$$

where ${\mathcal{X}'}_x$ and ${\mathcal{X}'}_y$ are finite subsets of $\mathcal{X}'$, and are determined according to $x$ and $y$.

Starting from the aforementioned simple formalization of the convolution kernels, [129] generalized the concept of the convolution kernel, and introduced the *mapping kernel*, that is explained in the next section.

### 3.4.1 Mapping kernels

The *mapping kernel* framework has been introduced in [129] to define kernels between structured objects. Recalling that ${\mathcal{X}'}_x = \{x' \in \mathcal{X}' | (x', x) \in R\}$, the mapping kernel is defined so that $(x', x)$ moves in subset $M_{x,y}$ of ${\mathcal{X}'}_x \times {\mathcal{X}'}_y$ rather then the entire cross product ${\mathcal{X}'}_x \times {\mathcal{X}'}_y$.

Mapping kernel is defined as:

$$K(x,y) = \sum_{(x',y') \in M_{x,y}} k(x',y').$$

This framework defines kernels over large structures that can be described through smaller ones enumerated in a set $\mathcal{X}'$. The kernels are defined by the mapping set $M_{x,y} \subset {\mathcal{X}'}^2$ paired with a *base* kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. This kernel $K$ is positive definite for all positive definite base kernels $k$ if and only if the mapping $\{M_{x,y} | x, y \in \mathcal{X}\}$ is transitive (see [129]).

The mapping kernel generalizes Haussler's convolution kernel in two ways:

1. The range of the pairs $(x', y')$ can be a subset $M_{x,y}$ instead of the entire ${\mathcal{X}'}_x \times {\mathcal{X}'}_y$.

2. The set ${\mathcal{X}'}_x$ can be an arbitrary set not limited to a subset of $\mathcal{X}'$.

In other words, the convolution kernel is the special case of the mapping kernel when $\mathcal{M}_{x,y}$ is ${\mathcal{X}'}_x \times {\mathcal{X}'}_y$.

In addition, the family of sets $\{M_{x,y} | x, y \in \mathcal{X}\}$ are called a *mapping system*. In relation to positive definiteness of mapping kernels, [129] introduced the following important notion and theorem.

**Definition 3.1.** *A mapping system $\{M_{x,y} | x, y \in \mathcal{X}\}$ is transitive if and only if the following conditions are met:*

- *if $(x', y') \in M_{x,y}$ then $(y', x') \in M_{x,y}$ holds;*
- *if $(x', y') \in M_{x,y}$ and $(y', z') \in M_{x,y}$, then $(x', z') \in M_{x,y}$ holds.*

**Theorem 3.2.** *The following conditions are equivalent.*

1. *The mapping system $M_{x,y} | x, y \in \mathcal{X}$ is transitive.*
2. *For an arbitrary positive definite kernel $k : \mathcal{X}' \times \mathcal{X}' \to \mathbb{R}$, the mapping kernel derived from it is positive definite.*

This equation provides a generic approach to *build* kernels but not to *compute* them. In [128] it is introduced a topology of families of sub-structures $\tau$ to define the mapping set $M_{x,y}$ in order to define computationally tractable kernels and then some examples of kernels of this family are shown. We will describe such kernels in the following sections.

#### Extension of mapping kernels

Mapping kernel theorem assert that if a mapping system is transitive, the resulting mapping kernel derived from a positive definite evaluation system is always positive definite. It do not deny the possibility that mapping kernels with non-transitive mapping systems or using non-positive-definite kernels on sub-structures can be positive definite.

A new technique called *covering technique* that allows us to deal with some of these cases is presented in [125]. In this work weaker but still sufficient condition for positive definiteness of mapping kernels are given. This work can be useful for future definitions of new kernels in the sense that it expands the possibility to easily demonstrate positive definiteness of kernels.

### String kernels

In this thesis, we are not interested in the details about kernels for strings. However, some concepts from string kernels will be used later. The idea behind string kernels is to compare strings in terms of common substrings. The paper [66] provides an extensive survey for these types of kernels, most of which fits in the convolution kernels framework.

## 3.5 Tree kernels

The majority of tree kernels are instances of the framework of convolution kernels. Every kernel defines a different decomposition of the input trees, i.e. paths or

subtrees, as we will see in the next sections.

### 3.5.1 Kernels for unordered trees

In general defining kernels on unordered trees is more complex with respect to the case of ordered ones. Indeed, in this case there is a kind of equivalence class between every possible permutation of the children of a node. This implies that practically it is infeasible to define kernels based on "complex" sub-structures when dealing with this kind of trees. Recently there has been some work [87] on defining a reasonable and feasible kernel for unordered trees. This kernel counts the number of common subpaths shared by two trees (the idea is from information retrieval [79]). A subpath is a part of a path. The implementation is efficient and based on a variation of the *multikey quicksort algorithm* applied to the tree of suffixes. The idea is to enumerate prefixes of common suffixes of two trees, where a suffix is the string starting from a node and ending at the root.

The algorithm for computing the kernel function between two trees $T_1$ and $T_2$ can be summarized as follows.

1. Initialize the kernel function value $k$ as $k := 0$, the current position $h$ as $h := 1$, and the current set $S$ as the set of all nodes in $T_1$ and $T_2$.

2. For nodes included in the current set $S$,

   (a) choose one of the node labels in $S$ as pivot at random;

   (b) compare the node labels in $S$ with the pivot label, and divide the nodes into three sets $S_{small}$, $S_{large}$ and $S_{equal}$, that are, nodes with labels alphabetically larger than the pivot, nodes with smaller labels than the pivot, and the others (with the same label of the pivot), respectively;

   (c) if $S_{small}$ has at least one node from both $T_1$ and $T_2$, apply step 2 to $S := S_{small}$;

   (d) apply Step 2(c) to $S_{large}$;

(e) if $S_{equal}$ has at least one node from both $T_1$ and $T_2$, update $k$ by using

$$k := k + w_h L(T_1)L(T_2)$$

where $w_h$ is the weight parameter, $L(T_1)$ and $L(T_2)$ are the number of nodes in $S_{equal}$ originated from $T_1$ and $T_2$, respectively. Otherwise, exit from the current recursion. Set $h := h + 1$ and apply step 2 to $S := parents(S_{equal})$, that is the set of parent nodes of $S_{equal}$.

### 3.5.2 Kernels for ordered trees

The main issue of kernels for unordered trees is that they can rely only on path/walk sub-structures in order to avoid NP-hardness. Moreover, most of the applications of kernels for trees are on ordered trees, i.e. trees where a total order among the children of each node is defined.

Many works deal with this kind of structures. Most of these kernels fall into the framework of mapping kernels, described above in Section 3.4.1.

#### Tree edit distances kernel

The work in [128] shows an exhaustive topology of tree kernels that can be efficiently computed. [23] introduces an edit distance that measures the (dis)similarity between two trees. We now describe the concept of edit distance, which can be applied to trees and graphs as well. Given two graphs (trees) $X$ and $Y$, let us define an edit script $\sigma$ as a finite sequence of edit operations that converts $X$ into $Y$. An edit operation can be:

1. *deletion* of a vertex $x \in V(X)$
2. *insertion* of a vertex $y \in V(Y)$ into $X$
3. *substitution* of a vertex $y \in V(Y)$ for $x \in V(X)$.

Each operation has an associated, possibly different, cost. The cost of a script $c(\sigma)$ is the sum of the costs of all edit operations in $\sigma$. Then, the distance between $X$ and $Y$ can be defined as the minimum cost over all edit scripts that turns $X$ into $Y$. Moreover, the costs of all the possible scripts can be considered as in the following kernel from [129]:

$$K(X,Y) = \sum_{\sigma:X\to Y} e^{-\lambda c(\sigma)}.$$

This kernel is an instance of mapping kernels (see Section 3.4.1).

Another family of tree kernels measures the similarity of two trees based on the number of sub-structures that they share. The difference between these kernels is the type of sub-structures they look for. In the family of tree kernels that search for common subtrees, we will describe three of the most important ones, in increasing expressivity order.

### Subtree kernel

*Subtree kernel* is an extension of the string kernel proposed in [143] that applies on trees. The set of all proper subtrees of the input trees forms the *feature space* of this kernel. In fact, the kernel can be formulated as a weighted sum over all proper subtrees shared by two trees.

Let $X$ and $Y$ be two trees. The subtree kernel is defined as:

$$K_{subtree}(X,Y) = \sum_{x\in X}\sum_{y\in Y}\delta(x,y)w_x = \sum_{s\in\mathcal{A}^*} h_s(X)h_s(Y)w_s$$

where $x$ and $y$ are proper subtrees of $X$ and $Y$, $w_x$ is the weight associated to the tree $x$, $\mathcal{A}^*$ is the set of all possible subtrees, $h_s(X)$ counts the frequency of the subtree $s$ in $X$, and $\delta$ is Kronecker's delta function, that is a function evaluated to 1 if the two arguments matches, 0 otherwise.

The kernel complexity that arises from the formulation is quadratic. However, an efficient implementation has been proposed that, using suffix trees, lowers the

complexity to $O(n \log n)$, where $n$ is the maximum number of nodes in $X$ and $Y$ (see [143]).

### Subset tree kernel

Unlike *Subtree kernel*, that calculates the similarity of two graphs counting the number of shared proper subtrees, the kernel explained in this section is based on *subset trees*. It is worth to notice that generally the number of subset trees in a graph is larger than the number of *proper* subtrees, so there is a significant expressivity gap between these two kernels.

Let $t_1, \ldots, t_m$ be the distinct trees corresponding to the set of subset trees induced by a finite set of trees. We can define a feature space in which every feature represents one different subset tree.

Let $h_s(X)$ be the number of times the subset tree $t_s$ occurs in the tree $X$. Every tree will then be represented as the feature vector:

$$\phi(T) = [h_1(T), \ldots, h_m(T)].$$

The inner product between the feature vectors of two trees in this representation is the subset tree kernel, defined as:

$$k_{subsettree}(X, Y) = \phi(X)\phi(Y) = \sum_{s=1}^{m} h_s(X)h_s(Y).$$

The *subset tree kernel* can be calculated efficiently by a recursive formulation: let $I_s(x)$ with $x \in V_X$ be a function evaluated to 1 if the subset tree $t_s$ appears in $X$ and is rooted in $x$, 0 otherwise. Then $h_s(X) = \sum_{x \in V_X} I_s(x)$. We can define the kernel as:

$$\begin{aligned} k_{subsettree} &= \sum_{s=1}^{m} h_s(T)h_s(T') \\ &= \sum_{s=1}^{m} \sum_{v \in V_T} I_s(v) \sum_{v' \in V_{T'}} I_s(v) \\ &= \sum_{v \in V_T} \sum_{v' \in V_{T'}} C(v, v') \end{aligned}$$

where $C(v, v') = \sum_{s=1}^{m} I_s(v) I_s(v')$.

$C(v, v')$ can be computed recursively with the following definition:

1. if the productions of $v$ and $v'$ (the production of $v$ is the subset tree rooted in $v$ and that includes only its children) are different, then $C(v, v') = 0$;

2. if the productions of $v$ and $v'$ are the same, and $v$ and $v'$ have only leaf children (they are pre-terminal symbols), then $C(v, v') = \lambda$;

3. if the productions of $v$ and $v'$ are the same, and $v$ and $v'$ are not pre-terminals, then

$$C(v, v') = \lambda \left( \prod_{j=1}^{nc(v)} (1 + C(Ch_j(v), Ch_j(v'))) \right)$$

where $nc(v)$ is the number of children of $v$, $Ch_j(v)$ returns the j-th children of node $v$, and $0 < \lambda \leq 1$ is a parameter used for the downweighting of larger structures.

The computation of this kernel consists in a dynamic programming procedure, that is the fulfilling of an $|X| \times |Y|$ matrix. Being $n$ the number of nodes of the biggest input tree, the computational complexity is $O(n^2)$.

### Partial tree kernel

It is possible to enhance the *subset tree kernel* in order to further augment the corresponding feature space dimension.

Partial trees are often simply referred as subtrees. Indeed, all the possible subtrees of the graphs in the training set form the feature space induced by partial tree kernel. the

More in detail, the *Partial tree kernel* [101] measures the similarity of two trees in terms of the partial matching of their subtrees. The definition is quite similar to the *subset tree kernel* one, as the only modification is in the definition of the $C$ function:

$$k_{partialtree}(X,Y) = \sum_{x \in V_X} \sum_{y \in V_Y} C(x,y)$$

where $C(x,y)$ is now defined by the following cases:

1. if $x$ and $y$ have the same label, then $C(v,v') = 0$
2. otherwise

$$C(v,v') = \mu \left( \lambda^2 + \sum_{\boldsymbol{J}_1, \boldsymbol{J}_2, |\boldsymbol{J}_1| = |\boldsymbol{J}_2|} \lambda^{d(\boldsymbol{J}_1)+d(\boldsymbol{J}_2)} \prod_{j=1}^{|J_1|} C(Ch_{J_{1i}}(v), Ch_{J_{2i}}(v')) \right),$$

where $J_1, J_2, \ldots$ are index sequences associated with the ordered child sequences $Ch(v), Ch(v')$, $J_{1i}, J_{2i}$ point to the i-th children in the two sequences, $|J_1|$ returns the length of the sequence $J_1$ and $d(J_1) = J_{1|J_1|} - J_{11}$ (that is the number of gaps that has been introduced), while $\mu$ and $\lambda$ are two decaying factors.

Partial tree kernel can be computed in $O(p^3|V(X)||V(Y)|)$ where $p$ is the maximum out-degree of the trees $X$ and $Y$.

#### Other tree kernels

Many other kernels for ordered trees have been proposed in literature, e.g. the *route kernel* [4] and the *elastic tree kernel* [84]. For a complete survey see [42]. Here we will focus our attention on graph kernels, so we are not going to describe these kernels.

## 3.6 Kernels for graphs

The main issue with graph kernels is whether it is possible or not to define graph kernels that distinguish between all non-isomorphic graphs and, if not, how can efficient graph kernels be defined to capture most of the structural information of the graphs. The research on graph kernels has been made possible thanks to the

important contribution of Haussler [73] that gave us an effective way to define valid kernels on structured data, see Section 3.4.

We start our overview on graph kernels, following [67], with some first tries of defining kernels for graphs.

The first class of graph kernels we are going to consider is the class of kernels that allows to distinguish between all (non-isomorphic) graphs in the feature space. If a kernel does not allow us to distinguish between two graphs then there is no way any learning machine based on this kernel function can separate these two graphs (and thus correctly classify them if their target value is different). Investigating the complexity of graph kernels that distinguish between all graphs is thus an interesting problem.

Unfortunately, computing any complete graph kernel is at least as hard as deciding whether two graphs are isomorphic [67].

Let us now look at another interesting class of graph kernels. Intuitively, it seems reasonable to base the similarity of two graphs on their common subgraphs. It is possible to define a graph kernel that decomposes a graph into the set of its subgraphs. Alternatively we can consider the map $\Phi$ that has one feature $\Phi_h$ for each possible graph $h$, each feature $\Phi_h(X)$ measuring how many subgraphs of $X$ are isomorphic to $h$.

Again, this problem is known to be NP-hard, so it is highly improbable that an algorithm that can solve this problem in polynomial time exists.

Moreover, limiting the feature space to only some specific types of subgraphs, e.g. paths, does not change the problem complexity that remains NP-hard, as shown in [67].

After these initial discouraging results, research focused on the definition of less expressive (and thus less discriminative) kernels that are computable in polynomial time. It is worth to point out that the definition of good kernels for graphs is a challenging problem because of the inevitable tradeoff between the computational complexity and the expressiveness of the kernel, which in general directly affects the predictive performance.

In the following, some of the most important proposals found in the recent literature are discussed.

### 3.6.1 Random walk kernels

An alternative approach to the "naïve" kernels presented in the previous section consists in measuring the number of common walks with identical label sequences in two graphs. Although the number of common walks can be infinite, the inner product in this space can be computed in polynomial time by first building the product graph and then computing the limit of a matrix power series of its adjacency matrix. This is the idea of the *product graph kernel*. It is worth to notice that the computation of this kind of kernels is polynomial for undirected graphs only [69].

#### Product graph kernel

This kernel, one of the first to be proposed in [66] by Gärtner who was a key forerunner in this field, counts the common walks of two graphs. A specific type of graph product is used, the *direct product graph*, that is an instance of the tensorial product.

More formally, the direct product graph of two graphs $X$ and $Y$, referred as $P_\times = X \times Y = (V_\times, E_\times)$ , is the graph which vertex and edge sets are defined as follows:

$$V_\times = \{(x, y) : x \in V(X) \wedge y \in V(Y) \wedge L_X(x) = L_Y(y)\}$$

$$E_\times = \{((x, y), (x', y')) \in V_\times \times V_\times : (x, y) \in E(X) \wedge (y, y') \in E(Y) \wedge \{L_X(x, y) = L_Y(y, y')\}$$

Now let X and Y be two graphs, and let $A_\times$ be the adjacency matrix of their product graph $P_\times = (V_\times, E_\times)$ . With the weight sequence $\lambda = \lambda_0, \lambda_1 \ldots$, with $\lambda_i \in \mathbb{R}; \lambda_i \geq 0$ for each $i \in \mathbb{N}$, it is possible to define the product graph kernel as:

$$K_\times(X, Y) = \sum_{x,y=0}^{|V_\times|} \left[ \sum_{k=0}^{\infty} \lambda_k A_\times^k \right]_{x,y}$$

if the limit exists.

This limit can be computed efficiently for particular values of lambdas: indeed, it is possible to reduce the problem to geometric or exponential series. Both the reductions have $O(n^6)$ computational complexity.

A new reduction of the kernel calculation to the solution of a *Sylverster equation* has recently been proposed in [140]. This approach lowers the computational complexity of the kernel to $O(n^3)$.

### Marginalized kernel

Another kernel has been defined that has the same feature space of the product graph kernel, but it has a completely different formulation. For historical reasons, we will explain this kernel too, that has been independently proposed in [85].

Informally, this kernel is defined as the expected value of a kernel over all possible pairs of label sequences generated by random walks on two graphs. The procedure for the generation of a random walk in a graph $X$ consists in (i) select the starting vertex $v_1 \in V(X)$ according to a probability distribution $p_s(v_1)$ over all vertices in $V(X)$ (ii) in the i-th step, the vertex $v_i$ is extracted according to the transition probability $p_t(v_i|v_{i-1})$ to go in the vertex $v_i$ starting from $v_{i-1}$, or the random walk stops with a probability $p_q(v_{i-1})$. The sum of the probability of moving from a vertex to another in its neighborhood and the stop probability of the current vertex, should be equal to 1, that is:

$$\sum_{i=1}^{|V(G)|} p_t(v_i|v_{i-1}) + p_q(v_{i-1}) = 1.$$

Every random walk generates a vertex sequence $w = (v_1, \dots v_l)$ where $l$ is the random walk length (possibly infinite). The probability of a walk $w$ to be generated is:

$$p(w|G) = p_s(v_1) \left( \prod_{i=2}^{l} p_t(v_i|v_{i-1}) \right) p_q(v_l).$$

Every walk have a corresponding label sequence, obtained alternating the vertex and edge labels of the walk:

$$\begin{aligned} h_w &= (L(v_1), L(v_1, v_2), L(v_2), \ldots, L(v_l)) \\ &= (h_1, h_2, \ldots, h_{2l-1}). \end{aligned}$$

The probability for the sequence $h$ to be generated is the sum over all the probabilities of the walks $w$ that generate the same label sequence $h_w = h$:

$$p(h|G) = \sum_w \delta(h, h_w) \left\{ p_s(v_1) \prod_{i=2}^{l} p_t(v_i|v_{i-1}) p_q(v_l) \right\}$$

where $\delta$ is Kronecker's delta. It is now possible to define a kernel $k_z$ for label sequences, using a PD kernel for vertices $k_v$ and one for edges $k_e$:

$$k_z(h, h') = k_v(h_1, h'_1) \prod_{i=2}^{l} k_e(h_{2i-2}, h'_{2i-2}) k_v(h_{2i-1}, h'_{2i-1})$$

Finally, the graph kernel based on label sequences is defined as the expected value of $k_z$ over every possible label sequence pair $h$ and $h'$ from two graphs $X$ and $Y$:

$$k(X, Y) = \sum_h \sum_{h'} k_z(h, h') p(h|X) p(h'|Y)$$

This kernel is an instance of the framework of $R - convolution$ kernels, where the decomposition is the set of all possible label sequences generated from a random walk. The complexity of this kernel is $O(n^6)$.

In [94] two extensions of the marginalized kernel have been proposed, aiming to improve the efficiency and to address the tottering problem. The first modification consists in a process of *label enrichment*, i.e. the encapsulation in vertex labels of additional information concerning the neighborhood of the vertex. This is achieved through the *Morgan index*. In fact, every label is enriched with the information about how many fixed-length paths starts from the vertex. This leads to a higher *feature relevance* (i.e. there will be less common labels between two graphs), with a computational gain and a small accuracy improvement in some datasets. The

second contribution consists in the reformulation of the random walk kernel in order to avoid tottering between two vertices, but the problem persists in longer cycles. However, this modification leads to a significant improvement in kernel performance.

### 3.6.2 Cyclic pattern kernel

This kernel has been proposed in [80] as an alternative to walk kernels. It is polynomial in the number of vertices and simple cycles of the graph. For this reason, it can be applied only if the number of cycles is small. This is the case in chemical informatics, but it is not true in the general case or in other domains. We defined in Section 3.1.1 what a simple cycle is. Without going too much in the details, it can be shown that every cycle has a canonical representation, i.e. a string that represents uniquely the cycle. The set of cyclic patterns of a graph $X$ is defined as

$$C(X) = \{\pi(c) | c \in S(X)\}$$

where $\pi(c)$ is the canonical representation of a cycle $c$ and $S(X)$ is the set of simple cycles of $X$. Now in order to make the kernel more expressive, we can consider the graph obtained removing all the edges that are in a cycle in the original graph $X$. The resulting graph is a forest consisting of the set of bridges (i.e. all the edges that do not belong to simple cycles) of the graph. Therefore, we can generate a pattern from each maximal tree in this forest, and refer the set of all tree patterns of a graph as $T(X)$. The feature space of this kernel is thus composed of features representing cyclic and tree patterns of the graph. We can formulate the kernel as:

$$K_{CP}(X, Y) = |C(X) \cap C(Y)| + |T(X) \cap T(Y)|$$

where the $\cap$ operator returns the set of common cycles (trees), as expected. We recall that a property of kernel functions ensure the positive semidefiniteness of the kernel resulting from this operation.

### 3.6.3 Subtree pattern kernel

Another alternative to random walk kernels has been proposed by Ramon and Gärtner in [112]. It relies on the observation that existing graph kernels were all conceptually based on the common walks between two graphs. See Section 3.6.1 for some examples. One of the problems of these kernels is that one can easily find pairs of graphs that are mapped to the same point in the feature space. So [112] proposed a kernel based on counting the common *subtree patterns* between two graphs. Roughly, the subtree patterns considered are rooted subgraphs such that there exists a subtree homomorphic to the subgraph (pattern), and the number of distinct children of both root nodes in the pattern and tree are the same.

A high-level description for the algorithm that calculates the kernel for two graphs $X$ and $Y$ follows.
For each vertices tuple $(x, y) \in V(X) \times V(Y)$:

- apply a positive semidefinite kernel for vertices on $x$ and $y$;
- recursively apply the kernel on the neighbors sets of the vertices $x$ and $y$, one level at time, until a pre-defined level $h$ is reached.

Notice that the algorithm analyzes one neighborhood level at time, considering all possible combinations of neighbors. It is not defined any ordering between vertices.

This kernel has a richer feature space compared to random walk kernels, but the calculation is more expensive and grows exponentially with the maximum depth $h$ of the considered subtree patterns.

For this reason, in the original paper no experimental results for this kernel are given.

### 3.6.4 Shortest path kernels

The motivation that led to the definition of this kernel was again to find a practical and effective similarity measure for graphs, because walk kernels were not so expressive, while other kernels, e.g. the subtree pattern kernel, were hard to compute.

The idea behind these kernels, proposed in [17], is to compare all-pairs shortest paths between two graphs. The computation of shortest paths is known to be a polynomial-time problem. There are many algorithms for solving this problem, that are for example *Dijkstra*, that calculate shortest paths starting from a vertex in $O(m + n \log n)$ ($n$ number of vertices, $m$ number of edges) time, or *Floyd-Warshall* that computes all-pairs shortest paths in $O(n^3)$.

The use of shortest paths in the kernel definition leads to a problem. The shortest path between two vertices in a graph can not be unique and there is no way for deterministically select one path among the others. Notice that to non-deterministically choose one shortest-path is not a solution because this would lead to a non positive semidefinite kernel function, while keeping all shortest paths would lead to a NP-hard kernel. Nevertheless, the length of the shortest paths between two vertices is unique, so in [17] it is proposed to use this property for the kernel definition.

Starting from a graph $X$, we can derive its *shortest path graph* $S(X)$ that has the same set of vertices of the original graph, that is $V(S(X)) = V(X)$, and that have an edge connection between two vertices $x$ and $y$ if and only if in the original graph at least one path exists that connects the two vertices. A label, that is the length of the shortest of these paths, is then associated to every edge.

This shortest path graph can be obtained using Floyd-Warshall algorithm in $O(n^3)$.

The shortest path kernel on two graphs $X$ and $Y$ is then defined as:

$$k_{shortest-path}(X, Y) = \sum_{e \in E(S(X))} \sum_{e' \in E(S(Y))} k^1_{walk}(e, e')$$

where $k^1_{walk}$ is a positive semidefinite kernel on 1-length walks, e.g. a kernel on edges.

The complexity of the kernel is dominated by the number of comparisons, i.e. one for each possible combination of edges from the two graphs. Every graph has at most $n^2$ edges, so the total complexity of the kernel is $O(n^4)$.

### 3.6.5 Graphlet kernel

Graphlet kernel [123] is a kernel that tries to approximate the *all-subgraphs* kernel, discussed in Section 3.6.

The idea is to measure the similarity between graphs in terms of the common subgraphs of a fixed (low) size.

Fixed the size $i$ of graphlets, the kernel can be defined as:

$$k_i(X, Y) = \sum_{S \in \mathcal{M}_i(X)} \sum_{S' \in \mathcal{M}_i(Y)} \delta(S, S')$$

where $\mathcal{M}_i(X)$ is the set of all possible matrices of dimension $i$ obtained deleting $n - i$ rows and the corresponding columns from the original adjacency matrix of the graph $X$ (that is, the set of all graphlets of size $i$ of $X$), and $\delta$ is the Kronecker's delta function. It is worth to notice that the $\delta$ function incorporates the graph isomorphism problem, so with the direct application of the formula, the kernel would be computationally unfeasible. However, when dealing with unlabeled graphs, one can pre-compute the set of all possible subgraphs of dimension $i$ and pre-compute the isomorphism. In this way the kernel computation becomes polynomial.

### 3.6.6 Weisfeiler-Lehman kernels

The first instance of this family of kernels was proposed in [122]. After that, recently a framework that generalizes the first definition has been proposed in [124].

The idea behind this framework is to represent every graph in the dataset as a sequence of graphs, each one with a different labeling function that encapsulates a specific iteration of the Weisfeiler-Lehman test of isomorphism.

More formally, the sequence of Weisfeiler-Lehman graphs for a graph $X = (V, E, L)$ is

$$\{X_0, X_1, \ldots, X_h\} = \{(V, E, L_0), (V, E, L_1), \ldots, (V, E, L_h)\}$$

where $X_0 = X, L_0 = L$ and the labeling function $L_i$ represents the relabeling of the original graph obtained applying the Weisfeiler-Lehman isomorphism test up to height $i$.

The general Weisfeiler-Lehman kernel framework can be formulated, being $k$ any positive semidefinite kernel for graphs:

$$K_{WL}^h(X, Y) = k(X_0, Y_0) + k(X_1, Y_1) + \ldots + k(X_h, Y_h). \tag{3.1}$$

Among the various instances of this framework that has been proposed, the first and most impactful is the Weisfeiler-Lehman subtree kernel.

### Weisfeiler-Lehman subtree kernel

This kernel, proposed in [122], is quite similar to the kernel explained in 3.6.3 with the difference that this kernel considers all subtree patterns up to height $h$, while *subtree pattern kernel* considers only the subtrees of exactly height h. Moreover, the further checks whether the neighborhoods of two nodes $x \in X$ and $y \in Y$ match exactly, whereas the latter considers all pairs of matching subsets of the neighborhoods of $x$ and $y$ . These differences lead to a consistent speedup of the kernel calculation: indeed this kernel can be computed in $O(hm)$ time on a pair of graphs (where $m$ is the maximum number of edges of the two graphs), and in $O(Nhm + N^2hn)$ on $N$ graphs (where $n$ is the maximum number of nodes of the graphs in the dataset).

For defining the WL-subtree kernel let the base kernel of equation 3.1 be the kernel counting pairs of matching node labels in two graphs:

$$k(X, Y) = \sum_{x \in V(X)} \sum_{y \in V(Y)} \delta(L(x), L(y))$$

where $\delta$ is the Kronecker delta function. Adopting this as base kernel, $K_{WL}^h$ becomes the subtree kernel defined in [122].

This kernel is a milestone in graph kernels because of its low complexity, and its good experimental results on real-world classification tasks.

Other kernels that belong to this framework have been defined. Just to mention, the WL-edge kernel, which counts matching pairs of edges with identically labeled endpoints, that have a complexity of $O(N^2m^2)$ per iteration.

Another example is the WL-shortest path kernel that can be computed in $O(N^2m^4)$.

It is worth to notice that these kernels can be computed also explicitly, keeping in memory the entire feature space, but in most cases this is not practical.

### 3.6.7 Neighborhood subgraph pairwise distance kernel

This kernel is an extension of the kernel explained in Section 3.6.6. This is a recent work, proposed in [38].

For the definition of this kernel, some additional notation is needed. The *distance* between two vertices of a graph $X$ is the length of the shortest path between them. The *neighborhood* of radius $r$ of a vertex $x \in V(X)$ is the set of vertices at a distance less than or equal to $r$ from $x$. In a graph, an induced subgraph of a set of vertices $W$ is the graph that have $W$ as vertices, and contains every edge of the original graph whose endpoints are in $W$. The *neighborhood subgraph* of radius $r$ of a vertex $x$ is the subgraph induced by the neighborhood of radius $r$ of $x$. It is denoted by $N_r^x(X)$.

We can define a relation $R_{r,d}(A^x, B^y, X)$ between two rooted graphs $A^x$, $B^y$ and a graph $X$ to be true iff both $A^x$ and $B^y$ are in $\{N_r^v | v \in V(X)\}$ (where the set inclusion is up to isomorphism) and the distance between $u$ and $v \in V(X)$ is exactly $d$. In other words, the relation selects all pairs of neighborhood graphs of radius $r$ whose roots are at distance $d$ in a given graph $X$. We can define an auxiliary kernel for graphs as the convolution kernel that uses this relation:

$$k_{r,d}(X,Y) = \sum_{\substack{A^v, B^u \in R_{r,d}^{-1}(X) \\ A'^v, B'^u \in R_{r,d}^{-1}(Y)}} \delta(A^v, A'^v)\delta(B^u, B'^u)$$

where $\delta$ is the Kronecker delta function. In words, $k_{r,d}$ counts the number of identical pairs of neighborhood subgraphs of radius $r$ at a distance $d$ in two graphs. Figure 3.1 shows an example of such features.

Finally, the *neighborhood subgraph pairwise distance* kernel is defined as:

$$K(X,Y) = \sum_{r} \sum_{d} k_{r,d}(X,Y)$$

For efficiency reasons, we can further define a kernel that imposes an upper bound on radius and distance parameters:

$$K_{r*,d*}(X,Y) = \sum_{r=0}^{r*} \sum_{d=0}^{d*} k_{r,d}(X,Y)$$

The complexity of the kernel is $O(|V||V_h||E_h| \log |E_h|)$, where $|V_h|$ and $|E_h|$ are the number of nodes and the number of edges of the subgraph obtained by a breadth-fist visit of depth $r*$. The authors state that, for small values of the subgraph size and distance, the complexity of the kernel becomes practically linear.

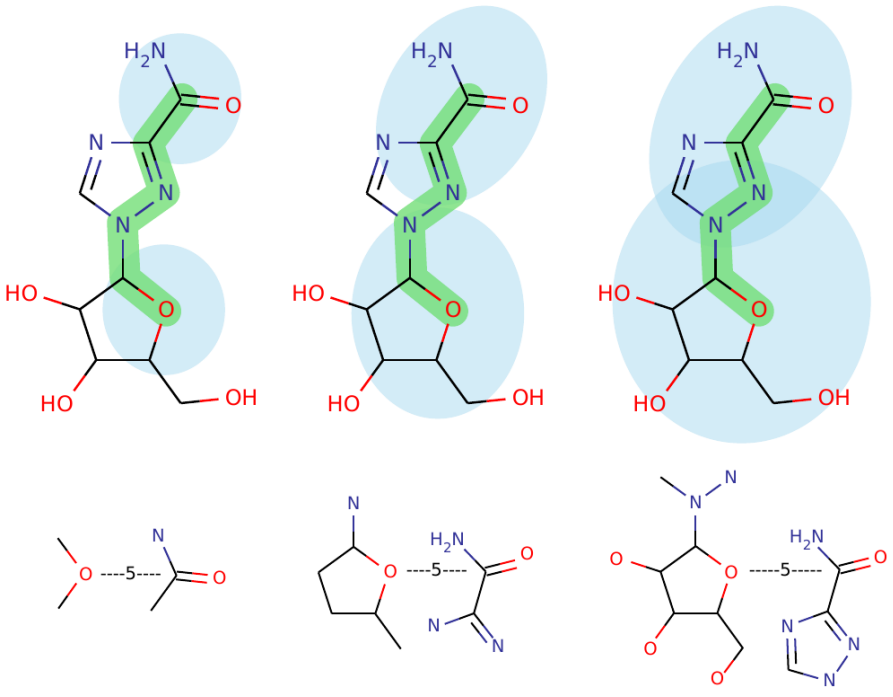


**Figure 3.1**: Example of some features generated from the application of the NSPDK kernel on an example graph representing a chemical compound.

### 3.6.8 Other graph kernels

In this section we will mention other kernels for graphs from the literature. The paper [76] proposes a kernel for reaction function prediction that is based on subpaths and is particularly efficient to compute thanks to the adoption of an efficient compressed path index. In the papers [126, 127] the author proposes a new framework for the definition of kernels for structured data. This framework is a generalization of the mapping kernel framework presented in Section 3.4.1. The partitionable kernels framework allows for the definition of new kernels for structured data with a different formulation respect the existing ones, maintaining the properties of recursive computation of the kernels. In the future, more kernels may be defined according to this framework.

# Part II

# Original Contributions

# Chapter 4

# A new framework for the definition of DAG-based graph kernels

> There are two possible outcomes: if the result confirms the hypothesis, then you've made a measurement. If the result is contrary to the hypothesis, then you've made a discovery.
>
> Enrico Fermi

As stated in Section 3.6, among all machine learning methods defined on structured data, kernel methods have important advantages because input instances are mapped in a (large) space only implicitly. This implicit mapping can bring important computational benefits with respect to other techniques that explicitly map examples in high-dimensional vectors.

We stated also that, thanks to the modularity of these methods, for a successful application on structured datasets it suffices to define a good kernel function for the task, that is a kernel that drops as little useful information (for the task) as possible, while being efficient from a computational point of view.

In general, it is not possible to define a universally good kernel function, because of the no free lunch theorem [152] that states that no learning algorithm can achieve

good performance in all tasks because, for making learning from examples possible, one should make strong assumptions on the data.

For this reason, different kernels for structured data have been defined, each one achieving good performance on specific application areas, and each one having different (but still polynomial) complexities (see Section 3.6).

The majority of the kernels that have been proposed in literature are based on walks or subtree patterns, that are subtrees where each vertex can appear multiple times (similar to the difference between paths and walks). Walk-patterns have been considered because of the NP-hardness of kernels based on paths or other subgraphs (see Section 3.6). This approach leads to tractable problems, but there is a major problem related to the feature spaces of these kernels commonly referred as *tottering*, i.e. a small subset of nodes can be visited several times, inducing artificially high similarity values between two graphs.

These kernels shown good results on some tasks, but in general the tottering problem badly affects the predictive performance of the kernel. Other kernels have been proposed that, considering simpler substructures such as shortest paths (see Section 3.6.4),but avoiding the tottering problem. On the other hand, often the considered structures are too simple resulting in poor predictive performance.

In general, the challenge with graph kernels is the unavoidable tradeoff between the efficiency and the expressive power of the kernel.

The research proposed in this chapter aims at improving the state-of-the art on kernel methods for graphs defining kernels that are more expressive compared to existing ones, with the smallest possible increase in computational complexity.
Moreover, we propose a framework for the definition of graph kernels, with various instantiations in order to have appropriate kernels for different application areas.
Among state-of-the art kernels for graphs, the most effective one (among those that have an acceptable computational complexity) seems to be the WL-subtree kernel (see Section 3.6.6). This kernel outperforms walk kernels in all the datasets where it has been tested. This gives the intuition that subtree-patterns are in general more expressive than other decompositions.

Following this intuition, a first idea for developing new graph kernels may be to define a kernel framework that allows for the definition of multiple kernels. With the possibility to instantiate several kernels from the framework, it is possible to balance the tradeoff between the expressivity and efficiency of the resulting kernel, depending on the specific task and on the required generalization performance.

In this field, the theoretical soundness of a kernel function has to be supported by experimental evidence.

However, for many real-world problems we do not know which kind of structural features are relevant for the specific task. In such cases, the definition of an effective kernel needs to pass through a trial-and-error step. Moreover, it is not uncommon for an expressive kernel to perform poorly on some tasks because the class of features it considers are not-relevant for the task.

This chapter is organized as follows. In Section 4.1-4.4 we present the first contribution, that is a new family of graph kernels. Moreover, in Section 4.3.1 we propose a way to speed up the kernel computation using hash tables. In Section 4.5 we will analyze how to apply feature selection techniques in order to make the final model more compact.

## 4.1 A new DAG-based kernel framework for graphs

In this section we propose a work in the direction of defining new kernels that outperform the state-of-the-art in terms of expressivity.

The main idea is to define a new kernel framework for graphs that allows to define both expressive and efficient kernels.

These kernels decompose a graph $G$ into a (multi)set of DAGs, guaranteeing that isomorphic graphs are represented exactly by the same (multi)set (this is important for the function to be a valid kernel). The kernels for graphs are defined as the sum of the values of a local kernel for DAGs, over all pairs of DAGs in the multisets. Different instantiations of the DAGs kernel result in different graph kernels. Since no kernel specifically designed for DAGs was described in literature, the adopted

strategy was to extend the definition of a well known class of tree kernels (see Section 3.5.2) to the DAG domain, via an ordering between DAG vertices and the extension of tree kernels on the DAG domain.

The following sections are divided into two parts. The first one describes a framework for exploiting tree kernels in a graph domain. First the decomposition of a graph into a set of DAGs is described in Section 4.1.1 and then the extension of a class of tree kernels to the DAG domain is discussed in Section 4.2. Optimizations generally applicable to the framework are discussed for the single kernel evaluation in Section 4.2.3 and for computing the kernel matrix in Section 4.2.4.

The second part focuses on two instances of the framework: one which is based on the Subtree kernel for trees and one based on a novel kernel, first introduced in this thesis. Section 4.3.1 shows specific optimizations for the Subtree kernel. Section 4.3.2 introduces a graph kernel based on the novel tree kernel. A theoretical comparison between the proposed kernels and the state-of-the-art ones (Fast Subtree kernel and gBoost) is reported in Section 4.3.3. A set of experiments on real benchmark datasets is described in Section 4.4. These kernels shown promising results in terms of classification performance on real-world datasets and in terms of computational complexity, giving evidence that it is possible to define flexible kernels that return state-of-the-art results on different graph domains while preserving competitive computational performances.

The section is a significantly reorganized and extended version of [44]: it adds a novel tree kernel and consequently a novel instance of the framework. Theoretical discussion as well as practical evaluation on novel real world datasets is provided.

### 4.1.1 Decomposition of a graph into DAGs and derived graph kernels

The first step of the framework we propose for the definition of kernels for graphs consists in decomposing the graph into a set of DAGs:

1. given a graph $G$, for each $v \in V_G$, one unordered rooted DAG, say $DD_G^v$, is generated. $G$ is then represented by the multiset of unordered rooted DAGs.

2. The kernel for graphs is defined as the sum of the evaluations of a local kernel for DAGs, over all pairs of DAGs in the multiset.

We first describe our proposal for decomposing a graph $G$ into $|V_G|$ DAGs, one for each $v \in V_G$. A requirement of this phase is for isomorphic graphs to be represented by exactly the same multiset of DAGs. The decomposition is based on a simplification of the graph isomorphism testing proposed by [61]. Given a node $v_i$, the idea is to keep in $DD_G^{v_i}$ all nodes of $G$ and those edges belonging to the shortest paths between $v_i$ and any $v_j \in V_G$. An efficient way to build $DD_G^{v_i} = (V_G, E_G^{v_i}, L)$ is to perform a breadth-first visit on the graph starting from node $v_i$ and applying the following rules:

1. during the visit a direction is given to each edge, if $v_j$ is reached from $v_i$ in one step, then $(v_i, v_j) \in E_G^{v_i}$ (note that edge $(v_j, v_i)$ is not added to $E_G^{v_i}$);

2. edges connecting nodes reached at level $l$ of the visit to nodes reached at level $g < l$ are not added to $E_{v_i}$. Such edges would induce a cycle in $DD_G^{v_i}$.

For every choice of $G$ and $v_i$, a single *Decompositional Dag* $DD_G^{v_i}$ is generated. By repeating the procedure for each node of the graph, $|V|$ DAGs are obtained. Figure 4.1 shows the four $DD$s obtained from the undirected graph in Figure 4.1-a. Note that when the same node is reached simultaneously (at the same level of the visit) from different nodes, then all involved edges are preserved. For example, when considering the visit at level 2 starting from node **s**, the node **d** is reached simultaneously by edges $(\mathbf{b}, \mathbf{d})$ and $(\mathbf{e}, \mathbf{d})$, and both of them are preserved in the corresponding Decompositional DAG (see Figure 4.1-b). On the contrary, edge $(\mathbf{b}, \mathbf{e})$ is not inserted into the corresponding Decompositional DAG: when the visit at level 2 attempts to traverse it (either starting from node **b** or **e**) a node already visited at level 1 (i.e., **e** or **b**, respectively) is reached.

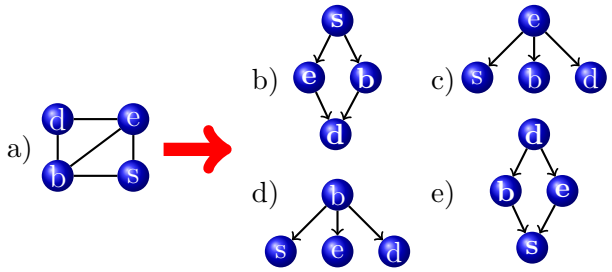


**Figure 4.1**: Example of decomposition of a graph a) into its 4 DDs b-e).

Some edges might not be added to a $DD_G$ because they would induce a cycle. However, $|V_G|$ DAGs are created from $G$, each one related to a different starting node of the breadth-first visit. Thus, every undirected edge $e_{ij}$, which is not a self-loop, will appear at least in two $DD_G$, i.e. $DD_G^{v_i}$ and $DD_G^{v_j}$.

**Theorem 4.1.** *Given two graphs $G_1$ and $G_2$, define the multisets of Decompositional DAGs associated to $G_1$ and $G_2$ as $DD(G_1) = \{DD_{G_1}^{v_i} | v_i \in V_{G_1}\}$ and $DD(G_2) = \{DD_{G_2}^{v_j} | v_j \in V_{G_2}\}$. If $G_1$ and $G_2$ are isomorphic, then $DD(G_1)$ and $DD(G_2)$ are identical.*

*Proof.* $G_1$ and $G_2$ are assumed to be isomorphic with respect to a function $f : V_{G_1} \to V_{G_2}$. Since the set of nodes of the $DD_G$s are identical by construction, it remains to be shown that the set of edges is identical. We prove the theorem by contradicting the thesis: we assume that $(v_i, v_j) \in E_{G_1}^{v_1}$ and $(f(v_i), f(v_j)) \notin E_{G_2}^{f(v_1)}$ (or vice-versa). If $(f(v_i), f(v_j)) \notin E_{G_2}^{f(v_1)}$, there is a shorter path connecting $f(v_1)$ to $f(v_j)$. Since $G_1$ and $G_2$ are isomorphic the corresponding path from $v_1$ to $v_j$ in $G_1$ must be shorter than the one comprising the edge $(v_i, v_j)$. This contradicts the fact that $(v_i, v_j) \in E_{G_1}^{v_1}$, i.e. the edge $(v_i, v_j)$ belongs to the shorter path connecting $v_1$ and $v_j$. □

Note that the DAGs resulting from the decomposition are not ordered.

Let assume a positive definite (PD) kernel $K_{DAG}$ for DAG is available. Then, we can define a kernel for graphs as follows:

$$DDK_{K_{DAG}}(G_1, G_2) = \sum_{\substack{D_1 \in DD(G_1) \\ D_2 \in DD(G_2)}} K_{DAG}(D_1, D_2). \tag{4.1}$$

The above kernel is positive semidefinite since it is an instance of the convolution kernel framework (see Section 3.4), where the relation $R$ is defined on $X_1 \times \ldots \times X_{|V|} \times G$, with $X_1 \times \ldots \times X_{|V|}$ being the multiset of Decompositional DAGs obtained from $G$. We will refer to this general class of kernels as $DD$ kernels.

In the following, we define a family of positive definite kernels for DAGs.

## 4.2 Extending tree kernels to DAGs

Equation (4.1) reduces the computation of a kernel for graphs to the combination of a set of kernels for DAGs. At the time of writing, few papers in literature propose kernels for DAGs [133, 117, 55]. The kernel in [133] is defined as the inner product of the common attribute sequences, that are the sequences obtained concatenating the labels of the nodes that appear in the sub-paths of the graph. Its complexity is $O(kn^2)$, where $k$ is the maximum length of the considered sub-paths. Thus, the resulting DDK kernel presented in Equation 4.1 would have a complexity of $O(kn^4)$, that is too high for our goal of having computational complexity comparable with the fastest state-of-the-art kernels (as stated in the introduction of this Chapter). Another kernel proposed in literature [55] exploits the same idea but with $O(n^3)$ complexity. Moreover, the the feature space of the resulting kernel would be very similar to the subpath kernel presented in Section 3.6.4. The third existing kernel, proposed in [117], calculates the sum of kernel values for all pairs of possible substructures of two input graphs. This kernel is applicable in our scenario, but the computational complexity of every kernel calculation is $O(\rho^2 n^2)$, that would result in a computational complexity of $O(\rho^2 n^4)$ for the resulting DDK kernel presented in Equation 4.1.

In order to derive faster alternatives for the kernels present in literature, the strategy we propose is to extend the definition of tree kernels to the rooted DAG domain. Our proposal owns the additional benefit that a large number of kernels for trees is already available. However, while there is a vast literature on kernels for ordered trees (see Section 3.5.2), just a few kernel functions for unordered trees are

defined (see Section 3.5.1). In order to broaden the applicability of our approach, we define an ordering between DAG nodes (Section 4.2.1) such that kernels for ordered trees can be applied, too. In Section 4.2.2 it is shown how to extend the definition of the most popular class of kernels for ordered trees, the convolution kernels, to the DAG domain. One nice aspect of the DAG-kernels proposed in this chapter is that, by representing the multiset of DAGs generated by a graph $G$ via an annotated DAG (BigDAG), where sub-structures shared by DAGs in the multiset are represented only once and their frequency of occurrence is annotated, the computation of the kernel can be sped up for common classes of graphs. In Section 4.2.3 we formally characterize the computational gain by providing a general upper bound on the number of nodes of the BigDAG and verify, on popular graph datasets, that the actual computational burden is far from the worst case expected complexity.

### 4.2.1 Ordering DAG vertices

The procedure described in Section 4.1.1 yields a multiset of Decompositional DAGs from a graph. However, such DAGs are not ordered. Despite the fact that it has been demonstrated that the computation of any tree kernel for unordered trees with subtrees as their sub-structures is $\sharp P$-complete [83], some kernels for unordered trees have been defined in literature [87, 142]. Such kernels, in order to be tractable, tend to restrict the type of sub-structures used as features, thus potentially limiting their expressiveness. Indeed, a set of experiments performed with the kernel in [87] on the datasets described in Section 4.4 show that the kernel never improves, in terms of classification performances, over the state of the art (for this reason we do not report the results in the experimental section).

In order to broaden the applicability of our approach, in this section we define a *strict partial order* among vertices of a DAG which allows us to employ the kernels defined in Section 3.5.2. The ordering makes use of a unique representation of subtrees as strings inspired by [142]. Here we modify such mapping by employing perfect hash functions to encode subtrees. As a consequence we obtain a compressed subtree representation, which will be also used in section 4.3.1 to implement one of

the kernels proposed in this chapter. The compressed representation is here used to define an ordering between DAG vertices:

**Definition 4.1.** ***Strict partial order relation $\dot{<}$ between DAG vertices*** *Let $\kappa()$ be a perfect hash function and $\#, \lceil, \rfloor \notin \mathcal{A}$ be symbols never appearing in any node label, then*

$$\pi(v) = \begin{cases} \kappa(L(v)) & \text{if } v \text{ is a leaf node} \\ \kappa\big(L(v)\lceil \pi(ch_v[1])\#\pi(ch_v[2])\ldots\#\pi(ch_v[\rho(v)])\rfloor\big) & \text{otherwise,} \end{cases}$$

*where the children of $v$ are recursively ordered according to their $\pi()$ values. Then $v_i \dot{<} v_j$ if $\pi(v_i) < \pi(v_j)$, where $<$ is the relation of order between alphanumeric strings.*

Notice that $\pi(v_i) = \pi(v_j) \Leftrightarrow \neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$, i.e. $\pi(v_i) = \pi(v_j)$ if and only if the nodes $v_i$ and $v_j$ are not comparable. In such case, many orderings for non comparable children nodes in Definition 4.1 are possible. We will show in Section 4.2.2 that any of them yields the same representation in terms of features and thus the related kernel functions for graphs are well defined. For this reason we avoid to give a specific ordering between non comparable vertices. If two vertices $v_i$, $v_j$ have different node labels or at least two children $ch_{v_i}[l]$, $ch_{v_j}[m]$ (possibly $l = m$) have a different representation, then the use of the symbols $\#, \lceil, \rfloor$ and the fact that $\kappa()$ is perfect ensures that they can not have the same $\pi()$ value (a detailed demonstration in the case $\kappa()$ is the identity function can be found in [142]). The ordering between $\pi()$ values is a strict partial order, i.e. irreflexive, transitive and asymmetric, since it is based on the alphanumeric ordering $<$ of strings. Then the ordering $\dot{<}$ between DAG vertices described in Definition 4.1 is also a strict partial order.

We now show that if two DAGs $DD_{G_1}^{v_i}$ and $DD_{G_2}^{v_j}$ are isomorphic, then the root nodes of the DAGs are not comparable with respect to the ordering in Definition 4.1:

**Theorem 4.2.** *if two DAGs $DD_{G_1}^{v_i}$ and $DD_{G_2}^{v_j}$ are isomorphic, then $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$.*

*Proof.* Let $f : V_{G_1} \to V_{G_2}$ be an isomorphism between $DD_1^{v_i}$ and $DD_2^{v_j}$. We prove the thesis by induction. Let $f(v_i) = v_j$, since the nodes are isomorphic $L(v_i) = L(v_j)$. If $v_i$ and $v_j$ are leaf nodes, then $\pi(v_i) = \pi(v_j)$ and consequently $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$. Otherwise, by inductive hypothesis $\forall l.1 \leq l \leq \rho(v_i).\ \pi(ch_{v_i}[l]) = \pi(ch_{f(v_i)}[l])$ and $L(v_i) = L(f(v_i))$, thus $\pi(v_i) = \pi(f(v_i)) = \pi(v_j)$. □

In the following, we will denote as $ODD_G^{v_i}$ the Decompositional Dag rooted at $v_i$ and ordered according to Definition 4.1. If the $\pi()$ values are computed according to a post order visit of the DAG, then the values $\pi(ch_v[l])$ for $1 \leq l \leq \rho(v)$ are already available for computing $\pi(v)$. Thus the time complexity of the ordering phase of the DAG is $O(|V|\rho \log \rho)$ where the term $\rho \log \rho$ accounts for the ordering of the children of each node.

The reason for defining the ordering according to Definition 4.1 is that it must be efficient to compute and must ensure that the swapping of non comparable nodes does not affect the feature space representation of the DAG. After introducing the extension of tree kernels to DAGs in Section 4.2.2, we will show that our ordering meets both constraints.

### 4.2.2 Tree-based kernels for ordered DAGs and graphs

Here we define a family of kernels for ordered DAGs based on tree-kernels. The basic idea is to project sub-DAGs to a tree space and then apply a tree kernel on the projections.

Let a tree visit be a function $T(v)$ that, given a node $v$ of a $ODD_G^{v_i}$, returns the tree resulting from the visit of the DAG starting from $v$. Figure 4.2 gives an example of tree visits. Note that the trees resulting from the visit starting in node **s** are identical for the DAGs in Figure 4.2-a and Figure 4.2-b even if the DAGs are not isomorphic. However, the DAGs in Figure 4.2-a and Figure 4.2-b can be discriminated if we consider the multiset of trees returned by the tree visits starting in all nodes of the DAGs: one tree is produced by the visit starting in node **d** of the DAG in Figure 4.2-a and two trees are generated by the two visits starting in

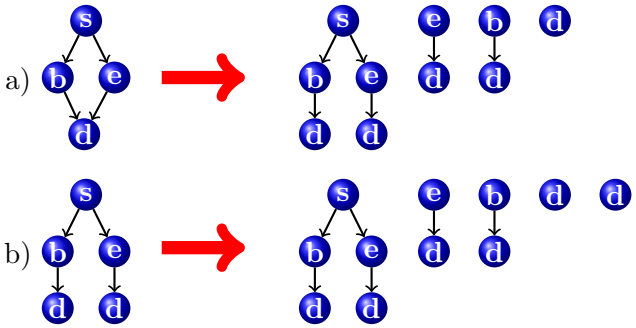


**Figure 4.2**: Two DAGs (left) and their associated tree visits $T()$ starting from each node.

the nodes labeled with **d** in Figure 4.2-b. Moreover, note that the set of tree visits would be different even if the nodes labeled with **d** were not leaves. The concept of tree visit is used in the following

**Theorem 4.3.** *Given the ordering $\dot{<}$ in Definition 4.1, $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$ if and only if $T(v_i)$ and $T(v_j)$, obtained as visits of the sub-DAGs rooted at $v_i$ and $v_j$, are identical.*

*Proof.* If $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$ then $\pi(v_i) = \pi(v_j)$. Recalling that $\kappa()$, the function on which $\pi()$ is based on, is a perfect hash function, we prove the thesis by induction. If $v_i, v_j$ are leaf nodes, then $\pi(v_i) = \pi(v_j) \Leftrightarrow L(v_i) = L(v_j)$. If $v_i, v_j$ are not leaf nodes, then $\forall l.1 \leq l \leq \rho(v_i)$ $T(ch_{v_i}[l])$ is identical to $T(ch_{v_j}[l])$ for inductive hypothesis, and then it must be $L(v_i) = L(v_j)$ since $\pi(v_i) = \pi(v_j)$; therefore $T(v_i)$ is identical to $T(v_j)$. Now we show that if $T(v_i)$ is identical to $T(v_j)$, then $\pi(v_i) = \pi(v_j)$ by induction. The base case has already been proved by the equality $\pi(v_i) = \pi(v_j) \Leftrightarrow L(v_i) = L(v_j)$. By inductive hypothesis $\pi(ch_{v_i}[m]) = \pi(ch_{v_j}[m])$ for each child $m$ of $v_i$ and $v_j$. Then $\pi(v_i) = \pi(v_j)$ and $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$. □

If we consider the tree kernels defined in Section 3.5.2, then we can write

$$KDAG_{K_T}(D_1, D_2) = \sum_{\substack{v_1 \in V_{D_1} \\ v_2 \in V_{D_2}}} C(root(T(v_1)), root(T(v_2))), \tag{4.2}$$

where $root()$ is a function returning the root node of a tree, $T()$ are tree-visits and the $C()$ function is the one related to the corresponding tree kernel $K_T$. Theorem 4.3 ensures that the mapping in feature space of each tree visit is well defined since the swapping of any two non comparable children of a node does not change the resulting tree visit. This, in turn, ensures that:

- $C(root(T(ch_{v_1}[i])), root(T(ch_{v_2}[j]))) = C(ch_{root(T(v_1))}[i], ch_{root(T(v_2))}[j])$, for every $1 \leq i \leq \rho(v_1), 1 \leq j \leq \rho(v_2)$, which allows us to use the algorithms described in [36, 102], thus significantly reducing the computational burden;
- $C()$ remains a valid local kernel.

Given two isomorphic DAGs $DD^{v_i}_{G_1}$ and $DD^{v_j}_{G_2}$, by applying Theorem 4.2 and Theorem 4.3 it can be concluded that $DD^{v_i}_{G_1}$ and $DD^{v_j}_{G_2}$ yield the same set of tree visits and thus have the same representation in feature space for each tree kernel. Finally, the kernel of Equation (4.2) is positive semidefinite since it is an instance of convolution kernel framework [73], where the relation $R$ is defined on $X_1 \times \ldots \times X_{|V|} \times ODD$, with $X_1 \times \ldots \times X_{|V|}$ being the set of tree-visits obtained from $ODD$. The feature space, i.e. the set of features associated with the kernels of Equation (4.2), coincides with the one of the tree kernel $K_T$. However, note that a DAG with a "diamond" shape, such as the DAG in Figure 4.2-a, has a different representation with respect to the DAG of Figure 4.2-b: while the non-zero features are the same, the feature related to the leaf node **d** occurs once in Figure 4.2-a, while it occurs twice in Figure 4.2-b.

After introducing the DAG-kernels used in this section, we can now motivate why the ordering in Section 4.2.1 has been employed. We recall that the requirements for the ordering are to be efficiently computable and to ensure that the swapping of non comparable nodes does not affect the feature space representation of the DAG. More "standard" orderings would not meet both constraints. For example, a total order between DAG vertices would require an algorithm with GI-complete computational complexity [113]. On the contrary, simple partial orderings, such as the one based on reachability of the nodes, would not allow us to consistently order structurally

different sub-DAGs. This makes impossible to have a consistent mapping in feature space for any kernel whose features are tree structures, which basically includes the vast majority of tree kernels.

On the basis of Equation (4.2), given a tree-kernel $K_T$, we can define a kernel $K_{K_T}$ between two graphs $G_1$ and $G_2$ as follows:

$$ODDK_{K_T}(G_1, G_2) = \sum_{\substack{D_1 \in ODD(G_1) \\ D_2 \in ODD(G_2)}} KDAG_{K_T}(D_1, D_2) \tag{4.3}$$

where $ODD(G)$ is the multiset $\{ODD_G^{v_i} | v_i \in V_G\}$, $KDAG$ is defined according to Equation (4.2) and depends on a kernel for trees $K_T$. In the following, we refer to this class of kernels as $ODD$ kernels. As we noted in Section 4.1.1, self-loops are ignored during the decomposition from graph to multiset of DAGs. However, information about self-loops can be incorporated into node labels. For example, let $* \notin \mathcal{A}$ be a symbol not appearing in any node label. Then, the label of any node with a self-loop can be concatenated with $*$ in order to distinguish it from a node with the same label but without self-loop. From a computational point of view, let $Q(n)$ denote the worst-case complexity of the tree kernel $K_T$, where $n = \max_{\substack{D_1 \in ODD(G_1) \\ D_2 \in ODD(G_2)}} \{|V_{D_1}|, |V_{D_2}|\}$, then the kernel of Equation (4.3) has time complexity $O(|V_{G_1}||V_{G_2}| \cdot Q(n))$. Thus, in the worst case where $n = \max\{|V_{G_1}|, |V_{G_2}|\}$, using ST, SST, and PT as tree-kernels, leads to a time complexity of $O(n^3 \log n)$, $O(n^4)$ and $O(\rho^3 n^4)$, respectively.

In Sections 4.2.3-4.2.5 we show how the computation of $ODDK_{K_T}(G_1, G_2)$ can be optimized.

### 4.2.3 Speeding up the single kernel evaluation

We now show how to speed up the computation of Equation (4.3) when $K_T$ is a convolution kernel where the input tree is decomposed into proper subtrees. All the kernels described in Section 3.5.2 and the vast majority of tree kernels defined in literature meet this constraint. The strategy for speeding up kernel computation is based on avoiding to recompute $C()$ values for identical proper subtrees appearing

in different DAGs. The $|V_G|$ Decompositional DAGs generated according to the procedure described in Section 4.1.1 can be represented by a single Annotated DAG, named $BigDAG$, where each node is annotated with the frequency of appearance of the proper subtree rooted at $v$, i.e. $T(v)$, in all the ODDs of the graph. An example of $BigDAG$ construction is shown in Figure 4.3. In the figure, the subtree rooted in **b** in the two graphs are merged together in the $BigDag$ since all of their descendant nodes, i.e. **c**, are identical. Note that, on the contrary, the nodes labeled with **a** are not merged together because, in one case the second child of **a** is **d** and in the other the second child is the subtree rooted at **c**. The algorithm for computing the

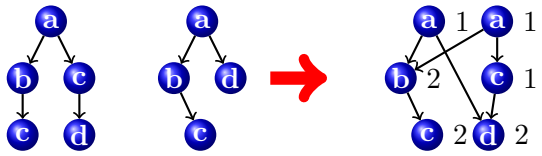


**Figure 4.3**: Two DAGs (left) and their associated $BigDAG$. Numbers on the right of the nodes represent their frequencies.

$BigDAG$ can be found in [5]. Its complexity is $O(|V_G|^2 \log |V_G|)$, thus it does not affect the worst-case complexity of any of the kernels considered here. The kernel of Equation (4.3) can be rewritten as

$$ODDK_{K_T}(G_1, G_2) = \sum_{\substack{u_1 \in V(BigDAG(G_1)) \\ u_2 \in V(BigDAG(G_2))}} f_{u_1} f_{u_2} C(u_1, u_2), \tag{4.4}$$

where $f_u$ is the frequency of the ordered DAG rooted at $u$ in $ODD(G)$. We want to stress the fact that the $BigDAG$ does not loose any information about the ODDs, thus Equation (4.4) and Equation (4.3) are equivalent. The speed up due to the $BigDAG$ depends on the number of identical sub-structures found in the ODDs. We now derive an upper bound on the number of nodes of the $BigDAG$ in terms of the in-degree of the nodes and the size of the cycles in the original graph $G$. We first derive the bound for a specific class of graphs and then extend it to general graphs.

Without loss of generality, we will assume in the following to deal with connected and undirected graphs. Let's consider a polytree $P$, i.e. a graph for which at most one undirected path exists between any two nodes. Let us consider any node $v_i \in P$ with degree $\rho(v_i)$. The procedure for obtaining a Decompositional DAG, when applied to all the nodes of $P$, generates exactly $\rho(v_i)+1$ different patterns of connectivity for $v_i$. In fact, since $P$ is a polytree, only 1 edge at a time can be entering $v_i$ while all the remaining edges are outgoing. Moreover, we have to consider the case where $v_i$ is the starting node for the visit and all the edges are outgoing. These considerations are valid for all nodes belonging to $P$, thus $BigDAG$ will exactly have $\sum_{i=1}^{|P|}(\rho(v_i)+1)$ nodes.

Let's now turn our attention to general graphs. Given a graph $G$ we can represent it as a polytree $P(G)$ by iteratively grouping the nodes forming local cycles into a single node $v \in P(G)$ until no more cycles can be grouped (see Figure 4.4 for an example). Let us define $n_{v_i}$ as the number of nodes of $G$ represented by $v_i$ in $P(G)$ and $q = |P|$. The values $\rho(v_i)$ represent now the sum of the incoming edges of all the nodes represented by $v_i$. Then, the bound can be computed by applying the same reasoning for polytrees, the only difference being that $v_i$ contains $n_{v_i}$ choices for starting the visit from a node "inside" $v_i$, thus generating $n_{v_i}$ different substructures with, in the worst case of a single cycle involving all the nodes of $G$ represented by $v_i$, a total of $n_i^2$ nodes added to the $BigDAG$. The number of nodes added to the $BigDAG$ is thus bounded by:

$$|BigDAG(G)| \leq \sum_{i=1}^{q}(\rho(v_i)+n_{v_i}^2). \tag{4.5}$$

Note that $|BigDAG(G)| \leq |V_G|^2$, being the worst case when there's only one cycle of length $|V_G|$. Let us define the length of the longest cycle in $G$ to be $o$. The bound then becomes $o|V_G|+\sum_i \rho(v_i)$, since $\sum_i n_{v_i} = |V_G|$. Note that $\sum_i \rho(v_i) \leq 2q$ otherwise $P(G)$ wouldn't be a polytree. Just to give an example, if $o = |V_G|^{\frac{1}{2}}$, then $|BigDAG(G)| \leq 2q + \sum_i n_{v_i} n_{v_i} \leq 2q + |V_G|^{\frac{1}{2}} \sum_i n_{v_i} = 2q + |V_G|^{\frac{3}{2}}$.

Note that if the number of nodes in the $BigDAG$ is $O(|V_G|^{\frac{3}{2}})$, then the complexity of the kernels reduces significantly, for example the application of the subset tree

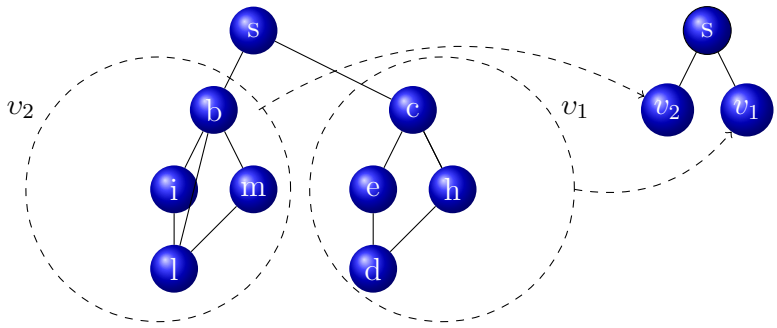


**Figure 4.4**: Example of a polytree representing a graph with two "complex" polytree nodes $v_1$ and $v_2$ representing local cycles.

kernel would have a complexity of $O(|V_G|^3)$, thus reducing the complexity of a factor $|V_G|$. Section 4.4 will discuss the actual reduction of the computational burden due to the *BigDAG* on benchmark datasets.

### 4.2.4 Speeding up the kernel matrix computation

When using a kernel method, such as the SVM, the tuning of the hyperparameters through a validation set or through cross-validation is common practice. If the kernel function depends on multiple parameters, then it is often more efficient to precompute the kernel matrix of the dataset[1] than computing the kernel values on demand. Other kernel methods, e.g. KPCA, require the full kernel matrix to be computed.

In this section, we discuss how the computation of the full kernel matrix can be optimized. The *BigDAG* has been used to avoid multiple calculations of kernel values for the same structures in different ODDs. The same idea can be extended to the whole training set avoiding multiple calculations of the kernel values for the same structures in different examples. Note that sub-structures shared by different graphs *should* exist, otherwise the kernel would be extremely sparse for the considered training set, and not worth to be used for learning. Let's assume that the training

[1]The $i, j$ element of the kernel matrix corresponds to the evaluation of the kernel function on the $i$-th and $j$-th examples of the dataset

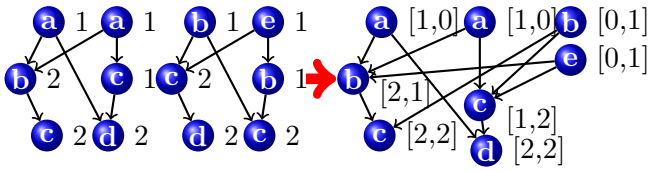


**Figure 4.5**: Two $BigDAGs$ (left) and their associated $Big^2DAG$. Arrays on the right of nodes represent the frequencies associated to each example.

set contains $M$ graphs. Then, we can obtain a compact representation of all the $BigDAGs$ coming from the $M$ graphs by applying the same algorithm for building the $BigDAG$ starting from a multiset of ODDs, the only difference being that, while a sub-DAG in a $BigDAG$ may have a frequency greater than one. The idea is implemented as follows. An annotated DAG, we call it $Big^2DAG$, is created starting from all the $BigDAGs$ generated by the training set, where each different structure is represented only once. A (sparse) vector $F_{v_i}$, representing the frequency of the structure rooted at that node in all graphs, is associated to each node of the $Big^2DAG$. For example, $F_{v_i}[j]$ represents the frequency of the structure rooted at node $v_i$ in $BigDAG(G_j)$. Figure 4.5 shows an example of $Big^2DAG$ construction. While the frequency associated to each node in $BigDag(G_i)$ is the frequency of the proper subtree rooted at that node in all the ODDs related to the graph $G_i$, in the same way, the vector $F_{u_i}$ represents the frequencies of the proper subtree rooted in node $u_i$ in the various training graphs. When the graphs $G_i$ and $G_j$ have been used to construct the $Big^2DAG$, eq. (4.4) can be rewritten as:

$$ODDK_{Big^2DAG}(G_i, G_j) = \sum_{u_1,u_2 \in V(Big^2DAG)} F_{u_1}[i] * F_{u_2}[j] * C(u_1, u_2). \tag{4.6}$$

Since eq. (4.4) corresponds to the entry $i, j$ of the kernel matrix, in order to compute the whole kernel matrix efficiently it is sufficient to construct the $Big^2DAG$ from all the graphs in the training set and then apply eq. (4.4) to any pair of graphs in the training set.

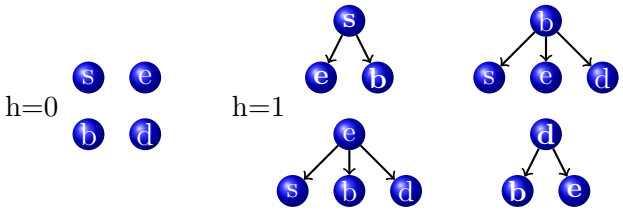


**Figure 4.6**: Example of decomposition of the graph shown in Figure 4.1-a) into its DDs with visits up to depth $h = 1$.

### 4.2.5 Limiting the depth of the visits

It is a common practice in many graph kernels (see Section 3.6 for some examples) to limit the features that are generated in order to reduce the computational burden when evaluating the kernel. We apply this approach by limiting the depth of the visits during the generation of the multiset of DAGs. Our aim is

1. to further reduce the computational complexity of the kernel when large graphs are involved;

2. moreover, for some tree kernels, to add features that otherwise would be discarded if only unlimited visits were performed. Examples of such kernels are the *subtree kernel* and the one presented in Section 4.3.2.

The above idea is implemented as follows. Given a graph $G$, for any $j \in \{0..h\}$ and for any $v_i \in V_G$, an ODD is generated with the additional constraint that the maximum depth is at most $j$ (we will refer to it as $ODD_G^{v_i,j}$). All the $ODD_G^{v_i,j}$ are merged together and form the $BigDAG(G)$ as described in Section 4.2.3. The kernel definition is the same as in eq. (4.3), the only difference being that $ODD(G)$ is now replaced by $\{ODD_G^{v_i,j} | v_i \in V_G, j \in \{0 \dots h\}\}$. This kernel is referred to as $K_{ODD-ST_h}$ in the following. Notice that, while new features are introduced when kernels such as ST are employed and the visits are limited, all DAGs generated by visits of depth $j > h$ are lost (see Figure 4.6).

Given a node, in the worst case no more than $O(\rho^h)$ nodes are generated by all the visits up to depth $h$. In fact, the total number of generated nodes is

$$H = \sum_{j=0}^{h}(h+1-j)\rho^j = \frac{(h+1)-(h+2)\rho+\rho^{h+2}}{(1-\rho)^2}.$$

Therefore the $BigDAG$ of a graph cannot have more than $nH$ nodes[2]. The time complexity of $BigDAG$ creation is thus dominated by a $O(nH\log(nH))$ term. If $\rho$ is constant, then $H$ is constant as well and the complexity reduces to $O(n\log n)$. Note that limiting the visits allows us to apply the kernel for graphs in Equation (4.3) with any tree kernel in Section 4.2.2 having the same worst case complexity of the corresponding tree kernel: for example, when computing eq.Equation (4.3) with the *subtree kernel* the total worst case complexity is $O(n\log n)$.

## 4.3 Two graph kernels based on the framework

Various instances of the proposed framework can be obtained from the kernels described in Section 3.5.2. However, a naive application of such kernels may lead, on most datasets, to graph kernels not competitive from the computational point of view. This section, instead, focuses on obtaining fast graph kernels. Specifically, two instances of the framework are presented in the following: Section 4.3.1 focuses on the graph kernel based on the ST Kernel and show how to optimize its computation. Section 4.3.2 introduces a novel tree kernel, which is more expressive than ST.

### 4.3.1 A graph kernel based on the Subtree Kernel

In the previous section we discussed how the $BigDAG$, and consequently the $Big^2DAG$, can improve the speed of the kernel matrix computation. If the tree kernel $K_T$ in Equation (4.4) is the *subtree kernel*, the $BigDAG$ yields an explicit (and compact)

[2]It is worth to notice that the bound is not tight, since the same leaf nodes are generated in turn by visits of neighbours nodes with depth 0 and 1.

representation of the feature space of the kernel: by construction each node $u$ is associated to the unique proper subtree $T(u)$ rooted at $u$. Such subtree is uniquely associated to a feature and $f_u$ is the frequency of $T(u)$ in all the ODDs related to the input graph. By definition of the *subtree kernel*, only matching identical subtrees contribute to the kernel, that is $C(u_1, u_2) \neq 0 \Leftrightarrow T(u_1) = T(u_2)$. Exploiting this property leads to an efficient implementation. In fact, if $T(u_1) = T(u_2)$, we recall from Section 3.5.2 that $C(u_1, u_2) = \lambda^{|T(u_1)|}$ and eq. (4.6) can be computed with a single scan of the $Big^2DAG$ built from $G_i$ and $G_j$:

$$ODDK_{Big^2DAG}(G_i, G_j) = \sum_{u_1,u_2 \in V(Big^2DAG)} F_{u_1}[i] * F_{u_2}[j] * C(u_1, u_2) = \sum_{u \in V(Big^2DAG)} F_u[i] * F_u[j] * \lambda^{|u|}. \tag{4.7}$$

As Equation (4.7) shows, the information needed to compute the kernel, for each feature $f$ and graph $G$, is: *i)* a representation of the proper subtree $f$; *ii)* the frequency of $f$ in $G$; *iii)* the value $\lambda^{|f|}$.

We now discuss a more compact representation for proper subtrees than the $Big^2DAG$ which keeps all relevant information for computing the kernel. Recalling that the evaluation of a kernel function corresponds to a dot product in a feature space induced by the kernel, i.e. $K(G_i, G_j) = \langle \phi(G_i), \phi(G_j) \rangle$, we consider the explicit feature space representation $\phi(G)$ of a graph $G$, as induced by the kernel in Equation (4.7): let us assume a mapping $\pi()$ from proper subtrees to indices of the vector $\phi(G_i)$ is available, if the value of the element of index $\pi(u)$ is $\phi_{\pi(u)}(G_i) = F_u[i]\lambda^{\frac{|u|}{2}}$ then

$$\sum_{u \in V(Big^2DAG)} F_u[i] * F_u[j] * \lambda^{|u|} = \sum_{u \in V(Big^2DAG)} \phi_{\pi(u)}(G_i)\phi_{\pi(u)}(G_j). \tag{4.8}$$

Since the last term in Equation (4.8) corresponds to evaluating only the non-zero elements of the feature space representation of $G_i$ and $G_j$, we can express Equation (4.8) as $\langle \phi(G_i), \phi(G_j) \rangle$ by setting $\phi_s(G) = 0$ for all $s$ such that $\nexists u \in V(G).\pi(u) = s$.

We propose a hash-based implementation of the $\phi(G)$ in Equation (4.8). The function $\pi(u)$ returning the unique id of the feature/subtree $T(u)$ is the one in Definition 4.1. Notice that, by using such $\pi()$ function to encode any DAG vertex $u$,

the size of each id, and consequently the time to compute it, is bounded by $\rho(u)$ and it is independent from the number of descendants of $u$. If the vertices are processed in inverse topological order, each feature is guaranteed to get a unique id. Notice that, since the tree kernel is ST, the number of features corresponds to the number of nodes in the $BigDAG$.

We finally show that $\phi(G)$ can be computed directly from the input graph without constructing first the $Big^2DAG$. The process is detailed in Algorithm 6, in which the sparse representation of the feature space of a graph $G$, induced by the ST kernel with visits limited to depth $h$, is computed and represented via a hash map. Given a node of the graph, first the $DD_G^{v,h}$ is constructed. The vector $X[]$ represents the encodings, according to Definition 4.1, of the subtrees resulting from the limited visits. Specifically $X_u[d]$ is the encoding of the proper subtree rooted at $u$ and obtained from a breadth-first visit limited to $d$ levels. The vector $S[]$ represents the size of the subtrees encoded by the corresponding element of $X[]$. Finally, $\phi$ stores the weights of the extracted features. The fact that in line 4 the visit is post-order and that the variable $i$ in line 9 of Algorithm 6 ranges from 1 to $\max_{v \in \triangle_u} depth(v) - depth(u)$, ensures that the $X_{ch_u[j]}[i-1]$ values of the children nodes of $u$ have been properly computed before being first accessed. Lines 5-11 compute the encoding of the subtree composed by the single node $u$, obtained from the visit limited to 0 levels, and add the corresponding element to $\phi$. If $u$ is not a leaf node, all subtrees obtained from limited visits up to $(\max_{v \in \triangle_u} depth(v) - depth(u)) \leq h$ levels, i.e. the visits termined at level $h$ or when a leaf node is encountered, are generated. The ordering of children nodes in line 14 is the one of Definition 4.1. The macro **compute-st-features** in line 15 executes the code in the bottom part of Algorithm 6, which computes the encoding of the subtree resulting from the visit limited to $i$ levels and add the corresponding entry to $\phi$.

Since the number of nodes of the DAG generated in line 3 of Algorithm 6 is at most $H$, the complexity of the algorithm is $O(|V_G|Hh\rho \log \rho)$. Evaluating the kernel between two graphs $G_1$ and $G_2$ then reduces to look for matching entries in the hash tables returned by Algorithm 6. If we consider $H$ constant, the complexity

of a kernel evaluation then is $|V_G| \log |V_G|$, where the term $\log |V_G|$ accounts for the worst case complexity for looking up a key in a hash table.

**Algorithm 6** An algorithm for computing the feature space representation of a graph $G$ according to the kernel $ST_h$.

**Input:** a graph $G$, $h$ (maximum depth of the visit)
**for each** $v \in V(G)$ **do**
  generate $DD_G^{v,h}$ as described in Section 4.1.1
  **for** $u \in$ **post-order-visit(**$DD_G^{v,h}$**) do**
    $X_u[0] = \kappa(L(u))$
    $S_u[0] = 1$
    **if** $X_u[0]$ is a new feature **then**
      $\phi_{X_u[0]} = \lambda^{\frac{1}{2}}$
    **else**
      $\phi_{X_u[0]} = \phi_{X_u[0]} + \lambda^{\frac{1}{2}}$
    **end if**
    **if** $u$ is not a leaf **then**
      **for** $1 \leq i \leq \max_{v \in \triangle^u} depth(v) - depth(u)$ **do**
        sort the children of $u$
        **compute-st-features** //see macro below
      **end for**
    **end if**
  **end for**
**end for**
**Output:** $\phi$, the set of features of $G$

**compute-st-features**

$X_u[i] = \kappa\Big(L(u)\Big\lfloor X_{ch_u[1]}[i-1] \# X_{ch_u[2]}[i-1] \ldots \# X_{ch_u[\rho(u)]}[i-1]\Big\rfloor\Big)$
$S_u[i] = 1 + \sum_{j=1}^{\rho(u)} S_{ch_u[j]}[i-1]$
**if** $X_u[i]$ is a new feature **then**
  $\phi_{X_u[i]} = \lambda^{\frac{S_u[i]}{2}}$
**else**
  $\phi_{X_u[i]} = \phi_{X_u[i]} + \lambda^{\frac{S_u[i]}{2}}$
**end if**

Algorithm 6 is applied to a single graph. However, an equivalent of the construction of the $Big^2DAG$ (Section 4.2.4) can be obtained by a simple merge operation between the hash tables resulting from the application of Algorithm 6 to a set of graphs.

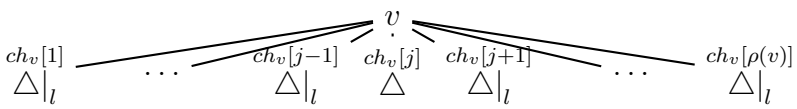


**Figure 4.7**: A feature for the tree kernel ST+. Given the node $v$ and the index $j$, the feature is composed by $v$, the proper subtree rooted at the $j$-th child and the subtrees resulting from a limited visit of $l$ levels for the other children.

### 4.3.2 A graph kernel based on a novel tree kernel

In Section 4.3.1 a graph kernel based on the ST kernel for trees has been defined. It has $|V_G| \log |V_G|$ complexity but, in practice, it is competitive, in terms of speed, with the fastest graph kernels (see Section 4.4). However, being it based on the ST kernel, the associated feature space might not be expressive for certain tasks.

The kernel we introduce in this section enlarges the feature space of the kernel in Section 4.3.1, with a modest increase in computational burden.

The associated tree kernel, which is a novel contribution in itself, is referred to as ST+. The set of features related to the ST+ kernel is a superset of the features of ST and a subset of the features of PT. Figure 4.7 describes a generic novel feature introduced by ST+. Notice that it depends on $v \in T$, the index of a child $j$ and a limit $l$ on the depth of the visits. While for the ST kernel there is one feature for each $v \in T$, ST+ associates at most $\rho(v)h$ features for any $v \in T$. Figure4.8 depicts a partial feature space representation of a tree according to ST+. For each node $v \in T$, for example the node hightlighted in Figure 4.8-a, the algorithm inserts at most $\rho(v)h + 1$ features:

- the proper subtree rooted at $v$, which in our example is the one in Figure 4.8-b;
- given the $j$-th child of $v$, the subtree composed by
  - $v$,
  - the proper subtree rooted at the $j$-th child of $v$,
  - the subtrees resulting from a visit limited to $1 \leq l \leq h$ levels starting from the other children of $v$

is added as feature. As $l$ ranges from 0 to $h$, the features/subtrees from Figure 4.8-c to Figure 4.8-e are added.

Substituting line 15 of Algorithm 6 with Algorithm 7, an algorithm for computing the features of ST+ directly from a graph $G$ is obtained. The variables $u$, $i$ are set in Algorithm 6 and represent the current node of the $DD_G^v$ and the current level of the visit, respectively. The features of the ST kernel are added in line 1 by the macro **compute-st-features**. Notice that it sets the vectors $X[]$ and $S[]$. The variables $x$ and $s$ represent the encoding of the novel features of ST+ as exemplified in Figure 4.7.

**Algorithm 7** A macro for computing the features of the ST+ kernel. It is supposed to replace line 15 of Algorithm 6.

**compute-ST+features**

1: **compute-st-features** // macro defined in Algorithm 6
2: **for** $1 \leq j \leq \rho(u)$ **do**
3: **for** $0 \leq l < i$ **do**
4: $(M_1, \ldots, M_{\rho(u)})$=sort$\Big(X_{ch_u[1]}[l], \ldots, X_{ch_u[j-1]}[l], X_{ch_u[j]}[i], X_{ch_u[j+1]}[l], \ldots, X_{ch_u[\rho(u)]}[l]\Big)$
5: $x = \kappa\Big(L(u)\Big\lceil M_1 \# \ldots \# M_{\rho(u)}\Big\rfloor\Big)$
6: $s = 1 + S_{ch_u[j]}[i] + \sum_{z=1, z\neq j}^{\rho(u)} S_{ch_u[z]}[l]$
7: **if** $x$ is a new feature **then**
8: $\phi_x = \lambda^{\frac{s}{2}}$
9: **else**
10: $\phi_x = \phi_x + \lambda^{\frac{s}{2}}$
11: **end if**
12: **end for**
13: **end for**

The complexity of Algorithm 6 where line 15 is substituted with Algorithm 7 is $O(|V_G| H h^2 \rho^2 \log \rho)$. The complexity of a kernel evaluation then is $|V_G| \log |V_G|$ if we consider $H$ constant.

### 4.3.3 Feature spaces comparison of some graph kernels

A way to get insights on the difference between graph kernels is to analyze what types of inputs they are able to discriminate. While this is not a sufficient requirement

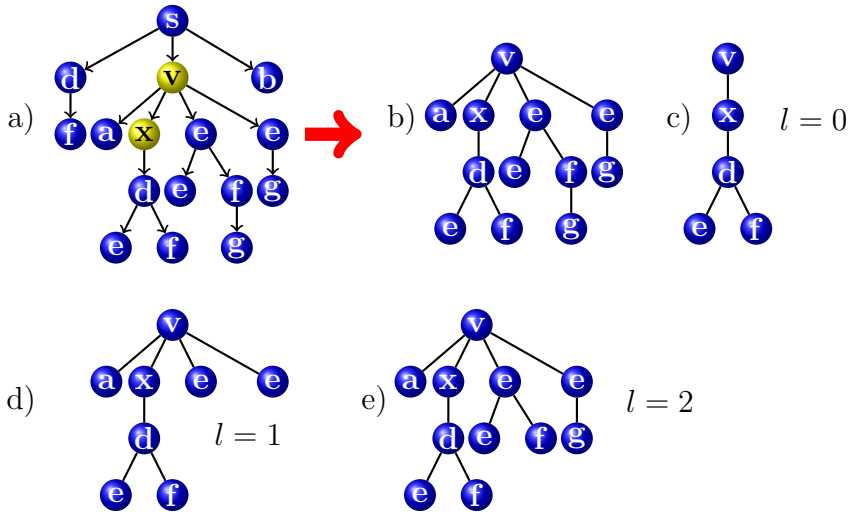


**Figure 4.8**: Feature space representation related to the kernel ST+ for an example tree: a) the input tree; b) the proper subtree rooted at the node labeled as **v**; c)-e) given the child **x** of **v**, the features related to visits limited to $l$ levels.

for successfully solving a learning task, it becomes necessary, for example, in a classification problem when two graphs, with the same representation in feature space, have different target classes. Specifically we are going to present some cases in which $\phi(G_1) = \phi(G_2)$ when $G_1$ is not isomorphic to $G_2$. The analysis focuses on gBoost, the Fast Subtree and the proposed $ODDK_{ST_h}$ and $ODDK_{ST+}$ kernels. Since gBoost extracts only a set of informative subgraphs, it is sufficient to consider $G_1 = \{g\} \cup \{g_1\}$ and $G_2 = \{g\} \cup \{g_2\}$, where $g$ is an informative subgraph and $g_1$, $g_2$ are not. Clearly $\phi(G_1) = \phi(G_2) = \phi(g)$.

When considering the Fast Subtree Kernel, it can be observed that, in order for two graphs to get the same representation in feature space, they need to have the same number of vertices and the same statistics on the out-degree of vertices. In addition, $ODDK_{ST_h}$ also requires the same distribution on the length of all the pairwise shortest paths. This is not the case for the Fast Subtree Kernel. Figure 4.9 shows an example of two graphs having the same feature space representation. Note that $ODDK_{ST_h}$ is able to discriminate those two graphs: the left one has a shorter path of length 4, while the longest one for the graph on the right has length 3.

This implies that there are at least two different tree visits and consequently two different features for both $ODDK_{ST_h}$ and $ODDK_{ST+}$: the proper subtrees whose root correspond to the root of the tree visits.

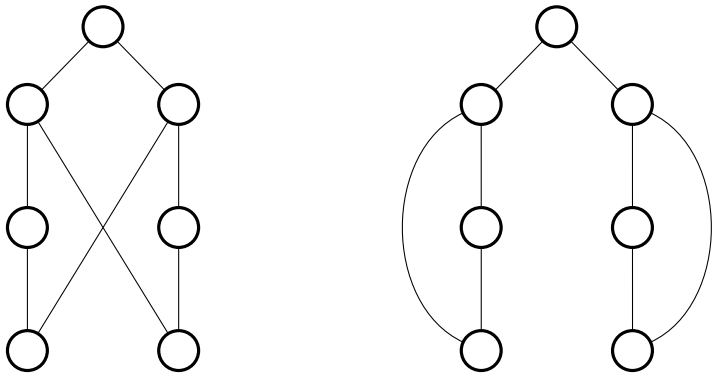

**Figure 4.9**: An example of two different graphs having the same representation in feature space according to the Fast Subtree kernel. All the nodes have the same label.

Let us turn our attention to the $ODDK_{ST_h}$ and $ODDK_{ST+}$ kernels. Clearly, two graphs differing just in self-loops, yield identical multisets of DAGs. However, the information about the presence of self-loops could be transferred into the representation of the corresponding node, for example by adding a field to the representation of the label. A less trivial example of non-injective mapping for the $ODDK_{ST_h}$ kernel is shown in Figure 4.10. Since the set of DAGs generated from the graphs in the figure are identical, the feature spaces related to the $ODDK_{ST+}$ kernel are identical as well.

## 4.4 Experimental results

While Equation (4.3) can be instantiated with any tree kernel, in order to reduce the time required for the experimentation, only the kernels in Section 4.3, the subtree kernel with visits of limited depth, i.e. $ODDK_{ST_h}$, and the kernel $ODDK_{ST+}$, are considered in the following. The reason for such choice lies in the fact that

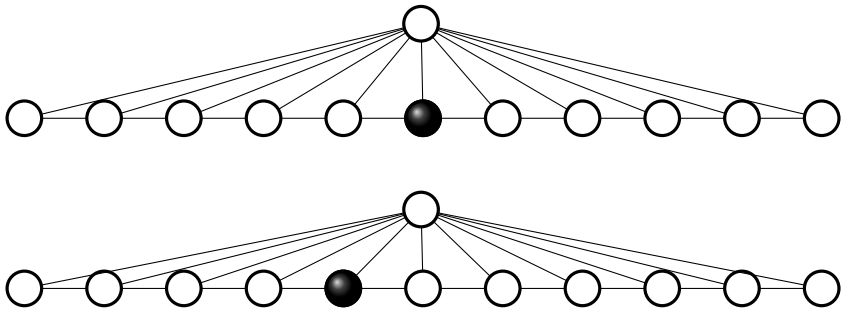

**Figure 4.10**: An example of two different graphs having the same representation in feature space according to the $ODDK_{ST_h}$ and $ODDK_{ST+}$ kernels. All the nodes have the same label.

previous sections heavily focused on $ODDK_{ST_h}$ and $ODDK_{ST+}$, especially from the computational point of view.

Typically, learning algorithms depend on a set of parameters. For the assessment of the predictive performance of an algorithm, we need a way to estimate the optimal values for such parameters. We will present two types of experiments that adopt different parameter selection procedures, in order to compare the performance with other algorithms present in literature [122, 38, 115], and to assess in the most unbiased way the predictive performances of the proposed methods.

The first experiments are performed in K-fold cross validation, with K=10. We compute the cross validation values over a grid of parameters, and choose the parameters that minimize the CV error to be our best parameters estimates. This evaluation method allows us to compare the performance of the proposed algorithms vs other previously published results, for which we were not able to perform new experiments because no software was publicly available. While this first method allows to estimate the best accuracy a predictive algorithm can reach in practice, when the cross validation is used for model selection (i.e. for parameter selection), as in our case, it is known that the results can be overly optimistic [26, 134]. Even if the effect applies to all learning algorithms, so the comparison should be fair, we decided to use also another parameter selection method, that more closely reflects the performance expected from the algorithm on unseen data.

The second type of experiments adopt a technique commonly referred to as *nested* k-fold cross validation. For each fold of the K-fold cross validation, we performed another *inner* K-fold cross validation, in which we select the best parameters for that particular fold. Then, using only the best parameters for each of the K folds, we classify the remaining examples (the test set for the particular iteration of K-fold). With this second approach, the parameters are optimized, for each fold, on the training dataset only, similarly to the approach adopted in [122].

The $10 \times 10$-CV statistical test proposed in [20] has been used for assessing the significance of the results.

### 4.4.1 Dataset Description

The experimentation has been performed on six datasets: Mutag [47], CAS[3], CPDB [77], AIDS [151], NCI1 [145] and GDD [51]. All datasets involve chemical compounds and represent binary classification problems. In all data sets nodes are labeled and there are no self-loops. Mutag, CAS, CPDB are dataset of mutagenic compounds whose target class represents whether or not they have a mutagenic effect. NCI1 is a balanced subset of a dataset of chemical compounds screened for activity against non small cell lung cancer. AIDS is an antiviral screen datasets.

The GDD dataset comprises X-ray crystal structures of proteins. Each protein is represented by a graph, in which the nodes are amino acids and two nodes are connected by an edge if they are less than 6° Angstroms apart. The examples are divided into enzymes and non-enzymes classes on the basis of Enzyme Commission number, annotations in the Protein Data Bank and Medline abstracts [51].

Table 4.1 summarizes the statistics of the datasets.

### 4.4.2 Results and Discussion

The efficiency of the proposed $ODDK_{ST_h}$ and $ODDK_{ST+}$ kernels are first compared. In Section 4.2.3 we have proposed a technique for speeding up the kernel computation

[3] http://www.cheminformatics.org/datasets/bursi

| Dataset | graphs | pos(%) | avg atoms | avg edges |
|---|---|---|---|---|
| Mutag | 188 | 66.48 | 45.1 | 47.1 |
| CAS | 4337 | 55.36 | 29.9 | 30.9 |
| CPDB | 684 | 49.85 | 14.1 | 14.6 |
| AIDS | 1503 | 28.07 | 58.9 | 61.4 |
| NCI1 | 4110 | 50.04 | 29.87 | 32.3 |
| GDD | 1178 | 58.65 | 284.31 | 2862.63 |

**Table 4.1**: Statistics of MUTAG, CAS, CPDB, AIDS, NCI1 and GDD datasets: number of graphs, percentage of positive examples, average number of atoms, average number of edges.

by compactly representing the ODDs by means of an Annotated DAG (BigDAG). The complexity of the DAG kernels, for example the ones described in Section 4.3, depends on the number of nodes of the BigDAG. In order to compare the bound on the number of nodes in Equation (4.5) and how it affects the complexity of the kernel, we have computed the number of nodes in each BigDAG as a function of the number of nodes of each graph. The plots in Figure 4.11 refer to the NCI1 dataset while the plots in Figure 4.12 refer to the CAS dataset. We have also plotted the polynomial function interpolating these points: such function is $n^{1.3691}$ for NCI1 and $n^{1.3652}$ fo CAS. In order to have a comparison with the values of the bounds discussed in Section 4.2.3, the functions $n^{\frac{3}{2}}$ and $n^2$ are plotted as well. The points of the plot, i.e. the numbers of nodes inserted in the BigDAG are, for the vast majority, under the curve $n^{1.5}$ for both datasets. The size of the BigDAG tends to an asymptotic value which depends on the outdegree of the nodes and the size of the longest cycle (see Section 4.2.5). In this case, the "actual" complexity of the kernel is proportional (via $H$) to $n \log n$. Similar results are obtained for the other datasets. The interpolating functions are: $n^{1.2774}$ for MUTAG, $n^{1.3001}$ for CPDB, $n^{1.2571}$ for AIDS and $n^{1.672}$ for GDD.

Figure 4.13 and Figure 4.14 report the time needed to compute the kernel matrix, as a function of $h$, for the $ODDK_{ST_h}$, $ODDK_{ST+}$, NSPDK and the FS kernels on

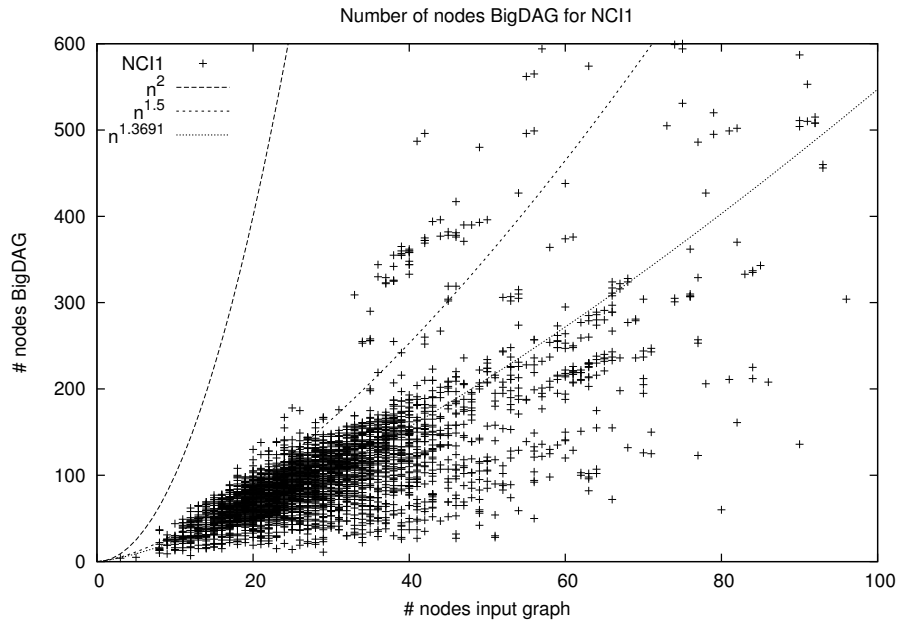


**Figure 4.11**: Number of nodes inserted into each BigDAG as a function of the nodes of the graphs.

NCI1 and CAS, respectively. All the experiments are performed on a PC with two Quad-Core AMD Opteron(tm) 2378 Processors and 64GB of RAM. Given that the available implementations of the two kernels are in different programming languages, the plots of the FS kernel were added just for a qualitative comparison. Notice that, as $h$ increases, the time required for computing the kernel matrix seems to increase almost linearly for FS and $ST_h$. This can be explained with the fact that smaller graphs do not allow too deep visits, so the increase in complexity due to the increase of the depth visit is compensated by the smaller number of graphs where visits of that depth can be actually performed.

We then compare the predictive abilities of the kernels in Section 4.3 to the Fast Subtree Kernel (FS), the Neighborhood Subgraph Pairwise Distance Kernel (NSPDK) and gBoost. In addition, results from the following algorithms, when available, are provided from [115] and [122]: Gaston, Correlated Pattern Mining (CPM), MOLFEA, Marginalized Graph Kernel (MGK) and SVM with frequent pattern mining (freqSVM). The proposed kernel, as each of the kernel functions

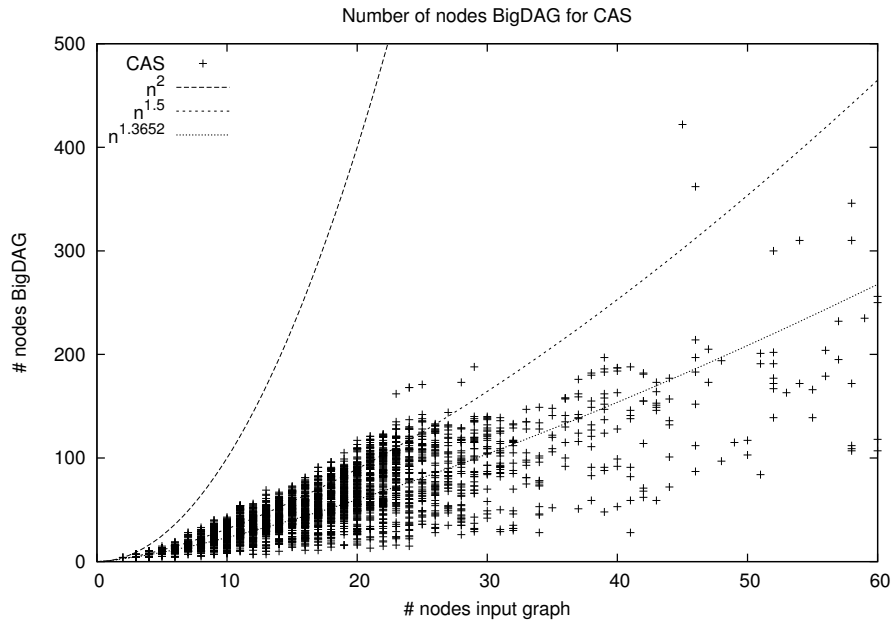


**Figure 4.12**: Number of nodes inserted into each BigDAG as a function of the nodes of the graphs.

in the following, was employed together with a Support Vector Machine. For the sake of comparison with the above mentioned results, the accuracy of each algorithm was measured by selecting a posteriori the best parameters among the average of 10 repetitions of a 10−fold cross validation. The values of the parameters of the $ODDK_{ST_h}$ and ST+ kernels have been restricted to: $\lambda = \{0.1, 0.2, \ldots, 2.0\}$, $h = \{1, 2, \ldots, 10\}$. The parameter selection procedure for the Fast Subtree Kernel replicates the one described in [122], i.e. only the parameter $h = \{1, 2, \ldots, 10\}$ is optimized. gBoost has many parameters, but following the parameter selection process exposed in [115], only $\mu$ (that controls training accuracy) has been optimized among the values $\{0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$. For the parameters of the other methods, see to the corresponding papers. Table 4.3 reports the average accuracy, the standard deviation and the ranking obtained by the considered methods on the six datasets.

According to the experimental results, gBoost tends to have a low ranking on most datasets, i.e. third, fourth and the fifth on three datasets. On the contrary, the

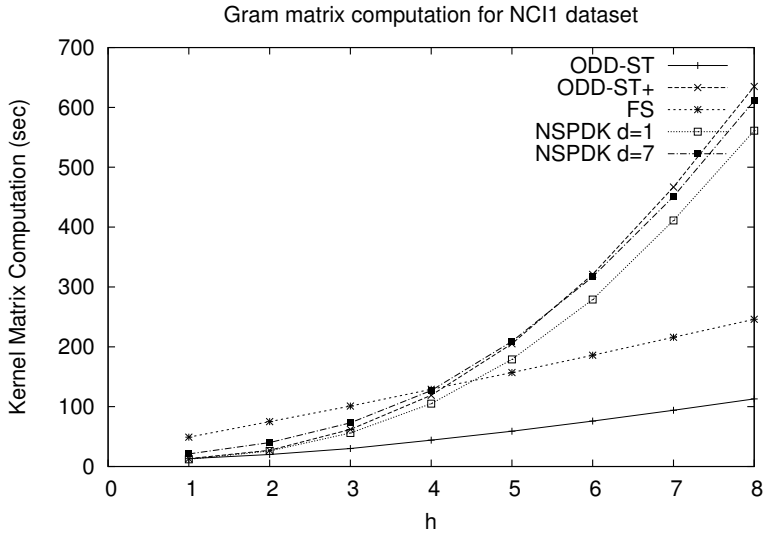


**Figure 4.13**: Time needed to compute the kernel matrix, as a function of $h$, for the $ODDK_{ST_h}$, $ODDK_{ST+}$ and the FS kernels on NCI1.

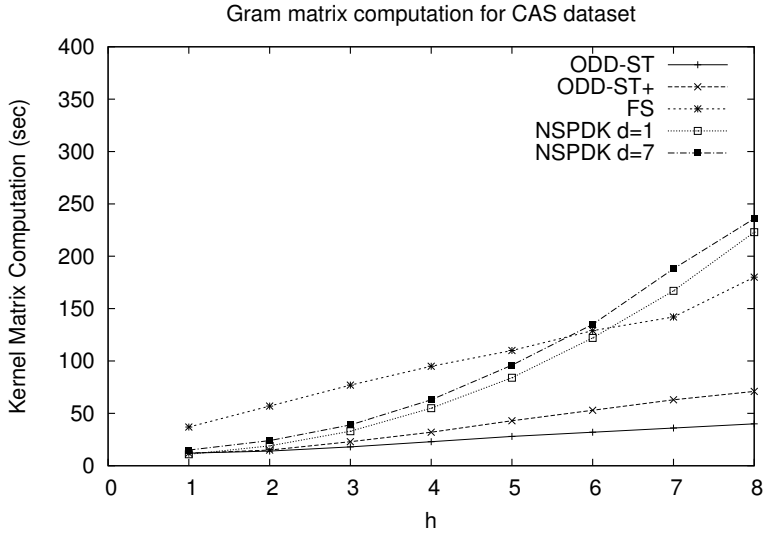


**Figure 4.14**: Time needed to compute the kernel matrix, as a function of $h$, for the $ODDK_{ST_h}$, $ODDK_{ST+}$ and the FS kernels on CAS dataset.

Fast Subtree, while placing first on two datasets, has a lower ranking, i.e. second, fourth, fifth and sixth, on the other datasets. Its average ranking is 3.16. By a *curious coincidence*, 3.16 is exactly the average ranking of NSPDK kernel as well. Thus, 3.16 is the average ranking to compare to our proposed kernels. $ODDK_{ST+}$ has best accuracy in three out of six datasets. On the remaining datasets it places in second position. Notice that $ODDK_{ST+}$ almost always increases its accuracy with respect to $ODDK_{ST_h}$ (the only exception being the CPDB dataset). The average rankings of $ODDK_{ST+}$ and $ODDK_{ST_h}$ are 1.5 and 2.83, respectively. These generally good results may be attributed to the fact that $ODDK_{ST_h}$ and $ODDK_{ST+}$ have associated a large feature space, which makes them more adaptable to different tasks.

As stated at the beginning of this section, when the cross validation is used for model selection (i.e. for parameter selection), as in our case, it is known that the results can be overly optimistic [26, 134]. In the following, we will present and discuss a second type of experiments performed with the technique commonly referred as *nested* 10-fold cross validation. Table 4.4 reports the average accuracy results obtained with this method for parameter selection. Note that in this case, there is no single setting of best parameters for each kernel/dataset, since different parameters are selected for each fold of each dataset. The values in accuracy for all the datasets are lower with respect to the results in 10-fold cross validation reported in Table 4.3. However, the relative ranking of the kernels for each dataset is similar. The only dataset in which there is a difference is MUTAG. We argue that this happens because the dataset is small, and the parameter selection procedure in this case is not reliable. It is evident looking at the high standard deviation presented by all the tested kernels on this dataset.

In order to better understand the differences in the performances of the various kernels, we analyzed the statistical significance among the accuracy results. We assessed the significance using the $10 \times 10$-CV statistical test proposed in [20] with confidence level 95%.

As expected, on small datasets (MUTAG and CPDB) the difference between the

| Kernel | CAS | AIDS | NCI1 | GDD |
|---|---|---|---|---|
| FS | 77 (h=3) | 30 (h=9) | 246 (h=8) | 405 (h=1) |
| NSPDK | 24 (h=2,d=6) | 217 (h=8,d=6) | 192 (h=5,d=4) | 395 (h=2,d=6) |
| $ODDK_{ST_h}$ | 18 (h=3) | 56 (h=7) | 44 (h=4) | 29 (h=1) |
| $ODDK_{ST+}$ | 32 (h=4) | 111 (h=8) | 205 (h=5) | 199 (h=2) |

**Table 4.2**: Average seconds required for computing the kernel matrix on CAS, AIDS, NCI1 and GDD dataset with the optimal parameters. The parameters influencing the speed of the kernel are reported between brackets.

various kernels is not statistically significant. More interestingly, in other datasets the difference in performance is significant. In the AIDS and CAS datasets, NSPDK, $ODDK_{ST_h}$ and $ODDK_{ST+}$ perform significantly better than FS. In the NCI1 dataset, FS and $ODDK_{ST+}$ performs significantly better than NSPDK that, in turn, performs better than $ODDK_{ST_h}$. In the GDD dataset, FS, $ODDK_{ST_h}$ and $ODDK_{ST+}$ perform significantly better than NSPDK.

Analyzing the significativity results, $ODDK_{ST+}$ never performs significantly worse respect the other kernels, while $ODDK_{ST_h}$ performs significantly worse than competitors in only one dataset: NCI1.

Table 4.2 reports the average computational time for a single fold with the optimal parameters on the four largest datasets: CAS, AIDS, NCI1, GDD. The parameters influencing the speed of the kernel are reported between brackets. Note that, looking at Table 4.2, Table 4.3 and Table 4.4 together, the $ODDK_{ST_h}$ kernel tends to have the best accuracy/speed ratio, being comparable to NSPDK in accuracy but faster in the calculation of the kernel matrix.

| *Kernel* | Mutag | CAS | CPDB | AIDS | NCI1 | GDD |
|---|---|---|---|---|---|---|
| Gaston | - | 79.0 (6) | - | - | - | - |
| MOLFEA | - | - | - | 78.5 (6) | - | - |
| CPM | - | 80.1 (5) | 76.0 (8) | 83.2 (3) | - | - |
| MGK | 80.8 (6) | 77.1 (8) | 76.5 (7) | 76.2 (8) | - | - |
| freqSVM | 80.8 (6) | 77.3 (7) | 77.8 (5) | 78.2 (7) | - | - |
| gBoost | 85.2 (5) | 82.5 (3) | 78.8 (3) | 80.2 (4) | 70.8 (5) | - |
| FS | $89.54_{\pm 0.99}$ (2) (c=100,h=1) | $81.12_{\pm 0.17}$ (4) (c=0.1,h=3) | $76.72_{\pm 0.96}$ (6) (c=0.1,h=2) | $78.45_{\pm 0.67}$ (5) (c=0.01,h=9) | $86.12_{\pm 0.21}$ (1) (c=0.01,h=8) | $79.63_{\pm 0.27}$ (1) (c=0.1,h=1) |
| NSPDK | $88.61_{\pm 0.60}$ (4) (c=1,h=1,d=4) | $84.35_{\pm 0.12}$ (2) (c=1,h=3,d=6) | $78.12_{\pm 0.71}$ (4) (c=1,h=1,d=2) | $83.96_{\pm 0.42}$ (2) (c=10,h=8,d=6) | $84.98_{\pm 0.16}$ (3) (c=1,h=5,d=4) | $76.09_{\pm 0.35}$ (4) (c=100,h=2,d=6) |
| $ODDK_{ST_h}$ | $89.28_{\pm 0.74}$ (3) (c=10,h=6, $\lambda$ =0.5) | $83.97_{\pm 0.21}$(3) (c=10,h=3, $\lambda$ =1.6) | $79.62_{\pm 0.59}$ (1) (c=10,h=2,$\lambda$=1.6) | $83.6_{\pm 0.34}$(3) (c=10,h=8,$\lambda$=1.8) | $82.71_{\pm 0.20}$ (4) (c=100,h=4,$\lambda$=1.04) | $77.92_{\pm 0.20}$(3) (c=10,h=1,$\lambda$=1.4) |
| $ODDK_{ST_+}$ | $89.92_{\pm 0.91}$ (1) (c=10,h=7,$\lambda$=0.3) | $84.40_{\pm 0.18}$ (1) (c=10,h=4,$\lambda$=1.2) | $79.47_{\pm 0.52}$ (2) (c=100,h=2,$\lambda$=0.5) | $83.99_{\pm 0.45}$ (1) (c=100,h=8,$\lambda$=2) | $85.46_{\pm 0.18}$ (2) (c=10,h=5,$\lambda$=1.02) | $78.37_{\pm 0.21}$ (2) (c=10,h=2,$\lambda$=1.02) |

**Table 4.3**: Average accuracy results (when available) ± standard deviation in 10-fold cross validation for Gaston, MOLFEA, Correlated Pattern Mining, Marginalized Graph Kernel, SVM with Frequent Mining, gBoost, the Fast Subtree, the Neighborhood Subgraph Pairwise Distance, the $ODDK_{ST_h}$ and the $ODDK_{ST+}$ kernels obtained on MUTAG, CAS, CPDB, AIDS, NCI1 and GDD datasets. The rank of the kernel is reported between brackets.

| *Kernel* | Mutag | CAS | CPDB | AIDS | NCI1 | GDD |
|---|---|---|---|---|---|---|
| FS | $83.53_{\pm 2.5}$ (3) | $81.13_{\pm 0.21}$ (4) | $73.37_{\pm 0.84}$ (4) | $75.35_{\pm 0.78}$ (4) | $\underline{84.79_{\pm 0.32}}$ (1) | $76.29_{\pm 1.22}$ (1) |
| NSPDK | $84.53_{\pm 1.33}$ (1) | $83.58_{\pm 0.37}$ (2) | $75.23_{\pm 1.77}$ (3) | $81.94_{\pm 0.41}$ (2) | $83.45_{\pm 0.43}$ (3) | $74.09_{\pm 0.91}$ (4) |
| $ODDK_{ST_h}$ | $83.82_{\pm 1.71}$ (2) | $83.34_{\pm 0.31}$(3) | $76.87_{\pm 1.64}$ (1) | $81.93_{\pm 0.72}$(3) | $82.10_{\pm 0.42}$ (4) | $75.27_{\pm 0.68}$(3) |
| $ODDK_{ST_+}$ | $83.44_{\pm 2.13}$ (4) | $83.90_{\pm 0.33}$ (1) | $75.66_{\pm 1.33}$ (2) | $82.33_{\pm 0.85}$ (1) | $84.60_{\pm 0.47}$ (2) | $75.33_{\pm 0.81}$ (2) |

**Table 4.4**: Average accuracy results ± standard deviation in nested 10-fold cross validation for the Fast Subtree, the Neighborhood Subgraph Pairwise Distance, the $ODDK_{ST_h}$ and the $ODDK_{ST+}$ kernels obtained on MUTAG, CAS, CPDB, AIDS, NCI1 and GDD datasets. The rank of the kernel is reported between brackets.

## 4.5 Model compression

As we saw in Section 2.5, feature selection consists in deleting non-informative features in order to reduce noise and to increase performance. The application of feature selection in the context of kernels for graphs is not usual because of the implicit definition of the feature space. Only when adopting other learning methods for graphs, feature selection has been applied (e.g. [115]) but considering only the feature frequency, and disregarding target information.

For a learning algorithm to be efficient, it is important to contain the size of the learned model, because it directly influences the complexity of the classification phase. One can notice that, in the high dimensional feature space in which kernel methods work, many features may be non-relevant for the task. Therefore, dropping these features not only reduces the noise in the data but also reduces the dimension of the feature space (and thus slightly improves kernel performance). It is worth to notice that we do not want to define new learning algorithms that produce smaller models with respect to state-of-the-art learning algorithms, since some work on this field has already been done, e.g. in [91]. The direction we want to explore is the application of feature selection techniques directly in the feature space of kernels for structured data.

Feature selection techniques can be applied to several kernels that can be efficiently computed with an explicit representation of the feature space, i.e. the Fast Subtree kernel (see Section 3.6.6), the Neighborhood Subgraph Pairwise Distance kernel (see Section 3.6.7) and the ODD kernels presented in this chapter in Section 4.3.

Existing feature selection techniques are designed for vector spaces. Thus, even if the feature space of kernel methods is actually an Euclidean space, each feature represents the presence or the absence of a specific (sub)structure into the instances. This introduces a major complication, because the features are strictly dependent, e.g. the feature associated with a certain tree can be activated only if all the features corresponding to its subtrees are activated as well.

We can apply two different strategies:

- consider the feature space as-is as a vectorial space, making possible to apply all the well-known feature selection techniques on these features. This is the most straightforward way, and we explored this approach in Section 4.5.1.
- taking into account the dependencies between features, including them in the feature selection process. This would be a more powerful approach, but it is necessary to keep track of feature dependencies in a non-straightforward way. This line of research is not considered in this thesis and is left as a future work.

The challenge of this research line is to define new measures and new algorithms for the weighting of features, considering the structured nature of the features. Currently, there is no work in literature that covers this topic (on graph data).

Evaluating a feature selection technique is not straightforward. A way to test the quality of the method is to compare the classification performance of various learning (kernel) methods with different dimensionality reduction techniques on various datasets.

Independently from feature selection, in some cases it may be convenient in kernel methods to represent the model obtained from learning not maintaining in memory all the selected support graphs as usual, but keeping a vector of the features that occur in those examples as explained in Section 4.3.1. This representation may lead to a more compact model, for example in the case of the studied kernel from the framework introduced in Section 4.1, and can be further reduced via feature selection.

### 4.5.1 Application of feature selection to graph kernels

The work presented in this section has been published in [43]. One nice characteristic of the kernel presented in Section 4.1 is the possibility to perform *feature selection* thanks to the explicit representation of the feature space. In particular, adopting

the explicit feature space representation proposed in Section 4.3.1, it becomes possible to delete some of the features from the hash map without problems, since the structural dependencies among features are not considered in this representation. Notice that this is not the case for other representations such as the BigDAG presented in Section 4.2.3. On the other hand, the BigDAG representation maintains the information about feature dependencies, and thus is apt for different and more powerful feature selection strategies. The definition of these strategies however is left as a future work.

In our case, feature selection is appealing since it can reduce the number of features to store in memory for representing the model generated via learning. Of course, a significant reduction in number of features should not reduce too much the performance of the model, which otherwise becomes useless.

In the literature, several feature selection principles have been proposed (see [31] for a description of many of them). A typical approach is to compute a statistical measure for estimating the relevance of each feature with regard to the target concept, and to discard the less-correlated ones.

In order to do as-well-as-possible, the measure to apply should take advantage of all available information, which in our learning scenario means to care about both feature frequency and target information. We have decided to study one measure that possesses these characteristics, the F-score (defined in Equation (2.8)). This measure has been widely adopted in various tasks including, for example, information retrieval. We recall that features that get small values of F-score are not very informative with respect to the binary classification task. Thus, features with F-score below a given threshold can be removed. Given a predefined tolerance on the drop of classification performance, a suitable value for such threshold can be estimated using a validation set or a cross-validation approach. In both the cases, the feature selection is performed on the training dataset only.

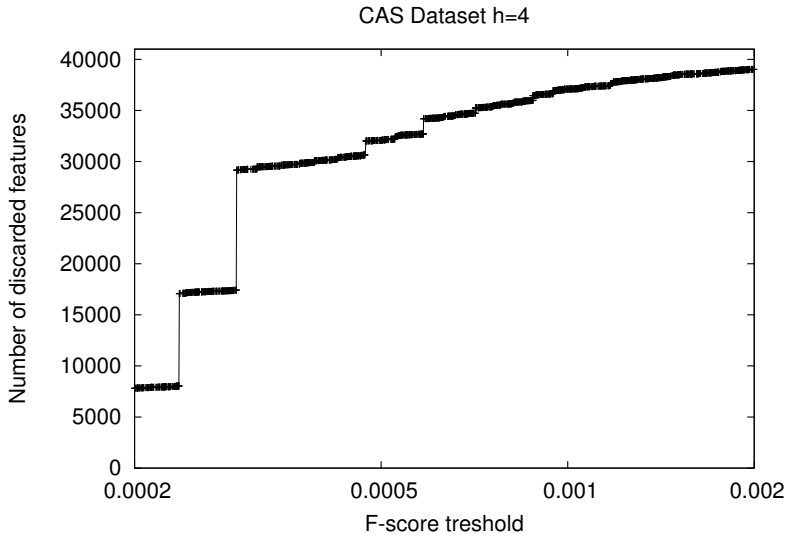


**Figure 4.15**: Number of discarded features as a function of the F-score threshold for the CAS dataset.

### 4.5.2 Experimental results

This section analyzes, from an empirical point of view, the effects of reducing the size of the model by making use of its explicit representation and by applying the feature selection technique based on the F-score (eq. 2.8). The experiments were performed on the CAS[4] and AIDS [151] datasets, previously described in Section 4.4.

All the experiments in this section involve the Support Vector Machines together with the kernel defined in Section 4.1. We recall that on these datasets this kernel (with $h = 4$) obtains state-of-the-art classification performances. The feature selection process is performed by choosing a set of threshold values and then discarding all those features whose F-score, computed according to eq. 2.8, is lower than such threshold on the training dataset. We first analyze the size of the model, in terms of number of features, after the application of various threshold values. Figure 4.15 and Figure 4.16 report the resulting plots. The number of discarded features is not linear with respect to the threshold value; in particular, there are high nonlinearities around the values 0.00023 and 0.0003 for CAS and one around the value 0.0006 for

[4]http://www.cheminformatics.org/datasets/bursi

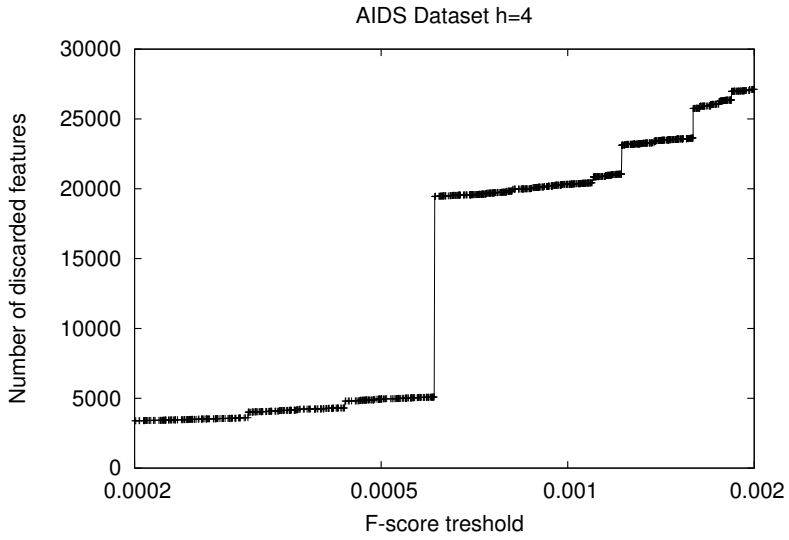


**Figure 4.16**: Number of discarded features as a function of the F-score threshold for the AIDS dataset.

AIDS.

We then checked how the accuracy of the learning algorithm was affected when reducing the number of features through the F-score. We performed a 10-fold cross validation on the datasets and reported mean accuracies and corresponding standard deviations. Figure 4.17 plots the values of accuracy and standard deviation with respect to the F-score threshold and the number of features that survived the selection process. Notice that between threshold values 0 and 0.0003 the accuracy decreases from 0.847 to 0.839, while the number of features retained decreases from $40,495$ to $11,549$ on average. This means that the size of the model, i.e. the number of features, is reduced to about $\frac{1}{4}$ with no significant loss in accuracy. Increasing the threshold to 0.001, the associated accuracy becomes 0.832 and the number of features becomes $3,477$, i.e. less than $\frac{1}{10}$ of the original size with a total accuracy loss of 0.015. After the threshold value 0.005 the number of features is 441, but the accuracy drops to 0.789. Clearly, too many significant features are discarded and the learning algorithm is not able to create a meaningful model. A similar behavior can be observed in Figure 4.18 for the AIDS dataset. However, notice that the

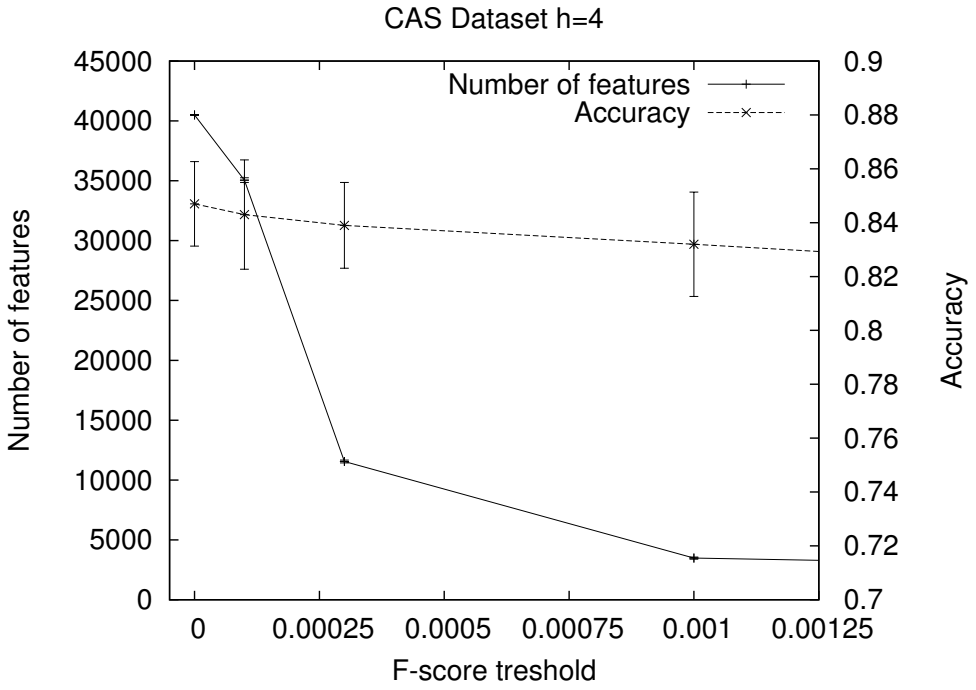


**Figure 4.17**: Accuracy (with standard deviation) and number of features as a function of the F-score threshold for the CAS dataset. Standard deviation for the number of features is not visible because it is very small.

0.0003 threshold has a slight accuracy improvement with respect to 0 (from 0.825 to 0.829) and the number of features decreases from $30,843$ to $27,169$. Figure 4.18 does not show the behavior for higher F-score values, however it is interesting to notice that with threshold value set at 0.05 the accuracy is 0.785, so 0.042 less than the reference figure, but with just 90 features.

Figure 4.19 plots the number of support graphs as a function of the F-score thresholds. Notice that, for threshold values less than 0.001, the number of support graphs drops, while there is no significant accuracy loss. This means that the learning algorithm is able to produce a simpler but effective model.

One can argue that eliminating features can lead a graph to lose all of its features in the representation. This graph then will not be represented at all in the model. Figure 4.20 shows that this happens only for high F-score values, so this does not

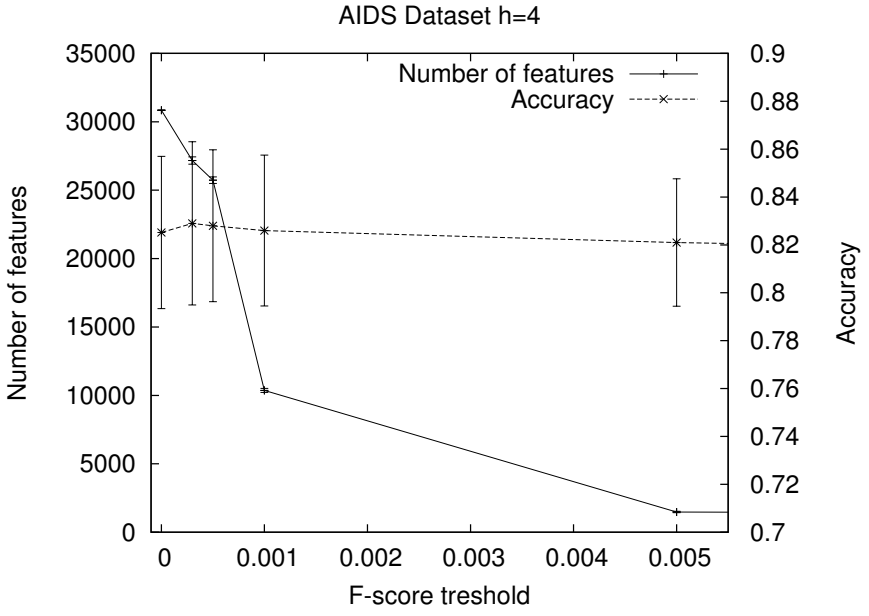


**Figure 4.18**: Accuracy (with standard deviation) and number of feature as a function of the F-score threshold for the AIDS dataset. Standard deviation for the number of features is not visible because it is very small.

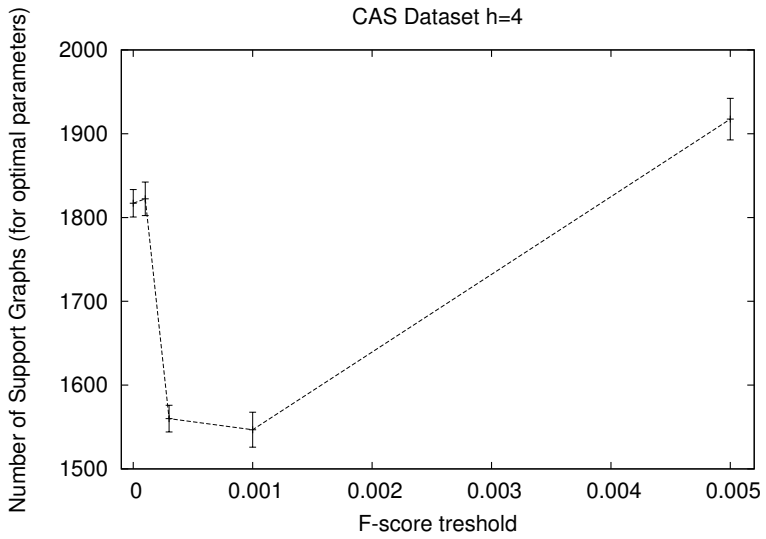


**Figure 4.19**: Number of support graphs (with standard deviation) as a function of the F-score threshold for the CAS dataset.

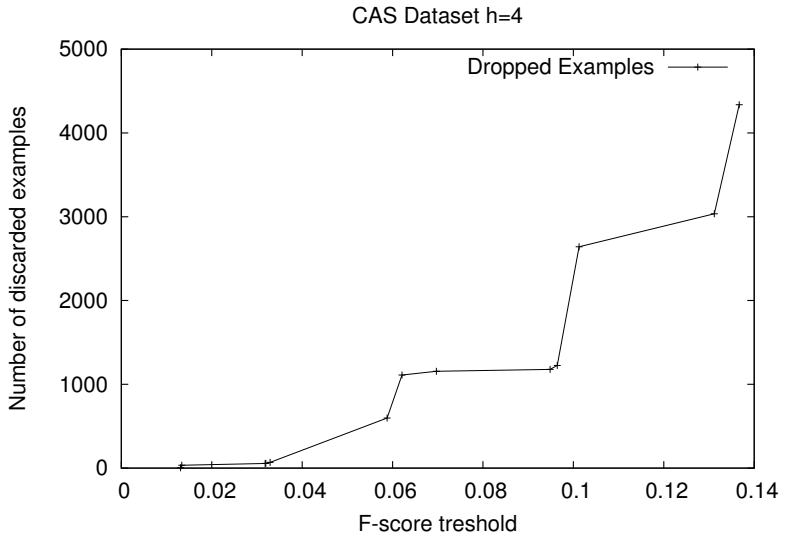


**Figure 4.20**: Number of graphs that lose all their features as a function of the F-score threshold value, for the CAS dataset.

happen with the range of F-score values we considered acceptable looking at the accuracy.

A final remark that is worth to make is the following. If the number of explicit features needed to represent a model is larger than the total number of vertices in the support graphs, then it is more convenient from the storage point of view to use the representation of the model in the dual space. We have verified that this is not the case for the two studied datasets. In fact, for the CAS datasets, the total number of vertices in the support graphs without feature selection is around $240,300$, while for the AIDS dataset, the same number is around $146,158$. Of course, these values are far larger than the number of features retained by the models, with or without feature selection.

# Chapter 5

# Learning algorithms for streams of graphs

> A paper should be like a mini skirt: long enough to cover everything, but short enough to keep it interesting.
>
> Anonymous

Recalling from Section 2.7.1, the main challenges of stream data mining are to accurately capture the continuous changing decision concepts and to scale up to large volumes of stream data. The nature of data streams requires the use of algorithms that involve at most one pass over the data and that try to keep track of time-evolving features (concept drifting). Moreover, a stream algorithm should use at most a fixed amount of memory. The research illustrated in Section 4.5 is useful in order to reduce the model dimension, that is a key prerequisite for the research discussed in this section.

The development of fast and robust stream learning algorithms for structured data is a very interesting and challenging task. Recently some work in this direction, involving learning on streams of trees [72], has been performed. Another recent work [2] considers the case of online learning on a stream of graphs. This work focuses on efficiency, using an approximated discriminative subgraph mining approach for graph classification. The paper considers the case of a stream of edges from a limited number of very big graphs. This case does not cover all the graph stream scenarios.

For example, if we want to classify a stream of chemical compounds, composed of a big number of small graphs, the proposed approach seems not adequate. Moreover, the proposed technique implicitly defines a similarity measure between graphs based on the edges two graphs share. Thus, there is no flexibility in the adopted similarity measure.

In Section 3.2 we gave some examples of application domains where learning on streams of graphs can be applied, e.g. fault diagnosis systems for sensor networks, malware detection, image classification or the discovery of new drugs.

Being the application domains so different, the goal we want to achieve is to define a framework for learning on streams of graphs that allows for the instantiation with different graph similarity measures, i.e. the framework has to be flexible. In this chapter we will focus on the problem of classification of graphs in a stream. However, the findings of this chapter can be easily extended to the other learning problems discussed in Section 2.1. Moreover, it is necessary to respect the strict constraints of stream learning, i.e. a strict bound on the memory occupation and the linear-time computational complexity, as stated in Section 2.7.1. Kernel methods turn to be a good approach also in this field. A stream learning algorithm that adopts kernel functions could be instantiated with various kernels for graphs, giving the possibility to select the kernel that is more appropriate for a given task. Nonetheless, the application of kernel methods to graph streams is not straightforward due to the computational problems that arise.

In this chapter we will give some feasible solutions for the definition of such framework, proposing modified versions of state-of-the-art linear-time classification algorithms as online passive aggressive presented in Section 2.7.4. The proposed modifications are easily extensible to all the algorithms based on the Online Stochastic Gradient Descent technique presented in Section 2.7.3 and to any graph kernel that is fast enough to respect the linear-time processing constraint imposed by the stream setting.

Moreover, if the adopted kernel allows for an explicit feature space representation, as the kernels belonging to the framework proposed in Chapter 4, it is possible to

adopt a different formulation for the learning algorithm working directly in the feature space induced by the kernel, and that is practically faster with respect to the classical formulation with kernel functions applied to examples in the input space. To define the framework, we have to modify the existing algorithms in order to:

- make a selection of features (based on some estimation of relevance) in order to consider only relevant information and to limit the dimension of the model, i.e. to respect the strict memory constraints;
- address the problem of concept drifting. This can be done by adapting the model for each data chunk that arrives.

For addressing the first problem, there are several techniques for performing feature selection. We will propose some alternatives and test the results of different policies in Section 5.1.1. Note that the research presented in Section 4.5.1 is the starting point for the policies presented in that section. To address the second problem, one way to identify outdated information is using a committee of classifiers, trained from different portions of the data following the concept of sliding windows over the examples: that is, every classifier is trained from a different, possibly overlapping, subset of the data available so far. This approach, presented in [92] for structured data, has the drawback of increased computational complexity though. Another strategy to face the second problem is to address concept drift in the feature selection step, i.e. when concept drift occurs, the learning algorithm is able to modify the current hypothesis in an effective way. This is the strategy we adopt in the proposed algorithms.

As far as we know, in literature there are no other works on learning on streams of graphs except the ones cited in this section. We recall that to assign a label to the nodes of a graph is a different problem w.r.t. the one we are considering.

With kernel-based learning algorithms, as the one presented in Section 2.7.3, we have the choice to represent the hypothesis explicitly or as a weighted sum over the support vectors (*representer theorem*, see Section 2.4). When the learning is

performed on structured data as in the case we are considering, or more in general with nonlinear kernels, usually it is not possible to use the sparse explicit version of the algorithms. With some recently proposed kernels, such as the one proposed in Chapter 4, it is possible to slightly modify explicit algorithms to deal with the explicit feature space representation of the examples. In Section 5.1 we will discuss if it is convenient to adopt such representation for the learned model. In Section 5.2 we will discuss a novel approach to represent in approximation the explicit model.

## 5.1 Budget online Passive-Aggressive on graph data

In this section, we study three algorithms together with different strategies for managing the budget on graph streams. The goal of the work in this section is to show that, when dealing with graph data, it is convenient, both from the accuracy and the performance point of view, to use algorithms working with explicit features. The algorithms listed in Section 2.7.4 assume that each example has a fixed size representation. This assumption clearly does not hold for structured data and specifically for graphs. Given the variable size of graph data, we make use of the following measure for computing the size of the model for Algorithm 5:

$$|M| = \sum_{G_j \in M} (|V_{G_j}| + |E_{G_j}| + 1), \tag{5.1}$$

where 1 takes into account the value $\tau_j$. This measure is optimistic with regard to the particular data structure adopted to represent the graph, meaning that any possible representation of the graph needs at least this amount of memory. Indeed, we take into account one memory unit per node plus one memory unit per edge.

The first algorithm we define (Algorithm 8) is a modification of Algorithm 5 with a different removal rule: when $G_t$ has to be inserted, instances are removed from $M$ until $|M| + |V_{G_t}| + |E_{G_t}| + 1 \leq B$, where $|M|$ is computed according to Equation (5.1).

**Algorithm 8** Dual Passive-Aggressive algorithm for online kernel learning on graphs on a budget.

**Input:** $\beta$ (algorithm dependent), $B$ (budget size)
Initialize $M$: $M = \{\}$
**for each round** $t$ **do**
  Receive an instance $G_t$ from the stream
  Compute the score of $G_t$: $S(G_t) = \sum_{i=1}^{|M|} y_i \tau_i K(G_i, G_t)$
  Receive the correct classification of $G_t$: $y_t$
  **if** $y_t S(G_t) \leq \beta$ ($G_t$ incorrectly classified) **then**
    update the hypothesis:
    **while** $|M| + |V_{G_t}| + |E_{G_t}| + 1 > B$ **do**
      select an element $G_j \in M$ for removal
      $M = M \setminus \{G_j\}$
    **end while**
    $M = M \cup \{(y_t \tau_t, G_t)\}$
  **end if**
**end for**

The complexity of dual online algorithms depends on the number of graphs in $M$ and the complexity of the employed kernel function. In settings in which the number of features associated to a kernel is not significantly greater than the size of the input (for example [44]), the evaluation of the kernel function may be greatly speeded up if performed as dot product of the corresponding feature vectors. This observation has been proposed in [130]. The actual size of vectors $\phi(G)$ can be much less than the feature space size if only non-null elements of $\phi(G)$ are represented in sparse format. In fact, when using graph kernels, it is typical that only a small fraction of the features that constitute the graph domain are not-null, i.e. the representation for the graph is sparse. We will refer to the size of $\phi(G)$ according to the sparse representation as $|\phi(G)|$. This observation leads to the *primal/dual* algorithm (referred to as *mixed* in the following) presented in Algorithm 9.

**Algorithm 9** *Mixed* Passive-Aggressive algorithm for online learning on a budget.

**Input:** $\beta$ (algorithm dependent), $B$ (budget size)
Initialize $M$: $M = \{\}$
**for each round** $t$ **do**
  Receive an instance $G_t$ from the stream
  Compute the score of $G_t$: $S(G_t) = \sum_{\phi(G_j) \in M} y_j \tau_j \phi(G_j) \cdot \phi(G_t)$
  Receive the correct classification of $G_t$: $y_t$
  **if** $y_t S(G_t) \leq \beta$ ($G_t$ incorrectly classified) **then**
    update the hypothesis:
    **while** $\sum_{\phi(G_j) \in M} |\phi(G_j)| + |\phi(G_t)| > B$ **do**
      select an element $\phi(G_j) \in M$ and remove it: $M = M \setminus \{\phi(G_j)\}$
    **end while**
    $M = M \cup \{y_t \tau_t \phi(G_t)\}$
  **end if**
**end for**

Note that Algorithm 9, although expected to be faster than Algorithm 8, needs to use more memory since every example is now explicitly represented by its set of features. The size of the model for this algorithm can be calculated as:

$$|M| = \sum_{G_j \in M} |\phi(G_j)|, \tag{5.2}$$

where $\phi(\cdot)$ refers to the specific adopted kernel function. In words, for each example in the model the memory occupancy corresponds to the number of non-null features.

Finally, we introduce a budget online algorithm working in the feature space. The idea is to replace all elements of $M$ with their sum: $w = \sum_{\phi(G_j) \in M} y_j \tau_j \phi(G_j)$, as in Algorithm 2. The difference is that in this case $w$ is sparse and very high dimensional, requiring specific techniques in order to limit the memory occupancy of the explicit model. Moreover, by so doing, we lose the connection between features and the instances they belong to. Consequently, during the update of the hypothesis it is not possible to select a whole vector $\phi(G)$ for removal. We propose to remove single features from $w$ when the budget is full, i.e. $|w| = B$ and to insert a new non-zero feature in $w$.

**Algorithm 10** *Primal* Passive-Aggressive online learning on a budget.

**Input:** $\beta$ (algorithm dependent)
Initialize $w$: $w_0 = (0, \ldots, 0)$
**for each round** $t$ **do**
    Receive an instance $G_t$ from the stream
    Compute the score of $G_t$: $S(G_t) = w_t \cdot \phi(G_t)$
    Receive the correct classification of $G_t$: $y_t$
    **if** $y_t S(G_t) \leq \beta$ ($G_t$ incorrectly classified) **then**
        **while** $|w + \phi(G_t)| > B$ **do**
            select a feature $j$ and remove it from $w$
        **end while**
        update the hypothesis: $w_{t+1} = w_t + \tau_t y_t \phi(G_t)$
    **end if**
**end for**

Note that the use of the sparse vector $w$ instead of the set $M$ allows Algorithm 10 to save a significant amount of memory while still being faster than Algorithms 8 and 9.

In Section 5.1.2 we will compare the three proposed algorithms, showing the superiority of Algorithm 10.

### 5.1.1 Removal policies

In all the algorithms introduced in the previous section, we have not specified how to select the examples/features to be removed when the budget is full. In this section, we describe the policies we have explored. Of course, all the algorithms are able to face (in a way or the other) concept drift since they use a budget. However, different policies can potentially lead to different performances in presence of concept drift. The explored policies mimic the different approaches presented in literature and briefly described in Section 2.8.

For Algorithms 8 and 9 (*Dual* and *Mixed*), we have explored the following policies:

- "random", examples are removed randomly with uniform probability; after preliminary experiments, we decided not to include this policy in our experimental evaluation because it tends to have worse performances [149].

- "oldest", the oldest examples are removed;
- "$\tau$", the examples with lowest $\tau$ values are removed. If more than one example has such $\tau$ value, the candidate is randomly selected.

The "random" and "oldest" policies are also used for Algorithm 10 (*Primal*) with the difference being that features are removed instead of examples. For the same algorithm, we have also explored the following specific policies:

- "Weight": first, all the features of the input instance that are already present in the model, are inserted. This maximizes the information available to the algorithm without increasing memory occupation. Next, for each remaining feature $f$ of the input instance, the feature of the model with lowest weight $w_i$, i.e. the weight associated to the feature $f_i$ in the current hypothesis, is selected for removal. However, if all the features in the model have weight higher than $f$, then $f$ is not inserted. This policy does not require to keep any additional information for each non-null feature because only the weight of the feature is used.
- "F-score": it is similar to the "weight" policy where the weight $w_i$ is replaced by the F-score, computed according to Equation (2.8) for the i-th feature.

  Moreover, we defined an incremental version of the F-score in Equation (5.3) that will be presented in the following section. If we use this formulation, the size of the model becomes four times the number of non-null features since, for each feature $f_i$, we need to keep its cumulative frequency in the positive ($f_i^+$) and negative ($f_i^-$) examples and the cumulative squared frequencies ($f_i^{2,+}$ and $f_i^{2,-}$).

### Incremental computation of F-score

The original F-score formulation, presented in Equation 2.8, cannot be applied as it is to a stream since instances arrive one at a time. However, it is not difficult to rewrite an incremental version of the F-score. Let $\mathcal{I}_t^+$ ($\mathcal{I}_t^-$) be the set of positive

(negative) instances which have been observed from the stream after having read $t$ instances, then the F-score $Fs(i,t)$ can be rewritten by using the following quantities:

$$n_t^+ = |\mathcal{I}_t^+|, \quad f_i^+(t) = \sum_{j \in \mathcal{I}_t^+} f_i^j, \quad f_i^{2,+}(t) = \sum_{j \in \mathcal{I}_t^+} (f_i^j)^2,$$

$$n_t^- = |\mathcal{I}_t^-|, \quad f_i^-(t) = \sum_{j \in \mathcal{I}_t^-} f_i^j, \quad f_i^{2,-}(t) = \sum_{j \in \mathcal{I}_t^-} (f_i^j)^2,$$

where $f_i^j$ is the frequency of the i-th feature in the j-th instance.
In fact, we have:

$$AVG_{i,t}^+ = \frac{f_i^+(t)}{n_t^+}, \; AVG_{i,t}^- = \frac{f_i^-(t)}{n_t^-}$$

$$AVG_{i,t} = \frac{f_i^+(t) + f_i^-(t)}{n_t^+ + n_t^-}$$

and

$$Fs(i,t) = \frac{(AVG_{i,t}^+ - AVG_{i,t})^2 + (AVG_{i,t}^- - AVG_{i,t})^2}{D_t^+ + D_t^-} \tag{5.3}$$

where

$$D_t^+ = \frac{f_i^{2,+}(t) - 2AVG_{i,t}^+ f_i^+(t) + n_t^+ (AVG_{i,t}^+)^2}{n_t^+ - 1},$$

$$D_t^- = \frac{f_i^{2,-}(t) - 2AVG_{i,t}^- f_i^-(t) + n_t^- (AVG_{i,t}^-)^2}{n_t^- - 1}.$$

By defining $\delta^+(t+1) = 1$ if the $(t+1)$th instance is positive; otherwise $\delta^+(t+1) = 0$, and $\delta^-(t+1) = 1 - \delta^+(t+1)$, the quantities of interest can be updated incrementally as follows:

$$n_{t+1}^+ = n_t^+ + \delta^+(t+1), \quad f_i^+(t+1) = f_i^+(t) + \delta^+(t+1) f_i^j,$$

$$f_i^{2,+}(t+1) = f_i^{2,+}(t) + (\delta^+(t+1) f_i^j)^2.$$

$$n_{t+1}^- = n_t^- + \delta^-(t+1), \quad f_i^-(t+1) = f_i^-(t) + \delta^-(t+1) f_i^j,$$

$$f_i^{2,-}(t+1) = f_i^{2,-}(t) + (\delta^-(t+1) f_i^j)^2.$$

It should be noticed that $n_t^+$ and $n_t^-$ do not depend on the feature index, while the other quantities do.

### 5.1.2 Experimental results

In this section, we report the performance of the various algorithms introduced in the previous section on two graph datasets: the first one composed of chemical compounds and the second one composed of images. We start by describing how the datasets were obtained. Then we introduce the experimental setup and the adopted evaluation measure. Finally, the obtained results are illustrated and discussed in the same section.

#### Chemical dataset

We have created graph streams combining two graph datasets available from the PubChem website (http://pubchem.ncbi.nlm.nih.gov).

PubChem is a source of chemical structures of small organic molecules and their biological activities. It contains the bioassay records for anti-cancer screen tests with different cancer cell lines. Each dataset belongs to a certain type of cancer screen. For each compound, an activity score is reported. The activity score for the selected datasets is based on increasing values of -LogGI50, where GI50 is the concentration of the compound required for 50% inhibition of tumor growth. A compound is classified as active (positive class) or inactive (negative class) if the activity score is, respectively, above or below a specified threshold. By varying the threshold, we are able to simulate a drift on the target concept.

Our dataset is a combination of the "AID: 123" and "AID: 109" datasets from PubChem. In "AID:123", growth inhibition of the MOLT-4 human leukemia tumor cell line is measured as a screen for anti-cancer activity. The dataset comprises $40,876$ compounds tested at 5 different concentrations. The average number of nodes for each graph representing a compound in this dataset is 26.8, while the average number of edges is 57.68. In "AID:109", growth inhibition of the OVCAR-8 human ovarian tumor cell line is measured as a screen for anti-cancer activity on $41,403$ compounds. The average number of nodes for each compound is 26.77, while the average number of edges is 57.63. For each dataset, we used two different

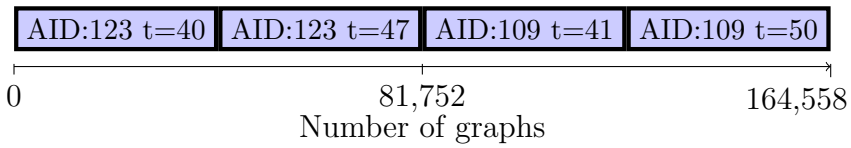


**Figure 5.1**: Composition of the first stream of graphs on chemical data. Four different target concepts are obtained by using different threshold values ($t$) on the activity scores of the datasets.

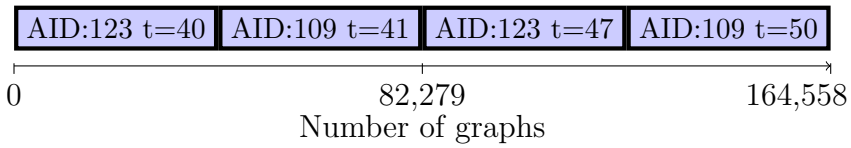


**Figure 5.2**: Composition of the second stream of graphs on chemical data. Four different target concepts are obtained by using different threshold values ($t$) on the activity scores of the datasets.

threshold values to simulate the concept drift: the median of the activity scores and the value such that approximately 3/4 of the compounds are considered dataset to be inactive (negative target).

Finally, the stream has been obtained as the concatenation of "AID: 123" with threshold 1, "AID: 123" with threshold 2, "AID: 109" with threshold 1, "AID: 109" with threshold 2 as shown in Figure 5.1. We call this stream Chemical1. In order to assess the dependency of the results on the order of concatenation of the datasets, we created a second stream as: "AID: 123" with threshold 1, "AID: 109" with threshold 1, "AID: 123" with threshold 2, "AID: 109" with threshold 2 (Figure 5.2). We call this stream Chemical2. Note that two streams are composed by four different concepts and each stream comprises a total of $164,558$ graphs. It should be stressed that the selected datasets represent very challenging classification tasks, independently of the value selected as the activity score threshold.

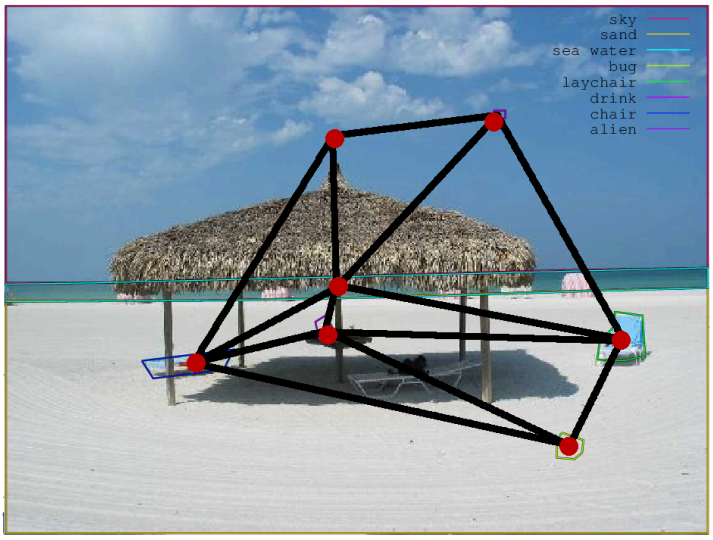


**Figure 5.3**: An example of graph construction from an annotated image.

### Image dataset

We created a stream of graphs from the *LabelMe* dataset[1]. The dataset comprises a set of images whose objects are manually annotated via the LabelMe tool [114]. The images are divided into several categories. We have removed those images having less than 3 annotations. We have selected six categories amongst the ones having the largest number of images: "office" (816), "home" (928), "houses" (1,294), "urban_city" (865), "street" (1,069), "nature" (370). In total we considered 5,342 images.

We then transformed each image into a graph: the annotated objects of the image become the nodes of the graph. The edges of the graph are determined according to the Delaunay triangulation [132]. The basic idea of the Delaunay triangulation is to connect spatially neighboring nodes. Figure 5.3 gives an example of the construction of a graph from an image. The average number of nodes per graph is 14.37 and the

[1]http://labelme.csail.mit.edu/Release3.0/browserTools/php/dataset.php

average number of edges is 63.61.

The stream is made up of six parts, for each of them one of the categories is selected as the positive class while the remaining ones represent the negative class; each one of the $5,342$ images appears six times in the stream: once with a positive class label, and 5 times with negative class label. The total number of examples composing the stream is $32,052$.

### Experimental setup

Algorithms 8 and 9 have been used as baseline approaches. We recall that the $\tau$ values are computed accordingly to Equation 2.6 and Equation 2.5. The presented approach can be easily applied to all of the budgeted online learning algorithms presented in Section 2.7.4. However, we focus on Passive-Aggressive for its very competitive performances (from both the accuracy and computational complexity point of view). The $C$ parameter has been fixed to 0.01 for both the chemical and image datasets after a set of trial experiments.

All the proposed algorithms use the same amount of memory, expressed as Budget, that is calculated for *Dual* as described in eq. 5.1, for *Primal*/*Dual* as described in eq. 5.2 and for *Primal* as described in Section 5.1.1.

Since all the proposed learning algorithms use the same kernel, it is possible to fix the kernel parameters, and to compare the performances varying only the parameters of the learning algorithm, i.e. the budget. The kernel parameters were set to the best values we obtained on a set of trial experiments on the stream: $h = 3$ and $\lambda = 1.6$ for the chemical datasets and $h = 3$ and $\lambda = 1.0$ for the image dataset. It is worth to notice that, for both the chemical and the image datasets, the kernel achieves the best performance for $h > 1$, proving that the structural information in the data is actually correlated with the learning tasks. As budget values we experimented with $10,000$ and $50,000$ memory units (assuming each memory unit can store a floating point or integer number). Higher budget values were not tested since the time needed for *Dual* to terminate became excessive.

Since the class distribution on the streams is unbalanced, the Balanced Error Rate was adopted as an error measure for our experiments [31]:

$$\text{BER} = \frac{1}{2}\left(\frac{\text{fp}}{\text{tn} + \text{fp}} + \frac{\text{fn}}{\text{fn} + \text{tp}}\right),$$

where tp, tn, fp and fn are, respectively, true positive, true negative, false positive and false negative examples. Since the BER is an error measure, in order to ease the presentation of the results, we preferred to adopt $1 - \text{BER}$ as performance measure. The plots in Section 5.1.2 regarding the $1-\text{BER}$ measure are obtained as follows: the $1 - \text{BER}$ measure is sampled every 50 examples. In order to increase the readability of the results, we interpolated the resulting values with the Bézier curve. Note that the interpolation has the following side effect: when a drift in the stream occurs, the changes in the plot are not immediate.

### Results and discussion

Our first experiments are aimed at determining, for each of the three algorithms, the best policy for managing the budget. In this set of experiments, we tested the algorithms with a budget value $b = 10,000$. The comparison between the "oldest" and "$\tau$" policies for *Dual* on the Chemical1 dataset are reported in Figure 5.4 while the performances on the Image dataset are reported in Figure 5.5. The results for Chemical2 are very similar to the ones of Figure 5.4 and thus are omitted.

The comparison between the "oldest" and "$\tau$" policies for *Primal*/*Dual* on the Chemical1 dataset are reported in Figure 5.6. Figure 5.7 reports the performance on the Image dataset. It is worth to notice that, in the Image dataset, the "oldest" policy is not able to recover after the second drift. This happens because this policy does not remove examples from the budget according to their contribution to the model. This plot clearly show how different budget policies can strongly affect the accuracy of the trained model.

A comparison between the "F-score" and the "weight" policies for *Primal* on the Chemical1, Chemical2 and Image datasets can be found, respectively, in Figure 5.8, Figure 5.9 and Figure 5.10.

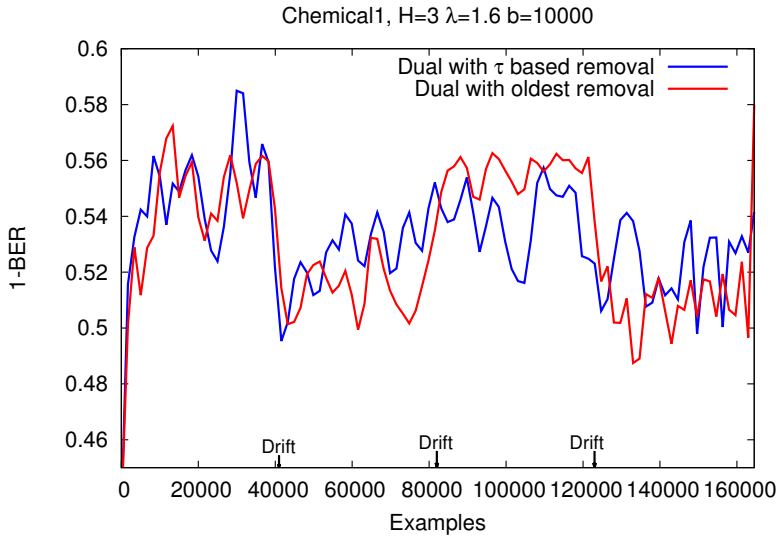


**Figure 5.4**: Comparison between the classification performances of the "$\tau$" and "oldest" policies applied to *Dual* with budget $10,000$ on the Chemical1 dataset.

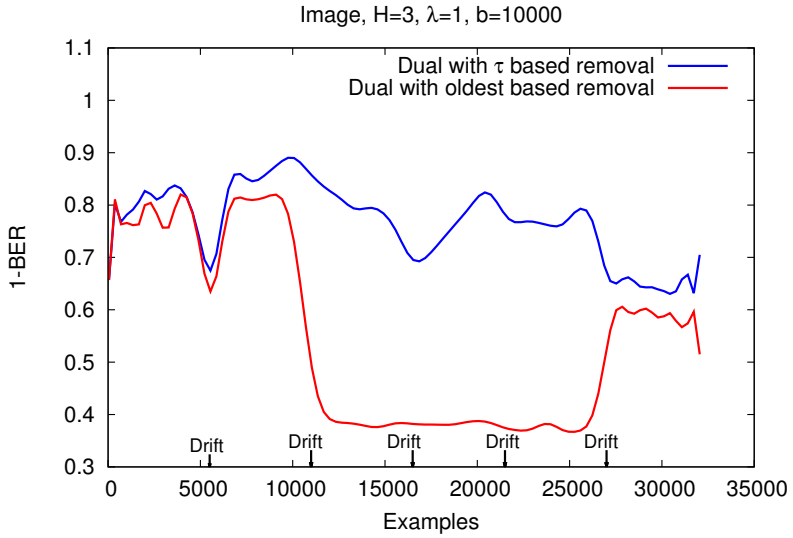


**Figure 5.5**: Comparison between the classification performances of the "$\tau$" and "oldest" policies applied to *Dual* with budget $10,000$ on the Image dataset.

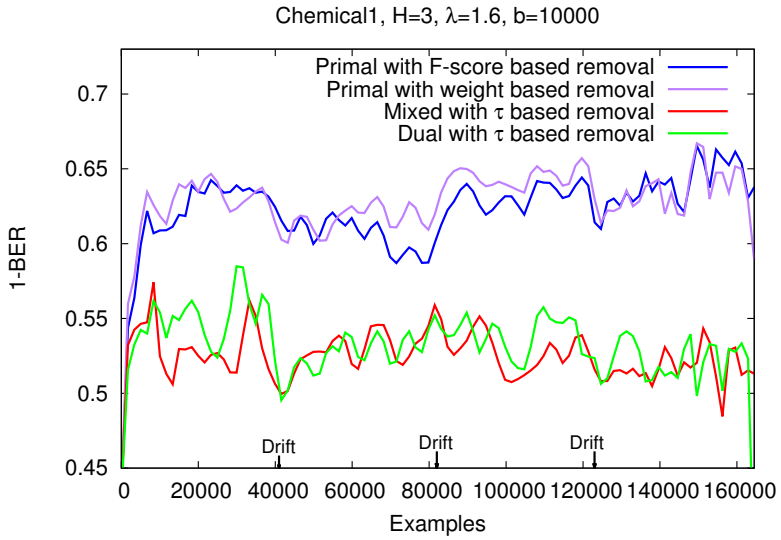


**Figure 5.6**: Comparison between the classification performances of the "$\tau$" and "oldest" policies applied to *Primal*/*Dual* with budget 10,000 on the Chemical1 dataset.

The plots do not clearly indicate the best policy, thus we report in Table 5.1 the expected $1 - \text{BER}$ value over the stream for each algorithm. The best policy for managing the budget for *Dual* and *Primal*/*Dual* is "$\tau$". The best policy for *Primal* is "weight". This might be surprising, since the "F-score" is able to select the most informative features to keep in the model. However, in order to compute F-score values, more information need to be kept in memory with respect to the "weight" approach: 5 budget units per feature versus 2. Our hypothesis is that the reduced discriminative power of the features selected by the "weight" policy is compensated by the fact that a higher number of them can be kept into the model.

Once the best policy for managing the budget has been selected, we proceeded by comparing the three algorithms with respect to classification and computational performances. Figure 5.8, Figure 5.9, and Figure 5.10 report the results of the classification performance comparison, with budget 10,000, on the three datasets. *Primal* has better performances. This is confirmed by its expected $1 - \text{BER}$ values

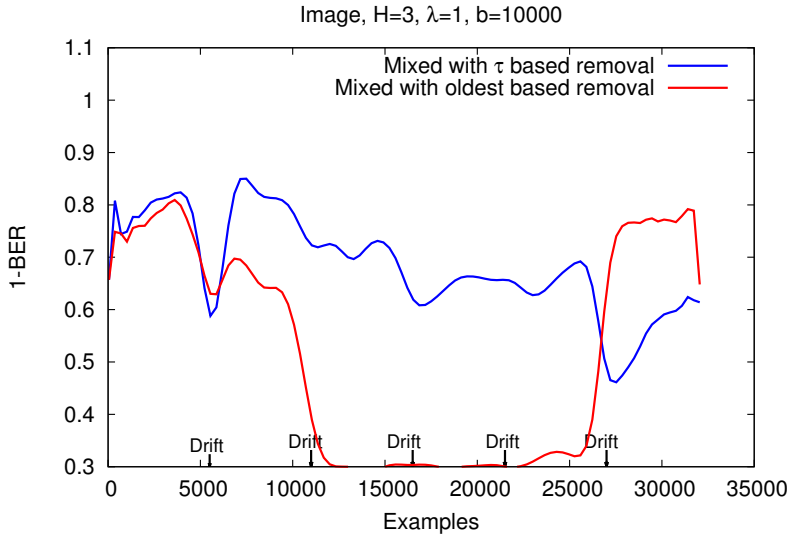


**Figure 5.7**: Comparison between the classification performances of the "$\tau$" and "oldest" policies applied to *Primal*/*Dual* with budget $10,000$ on the Image dataset.

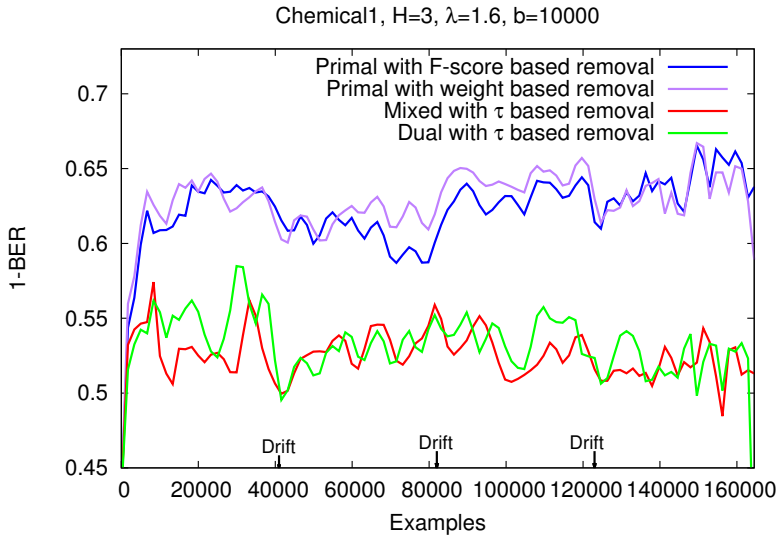


**Figure 5.8**: Performance curves of *Dual* and *Primal*/*Dual* with "$\tau$" policy, and *Primal* with "F-score" and "weight" policies on the Chemical1 dataset. Budget size is $10,000$.

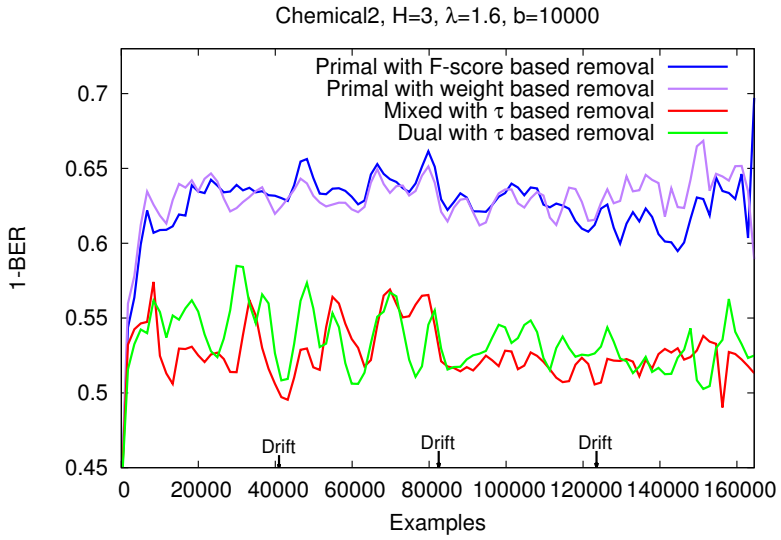


**Figure 5.9**: Performance curves of *Dual* and *Primal/Dual* with "$\tau$" policy, and *Primal* with "F-score" and "weight" policies on the Chemical2 dataset. Budget size is $10,000$.

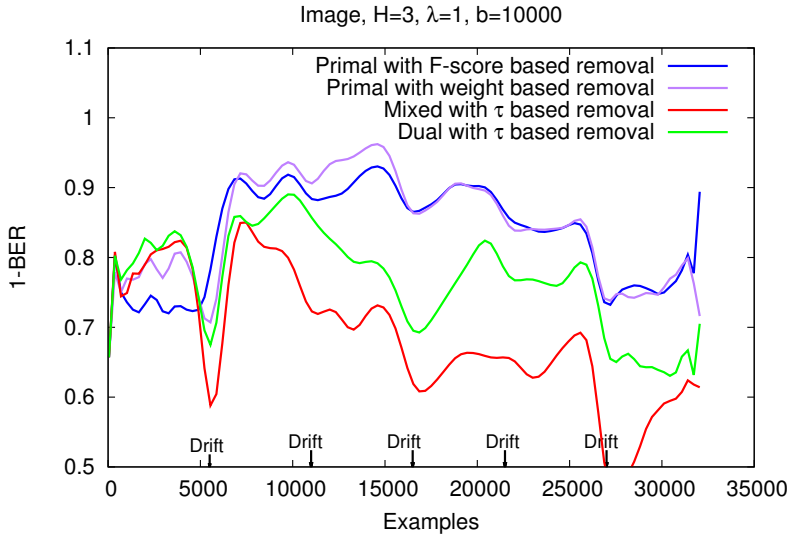


**Figure 5.10**: Performance curves of *Dual* and *Primal/Dual* with "$\tau$" policy, and *Primal* with "F-score" and "weight" policies on the Image dataset. Budget size is $10,000$.

| Algorithm/Dataset | Chemical1 | | Chemical2 | | Image | |
|---|---|---|---|---|---|---|
| Budget | 10000 | 50000 | 10000 | 50000 | 10000 | 50000 |
| *Dual*-$\tau$ | 0.534 | 0.547 | 0.535 | 0.546 | 0.768 | 0.822 |
| *Dual*-oldest | 0.531 | - | 0.515 | - | 0.547 | 0.770 |
| *Primal*/*Dual*-$\tau$ | 0.526 | 0.540 | 0.528 | 0.541 | 0.685 | 0.813 |
| *Primal*/*Dual*-oldest | 0.510 | 0.532 | 0.510 | 0.528 | 0.516 | 0.579 |
| *Primal*-F-score | 0.624 | 0.629 | 0.626 | 0.633 | 0.836 | **0.848** |
| *Primal*-weight | **0.629** | **0.642** | **0.630** | **0.642** | **0.845** | 0.846 |

**Table 5.1**: Expected $1-$BER values for Algorithms 8 (*Dual*), 9 (*Primal*/*Dual*), 10 (*Primal*) with budget $10,000$ and $50,000$, "$\tau$","weight","oldest", "F-score" policies for managing te budget on the Chemical1, Chemical2 and Image datasets. Best results are in bold.

in Table 5.1: 0.629 on Chemical1, 0.630 on Chemical2 and 0.845 on Image. The three figures show that both the "weight" and the "F-score" policies of *Primal* outperform the best policy of *Dual* and *Primal*/*Dual*.

Increasing the budget size up to $50,000$ yields the results reported in Figure 5.11, Figure 5.12 and Figure 5.13. Since *Dual* is very computationally demanding on chemical datasets, we tested only the most promising policy with budget $50,000$. While *Primal* is still better than *Dual* and *Primal*/*Dual*, the gap is reduced especially on the Image dataset. The values in Table 5.1 confirm the visual analysis. Since Algorithms 8 and 9 significantly benefit from the budget increase while *Primal* does not, suggest that, at least for these datasets, *Primal* is able to achieve good results with a smaller budget.

We finally compare the algorithms from a computational point of view. Table 5.2 reports the time required by each algorithm to process the whole stream of data for the Chemical1 and Image datasets using a budget size of $10,000$ and $50,000$. The running times are referred to executions on a machine with two Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz equipped with 256GB of RAM. Notice that the executions use a single core and a very limited amount of RAM. There is a clear gap

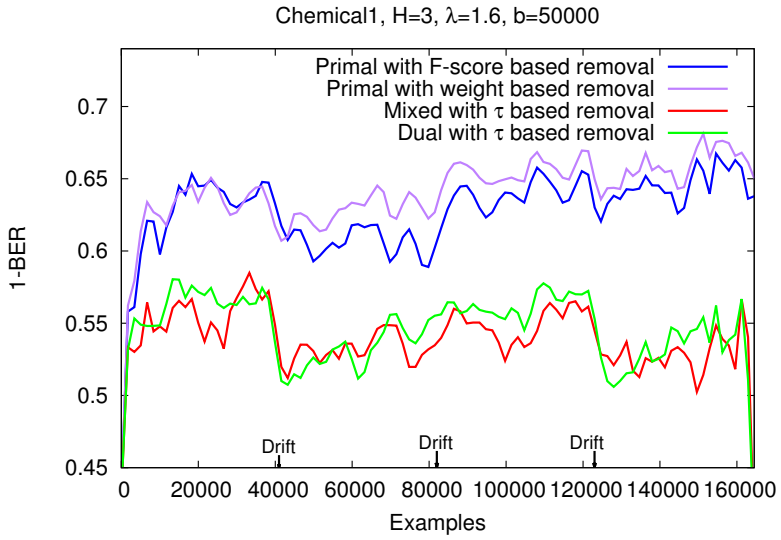


**Figure 5.11**: Performance curves of *Dual* and *Primal/Dual* with "$\tau$" policy, and *Primal* with "F-score" and "weight" policies on the Chemical1 dataset. Budget size is $50,000$.

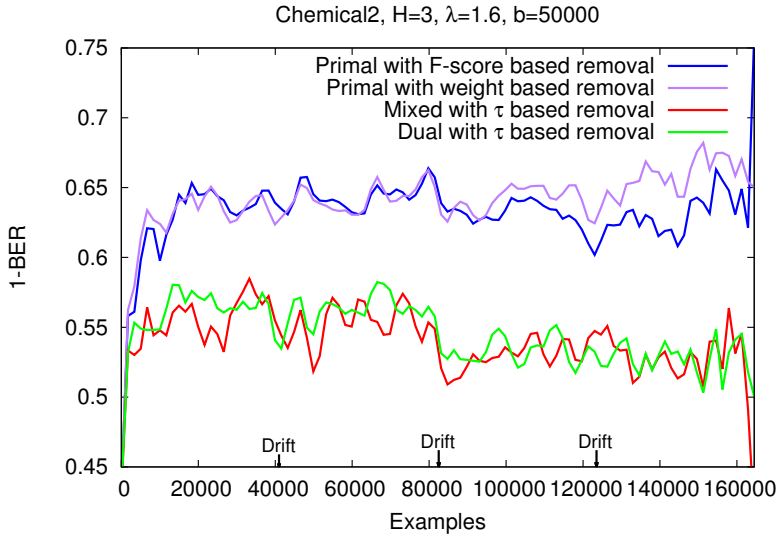


**Figure 5.12**: Performance curves of *Dual* and *Primal/Dual* with "$\tau$" policy, and *Primal* with "F-score" and "weight" policies on the Chemical2 dataset. Budget size is $50,000$.

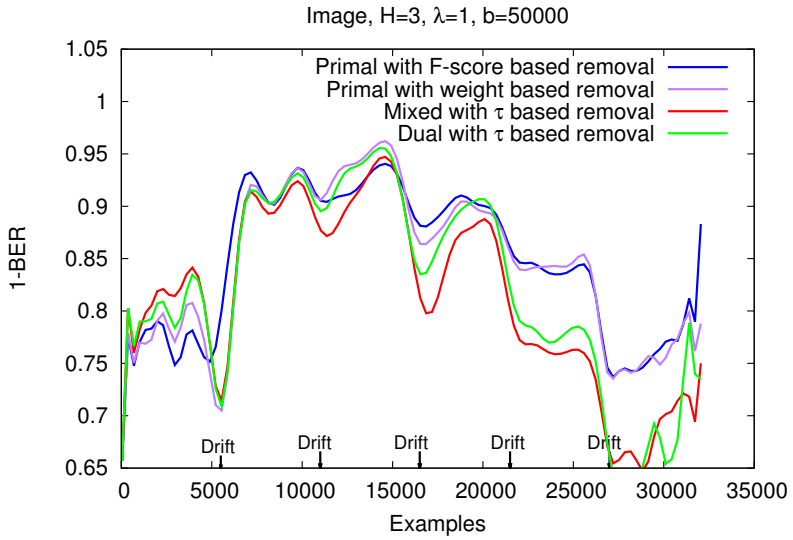


**Figure 5.13**: Performance curves of *Dual* and *Primal/Dual* with "$\tau$" policy, and *Primal* with "F-score" and "weight" policies on the Image dataset. Budget size is $50,000$.

between the computational times of Algorithms *Dual*, *Primal/Dual* and *Primal*. On average, the time needed by the explicit algorithm to process a single example is 0.01 seconds with budget $50,000$, while for the faster of the other two algorithms (*Primal/Dual*) it is 0.13 seconds.

*Dual* is efficient from the point of view of memory occupation, however, that is paid by a very low efficiency when computing the score for a new graph: the kernel values between the input graph and all the graphs in the model have to be computed

| Algorithm/Dataset | Chemical1 | | Image | |
|---|---|---|---|---|
| Budget | 10000 | 50000 | 10000 | 50000 |
| *Dual* | 5h49m06s | 31h59m18s | 58m 48s | 7h 34m |
| *Primal/Dual* | 25m43s | 6h12m58s | 15m 36s | 58m 5s |
| *Primal* | 11m30s | 26m58s | 6m 30s | 34m 7s |

**Table 5.2**: Execution times of Algorithms 8 (*Dual*), 9 (*Primal/Dual*), 10 (*Primal*) on the Chemical1 and Image datasets with budget size $10,000$ and $50,000$.

from scratch. As the values of Table 5.2 indicate, that makes the application of the dual algorithm to graph streams practically infeasible, especially when strict time constraints have to be satisfied. *Primal/Dual* is able to significantly speed up the score computation by storing the explicit feature space representation of each graph in the model. As a consequence, the size of the model may increase significantly, thus reducing the total number of graphs that can be kept in it: *Dual* is able to store in memory approximately 250 graphs of the chemical datasets with budget $10,000$, while *Primal/Dual* only 100 graphs. On the contrary, *Primal* keeps in the model only the most informative features, and so it is able to retain information of all graphs inserted in the model while preserving a very good efficiency. In particular, the primal algorithm made practically feasible the application of graph kernel methods to data streams.

## 5.2 Budget Passive-Aggressive with Lossy Counting

In this section, we move forward from the results presented in the previous section and suggest to use a state-of-the-art online learning algorithm, i.e. Primal Passive-Aggressive with budget for graphs presented in Algorithm 10 in conjunction with an adapted version of a classical result from stream mining, i.e. Lossy Counting (see Section 5.2.1), to efficiently manage the available budget so to keep in memory only relevant features. This approach is appealing because it does not require any particular feature selection technique, it gives a bound on the approximation error, and it is fast. Specifically, we extend Lossy Counting to manage both weighted features and a budget, while preserving theoretical guarantees on the introduced approximation error.

Experiments on real-world datasets show that the proposed approach achieves state-of-the-art classification performances, while being much faster than existing algorithms.

### 5.2.1 Online frequent pattern mining

#### Lossy Counting

The *Lossy Counting* is an algorithm for computing frequency counts exceeding a user-specified threshold over data streams [95]. Let $s \in (0, 1)$ be a support threshold, $\epsilon \ll s$ an error parameter, and $N$ the number of items of the stream seen so far. By using only $O\left(\frac{1}{\epsilon}\log(\epsilon N)\right)$ space to keep frequency estimates, *Lossy Counting*, at any time $N$ is able to *i)* list any item whose frequency exceeds $\epsilon N$; *ii)* avoid listing any item with frequency less than $(s - \epsilon)N$. Moreover, the error of the estimated frequency of any item is at most $\epsilon N$.

In the following the algorithm is sketched, for more details refer to [95]. The stream is divided into buckets $B_i$. The size of a bucket is $|B_i| = \lceil \frac{1}{\epsilon} \rceil$ items. Note that the size of a bucket is determined a priori because $\epsilon$ is a user-defined parameter.

The frequency of an item $f$ in a bucket $B_i$ is represented as $F_{f,i}$, the overall frequency of $f$ is $F_f = \sum_i F_{f,i}$. The algorithm makes use of a data structure $\mathcal{D}$ composed by tuples $(f, \Phi_{f,i}, \Delta_i)$, where $\Phi_{f,i}$ is the frequency of $f$ since it has been inserted into $\mathcal{D}$, and $\Delta_i$ is an upper bound on the estimate of $F_f$ at time $i$.

The algorithm starts with an empty $\mathcal{D}$. For every new event $f$ arriving at time $i$, if $f$ is not present in $\mathcal{D}$, then a tuple $(f, 1, i - 1)$ is inserted in $\mathcal{D}$, otherwise $\Phi_{f,i}$ is incremented by 1. Every $|B_i|$ items all those tuples such that $\Phi_{f,i} + \Delta_i \leq i$ are removed.

The authors prove that, after observing $N$ items, if $f \notin \mathcal{D}$ then $\Phi_{f,N} \leq \epsilon N$ and for each $(f, \Phi_{f,N}, \Delta_N) \in \mathcal{D}$ holds that $\Phi_{f,N} \leq F_f \leq \Phi_{f,N} + \epsilon N$.

### 5.2.2 Online frequent pattern mining with real weights

In this section, we propose an extension to Lossy Counting (LC) algorithm, presented in Section 5.2.1 that works on a fixed memory budget and able to deal with events weighted by positive real values. Then, in Section 5.2.3 we show how to apply the proposed algorithm to the Passive-Aggressive algorithm in the case of a stream of graphs.

**LCB: Lossy Counting with budget for weighted events**

The original Lossy Counting algorithm does not guarantee that the available memory will be able to contain the $\epsilon$-deficient synopsis $\mathcal{D}$ of the observed items. Because of that, we propose an alternative definition of Lossy Counting that addresses this issue and, in addition, is able to deal with weighted events. Specifically, we assume that the stream emits events $e$ constituted by couples $(f, \phi)$, where $f$ is an item id and $\phi \in \mathbb{R}^+$ is a positive weight associated to $f$. Different events may have the same item id while having different values for the weight, e.g. $e_1 = (f, 1.3)$ and $e_2 = (f, 3.4)$. We are interested in maintaining a synopsis that, for each observed item id $f$, collects the sum of weights observed in association with $f$. Moreover, we have to do that on a memory budget. To manage the budget, we propose to decouple the $\epsilon$ parameter from the size of the bucket: we use buckets of variable sizes, i.e. the $i$-th bucket $B_i$ will contain all the occurrences of events which can be accommodated into the available memory, up to the point that $\mathcal{D}$ will use the full budget and no new event can be inserted into it. This policy implies that the approximation error will vary according to the size of the bucket, i.e. $\epsilon_i = \frac{1}{|B_i|}$, where $|B_i|$ is the sum of the weights of all the events in bucket $B_i$, that is $|B_i| = \sum_{(f,\phi)\in B_i} \phi$. Having buckets of different sizes, the index of the current bucket $b_{current}$ is defined as $b_{current} = 1 + \max_k[\sum_{s=1}^{k} |B_s| < N]$. Deletions occur when there is no more space to insert a new event in $\mathcal{D}$. Trivially, if $N$ events have been observed when the $i$-th deletion occurs, $\sum_{s=1}^{i} |B_s| = N$ and, by definition, $b_{current} \leq \sum_{s=1}^{i} \epsilon_s |B_s|$.

Let $\phi_{f,i}(j)$ be the weight of the $j$-th occurrence of $f$ in $B_i$, and the cumulative weight associated to $f$ in $B_i$ as $w_{f,i} = \sum_{j=1}^{F_{f,i}} \phi_{f,i}(j)$. The total weight associated to bucket $B_i$ can then be defined as $W_i = \sum_{f\in\mathcal{D}} w_{f,i}$. We now want to define a synopsis $\mathcal{D}$ such that, having observed $N = \sum_{s=1}^{i} |B_s|$ events, the estimated cumulative weights are *less* than the true cumulative weights by at most $\sum_{s=1}^{i} \epsilon_i W_i$, where we recall that $\epsilon_i = \frac{1}{|B_i|}$. In order to do that we define the cumulated weight for item $f$ in $\mathcal{D}$, after having observed all the events in $B_i$, as $\Phi_{f,i} = \sum_{s=i_f}^{i} w_{f,s}$, where $i_f \leq i$ is the largest index of the bucket where $f$ has been inserted in $\mathcal{D}$. The deletion test

is then defined as

$$\Phi_{f,i} + \Delta_{i_f} \leq \Delta_i \tag{5.4}$$

where $\Delta_i = \sum_{s=1}^{i} \frac{W_s}{|B_s|}$. However, we have to cover also the special case in which the deletion test is not able to delete any item from $\mathcal{D}$[2]. We propose to solve this problem as follows: we assume that in this case a new bucket $B_{i+1}$ is created containing just a single *ghost* event $(f_{ghost}, \Phi_i^{min} - \Delta_i)$, where $f_{ghost} \notin \mathcal{D}$ and $\Phi_i^{min} = \min_{f \in \mathcal{D}} \Phi_{f,i}$, that we do not need to store in $\mathcal{D}$. In fact, when the deletion test is run, $\Delta_{i+1} = \Delta_i + \frac{W_{i+1}}{|B_{i+1}|} = \Phi_i^{min}$ since $W_{i+1} = \Phi_i^{min} - \Delta_i$ and $|B_{i+1}| = 1$, which will cause the ghost event to be removed since $\Phi_{ghost,i+1} = \Phi_i^{min} - \Delta_i$ and $\Phi_{ghost,i+1} + \Delta_i = \Phi_i^{min}$. Moreover, since $\forall f \in \mathcal{D}$ we have $\Phi_{f,i} = \Phi_{f,i+1}$, all $f$ such that $\Phi_{f,i+1} = \Phi_i^{min}$ will be removed. By construction, there will always be one such item.

**Theorem 5.1.** *Let $\Phi_{f,i}^{true}$ be the true cumulative weight of $f$ after having observed $N = \sum_{s=1}^{i} |B_s|$ events. Whenever an entry $(f, \Phi_{f,i}, \Delta)$ gets deleted, $\Phi_{f,i}^{true} \leq \Delta_i$.*

*Proof.* We prove by induction. Base case: $(f, \Phi_{f,1}, 0)$ is deleted only if $\Phi_{f,1} = \Phi_{f,1}^{true} \leq \Delta_1$. Induction step: let $i^* > 1$ be the index for which $(f, \Phi_{f,i^*}, \Delta)$ gets deleted. Let $i_f < i^*$ be the largest value of $b_{current}$ for which the entry was inserted. By induction $\Phi_{f,i_f}^{true} \leq \Delta_{i_f}$ and all the weighted occurrences of events involving $f$ are collected in $\Phi_{f,i^*}$. Since $\Phi_{f,i^*}^{true} = \Phi_{f,i_f}^{true} + \Phi_{f,i^*}$ we conclude that $\Phi_{f,i^*}^{true} \leq \Delta_{i_f} + \Phi_{f,i^*} \leq \Delta_{i^*}$.

□

**Theorem 5.2.** *If $(f, \Phi_{f,i}, \Delta) \in \mathcal{D}$, $\Phi_{f,i} \leq \Phi_{f,i}^{true} \leq \Phi_{f,i} + \Delta$.*

*Proof.* If $\Delta = 0$ then $\Phi_{f,i} = \Phi_{f,i}^{true}$. Otherwise, $\Delta = \Delta_{i_f}$, and an entry involving $f$ was possibly deleted sometimes in the first $i_f$ buckets. From the previous theorem, however, we know that $\Phi_{f,i_f}^{true} \leq \Delta_{i_f}$, so $\Phi_{f,i} \leq \Phi_{f,i}^{true} \leq \Phi_{f,i} + \Delta$. □

We notice that, if $\forall e = (f, \phi)$, $\phi = 1$, the above theorems apply to the not weighted version of the algorithm.

[2]E.g, consider the stream $(f_1, 10)$, $(f_2, 1)$, $(f_3, 10)$, $(f_4, 15)$, $(f_1, 10)$, $(f_3, 10)$, $(f_5, 1)$, .. and budget equal to 3 items: the second application of the deletion test will fail to remove items from $\mathcal{D}$.

Let now analyze the proposed algorithm. Let $\mathcal{O}_i$ be the set of items that have survived the last (i.e., $(i-1)$-th) deletion test and that have been *observed* in $B_i$, $\mathcal{I}_i$ be the set of items that have been *inserted* in $\mathcal{D}$ after the last (i.e., $(i-1)$-th) deletion test; notice that $\mathcal{O}_i \cup \mathcal{I}_i$ is the set of all the items observed in $B_i$, however it may be properly included into the set of items stored in $\mathcal{D}$. Let $w_i^{\mathcal{O}} = \sum_{f \in \mathcal{O}_i} F_{f,i}$ and $w_i^{\mathcal{I}} = \sum_{f \in \mathcal{I}_i} F_{f,i}$. Notice that $w_i^{\mathcal{I}} > 0$, otherwise the budget would not be fully used; moreover, $|B_i| = w_i^{\mathcal{O}} + w_i^{\mathcal{I}}$. Let $W_i^{\mathcal{O}} = \sum_{f \in \mathcal{O}_i} w_{f,i}$ and $W_i^{\mathcal{I}} = \sum_{f \in \mathcal{I}_i} w_{f,i}$. Notice that $W_i = W_i^{\mathcal{O}} + W_i^{\mathcal{I}}$, and that $\frac{W_i}{|B_i|} = \frac{w_i^{\mathcal{O}}}{|B_i|}[\widehat{W}_i^{\mathcal{O}} - \widehat{W}_i^{\mathcal{I}}] + \widehat{W}_i^{\mathcal{I}} = p_i^{\mathcal{O}}[\widehat{W}_i^{\mathcal{O}} - \widehat{W}_i^{\mathcal{I}}] + \widehat{W}_i^{\mathcal{I}}$, where $p_i^{\mathcal{O}} = \frac{w_i^{\mathcal{O}}}{|B_i|}$ is the fraction of updated items in $B_i$, $\widehat{W}_i^{\mathcal{O}} = \frac{W_i^{\mathcal{O}}}{w_i^{\mathcal{O}}}$ is the average of the updated items, $\widehat{W}_i^{\mathcal{I}} = \frac{W_i^{\mathcal{I}}}{w_i^{\mathcal{I}}}$ is the average of the inserted items. Thus $\frac{W_i}{|B_i|}$ can be expressed as a convex combination of the average of the updated items and the average of the inserted items, with combination coefficient equal to the fraction of updated items in the current bucket. Thus, the threshold $\Delta_i$ used by the $i$-th deletion test can be written as

$$\Delta_i = \sum_{s=1}^{i} p_s^{\mathcal{O}}[\widehat{W}_s^{\mathcal{O}} - \widehat{W}_s^{\mathcal{I}}] + \widehat{W}_s^{\mathcal{I}} \tag{5.5}$$

Combining eq. (5.4) with eq. (5.5) we obtain

$$\Phi_{f,i} \leq \sum_{s=i_f}^{i} p_s^{\mathcal{O}}[\widehat{W}_s^{\mathcal{O}} - \widehat{W}_s^{\mathcal{I}}] + \widehat{W}_s^{\mathcal{I}}.$$

If $f \in \mathcal{I}_i$, then $i_f = i$ and the test reduces to $w_{f,i} \leq p_i^{\mathcal{O}}[\widehat{W}_i^{\mathcal{O}} - \widehat{W}_i^{\mathcal{I}}] + \widehat{W}_i^{\mathcal{I}}$. If $f \notin \mathcal{I}_i$ (notice that this condition means that $i_f < i$, i.e. $f \in \mathcal{I}_{i_f}$), then the test can be rewritten as $w_{f,i} \leq p_i^{\mathcal{O}}[\widehat{W}_i^{\mathcal{O}} - \widehat{W}_i^{\mathcal{I}}] + \widehat{W}_i^{\mathcal{I}} - \sum_{s=i_f}^{i-1} \gamma_{f,s}$, where $\gamma_{f,s} = w_{f,s} - p_s^{\mathcal{O}}[\widehat{W}_s^{\mathcal{O}} - \widehat{W}_s^{\mathcal{I}}] + \widehat{W}_s^{\mathcal{I}}$ is the *credit/debit* gained by $f$ for bucket $B_s$. Notice that, by definition, $\forall k \in \{i_f, \ldots, i\}$ the following holds $\sum_{s=i_f}^{k} \gamma_{f,s} > 0$.

### 5.2.3 LCB-PA on streams of graphs

This section describes an application of the results in the previous section to the *Primal* PA algorithm presented in Algorithm 10. We will refer to the algorithm described in this section as LCB-PA. The goal is to use the synopsis $\mathcal{D}$ created and

maintained by LCB to approximate at the best $w$, according to the available budget. This is obtained by LCB since only the most influential $w_i$ values will be stored into $\mathcal{D}$.

A difficulty in using the LCB synopsis is due to the fact that LCB can only manage positive weights. We overcome this limitation by storing for each feature $f$ in $\mathcal{D}$ a version associated to positive weight values and a version associated to (the modulus) of negative values.

Let's detail the proposed approach in the following. First of all, recall that we consider the features generated by the $\phi()$ mapping described in Section 4. When a

**Algorithm 11** LCB-PA Algorithm.

1: Initialize $\mathcal{D}$: $\{\}$, $\Delta$: $\Delta_0 = 0$
2: **for each round** $t$ **do**
3: Receive an instance $G_t$ from the stream
4: Compute the feature set $\phi(G_t) = \{(f, \phi_{f,t}) | \text{f is a dimension in the feature space}\}$
5: Compute the score of $G_t$: $S(G_t) = w_t \cdot \phi(G_t)$
6: Receive the correct classification of $G_t$: $y_t$
7: **if** $y_t S(G_t) \leq 1$ **then**
8: **if** $|w + \phi(G_t)| > \mathcal{B}$ **then**
9: Deletion procedure:
10: delete all $f \in \mathcal{D}$ s.t. $\Phi_{f,i} + \Delta_{i_f} \leq \Delta_t$
11: **end if**
12: update the hypothesis: $\mathcal{D}_{t+1/2} = \mathcal{D}_t \cup \{(f, \Phi_{f,t} + \tau_t y_t \phi_{f,t}) | (f, \phi_{f,t}) \in \phi(G_t) \wedge (f, \cdot) \in \mathcal{D}\}\}$
13: update the hypothesis: $\mathcal{D}_{t+1} = \mathcal{D}_t \cup \{(f, \tau_t y_t \phi_{f,t}) | (f, \phi_{f,t} + \Delta_t) \in \phi(G_t) \wedge (f, \cdot) \notin \mathcal{D}\}$
14: **end if**
15: **end for**

new graph arrives from the stream (line 3 of Algorithm 11), it is first decomposed into a bag of features according to the process described in Section 4. Then the score for the graph is computed according to eq. (2.4) (line 4 of Algorithm 11). If the graph is misclassified (the test on line 7 of Algorithm 11 is true), then the synopsis $\mathcal{D}$ (i.e., $w$) has to be updated. In this scenario, an event corresponds to a feature $f$ of the current input graph $G$.

The weight $\phi_{f,i}(j)$ of a feature $f$ appearing for the $j$-th time in the $i$-th bucket $B_i$, is computed multiplying its frequency in the graph $G$ with the corresponding $\tau$ value computed for $G$ according to eq. (2.5), which may result in a negative weight.

$\Phi_{f,i}$ is the weighted sum of all $\phi_{f,i}(j)$ values of the feature $f$ since $f$ has last been inserted into $\mathcal{D}$. In this way, the LCB algorithm allows to maintain an approximate version of the full $w$ vector by managing the feature selection and model update steps (lines 7-10 of Algorithm 11).

In order to cope with negative weights, the structure $\mathcal{D}$ is composed by tuples $(f, |f|, \Phi^+_{f,i}, \Phi^-_{f,i})$, where $\Phi^+_{f,i}$ corresponds to $\Phi_{f,i}$ computed only on features whose graph $G$ has positive classification ($\Phi^-_{f,i}$ is the analogous for the negative class). Whenever the size of $\mathcal{D}$ exceeds the budget $\mathcal{B}$, all the tuples satisfying eq. (5.4) are removed from $\mathcal{D}$. Here $\Delta_i$ can be interpreted as the empirical mean of the $\tau$ values observed in the current bucket. Note that the memory occupancy of $\mathcal{D}$ is now $4|w|$, where $|w|$ is the number of features in $\mathcal{D}$. The update rule of eq. (2.5) is kept. However, when a new tuple is inserted into $\mathcal{D}$ at time $N$, the $\Delta_N$ value is added to the $\tau$ value computed for $G$. The idea is to provide an upper bound to the $\Phi^+_{f,N}$, $\Phi^-_{f,N}$ values that might have been deleted in the past from $\mathcal{D}$. Theorem 5.1 shows that indeed $\Delta_N$ is such upper bound.

### 5.2.4 Experiments

In this section, we report an experimental evaluation of the algorithm proposed in Section 5.2.3 (*LCB-PA*), compared to the algorithms presented in Section 5.1. Our aim is to show that the proposed algorithm has comparable, if not better, classification performances than the baselines while being much faster to compute.

#### Experimental Setup

We used the same datasets described in Section 4.5.2, and one new chemical dataset. This new dataset, referred to as *Chemical3*, is the concatenation of the datasets *Chemical1* and *Chemical2* described in the same section. The reason why we generated this new dataset is to understand the difference in the classifier behavior when starting from an empty hypothesis or when recovering from a concept drift.

We considered as baseline Algorithm 10 (*Primal*) and Algorithm 8 (*Dual*), both using the $\phi()$ mapping defined in Section 4.3.1. We compared them against Algorithm 11 presented in Section 5.2.3 (*LCB-PA* in the following). From the experiments reported in Section 5.1.2, we determined the best removal policy and kernel parameters. The best removal policy for *Primal* PA algorithm is the one removing the feature with lowest value in $w$. The best policy for the *Dual* PA removes the example with the lowest $\tau$ value. We did not consider the *Mixed* algorithm since its performance are lower w.r.t. *Primal* (see Section 5.1.2). The $C$ parameter has been set to 0.01, while the kernel parameters has been set to $\lambda = 1.6$, $h = 3$ for chemical datasets, and to $\lambda = 1$, $h = 3$ for the *Image* dataset.

### Results and discussion

The measure we adopted for performance assessment is the 1-(*BER*), explained in Section 5.1.2. We recall that higher values mean better performances. In our experiments, we sampled 1-BER every 50 examples. In Table 5.3 are reported, for each algorithm and budget, the average of the $1-$BER values over the whole stream.

It is easy to see that the performances of the *Dual* algorithm are poor. Indeed, there is no single algorithm/budget combination in which the performance of this algorithm is competitive with the other two. This is probably due to the fact that support graphs may contain many features that are not discriminative for the tasks. Let us consider the *Primal* and the *LCB-PA* algorithms. Their performances are almost comparable. Concerning the chemical datasets, Figure 5.14 and Figure 5.15 report the details about the performance of the *Chemical*1 stream. Similar plots can be obtained for *Chemical*2 dataset and thus are omitted. Figure 5.16 reports the performance plot on the *Chemical*3 stream with $\mathcal{B} = 50,000$. Observing the plots and the corresponding $1 - BER$ values, it's clear that on the three datasets the algorithm *Primal* performs better than *LCB-PA* for budget sizes up to $25,000$. For higher budget values, the performances of the two algorithms are very close, with *LCB-PA* performing better on *Chemical*1 and *Chemical*2 datasets, while the

| | Budget | Dual | Primal | LCB-PA | PA (B=$\infty$) |
|---|---|---|---|---|---|
| *Chemical*1 | 10,000 | 0.534 | **0.629** | 0.608 | |
| | 25,000 | 0.540 | **0.638** | 0.637 | |
| | 50,000 | 0.547 | 0.642 | **0.644** | **0.644** |
| | 75,000 | - | 0.643 | **0.644** | ($\mathcal{B} = 182,913$) |
| | 100,000 | - | **0.644** | **0.644** | |
| *Chemical*2 | 10,000 | 0.535 | **0.630** | 0.610 | |
| | 25,000 | 0.541 | **0.638** | **0.638** | |
| | 50,000 | 0.546 | 0.642 | **0.644** | 0.644 |
| | 75,000 | - | 0.644 | **0.645** | ($\mathcal{B} = 182,934$) |
| | 100,000 | - | 0.644 | **0.645** | |
| *Chemical*3 | 10,000 | 0.532 | **0.640** | 0.601 | |
| | 25,000 | 0.542 | **0.652** | 0.643 | |
| | 50,000 | 0.549 | **0.658** | **0.658** | **0.661** |
| | 75,000 | - | **0.660** | **0.660** | ($\mathcal{B} = 183,093$) |
| | 100,000 | - | **0.661** | **0.661** | |
| *Image* | 10,000 | 0.768 | 0.845 | **0.855** | |
| | 25,000 | 0.816 | 0.846 | **0.853** | |
| | 50,000 | 0.822 | 0.846 | **0.852** | 0.852 |
| | 75,000 | - | 0.846 | **0.852** | ($\mathcal{B} = 534,903$) |
| | 100,000 | - | 0.845 | **0.852** | |

**Table 5.3**: $1 - BER$ values for *Primal*, *Dual* and *LCB-PA* algorithms, with different budget sizes, on *Chemical*1, *Chemical*2, *Chemical*3 and *Image* datasets. Best results for each row are reported in bold. The missing data indicates that the execution did not finish in 3 days.

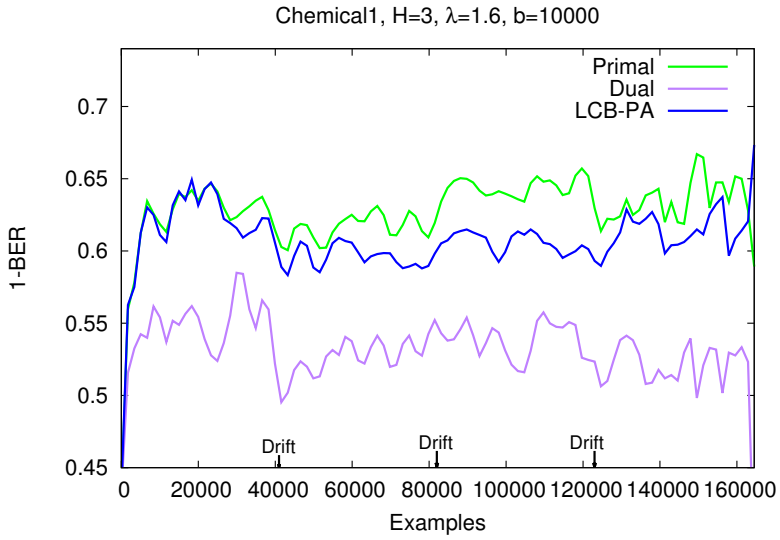


**Figure 5.14**: Comparison of $1 - BER$ measure for *Primal*, *Dual* and *LCB-PA* algorithms on *Chemical*1 dataset with budget $10,000$.

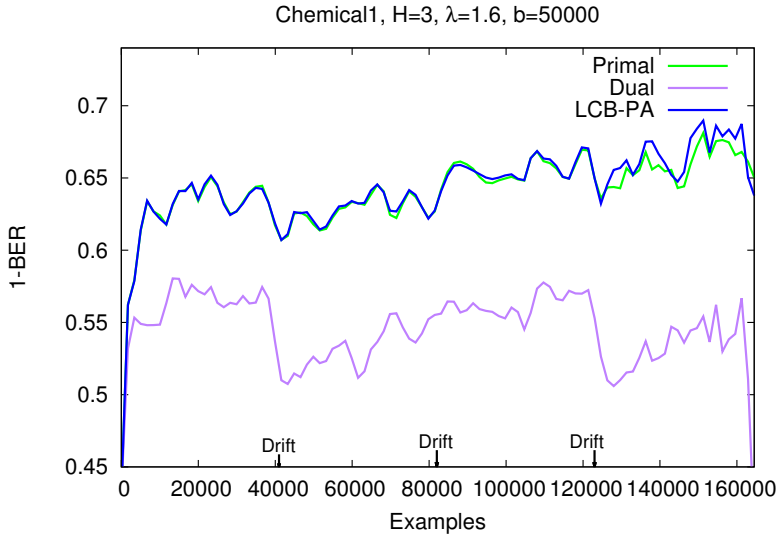


**Figure 5.15**: Comparison of $1 - BER$ measure for *Primal*, *Dual* and *LCB-PA* algorithms on *Chemical*1 dataset with budget $50,000$.

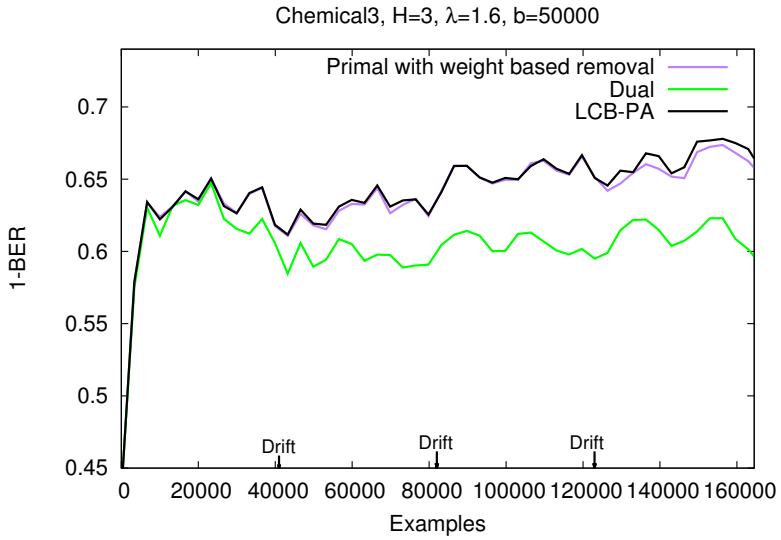


**Figure 5.16**: Comparison of $1 - BER$ measure for *Primal*, *Dual* and *LCB-PA* algorithms on *Chemical*3 dataset with budget $50,000$.

performances are exactly the same on *Chemical*3. Let's analyze in more detail this behavior of the algorithms. In *Primal* every feature uses 3 budget units, while in *LCB-PA* 4 units are consumed. In the considered chemical streams, on average every example is made of 54.56 features. This means that, with budget $10,000$, *Primal* can store approximatively the equivalent of 60 examples, while *LCB-PA* only 45, i.e. a 25% difference. When the budget increases, such difference reduces. The *LCB-PA* performs better then the *Primal* PA with budget size of $50,000$ or more. Notice that *LCB-PA*, with budget over a certain threshold, reaches or improves over the performances of the *PA* with $\mathcal{B} = \infty$.

On the *Image* dataset we have a different scenario. *LCB-PA* with budget $10,000$ already outperforms the other algorithms.

Table 5.4 reports the time needed to process the streams. It is clear from the table that *LCB-PA* is by far the fastest algorithm. *Dual* algorithm is slow because it works in the dual space, so it has to calculate the kernel function several times.

The plot in Figure 5.17 reports the memory occupancy of *Primal* and *LCB-PA* algorithms. Notice that when the *LCB-PA* performs the cleaning procedure,

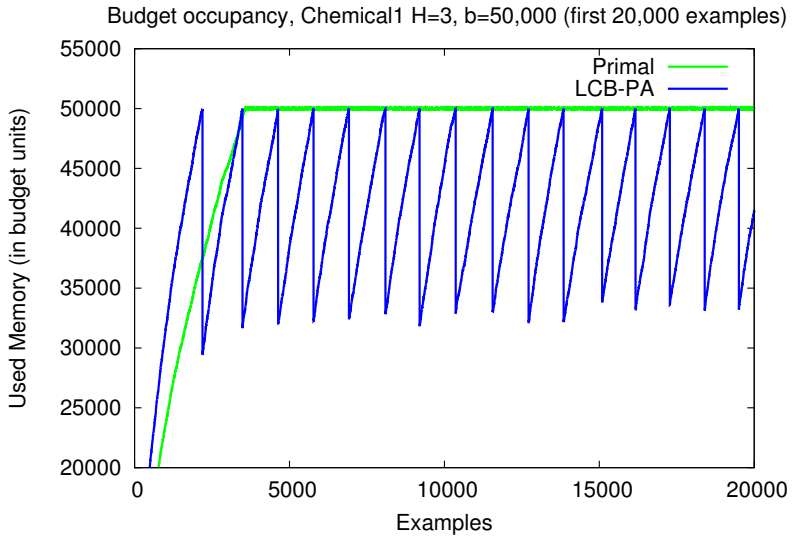


**Figure 5.17**: Evolution of memory occupation of *Primal* and *LCB-PA* algorithms on the first 20,000 examples of the *Chemical*1 dataset.

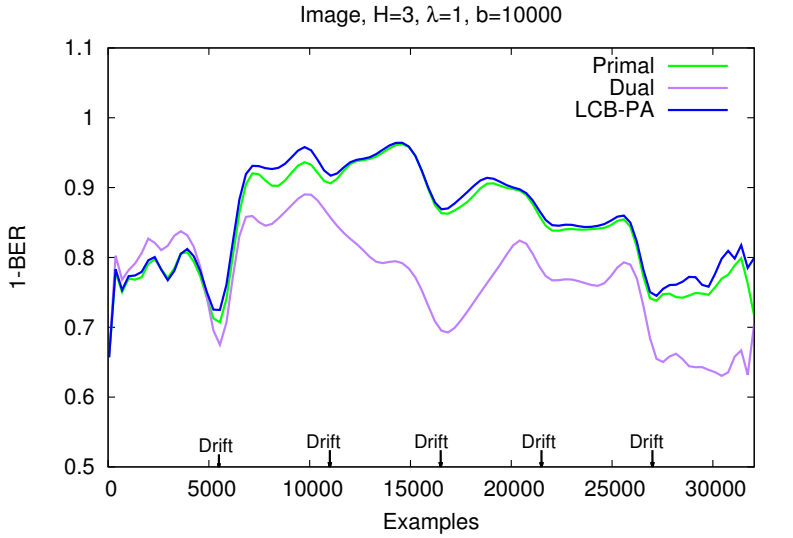


**Figure 5.18**: Comparison of $1 - BER$ measure for *Primal*, *Dual* and *LCB-PA* algorithms on *Image* dataset with budget $10,000$.

| | *Chemical*1 | | *Chemical*2 | |
|---|---|---|---|---|
| Budget | 10,000 | 50,000 | 10,000 | 50,000 |
| Dual | 5h44m | 31h02m | 5h42m | 31h17m |
| Primal | 44m19s | 1h24m | 43m54s | 1h22m |
| LCB-PA | 4m5s | 3m55s | 3m52s | 3m57s |
| | *Chemical*3 | | *Image* | |
| Budget | 10,000 | 50,000 | 10,000 | 50,000 |
| Dual | 10h50m | 60h31m | 49m34s | 6h18m |
| Primal | 1h32m | 2h44m | 7m21s | 33m18s |
| LCB-PA | 7m41s | 7m45s | 0m49s | 0m50s |

**Table 5.4**: Execution times of *Dual*, *Primal* and *LCB-PA* algorithms for budget values of 10,000 and 50,000.

it removes far more features from the budget then *Primal*. As a consequence, *Primal* calls the cleaning procedure much more often than *LCB-PA*, thus inevitably increasing its execution time. For example, on *Chemical*1 with budget 50,000, *Primal* executes the cleaning procedure 132,143 times, while *LCB-PA* only 142 times. Notice that a more aggressive pruning for the baselines could be performed by letting a user define how many features have to be removed. Such a parameter could be chosen a piori, but it would be unable to adapt to a change in the data distribution and there would be no guarantees on the quality of the resulting model. The parameter can also be tuned on a subset of the stream, if the problem setting allows it. In general, developing a policy to tune the parameter is not straightforward and the tuning procedure might have to be repeated multiple times, thus increasing significantly the overall computational burden of the learning algorithm. Note that, on the contrary, our approach employs a principled, automatic way, described in Equation (5.5), to determine how many and which features to prune.

# Chapter 6

# Application to RNA

> So Keys did what any dedicated researcher would do.. he threw up the data that didn't fit and published his results.
>
> from the movie "Fat Head"

In Chapter 3, we showed some examples of learning applications on graphs. There are other applications where it is possible to represent instances in a graph-structured form, but where it's not so straightforward. An example of this kind of applications is the function prediction of RNA sequences. If we are able to find a good representation, we can achieve good results with the application of state-of-the-art graph kernels. In the last few years, techniques of whole genome sequencing, that allow to determine the complete DNA sequence of an organism, have received increasing attention. The 98% of this sequence is not translated into proteins, and may encode functional RNA molecules. These molecules may have important roles in cells. However, the exact mechanisms are not understood yet (see Section 6.3.2).

For the problem faced in this chapter, the computational complexity of the method is a key aspect for the applicability of the techniques to real-world data, i.e. the target data set to analyze (the human genome) is a string of approximately $3.2 \times 10^9$ base pairs.

The application of graph kernels to this problem is made possible by the recent definition of fast kernel functions allowing for an explicit feature space representation (see Chapter 4) that, in conjunction with fast learning algorithms, gives a learning method that can be reliable on such a problem. This chapter is organized as follows: we start in Section 6.1 with some basic concepts about RNA. In Section 6.2 we formally state the problem we face from a machine learning point of view. In Section 6.3 we present the existing methods we adopted as baselines. Then in Section 6.4 we focus on the different issues that it is necessary to face in order to represent RNA as graphs. In Section 6.5 we present our novel approach based on graph kernels and the problems we need to solve. This section presents a novel kernel specifically designed for graphs representing RNA sequences, that is an extension of the kernel framework presented in Chapter 4. Section 6.6 presents promising experimental results.

## 6.1 Introduction to RNA

Ribonucleic acid (RNA) is a family of biological molecules that perform multiple vital roles in the coding, decoding, regulation, and expression of genes. Like DNA, RNA is assembled as a chain of components called nucleotides, but it is usually single-stranded. There are four types of nucleotides in RNA: Adenine (A), Cytosine (C), Guanine (G) and Uracil (U). RNA has various functions. For example messenger RNA (mRNA) is used by all cellular organisms to carry the genetic information that directs the synthesis of proteins. A non-coding RNA (ncRNA or functional RNA) is an RNA molecule that is not translated into a protein. The study of ncRNA genes is receiving increasing attention because of new accumulating evidence that large portions of genome are transcribed into RNA molecules of mostly unknown functions, as well as the discovery of novel non-coding RNA types and functional RNA elements. This class of RNA genes produces transcripts that function directly as structural, catalytic or regulatory RNAs, rather than expressing information on how to encode proteins like mRNA. Recent research focused on

function prediction of this type of RNA e.g. [54].

However, working on ncRNA sequences is difficult because they have little statistical signals. Indeed, as we will explain in Section 6.3.2, the analysis of these RNAs has revealed that they are highly structured.

## 6.2 Problem statement

In this section we will formally state the problem we are facing from a machine learning point of view.

With the evolution of computational techniques applicable to ncRNA sequences, novel challenges have been faced. A first problem was the identification of non-coding sequences from a single (long) genome sequence: one of the first proposed techniques is based on *neural networks* [24]. More recently, a more complex problem has been approached. This is the problem we deal with in this chapter. Given a set of small sequences known to be some particular ncRNA genes, the problem is to build a classifier that predicts whether or not new (unknown) sequences share the same function. From a machine learning point of view, this is a multiclass classification problem on sequence data. The dataset we consider has 47 different classes, and is a subset of the Rfam database [65] that comprises more than 2000 RNA families. The sequence length is fixed to 200 nucleotides. Since the considered RNA sequences are shorter than that, we added padding before and after the original sequence, respecting the same statistics on the nucleotide distribution.

The first methods that have been proposed in literature were based on comparative genome analysis [109]. Currently, the approach that seems to be the most promising is based on kernel methods: indeed, a kernel for ncRNAs that computes the similarity between sequences in terms of the similarity between its secondary structure has been proposed in [116] (see Section 6.3.3). Nonetheless this approach (as well as other methods that consider the RNA secondary structure) suffers from severe scalability problems.

In order to solve the issues of existing methods, we decided to follow a different approach. We transform every sequence into the corresponding secondary structures, defining a suited representation for these structures as graphs, and finally apply learning algorithms based on graph kernels. Several problems arise and have to be faced:

- multiple instance learning, i.e. the RNA secondary structure prediction algorithms output a set of candidate secondary structures rather than a single structure. Thus, we have a bag of structures for each RNA sequence. This problem requires special techniques to be solved in an efficient way;
- representation issues, i.e. we have to define a way to transform the secondary structures into graphs in a meaningful way (for our task).

In Section 6.5 we will discuss in detail the solution we propose for these problems.

## 6.3 Existing methods

There are several different methods for the classification of RNA molecules. Here we focus only on the most representative ones.

### 6.3.1 Sequence-based methods

The first class of methods consists in the homology-based ones, where the most known algorithm is BLAST (Basic Local Alignment Search Tool). To use it, a sequence of interest has to be submitted to the algorithm. BLAST will then compare the sequence the user submitted with the sequences in its database, and tell the user which database sequences most closely match the user-submitted one. Given a query q and a target sequence, BLAST:

- find substrings of length k (k-mers) of score at least t, also called hits. k is normally 3 to 5 for amino acids and 12 for nucleotides.

- Extend each hit to a locally maximal segment. Terminate the extension when the reduction in score exceeds a pre-defined threshold.

- Report maximal segments above score S.

To efficiently compute the similarity between sequences, the software starts computing the match of $k$-mers, that are subsequences of length $k$, and then expands the matching regions to find $k+1$-mers. This process is repeated Then it's possible to infer a class for the input sequence based on a majority vote among the matching ones.

### 6.3.2 Structure-based methods

#### RNA secondary structure

Like DNA, most biologically active RNAs contain self-complementary sequences that allow parts of the RNA to fold. The analysis of these RNAs has revealed that they are highly structured. Unlike DNA, their structures do not consist of long double helices but rather collections of short helices packed together into structures akin to proteins. The structure of ncRNAs is important because it is thought to provide insights into its biological function [96]. In the folding process, characteristic nucleotide base-pairing and stacking interactions play significant roles and are governed by molecular forces acting on and within any molecule in aqueous solutions (e.g. electrostatic interactions) [135]. RNA secondary structure is the 2-dimensional representation of the folding of the RNA sequence, driven from the bonds that arise between the nucleotides. The secondary structures of RNAs are generally composed of stems, hairpins, bulges, interior loops and multi-branches, and thus can be easily represented as graphs, as we will show in Section 6.4.2. The adopted shapes or folds can be highly complex and are capable of carrying out a variety of molecular functions, such as binding metabolites and proteins with high specificity. RNA is particularly suited for hybridizing with nucleotide sequences allowing for highly specific targeting of genes and genomic regions. Furthermore, it is conceivable that two

ncRNA molecules with completely different nucleotide compositions would still fold to form the same structure and have the same function. For example, the secondary structure of tRNA (transfer RNA) has a characteristic cloverleaf shape; however, the nucleotide composition of tRNA can vary to the degree that two tRNAs can have completely different sequences. Thus, methods that incorporate ncRNA secondary structure information can improve the accuracy of prediction.

Figure 6.1 shows an example of a stem structure, with the corresponding RNA sequence, while in Figure 6.2 it is shown an example of a typical secondary structure of tRNA, that is a particular class on non-coding RNA, showing all the aforementioned structures.

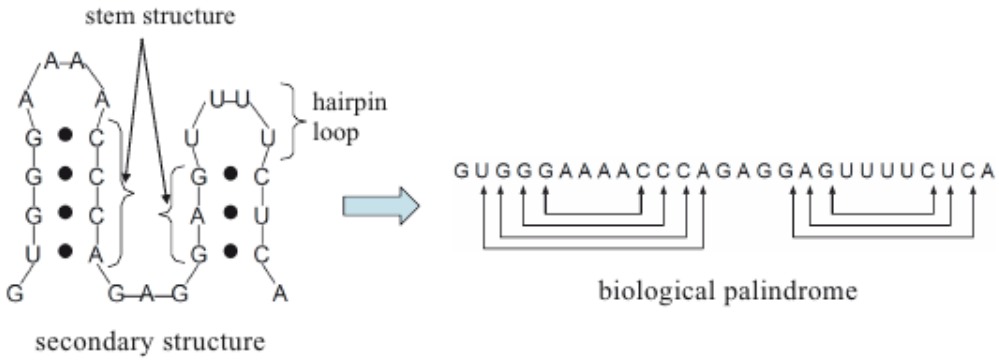


**Figure 6.1**: A typical secondary structure including stem structure of an RNA sequence (left) constitutes the so-called biological palindrome (right).

This representation of RNA sequences arises a problem, that is for a fixed RNA sequence we have no way to uniquely identify which will be its secondary structure. The reason is that experimental probing of the secondary structure is expensive and not always possible [150]. What we can do is to predict a possibly small number of different foldings (see Section 6.5.1), and estimate the probability of a folding to be the correct one (the folding that appears in nature) using some heuristics such as the minimal energy principle, see Section 6.4.1.

Moreover, while it is clear that secondary structure information is relevant for

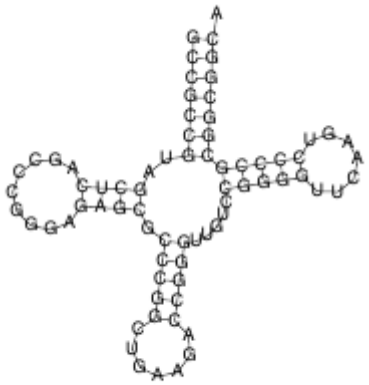


**Figure 6.2**: Typical secondary structure of tRNA.

the task of identifying and classifying ncRNA sequences, it is still not clear how to use this information. This will be the focus of this chapter, as we will explain in Section 6.5.

Graph topology indices derived from RNA structure have been used to predict the family of RNA sequences in the Rfam database [32]. Their approach consist in representing RNA sequences as a fixed-size vector of graph properties, defined over the graph representation of the sequence. In this way, it is possible to identify sequences that have a similar folding but that are highly dissimilar from the sequence point of view. However, using graph properties as "summaries" cause a significant information loss about the actual folding. The approach appears to be successful for the classification purpose only when considering highly dissimilar sequences. This method is thus not suited for the task of non-coding RNA function prediction. In Section 6.3.3 we will see another approach that at time of writing is considered at the state-of-the-art.

### Infernal

The other family of methods we consider exploit the secondary structure information and is based on covariance models. The most adopted software in this family is

Infernal (INFERence of RNA ALignment) [103]. It is a tool for searching databases of sequences for RNA structure and sequence similarities. It is an implementation of a special case of *profile stochastic context-free grammars* called covariance models (CMs). To build a model, INFERNAL starts from a pre-computed consensus structure (see Section 6.4.1), that is a secondary structure shared by different sequences in the same family. The assumption is that the secondary structure of most RNA molecules is strongly conserved in evolution. In Section 6.4 we will present how the secondary structure can be computed. A CM is like a sequence profile, but it computes a combination of sequence consensus and RNA secondary structure consensus, so in many cases, it is capable of identifying RNA homologs that conserve their secondary structure more than their primary sequence. CMs are closely related to profile Hidden Markov Models, but are more complex. CMs and profile HMMs both capture position-specific information about how conserved each column of the alignment is. However, in a profile HMM each position of the profile is treated independently, while in a CM base-paired positions are dependent on one another. The dependency between paired positions in a CM enables the profile to model covariation at these positions, which often occurs between base-paired columns of structural RNA alignments. In most cases, the particular RNA folding is determined by the Watson-Crick base pairs complementarity (G-C, A-U), rather than the specific nucleotide in a particular position. For this reason, for many of these base-pairs, it is not the specific nucleotides that make up the pair that is conserved by evolution, but rather that the pair maintains Watson-Crick base-pairing. The added signal from covariation can be significant when using CMs for homology searches in large databases.

### 6.3.3 Kernel methods

#### Stem kernel

Stem kernel [116] is a natural extension of the *all-subsequences string kernel* [121] for the application to RNA sequences. The feature space of the *all-subsequences*

```
# STOCKHOLM 1.0

tRNA1             GCGGAUUUAGCUCAGUUGGG.AGAGCGCCAGACUGAAGAUCUGGAGGUCC
tRNA2             UCCGAUAUAGUGUAAC.GGCUAUCACAUCACGCUUUCACCGUGGAGA.CC
tRNA3             UCCGUGAUAGUUUAAU.GGUCAGAAUGGGCGCUUGUCGCGUGCCAGA.UC
tRNA4             GCUCGUAUGGCGCAGU.GGU.AGCGCAGCAGAUUGCAAAUCUGUUGGUCC
tRNA5             GGGCACAUGGCGCAGUUGGU.AGCGCGCUUCCCUUGCAAGGAAGAGGUCA
#=GC SS_cons      <<<<<<<..<<<<.........>>>>.<<<<<.......>>>>>.....<

tRNA1             UGUGUUCGAUCCACAGAAUUCGCA
tRNA2             GGGGUUCGACUCCCCGUAUCGGAG
tRNA3             GGGGUUCAAUUCCCCGUCGCGGAG
tRNA4             UUAGUUCGAUCCUGAGUGCGAGCU
tRNA5             UCGGUUCGAUUCCGGUUGCGUCCA
#=GC SS_cons      <<<<.......>>>>>>>>>>>>.
//
```

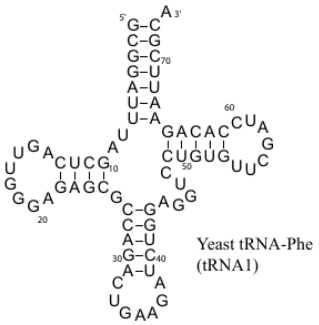


**Figure 6.3**: Example of a multiple RNA sequence alignment with secondary structure annotation.

*string kernel* is defined as all the possible subsequences of the input string, both the contiguous and non-contiguous ones. For example, two RNA sequences CUG and CAU have 4 common subsequences: $\epsilon$ (the empty string), C, U,and C-U, where - represents the bond between two nucleotides. The all-subsequences kernel calculates the inner product of the feature vectors by counting all common subsequences (considering gaps). The inner product of the two sequences CUG and CAU is 4.

The Stem kernel is a simple kernel for RNA secondary structures, that maps them in a feature space representing all the possible base pairs. Note that paired nucleotides have not to be neighbors in the sequence. This feature mapping is easily constructed when the secondary structure of the target is available, but in general this is not the case. The policy is to extend the same approach to count over all possible base-pair candidates, but this may be not feasible. For limiting the complexity of the method, the idea is to count the occurrences of possible stacking base-pairs, called stem structures. An example of stacking base pairs is in the sequence ACGU, where the pairs A-U and C-G are stacked. This situation is disambiguated introducing parenthesis in the notation, so in this case we represent this stem structure as (A(C-G)U). For example, stems of length 2 constitutes a 16-dimensional vector space: (A(A-U)U), (A(U-A)U), (A(C-G)U), (A(G- C)U), (U(A-U)A), (U(U-A)A), (U(C-G)A), (U(G-C)A), (C(A-U)G), (C(U-A)G), (C(C-G)G), (C(G-C)G), (G(A-U)C), (G(U-A)C), (G(C-G)C), (G(G-C)C).

The features of the Stem kernel are all possible non-contiguous stems of arbitrary length from the RNA sequence that is examined. The kernel calculates the inner product in the feature space implicitly, starting from RNA sequences, thus no additional information about the secondary structure is needed.

#### Marginalized kernel on RNA sequences

The work in [98] proposed the application of marginalized kernel to RNA sequences represented as a *labeled dual graph*, that is the representation in Figure 6.7B. In this representation, every node represents helical regions (sequences of paired nucleotides) and the edges represent the loops (sequences of non-paired nucleotides).

This representation uses only the information about the pairing of the nucleotides in the sequence, discarding the information about the type of structure (stems, hairpins, bulges,...). Such information is however coded in the resulting graph, but not in an explicit way.

Marginalized kernel, that is based on random walks and has been presented in Section 3.6.1, has been applied on these graphs. It showed promising performance. It made the way for the application of other graph kernels to RNA secondary structure graphs.

## 6.4 Computation of RNA secondary structure

Due to the importance of RNA secondary structure, several computational RNA folding tools have been developed. RNAalifold is one of the most adopted programs to predict the secondary structure starting from an RNA sequence [10]. It determines the folded secondary structure that minimizes the free energy by optimizing the intramolecular base pairing. We describe this technique in detail here because we adopted it for the conversion from RNA sequence to graph for our proposed method.

### 6.4.1 Minimum free energy structure

The secondary structure for ncRNA sequences is very important for understanding of the function of the molecule. However, as stated in Section 6.3.2, the experimental determination of the secondary structure is time consuming and expensive. For this reason, computational methods able to accurately predict it are of great interest. The minimization of the free energy (MFE) is a well-understood method for predicting the secondary structure which an RNA takes at equilibrium for a given temperature and pH. The free energy $\Delta G$ of a double strained structure relative to a single strand is defined as the sum of enthalpy, $\Delta H$, and the entropy, $\Delta S$, terms

$$\Delta G = \Delta H - T\Delta S$$

where T is the absolute temperature.

The structure with the lowest total free energy will be the most stable and probably the one present in nature.

An approximated approach for estimating the free energy has been proposed in [135], and it is adopted nowadays for the MFE computation.

The approach consists in the assignment of stability numbers to every pair and structure. These numbers results from experiments at 25 degrees C in a neutral buffer. The numbers are: (1) +1 for A-U pairs, (2) +2 for G-C pairs, (3) 0 for G U pairs, (4) -5 to -7 for hairpin loops, (5) -4 to -7 for interior loops, (6) -2 to -6 for bulges. For the exact weight assignment see Figure 6.4. The stability number for a given RNA secondary structure is the sum of the contribution of the pairs, loops, bulges and helices. The structure with the highest number is the most stable. Stable structures have thus positive values, measured in units of the free energy of forming an A-U base pair ($-1.2 kcal/mol$ at 25 degrees Celsius).

In order to calculate the minimum free energy structure for a given sequence, a dynamic programming approach has to be adopted. We can compute a base pair matrix with all possible pairs between A-U, G-C and G U nucleotides. A matrix for an example sequence is shown in Figure 6.5, in which 1, 2 and 0 represent the respective stability numbers. From the matrix, all possible secondary structures can

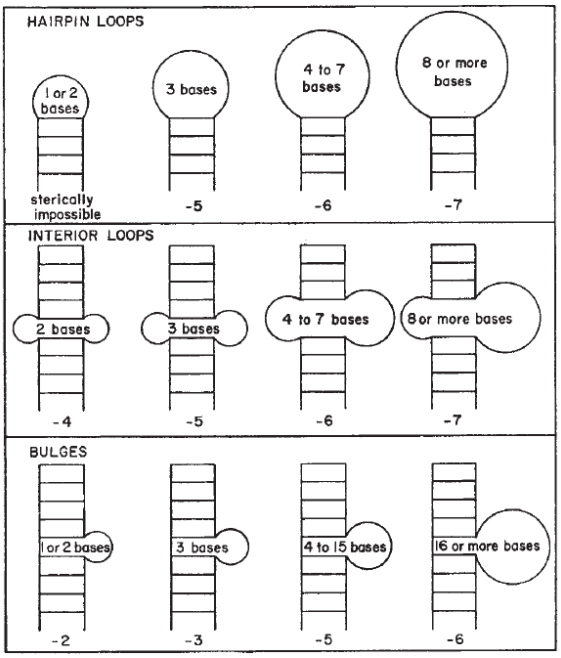


**Figure 6.4**: Contributions of loops and bulges with unbonded bases to the stability of the secondary structure. Figure from [135].

be investigated and their stability numbers computed. Figure 6.6 shows three of the most probable structures for the same sequence. In some cases, the method is unlikely to predict an unambiguous stable structure, but it is valuable in significantly limiting the number of possibilities.

### 6.4.2 RNA shape representation

The majority of the methods developed to predict ncRNA function are fully reliant on the ability to predict the secondary structure accurately. A large portion of recent research on RNA function prediction applies concepts developed in graph theory to the analysis of RNA structure (see references in [32]). There are many ways to represent RNA secondary structure with graphs (see Section 6.4.3), including the bracketed representation (where nucleotides are converted to nodes and bonds to edges), and the tree one (where base pairs are converted to 'stem' nodes and loop nucleotides are converted to 'loop' nodes). Each representation has different advan-

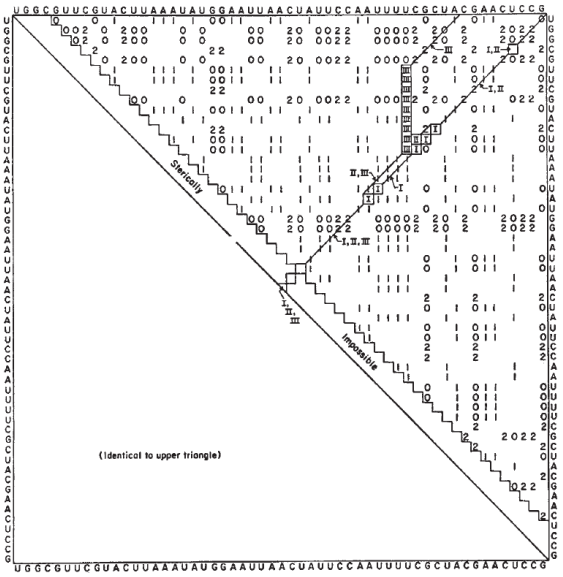


**Figure 6.5**: Example base pairing matrix for a sequence of 55 bases from R17 viral RNA. Figure from [135].

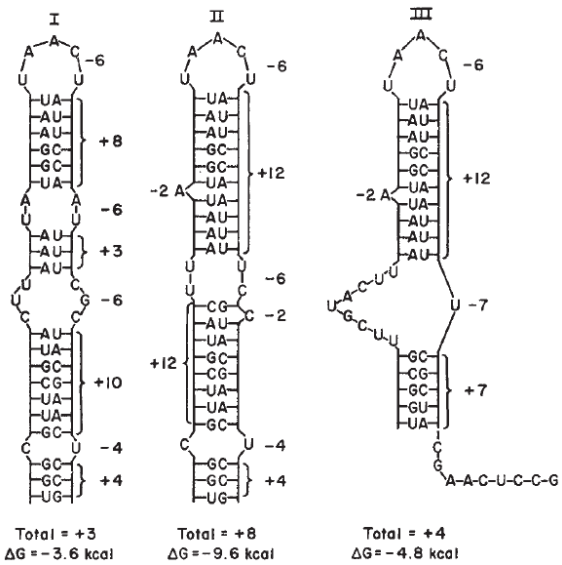


**Figure 6.6**: Three possible secondary structures for a sequence of 55 bases from R17 viral RNA. Figure from [135].

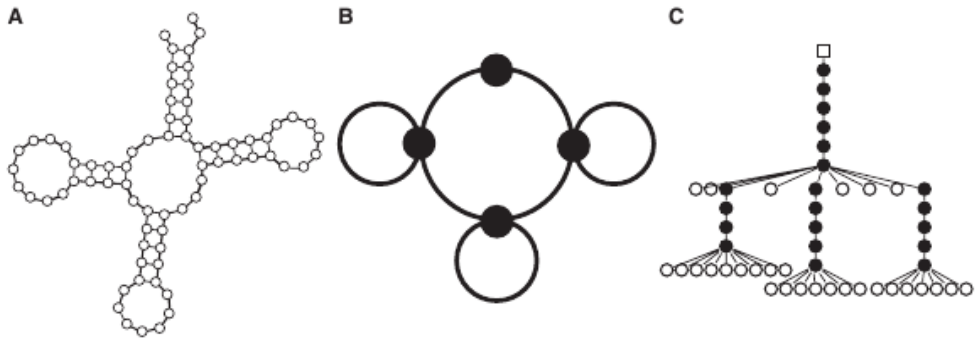


**Figure 6.7**: Example of different graph representations of RNA.

tages and disadvantages including information loss and complexity of calculation [57]. For an example see Figure 6.7, where "A" is the most straightforward representation where each nucleotide is a node, and each bond between nucleotides forms an edge. Figure 6.7 "B" gives an example of the bracketed representation, while Figure 6.7 "C" represents the secondary structure as a tree.

### 6.4.3 RNA abstract shapes

The minimum free energy principle for the prediction of RNA secondary structure has been presented in Section 6.4.1. However, this principle is not enough for the successful prediction of the secondary structure. Frequently, the minimum free energy configuration is not the one present in nature. Nonetheless, the configuration present in nature is usually one of the suboptimal configurations, i.e. one with a slightly higher free energy w.r.t. the MFE configuration. Thus, it is necessary to consider a large set of suboptimal solutions, often very similar to one another. On the other side, we do not want to analyze all the possible variations because the high number of structures to analyze would increase the computational complexity too much. Instead, we want to select only a small set of configurations, possibly with fundamental differences in their shape. This key idea is implemented in one of the state-of-the-art tools for RNA secondary structure prediction: RNAshapes [70]. Each shape of RNA belongs to a class of similar structures, each one with

a MFE representative. For a given energy range, the number of different shapes is considerably smaller than the number of different structures. For the definition of the abstract shapes, we need to partition the folding space into different classes of structures, thus abstracting from structural details. Given a particular folding (shape) there are different abstraction levels we can use for its representation. The shape type is the level of abstraction or dissimilarity that we adopt to decide whether two different foldings have the same shape or not. In general, helical regions are depicted by a pair of opening and closing square brackets and unpaired regions are represented as a single underscore. The differences in shape types are due to whether a structural element (bulge loop, internal loop, multi loop, hairpin loop, stacking region and external loop) contributes to the shape representation: five types are implemented. All shapes abstract from loop and stack lengths. In type 1, unpaired regions are represented by an underscore and stacking regions by a pair of squared brackets. The succeeding shape types gradually increase abstraction. In type 5, no unpaired regions are included and nested helices are combined. The differences among the five abstraction types are shown in the example in Figure 6.8. For the given sequence, one possible folding is reported. Then, according to the different shape types, the associated shape is reported.

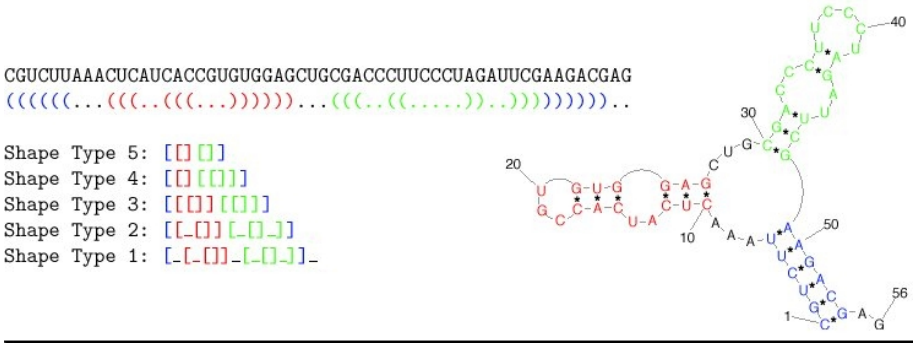


**Figure 6.8**: RNAshapes: image describing the differences between shape types.

In order to generate the set of foldings associated to a given sequence, we proceed analyzing one candidate secondary structure at a time. When analyzing different foldings of the same sequence, if the folding generates a new shape then it is added

to the set of foldings we consider, otherwise it is discarded.

## 6.5 Novel graph kernels for RNA sequences

Let us now focus on the application of kernel methods for graphs on RNA sequences. In literature there are some graph kernels specifically defined for RNA secondary structures.

Recent work on this topic proposed a kernel function that enhances the ability to measure the similarity of two RNA sequences from the viewpoint of secondary structure, that is the Stem kernel (see Section 6.3.3).

The application of graph kernels to RNA function prediction (that is, predicting the family which an RNA sequence belongs to) is made possible by the representation of the RNA secondary structure as a graph. As stated in Section 6.3, string kernels working directly on the RNA sequence do not constitute a successful approach since the majority of the information resides in the secondary structure. Only few kernels that consider the secondary structure have been proposed, but they show promising results. A major problem of these kernels is the computational complexity. For example, the Stem kernel have an asymptotic complexity of $O(n^4)$, that makes it unfeasible for datasets bigger than few hundred examples.

The goal of the work presented in the following is to extend kernel methods for structured data to RNA sequences.

The motivation leading to the application of kernels for graphs to the ncRNA family prediction is basically to solve the issues of existing tools (see Section 6.3), and to propose a method that can take into account the secondary structure of the sequences, and at the same time being feasible from a computational point of view.

The goal in this case is to develop fast kernels (near to linear time) because of the large amount of data we need to process. Moreover, examples in this case are RNA sequences. Since in the genome we don't know where a certain ncRNA sequence begins or ends, we have to deal with a very noisy signal, as explained in Section 6.5.1.

The problem presents various non-trivial issues that have to be addressed to achieve the goal of defining good kernels for this domain. In the following sections these main issues are presented. Preliminary experiments in Section 6.6 show promising results.

The contributions presented in this chapter can be summarized in:

- the definition of a way to represent RNA sequences as graphs, preserving as much information as possible;
- the definition of new graph kernels particularly suited for this domain.

### 6.5.1 Multiple instance learning

In machine learning, the multiple instance learning problem [50] is a variation on supervised learning (see Section 2.1). Instead of receiving a set of instances which are labeled positive or negative, the learner receives a set of bags that are labeled positive or negative. Each bag contains many instances. The most common assumption is that a bag is labeled negative if all the instances in it are negative. On the other hand, a bag is labeled positive if there is at least one instance in it which is positive. We will now explain why this problem arises when considering the problem of RNA function prediction.

As stated before, in the RNA domain the examples are simple nucleotide sequences. In these sequences, we don't really know where a particular ncRNA fragment starts or ends. This fact holds in nature, because there is no particular signal for the start or end of a region of interest. For this reason, this holds also for the dataset we considered in our experiments. This means that a significant part of the input sequence may be noise, i.e. nucleotide sequences not generating any RNA functional molecule. The problem arises from the fact that trying to fold the real sequence and a part of noise can give misleading results because the "right" folding can be completely different. A solution to this problem is to adopt a sliding windows policy. Beginning with position 1 of the input sequence, the analysis is repeatedly performed on subsequences of the specified size. At each calculation, the

resulting structures (the configurations with lowest free energy) are added to the set of structures for the given sequence and the window is moved, until the end of the input sequence is reached. Table 6.1 shows an example of the resulting subsequences starting from an RNA sequence of 100 nucleotides. The windows of size 20 and 50 are shown, with a 50% shift (i.e. two subsequent sequences overlap by the half of their length). In the end, we have a set of several foldings associated with the input sequence. Most of them will probably be noise, but the hope is that in some of them there is the signal we are looking for. Experimental results in Section 6.6 show an evidence for this claim. The dimension of the window is a very important parameter for the successful prediction of the actual shape of the RNA sequence. In particular, existing tools have different predictive performance depending on the size of the original sequence. Since we don't know the length of the real sequence, the only option we have is to try different window sizes, looking for more and more complex shapes. In particular, the bigger the window, the more complex the resulting shape will be. From biological knowledge, it has been shown that sequences of length up to 20 gives the simpler structures, consisting in a single shape. Sequences up to 50 nucleotides give more complex structures, giving some information about the connections between different shapes. Finally, for sequences bigger than that, a complex folding is generated. In general, the bigger the considered sequence, the more complex is the resulting folding, but the less probable that every particular folding is the one we find in nature (because the impact of noise in the overall folding is determinant for the final shape).

For the preliminary work presented in this chapter, we decided to face the multiple instance learning issue in the simplest way. We consider as a single example the set of all graphs generated from a particular sequence. This approach increases the example size of several times with respect to the length of the original sequence. Future extensions may consider the weighting of each structure according to the probability of that structure to be the real one.

| Original Sequence |
|---|
| GAAGAUCUGCCGCAACUGCAAGAUCAUCCGCCGCAAAGGUGUUGUGCGCG<br>UGAUCUGCACUGACCCGCGCCACAAGCAGCGCCAGGGUUGAUUUACAGGU |
| Window size 20 |
| 1 GAAGAUCUGCCGCAACUGCA 20<br>-0.50 .....(((.....))) []<br>11 CGCAACUGCAAGAUCAUCCG 30<br>-0.40 .((....)).......... []<br>21 AGAUCAUCCGCCGCAAAGGU 40<br>-2.50 .....(((.....))) []<br>31 CCGCAAAGGUGUUGUGCGCG 50<br>-2.90 .((((.....)))).. []<br>41 GUUGUGCGCGUGAUCUGCAC 60<br>-4.00 ...(((((.....))))) []<br>51 UGAUCUGCACUGACCCGCGC 70<br>-0.90 .....(((.....))). []<br>61 UGACCCGCGCCACAAGCAGC 80<br>-1.80 .....((((.....)).)) []<br>71 CACAAGCAGCGCCAGGGUUG 90<br>-3.60 .....((((.....)))) []<br>81 GCCAGGGUUGAUUUACAGGU 100<br>-1.40 (((...((.....)).))) [] |
| Window size 50 |
| 1 GAAGAUCUGCCGCAACUGCAAGAUCAUCCGCCGCAAAGGUGUUGUGCGCG 50<br>-13.70 ((.(((((((.....)).)))))).))(((((((((.....)))))).))) [][]<br>26 AUCCGCCGCAAAGGUGUUGUGCGCGUGAUCUGCACUGACCCGCGCCACAA 75<br>-17.90 ...((((.....))))(((((.(((((.((.....))..)))))))))) [][]<br>51 UGAUCUGCACUGACCCGCGCCACAAGCAGCGCCAGGGUUGAUUUACAGGU 100<br>-15.20 ..((((((....(((((((((........))))..)))))).....)))))) [] |

**Table 6.1**: Sliding window approach.

### 6.5.2 Representation issues

In Section 6.3.2 we showed how it is possible to predict the secondary structure of an RNA sequence, and that there exists a natural representation of this structure as a graph, where vertices represent the nucleotides and an edge between two vertices means that the corresponding nucleotides are paired in the folding. We also argued that we want to find similarities between structures, and not between the nucleotides these structures are made of. One way to face this problem is to consider multiple representations at the same time, just differing on the abstraction level of

representation.

Some possible secondary structure representations found in literature have been presented in Section 6.4.2, but no one seems to carry the right amount of information on the secondary structure. So the representation issue has to be addressed defining new representations for the RNA secondary structure.

We decided to adopt a combined representation with possibly several abstraction layers. Using one of the abstractions presented in Section 6.4.2, a layer may have a node corresponding to each structure (stems, loops, hairpins, bulges, ...). Each one of these nodes is connected to a *relationship node*, that is a node representing the relationships between two different layers. On the other hand, the same relationship node is connected to the corresponding vertices in a different abstraction layer, e.g. the nodes representing the nucleotides involved in the particular shape. An example of this representation can be found in Figure 6.9. In this figure, the yellow nodes are the relationship ones. Each one of these nodes is connected to the corresponding nodes in the two abstraction layers. For example, the leftmost one is connected to a stem node in one abstraction, and to the nucleotides forming the stem in the other abstraction.

It is worth to notice that this representation allows for multiple abstraction layers. Theoretically, each one of the possible secondary structure representations presented in Section 6.4.3 can be one of several layers interconnected by relationship nodes. At the time of writing we did not explore this path, the main reason being that adding more abstraction layers would lead to an increase in the computational demand of the kernel calculation. So from now on we will refer to the representation carrying only two abstraction layers as the one in Figure 6.9. Many graph kernels can be defined on this representation, keeping into account the presence of the relationship nodes that have not to be accounted for the overall kernel calculation (see Section 6.5.3 for our proposed kernel).

To evaluate a representation, we measured the classification performance of the SGD learning algorithm, that approximates the SVM solution (see Section 2.7.3) on different representations of RNA sequences. Our aim is to find a representation

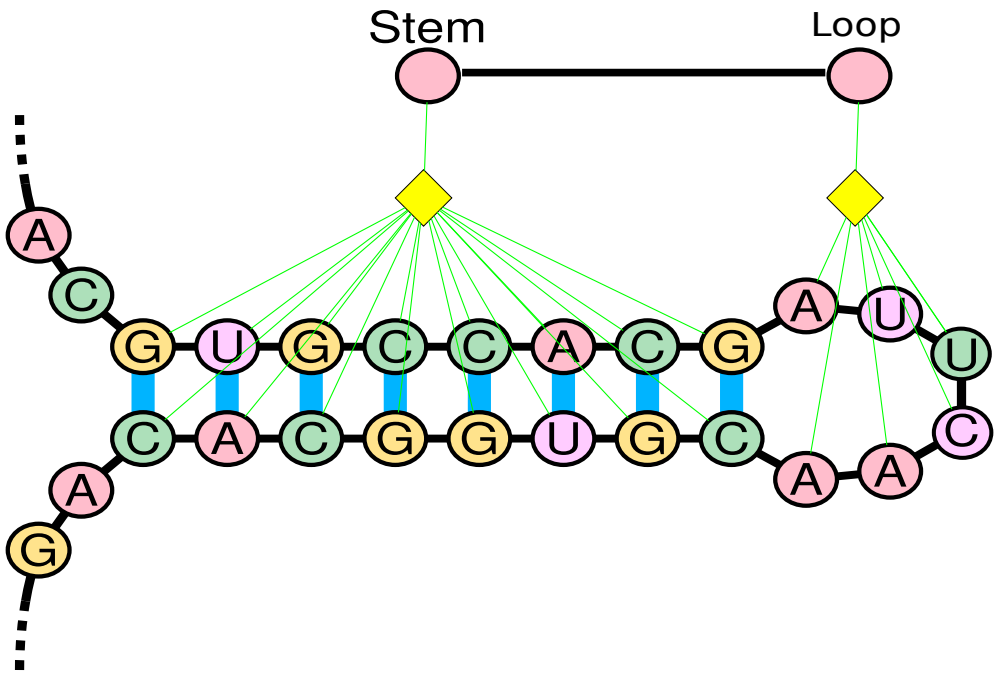


**Figure 6.9**: Example of different abstraction layers for the representations of RNA.

that performs well for the majority of ncRNA families, that is a representation that nearly contains all the useful information for the task of classifying non-coding RNA sequences.

### 6.5.3 Kernels exploiting different abstraction levels

In this section we will present two kernels designed for the graphs representing RNA sequences with different abstraction layers defined in Section 6.5.2.

#### Abstract NSPDK

The NSPDK kernel presented in Section 3.6.7 is easily extensible to the representation of RNA with two abstraction layers presented in Section 6.5.2. We only have

to decide the behavior of the relationship nodes. The idea is not to consider them as normal nodes, because of the following reasons:

- we don't want relationship nodes and the edges connecting them with other nodes to influence the kernel value; i.e. since relationship nodes do not have a label, we would have always node-match, that would slightly alter the kernel value;
- we want to keep a simple interpretation of the features. For this reason, we do not want mixed features, i.e. a subgraph spanning two different abstraction layers.

For these reasons, the proposed adaption consists in not considering the relationship nodes as actual nodes. That is, when computing a sub-structure of radius $r$ rooted in a certain node $v$, even if the relationship node is at a distance smaller than $r$ from $v$, it will not appear in the sub-structure. In other words, relationship nodes and the relative edges are "invisible" when computing the sub-structures. This first modification leads to the property that each sub-structure belongs to exactly one abstraction layer. Moreover, we do have to consider relationship nodes when computing the pairwise features. When merging two sub-structures at distance $d$, it's possible to consider structures at different abstraction levels, respecting the constraint that the distance between the two roots has to be $d$, considering in the path the relationship node.

### Abstract NSDDK

In this section, we present the Neighborhood Subgraph Decomposition DAG Kernel (*NSDDK*), particularly suited for graphs representing RNA structures defined in Section 6.5.2. The idea behind this kernel is to exploit the expressivity of pairwise features. That is, a single feature is made by two structures in the original graphs at predefined distance, limited by the kernel parameter $d$. In this way, compared to having only the single decomposition structures as features, we add some information

about the context in which a feature appears in the graph. This pairwise trick promotes sparsity, with benefits from both the computational and the predictive point of view for the task we are considering. This is the same concept behind the NSPDK kernel (see Section 3.6.7). The problem of NSPDK on this task is that it requires isomorphism between the two neighborhoods it considers. As we stated before, it's unlikely, for two ncRNA that share the same structure, to share the nucleotide sequence as well. This would result in a very sparse kernel, ending up to consider only small fragments of nucleotides. $ODDK_{ST_h}$ kernel, presented in Section 4.1, on the other hand has a feature associated to each subtree-fragment, allowing for a "softer" match. The idea is to apply the pairwise trick to the $ODDK_{ST_h}$ kernel, that is fast and shows good predictive performances. The problem is that the number of features $ODDK_{ST_h}$ produces is higher with respect to the ones NSPDK generates. The straightforward pairing of the features, resulting in a quadratic number of features, would lead to a quadratic kernel, which does not respect the computational complexity constraint stated in the problem statement (see Section 6.2).

We decided to combine the $ODDK_{ST_h}$ features generated from a node in the graph with the neighborhoods features generated starting from another node. In this case, the resulting kernel would have a linear number of features w.r.t. the $ODDK_{ST_h}$ kernel, hopefully maintaining the performance improvements of the pairwise trick. Similarly to the definition of NSPDK kernel in Section 3.6.7, for defining the NSDDK kernel it is convenient to define a relation $R_{r,d}(A^x, D^y, X)$ between two rooted graphs $A^x$, $D^y$ and a graph $X$ to be true iff $A^x$ is in $\{N_r^v | v \in V(X)\}$ (where the set inclusion is up to isomorphism), $D^y$ is in $\{\phi_{DDK}^{r,v} | v \in V(X)\}$ and the distance between $u$ and $v \in V(X)$ is exactly $d$. We recall that the function $\phi_{DDK}^{r,v}$ is the $\phi$ function of the $ODDK_{ST_h}$ kernel defined in Section 4.1. The relation selects all pairs made of one neighborhood subgraph of radius $r$ and a tree-feature of $ODDK_{ST_h}$ kernel at a distance $d$ in a given graph $X$. We can define an auxiliary kernel for graphs

as the convolution kernel that uses this relation:

$$k_{r,d}(X,Y) = \sum_{\substack{A^v, D^u \in R_{r,d}^{-1}(X) \\ A'^v, D'^u \in R_{r,d}^{-1}(Y)}} \delta(A^v, A'^v)\delta(D^u, D'^u),$$

where $\delta$ is the Dirac kernel. In words, $k_{r,d}$ counts the number of identical pairs made of a neighborhood subgraphs of radius $r$ and a DAG-feature at distance $d$ in two graphs.

We can formally define the *NSDDK* kernel as:

$$K(X,Y) = \sum_r \sum_d k_{r,d}(X,Y).$$

This kernel can be directly applied to the abstract representation of RNA sequences presented in Section 6.5.2. It's worth to notice that the decompositions $A$ and $D$ may belong to different abstraction layers. In this case, the path of length $d$ that connects the roots of the two structures have to include a *relationship node*. These mixed features are possible only using the described representation. We think these features that consider information from different points of view to be the most relevant ones.

## 6.6 Experiments

In this section, we present and analyze the preliminary results of the application of graph kernel methods to the task of ncRNA family prediction. We used as baseline a classifier based on BLAST and described in Section 6.3.1.

### 6.6.1 Datasets

The dataset adopted for experiments on ncRNA was extracted from Rfam [65]. Rfam is a database that aims to catalog non coding RNAs through the use of sequence alignments and covariance models. RNA sequences are grouped according to the family, that is a set of sequences that share the same function and a clear common

ancestor. First, the family assignment is done using the Infernal tool on a well-curated set of seed alignments. Then a large database of nucleotide sequences is searched for possible homologues. The search returns a ranked list. At this point, a human expert fixes a threshold to discriminate between real homologues and false hits.

At the time of writing, the Rfam database is one of the most accurate sources of annotated non-coding RNA sequences. The dataset was created starting from 47 Rfam families. We calculated the identity score (sequence similarity) between each pair of sequences in a family. Then, each family was splitted in two parts, one for training and one for validation. The split was done assuring that no sequence in a split would have sequence similarity grater than 50% with any sequence in the other split. To better simulate the real world scenario in which the start and end points of the sequences are not known, we added random padding (respecting the frequency of each nucleotide in the particular sequence) at the start and after the end of the sequence. The total length of the original Rfam sequence and the padding sums up to 200 nucleotides. The RNA families we selected and the number of examples foe each class are shown in Table 6.2. The problem is inherently a multiclass problem. We decided to split it into 47 single-class problems using the one-versus-all approach. Since the resulting datasets are unbalanced, we adopted as performance measure the area under the precision/recall curve (APR).

Moreover, we generated a reduced version of the dataset with only seven classes in order to be able to estimate the parameters in a fast way. This reduced dataset comprehends only the following classes: 14, 21, 24, 25, 26, 31, 42.

### 6.6.2 Experimental results

In the experimental section of this chapter, we consider as baselines the aforementioned BLAST tool, that is considered the state-of-the-art [97] and has been presented in Section 6.3. Other methods based on Covariance Models like INFERNAL that has been presented in the same section, and the kernel-based ones presented in Section 6.3.3 have not been applied since their computational complexity makes

| Class ID | ncRNA Rfam calsses | TRAINING | VALIDATION |
|---|---|---|---|
| 1 | RF00001:5S_rRNA | 593 | 290 |
| 2 | RF00005:tRNA | 360 | 170 |
| 3 | RF00015:U4 | 568 | 260 |
| 4 | RF00016:SNORD14 | 164 | 41 |
| 5 | RF00019:Y_RNA | 614 | 319 |
| 6 | RF00020:U5 | 134 | 60 |
| 7 | RF00026:U6 | 170 | 69 |
| 8 | RF00029:Intron_gpII | 286 | 136 |
| 9 | RF00031:SECIS_1 | 124 | 51 |
| 10 | RF00050:FMN | 75 | 34 |
| 11 | RF00059:TPP | 298 | 160 |
| 12 | RF00066:U7 | 121 | 59 |
| 13 | RF00097:snoR71 | 171 | 81 |
| 14 | RF00140:Alpha_RBS | 62 | 34 |
| 15 | RF00156:SNORA70 | 145 | 80 |
| 16 | RF00162:SAM | 56 | 23 |
| 17 | RF00163:Hammerhead_1 | 148 | 61 |
| 18 | RF00169:Bacteria_small_SRP | 98 | 47 |
| 19 | RF00263:SNORA68 | 56 | 28 |
| 20 | RF00322:SNORA31 | 100 | 48 |
| 21 | RF00406:SNORA42 | 88 | 42 |
| 22 | RF00409:SNORA7 | 605 | 274 |
| 23 | RF00420:SNORA61 | 217 | 114 |
| 24 | RF00504:Glycine | 373 | 89 |
| 25 | RF00557:L10_leader | 73 | 33 |
| 26 | RF00560:SNORA17 | 149 | 72 |
| 27 | RF00619:U6atac | 93 | 48 |
| 28 | RF00645:MIR169_2 | 57 | 24 |
| 29 | RF00655:mir-28 | 117 | 59 |
| 30 | RF00779:MIR474 | 124 | 65 |
| 31 | RF00875:mir-692 | 121 | 47 |
| 32 | RF00876:mir-684 | 65 | 27 |
| 33 | RF00906:MIR1122 | 1010 | 148 |
| 34 | RF00989:mir-492 | 142 | 56 |
| 35 | RF01016:mir-584 | 263 | 127 |
| 36 | RF01028:mir-633 | 70 | 33 |
| 37 | RF01055:MOCO_RNA_motif | 53 | 19 |
| 38 | RF01059:mir-598 | 480 | 237 |
| 39 | RF01063:mir-324 | 100 | 49 |
| 40 | RF01699:Clostridiales-1 | 114 | 59 |
| 41 | RF01705:Flavo-1 | 76 | 38 |
| 42 | RF01725:SAM-I-IV-variant | 76 | 34 |
| 43 | RF01731:TwoAYGGAY | 64 | 24 |
| 44 | RF01734:crcB | 58 | 29 |
| 45 | RF01739:glnA | 67 | 33 |
| 46 | RF01942:mir-1937 | 249 | 122 |
| 47 | RF02012:group-II-D1D4-7 | 63 | 27 |
| Total | | 9247 | 3953 |

**Table 6.2**: Table showing the number of examples for each ncRNA class in the training and validation sets.

their application to the considered datasets unfeasible. With this kind of methods a subsampling phase is necessary. However, the comparison of the kernel approach with other methods is left as future work.

For the assessment of the classification performance, we considered the AUC measure, that is the area under the Precision/Recall curve.

We fixed the parameters for RNAshapes using a line-search approach on the reduced version of the dataset. We validated the shape type parameter $t$ in $\{1, 2, 3, 4, 5\}$ and the parameter w of the window side in the set $\{20, 50, 75, 100\}$. For the classifier based on BLAST, we performed a majority voting among the top-matching sequences. The number of foldings parameter $M$ has been validated in the set $\{1, 3, 5, 10, 20, 30, 50\}$

Finally, for each dataset the kernel parameters have been optimized using a grid-search approach. For NSPDK, the parameters have selected from the following sets: $r = \{1, 2, 3\}, d = \{1, 2, 3\}$.

After a set of experiments on the reduced dataset, we fixed the set of parameters to adopt. Table 6.3 shows the performance on the validation set for the optimal parameters configuration for each class. The kernel approach outperforms BLAST in almost all the considered classes. Note that these results are not directly comparable with the results on the whole 47-class problem. Since there is not the need for the uniqueness for some parameters, i.e. it is possible to use multiple window sizes at the same time, we found a good set of parameters to be $w = \{20, 30, 75\}, M = 5, t = 4$.

Let us now focus on the full 47-class dataset. As we can clearly see from the results on the validation set, presented in Table 6.4, using BLAST for classification works well for some RNA families. Indeed, for some ncRNA families the sequence carries enough information about the function. In these cases, it's useless to apply techniques that consider the secondary structure because the additional computational complexity may not be worth the benefits from the classification point of view. In some cases, considering the structure may introduce additional noise to the data, resulting in worst classification performance. This happens to the classes 12, 14, 16, 19, 28 in the dataset. For other classes, the performance of the two methods

| Class ID | Validation set | |
|---|---|---|
| | BLAST | NSPDK |
| 14 | 1.00000 | 0.94663 |
| | (b=5) | (w=100 m=5 t=4) |
| 21 | 0.88690 | 0.92652 |
| | (b=10) | (w=20 m=5 t=4) |
| 24 | 0.87900 | 0.99951 |
| | (b=1) | (w=20 m=5 t=4) |
| 25 | 0.89794 | 0.91945 |
| | (b=50) | (w=20 m=5 t=4) |
| 26 | 0.96059 | 0.96156 |
| | (b=10) | (w=20 m=5 t=4) |
| 31 | 0.95591 | 0.98842 |
| | (b=20) | (w=50 m=5 t=4) |
| 42 | 0.52928 | 0.89303 |
| | (b=10) | (w=75 m=5 t=4) |

**Table 6.3**: Experimental results (Area under Precision/Recall curve) of NSPDK kernel and the BLAST baseline in the reduced ncRNA dataset. The optimal parameters are reported between brackets.

are comparable (classes 6, 26, 27 and 41), but the kernel approach is considerably slower. However, for the majority of the classes the kernel approach achieves significantly higher classification performance. In 38 out of 47 classes, kernel methods outperform BLAST. On average, the AUC value for BLAST is 0.66, while for the NSPDK kernel is 0.80. From the execution times points of view, the two methods run times have a huge difference. BLAST requires only few seconds for building the database and for looking for the top-matching sequences in the database from the query. On the other hand, Abstract NSPDK requires from 8 minutes to 2 hours (depending on the parameters) for the computation of the feature vectors of the training examples in the reduced dataset. The point here is that the proposed ap-

proach is the only one that considers the RNA secondary structure and is feasible for this dataset, i.e. other methods that consider the secondary structure applied to the reduced dataset would have required several hours or days for the computation.

Notice that kernels that do not account for the different abstraction levels, such as the $ODDK_{ST_h}$ kernel that considers single decompositions, perform poorly on this dataset. For what concerns the NSDDK kernel presented in Section 6.5.3, we were not able to complete the experiments, thus the analysis of this kernel is left as a future work. The preliminary experimental results presented in this section confirm the fact that the information considered in the RNA graph representation presented in Section 6.5.2 is useful for the task of ncRNA family prediction, and that the graph kernel approach is promising for the task of RNA function prediction.

| Class ID | Validation set | |
|---|---|---|
| | BLAST | NSPDK |
| 1 | 0.76065 | **0.82869** |
| 2 | 0.38904 | **0.63258** |
| 3 | 0.90148 | **0.99369** |
| 4 | 0.70689 | **0.80432** |
| 5 | 0.63467 | **0.78767** |
| 6 | **0.83047** | 0.82118 |
| 7 | 0.77439 | **0.7934** |
| 8 | 0.39317 | **0.87899** |
| 9 | 0.02315 | **0.06984** |
| 10 | 0.90392 | **0.93252** |
| 11 | 0.9005 | **0.95741** |
| 12 | **0.83744** | 0.76889 |
| 13 | 0.04367 | **0.20995** |
| 14 | **0.94948** | 0.79752 |
| 15 | 0.84183 | **0.93446** |
| 16 | **0.75302** | 0.69802 |
| 17 | 0.20389 | **0.40515** |
| 18 | 0.43001 | **0.66874** |
| 19 | **0.69319** | 0.60933 |
| 20 | 0.82743 | **0.88246** |
| 21 | 0.76755 | **0.83394** |
| 22 | 0.8562 | **0.98875** |
| 23 | 0.85023 | **0.93274** |
| 24 | 0.76338 | **0.95497** |

| Class ID | Validation set | |
|---|---|---|
| | BLAST | NSPDK |
| 25 | 0.66202 | **0.96678** |
| 26 | **0.85414** | 0.84969 |
| 27 | **0.68772** | 0.63412 |
| 28 | **0.93186** | 0.86177 |
| 29 | 0.77299 | **0.96774** |
| 30 | 0.82191 | **0.99275** |
| 31 | 0.84984 | **0.96258** |
| 32 | 0.87513 | **0.92935** |
| 33 | 0.65172 | **0.94471** |
| 34 | 0.78503 | **0.92344** |
| 35 | 0.86821 | **0.99589** |
| 36 | 0.85933 | **0.91913** |
| 37 | 0.16878 | **0.70015** |
| 38 | 0.66543 | **0.99927** |
| 39 | 0.79103 | **0.88071** |
| 40 | 0.71489 | **0.94816** |
| 41 | **0.68267** | 0.66265 |
| 42 | 0.06366 | **0.63009** |
| 43 | 0.36706 | **0.8188** |
| 44 | 0.09053 | **0.24882** |
| 45 | 0.59577 | **0.86439** |
| 46 | 0.80531 | **0.90373** |
| 47 | 0.28407 | **0.78127** |
| AVG | 0.66 | **0.80** |

**Table 6.4**: Experimental results (Area under Precision/Recall curve) of different kernels and the BLAST baseline in the ncRNA dataset with window size of 20, 30 and 75, shift 20% and $M = 5$ different shapes per window.

# Chapter 7

# Conclusions and future work

> The key to making programs
> fast is to make them do
> practically nothing. ;-)
>
> Mike Haertel (GNU grep)

The aim of this thesis was to propose feasible solutions to some of the current drawbacks of kernel methods applied to graph-structured data. The problems we considered are challenging and interesting because of the large number of application domains that can benefit from solutions to them.
We can summarize the main contributions of this thesis as follows.

One of the main drawbacks of the existing kernels for graphs is the tradeoff between the computational complexity and the predictive performance of the adopted kernel. The recent trend in graph kernels is to define fast kernel functions (near-linear time complexity), but in general they are not flexible i.e. a fast kernel has limits in the expressiveness. This is a problem when the used kernel function does not reach the expected predictive performance on the considered task. The idea underpinning our approach is to define a family of graph kernels which allows several alternate instantiations in order to give to the user the possibility to select one kernel or another depending on the addressed task. In Chapter 4 we proposed a framework for graph kernels based on the decomposition of a graph into a multiset of unordered DAGs. By extending the definition of convolution tree kernels to DAGs

and by defining an ordering of the nodes of the DAGs, we favored the application of a vast class of tree kernels to graph data. Our analysis focused on two kernels belonging to the framework, one of which based on a novel tree kernel, for which proposed specific optimizations. The classification performances of the proposed kernels on six benchmark datasets show that they are able to reach competitive results on practically all the considered datasets. Moreover, while their worst-case time complexity is more than linear (while other considered kernels have a linear complexity in the number of nodes or edges in the graphs) in practice they are competitive, if not faster than competing approaches. Future developments of this line of research will include the definition of different graph kernels from the framework, possibly defining DAG kernels that consider different kinds of sub-structures than the ones considered here.

We proposed in the same chapter a feasible approach for model compression based on the application of feature selection techniques in the explicit feature space associated to a kernel. Future research on this research line are the following. Having the explicit set of features associated to the support graphs, and having the relative importance measure for each feature (given by the feature selection phase and the learning phase), it is possible to map each feature back to the input examples, obtaining information on the most significant part of each example. For example, we can map a feature space made of subtrees back to the input examples assigning a weight to each vertex in the original graphs that is the sum over the weights of each tree in feature space where that node does occur. At the end, we obtain a value for each node in the dataset that is a measure indicating how much that node is important for the task, and thus finding out what are the "key" zones of each example. This is important for example in bioinformatics applications, i.e. for the RNA analysis. Another future extension is the application of more sophisticated feature selection techniques, considering the dependencies between structural features.

Another important problem that is recently attracting interest consists in the application of machine learning algorithms on streams of graphs. In Chapter 5

we analyzed the tradeoff between efficiency and efficacy of various online margin kernel perceptron algorithms with fixed budget for graph streams. We defined three algorithms as modified versions of online algorithms present in literature. One of them efficiently exploits the explicit representation of the feature space (via hash tables) of one of the kernels for graphs defined in this thesis. It is faster and more accurate than the other proposed algorithms. Experiments on real data show the effectiveness of our approach. This is a pioneering work in the area of kernel methods for streams of graphs which shows that working in the primal space is a viable way for applying kernel methods to graph streams.

In the same chapter, we presented a fast technique for budget maintenance in the explicit feature space. The idea is to estimate the weighted frequency of a stream of features based on an extended version of the *Lossy Counting* algorithm. It uses it for: *i)* pruning the set of features of the current solution such that it is ensured that it never exceeds a predefined budget; *ii)* prediction, when a feature not present in the current model is first encountered. The results on streams of graph data show that the proposed technique for managing the budget is much faster than competing approaches. Its classification performance, provided the budget exceeds a practically very low value, is superior to the competing approaches, even when no budget constraints are considered for them. As future work, it would be interesting to apply the same budget maintenance policy to other learning algorithms. Moreover, the technique may be applied to every scenario where a kernel with a sparse representation in the feature space is defined, i.e. learning on trees or strings. We plan to apply the technique to the Multiple Hyperplane Machine [6, 147], that shows very good predictive performances while being at the same time a very fast algorithm.

Finally, in Chapter 6 we presented an application of graph kernels to an important real world problem in bioinformatics, namely RNA function prediction, where several issues arise for the extraction of relevant information from the data. We showed an effective way to solve the different problems and, applying graph kernels, we achieved state-of-the-art results. The work presented in this chapter should be considered preliminary. Indeed, an extensive analysis of the novel graph kernel de-

fined in this thesis for this problem should be performed. Moreover, there is the need to analyze the results, understanding which kind of feature is the most important and eventually focusing on the generation of a reduced number of features (with computational benefits) without losing predictive performance. On the other hand, we proposed a representation for RNA sequences as sets of graphs. It would be interesting to analyze other representations, focusing on the reduction of the practical complexity of the method and/or on the improvement of the predictive performance.

# References


[1] C. C. Aggarwal. Managing and Mining Graph Data, volume 40 of Advances in Database Systems. Springer US, Boston, MA, 2010.

[2] C. C. Aggarwal. On Classification of Graph Streams. In SDM, 2011.

[3] R. Agrawal. Fast Algorithms for Mining Association Rules. In Proc 20th Int Conf Very Large Data Bases VLDB, pages 487–499, 1994.

[4] F. Aiolli, G. Da San Martino, and A. Sperduti. Route kernels for trees. In Proceedings of the 26th Annual International Conference on Machine Learning, pages 17–24, Montreal, Quebec, Canada, 2009. ACM.

[5] F. Aiolli, G. Da San Martino, A. Sperduti, and A. Moschitti. Fast On-line Kernel Learning for Trees. In Proceedings of the 2006 IEEE Conference on Data Mining, pages 787–791, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[6] F. Aiolli and A. Sperduti. Multiclass Classification with Multi-Prototype Support Vector Machines. Journal of Machine Learning Research, 6:817–850, 2005.

[7] C. Alippi, S. Ntalampiras, and M. Roveri. A Cognitive Fault Diagnosis System for Distributed Sensor Networks. IEEE Transactions on Neural Networks and Learning Systems, 24(8):1–14, 2013.

[8] T. Asai, H. Arimura, K. Abe, S. Kawasoe, and S. Arikawa. Online Algorithms for Mining Semi-structured Data Stream. Data Mining, IEEE Internl. Conf. on, page 27, 2002.

[9] F. Bach. Image Classification with Segmentation Graph Kernels e. In In Proc. CVPR, 2007.

[10] S. H. Bernhart, I. L. Hofacker, S. Will, A. R. Gruber, and P. F. Stadler. RNAalifold: improved consensus structure prediction for RNA alignments. BMC bioinformatics, 9:474, Jan. 2008.

[11] A. Bifet and R. Gavaldà. Mining adaptively frequent closed unlabeled rooted trees in data streams. In Proceeding of the 14th ACM SIGKDD Internl. Conf. on Knowl. Disc. and data mining, KDD '08, pages 34–42, New York, NY, USA, 2008. ACM.

[12] A. Bifet, G. Holmes, B. Pfahringer, and R. Gavaldà. Mining frequent closed graphs on evolving data streams. Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '11, page 591, 2011.

[13] A. Bifet, G. Holmes, B. Pfahringer, R. Kirby, and R. Gavaldá. New Ensemble methods for evolving data streams. In Proc. of the 15th Internl. Conf. on Knowl. Disc. and Data Mining, pages 139–148, 2009.

[14] H. Block. The Perceptron: A Model for Brain Functioning. I. Reviews of Modern Physics, 34(1):123–135, Jan. 1962.

[15] S. Bloehdorn and A. Rettinger. Graph Kernels for RDF Data. In The Semantic Web: Research and Applications, pages 134–148. 2012.

[16] A. Bordes. Fast Kernel Classifiers with Online and Active Learning. Journal of Machine Learning Research, 6:1579–1619, 2005.

[17] K. M. Borgwardt and H.-P. Kriegel. Shortest-Path Kernels on Graphs. Data Mining, IEEE International Conference on, 0:74–81, 2005.

[18] L. Bottou. Online learning and stochastic approximations. On-line learning in neural networks, 1998.

[19] L. Bottou and O. Bousquet. The Tradeoffs of Large-Scale Learning. In S. Platt, J.C. and Koller, D. and Singer, Y. and Roweis, editor, Advances in Neural Information Processing Systems, pages 161—-168. NIPS Foundation (http://books.nips.cc), 2008.

[20] R. R. Bouckaert. Estimating replicability of classifier learning experiments. In Twenty-first international conference on Machine learning - ICML '04, page 15, New York, New York, USA, 2004. ACM Press.

[21] A. Z. Broder. On the resemblance and containment of documents. Systems Research, pages 1–9.

[22] L. Brun and D. Villemin. Two New Graph Kernels and Applications to Chemoinformatics. pages 112–121, 2011.

[23] N. Carolina, S. Umverslty, and N. Carohna. The Tree-to-Tree Correction Problem. Computing, (3):422–433, 1979.

[24] R. J. Carter, I. Dubchak, and S. R. Holbrook. A computational approach to identify genes for functional RNAs in genomic sequences. Nucleic acids research, 29(19):3928–38, Oct. 2001.

[25] G. Cavallanti, N. Cesa-Bianchi, and C. Gentile. Tracking the best hyperplane with a simple budget Perceptron. Machine Learning, 69(2-3):143–167, Feb. 2007.

[26] G. Cawley and N. Talbot. Preventing over-fitting during model selection via Bayesian regularisation of the hyper-parameters. The Journal of Machine Learning Research, 8:841–861, 2007.

[27] N. Cesa-Bianchi, a. Conconi, and C. Gentile. On the Generalization Ability of On-Line Learning Algorithms. IEEE Transactions on Information Theory, 50(9):2050–2057, Sept. 2004.

[28] Y. Chang and C. Hsieh. Training and testing low-degree polynomial data mappings via linear SVM. The Journal of Machine Learning Research, (11):1471–1490, 2010.

[29] O. Chapelle, B. Scholkopf, and E. Zien A. Semi-Supervised Learning (Chapelle, O. et al., Eds. IEEE Transactions on Neural Networks, 20:542, 2009.

[30] Y.-W. Chen and C.-J. Lin. Combining SVMs with Various Feature Selection Strategies. In I. Guyon, M. Nikravesh, S. Gunn, and L. Zadeh, editors, Feature Extraction, volume 207 of Studies in Fuzziness and Soft Computing, pages 315–324. Springer Berlin / Heidelberg, 2006.

[31] Y.-w. Chen and C.-j. Lin. Combining SVMs with Various Feature Selection Strategies. In Feature Extraction, number 1, pages 1–10. 2007.

[32] L. Childs, Z. Nikoloski, P. May, and D. Walther. Identification and classification of ncRNA molecules using graph properties. Nucleic acids research, 37(9):e66, May 2009.

[33] F. Chu and C. Zaniolo. Fast and light boosting for adaptive mining of data streams. In Proc. of the 8th Pacific-Asia Conf. Advances in Knowl. Disc. and Data Mining (PAKDD'04), pages 282–292, Sydney, Australia, 2004.

[34] M. Collins and N. Duffy. Convolution Kernels for Natural Language. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, NIPS, pages 625–632. MIT Press, 2001.

[35] M. Collins and N. Duffy. Convolution Kernels for Natural Language. ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 14:625–632, 2001.

[36] M. Collins and N. Duffy. New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron. In Proceedings

of the 40th Annual Meeting on Association for Computational Linguistics, pages 263–270, Philadelphia, Pennsylvania, 2002. Association for Computational Linguistics.

[37] F. Costa and B. Bringmann. Towards Combining Structured Pattern Mining and Graph Kernels. In ICDM Workshops, pages 192–201, 2008.

[38] F. Costa and K. De Grave. Fast neighborhood subgraph pairwise distance kernel. In Proceedings of the 26th International Conference on Machine Learning, number v, 2010.

[39] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online Passive-Aggressive Algorithms. Journal of Machine Learning Research, 7:551–585, 2006.

[40] K. Crammer, J. S. Kandola, and Y. Singer. Online Classification on a Budget. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, NIPS. MIT Press, 2003.

[41] N. Cristianini and J. Shawe-Taylor. An Introduction to Support Vector Machines and Other K volume 1. 2000.

[42] G. Da San Martino. Kernel Methods for Tree Structured Data. PhD thesis, University of Bologna, 2009.

[43] G. Da San Martino, N. Navarin, and A. Sperduti. A memory efficient graph kernel. In The 2012 International Joint Conference on Neural Networks (IJCNN). Ieee, June 2012.

[44] G. Da San Martino, N. Navarin, and A. Sperduti. A Tree-Based Kernel for Graphs. In Proceedings of the Twelfth SIAM International Conference on Data Mining, pages 975–986, 2012.

[45] G. Da San Martino, N. Navarin, and A. Sperduti. A Lossy Counting Based Approach for Learning on Streams of Graphs on a Budget. In IJCAI 2013, Proceedings of the 23rd International Joint Conference on

Artificial Intelligence, Beijing, China, August 3-9, 2013., pages 1294–1301. IJCAI/AAAI, 2013.

[46] G. Da San Martino and A. Sperduti. Mining Structured Data. IEEE Comp. Int. Mag., 5(1):42–49, 2010.

[47] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. Journal of Medicinal Chemistry, 34(2):786–797, Feb. 1991.

[48] O. Dekel, S. S. Shwartz, Y. Singer, and S. Shalev-Shwartz. The Forgetron: A Kernel-Based Perceptron on a Budget. SIAM J. Comput., 37(5):1342–1372, 2008.

[49] V. Di Massa, G. Monfardini, L. Sarti, F. Scarselli, M. Maggini, and M. Gori. A Comparison between Recursive Neural Networks and Graph Neural Networks. The 2006 IEEE International Joint Conference on Neural Network Proceedings, pages 778–785.

[50] T. Dietterich, Thomas G.; Lathrop, Richard H.; Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. 89, 1997.

[51] P. D. Dobson and A. J. Doig. Distinguishing Enzyme Structures from Non-enzymes Without Alignments. Journal of Molecular Biology, 330(4):771–783, 2003.

[52] P. Domingos and G. Hulten. Mining High-Speed Data Streams. In Proc. of the 6th Internl. Conf. on Knowl. Disc. and Data Mining (KDD'00), pages 71–80, Boston. MA, 2000.

[53] K. K. Driessens and K. Driessens. Graph kernels and Gaussian processes for relational reinforcement learning. Machine Learning, pages 146–163, 2003.

[54] S. R. Eddy and H. Hughes. NON-CODING RNA GENES AND THE MODERN WORLD. Genetics, 2(December), 2001.

[55] C. H. Elzinga and H. Wang. Kernels for acyclic digraphs. Pattern Recognition Letters, 33(16):2239–2244, Dec. 2012.

[56] M. Eskandari and S. Hashemi. A graph mining approach for detecting unknown malwares. Journal of Visual Languages & Computing, 23(3):154–162, June 2012.

[57] D. Fera, N. Kim, N. Shiffeldrim, J. Zorn, U. Laserson, H. H. Gan, and T. Schlick. RAG: RNA-As-Graphs web resource. BMC bioinformatics, 5:88, July 2004.

[58] R. Field, H. Solomon, T. M. Cover, and P. E. Hart. On the Mean Accuracy of Statistical Pattern Recognizers. Electronics, I, 1968.

[59] T. Fink and J. L. Reymond. Virtual exploration of the chemical universe up to 11 atoms of C, N, O, F: assembly of 26.4 million structures (110.9 million stereoisomers) and analysis for new ring systems, stereochemistry, physicochemical properties, compound classes, and drug discove. J Chem Inf Model, 47(2):342–353, 2007.

[60] M. Fisher, M. Savva, and P. Hanrahan. Characterizing Structural Relationships in Scenes Using Graph Kernels. ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2011, 30(4), 2011.

[61] O. Flint. Graph Isomorphism Testing. CMS, 2010.

[62] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. ACM SIGMOD Records, 34(2):18–26, 2005.

[63] J. Gama and C. Pinto. Discretization from data streams: applications to histograms and data mining. In Proc. of the 2006 ACM symposium on Applied computing (SAC'06), pages 662–667, Dijon, France, 2006.

[64] B. J. Gantz, D. Reinsel, and B. D. Shadows. THE DIGITAL UNIVERSE IN 2020 : Big Data , Bigger Digital Shadow s , and Biggest Growth in the Far East Executive Summary: A Universe of Opportunities and Challenges. Technical report, IDC, 2012.

[65] P. P. Gardner, J. Daub, J. Tate, B. L. Moore, I. H. Osuch, S. Griffiths-Jones, R. D. Finn, E. P. Nawrocki, D. L. Kolbe, S. R. Eddy, and A. Bateman. Rfam: Wikipedia, clans and the "decimal" release. Nucleic acids research, 39(Database issue):D141–5, Jan. 2011.

[66] T. Gärtner. A survey of kernels for structured data. ACM SIGKDD Explorations Newsletter, 5(1):49, July 2003.

[67] T. Gartner, P. Flach, and S. Wrobel. On Graph Kernels : Hardness Results and Efficient Alternatives. Lecture notes in computer science, pages 129–143, 2003.

[68] T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. Lecture notes in computer science, pages 129–143, 2003.

[69] T. T. Gärtner, J. W. Lloyd, and P. a. Flach. Kernels and Distances for Structured Data. Machine Learning, 57(3):205–232, Dec. 2004.

[70] R. Giegerich, B. Voss, and M. Rehmsmeier. Abstract shapes of RNA. Nucleic acids research, 32(16):4843–51, Jan. 2004.

[71] C. Giraud-carrier. A Note on the Utility of Incremental Learning. Ai Communications, 13 (4):215–223, 2000.

[72] V. Grossi and A. Sperduti. Kernel-Based Selective Ensemble Learning for Streams of Trees. In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, volume 1, pages 1281–1287, 2009.

[73] D. Haussler. Convolution Kernels on Discrete Structures. Technical report, Department of Computer Science, University of California at Santa Cruz, 1999.

[74] H. He, S. Chen, K. Li, and X. Xu. Incremental learning from stream data. IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council, Oct. 2011.

[75] M. Heinonen and J. Rousu. Efficient Path Kernels for Reaction Function Prediction. 2009.

[76] M. Heinonen, N. Välimäki, V. Mäkinen, and J. Rousu. Efficient Path Kernels for Reaction Function Prediction. Bioinformatics Models, Methods and Algorithms, 2012.

[77] C. Helma, T. Cramer, S. Kramer, and L. D. Raedt. Data mining and machine learning techniques for the identification of mutagenicity inducing substructures and structure activity relationships of noncongeneric compounds. Journal of Chemical Information and Computer Sciences, 44:1402–1411, 2004.

[78] S. Hido and H. Kashima. A Linear-Time Graph Kernel. ICDM, 0:179–188, 2009.

[79] I. Hiroshi, H. Keita, H. Taiichi, and T. Takenobu. Efficient sentence retrieval based on syntactic structure. In Proceedings of the COLING/ACL on Main conference poster sessions, COLING-ACL '06, pages 399–406, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[80] T. Horváth, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '04, page 158, New York, New York, USA, 2004. ACM Press.

[81] G. Hulten, L. Spencer, and P. Domingos. Mining Time Changing Data Streams. In Proc. of the 7th Internl. Conf. on Knowl. Disc. and Data Mining (KDD'01), pages 97–106, San Francisco, CA, 2001.

[82] A. Kalousis and M. Hilario. Matching Based Kernels for Labeled Graphs. pages 4–7.

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

[84] H. Kashima and T. Koyanagi. Kernels for Semi-Structured Data. In Proceedings of the Nineteenth International Conference on Machine Learning, pages 291–298. Morgan Kaufmann Publishers Inc., 2002.

[85] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In Proceedings of the Twentieth International Conference on Machine Learning, pages 321–328. AAAI Press, 2003.

[86] M. Kelly, D. Hand, and N. Adams. The impact of changing populations on classifier performance. Proceedings of the fifth ACM SIGKDD ..., 32(2):367–371, 1999.

[87] D. Kimura, T. Kuboyama, T. Shibuya, and H. Kashima. A Subpath Kernel for Rooted Unordered Trees. In In Proceedings of the 15th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), Shenzeng, China, 2011.

[88] R. Klinkenberg. Learning drifting concepts: Example selection vs. example weighting. Intell. Data Anal., 8:281–300, August 2004.

[89] J. Z. Kolter and M. A. Maloof. Dynamic Weighted Majority: An Ensemble Method for Drifting Concepts. Journ. of Mach. Learn. Res., 8:2755–2790, 2007.

[90] S. O. Kuznetsov and M. V. Samokhin. Learning Closed Sets of Labeled Graphs for Chemical Applications. pages 190–208, 2005.

[91] Y.-j. Lee and O. L. Mangasarian. RSVM : Reduced Support Vector Machines. In Data Mining Institute, Computer Sciences Department, University of Wisconsin, pages 1–17, 2001.

[92] B. Li, X. Zhu, L. Chi, and C. Zhang. Nested Subtree Hash Kernels for

Large-Scale Graph Classification over Streams. 2012 IEEE 12th International Conference on Data Mining, pages 399–408, Dec. 2012.

[93] G. Li, M. Semerci, and M. J. Zaki. Graph Classification via Topological and Label Attributes.

[94] P. Mahé, N. Ueda, T. Akutsu, J.-L. Perret, and J.-P. Vert. Extensions of marginalized graph kernels. Twenty-first international conference on Machine learning - ICML '04, page 70, 2004.

[95] G. Manku and R. Motwani. Approximate frequency counts over data streams. In VLDB, pages 346–357, 2002.

[96] D. H. Mathews and D. H. Turner. Prediction of RNA secondary structure by free energy minimization. Current opinion in structural biology, 16(3):270–8, June 2006.

[97] P. Menzel, J. Gorodkin, and P. F. Stadler. The tedious task of finding homologous noncoding RNA genes. RNA (New York, N.Y.), 15(12):2075–82, Dec. 2009.

[98] S. R. Meraz, Richard F. & Holbrook. Classification of non-coding RNA using graph representations of secondary structure. Science, 2004.

[99] R. Michalski, J. Carbonell, and T. Mitchell. Machine learning: An artificial intelligence approach. Kaufman Publishers Inc.,Los Altos, CA, Jan. 1983.

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

[101] A. Moschitti. Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees.

[102] A. Moschitti. Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors,

ECML, volume 4212 of Lecture Notes in Computer Science, pages 318–329. Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings, Berlin, Germany, Sept. 2006.

[103] E. P. Nawrocki, D. L. Kolbe, and S. R. Eddy. Infernal 1.0: inference of RNA alignments. Bioinformatics (Oxford, England), 25(10):1335–7, May 2009.

[104] M. Neuhaus and H. Bunke. A Convolution Edit Kernel for Error-tolerant Graph Matching. Pattern Recognition, International Conference on, 4:220–223, 2006.

[105] M. Neuhaus and H. Bunke. Edit distance-based kernel functions for structural pattern classification. Pattern Recognition, 39(10):1852–1863, 2006.

[106] S. Nowozin, K. Tsuda, T. Uno, T. Kudo, and G. Bakir. Weighted Substructure Mining for Image Analysis. 2007 IEEE Conference on Computer Vision and Pattern Recognition, pages 1–8, June 2007.

[107] F. Orabona, J. Keshet, and B. Caputo. The projectron: a bounded kernel-based perceptron. In ICML '08 Proceedings of the 25th international conference on Machine learning, pages 720–727, 2008.

[108] N. C. Oza and S. Russell. Online bagging and boosting. In Proc. of 8th Internl. Workshop on Artificial Intelligence and Statistics (AISTATS'01), pages 105–112, Key West, FL, 2001.

[109] B. J. Parker, I. Moltke, A. Roth, S. Washietl, J. Wen, M. Kellis, R. Breaker, and J. S. Pedersen. New families of human regulatory RNA structures identified by comparative analysis of vertebrate genomes. Genome research, 21(11):1929–43, Nov. 2011.

[110] J. C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, 1998.

[111] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi. Graph kernels for chemical informatics. Neural networks : the official journal of the International Neural Network Society, 18(8):1093–110, Oct. 2005.

[112] J. Ramon and T. Gärtner. Expressivity versus efficiency of graph kernels. In Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences, pages 65–74, 2003.

[113] R. C. Read and D. G. Corneil. The graph isomorphism disease. Journal of Graph Theory, 1(4):339–363, 1977.

[114] B. C. Russell and A. Torralba. LabelMe: a database and web-based tool for image annotation. International journal of Computer Vision, 77(1-3):157–173, 2008.

[115] H. Saigo, S. Nowozin, T. Kadowaki, T. Kudo, and K. Tsuda. gBoost: a mathematical programming approach to graph classification and regression. Machine Learning, 75(1):69–89, Nov. 2008.

[116] Y. Sakakibara, K. Popendorf, N. Ogawa, K. Asai, and K. Sato. Stem kernels for RNA sequence analyses. Journal of bioinformatics and computational biology, 5(5):1103–22, Oct. 2007.

[117] K. Sato, T. Mituyama, K. Asai, and Y. Sakakibara. Directed acyclic graph kernels for structural RNA analysis. BMC bioinformatics, 9:318, Jan. 2008.

[118] B. Scholkopf and A. J. Smola. Learning with Kernels. MIT Press, Cambridge, MA, USA, 2001.

[119] M. Scholz and R. Klinkenberg. An Ensemble Classifier for Drifting Concepts. In Proceeding of 2nd Internl. Workshop on Knowl. Disc. from Data Streams, in conjunction with ECML-PKDD 2005, pages 53–64, Porto, Portugal, 2005.

[120] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: primal estimated sub-gradient solver for SVM. Mathematical Programming, 127(1):3–30, Oct. 2010.

[121] J. Shawe-Taylor and N. Cristianini. Kernel Methods for Pattern Analysis. Cambridge University Press, New York, NY, USA, 2004.

[122] N. Shervashidze and K. Borgwardt. Fast subtree kernels on graphs. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, Advances in Neural Information Processing Systems 22, pages 1660–1668, 2009.

[123] N. Shervashidze, K. Mehlhorn, T. H. Petri, S. V. N. Vishwanathan, and K. Borgwardt. Efficient graphlet kernels for large graph comparison. 5:488–495, 2009.

[124] N. Shervashidze and P. Schweitzer. Weisfeiler-Lehman Graph Kernels. Journal of Machine Learning Research, 12:2539–2561, 2011.

[125] K. Shin. Mapping kernels defined over countably infinite mapping systems and their application. Journal of Machine Learning Research, 20:367–382, 2011.

[126] K. Shin. Partitionable Kernels for Mapping Kernels. 2011 IEEE 11th International Conference on Data Mining, pages 645–654, Dec. 2011.

[127] K. Shin. A New Frontier of Kernel Design for Structured Data. In Proceedings of the 30th International Conference on Machine Learning (ICML-13), pages 401–409, 2013.

[128] K. Shin, M. Cuturi, and T. Kuboyama. Mapping kernels for trees. In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pages 961–968, 2011.

[129] K. Shin and T. Kuboyama. A generalization of Haussler's convolution kernel:

mapping kernel. In ICML '08: Proceedings of the 25th international conference on Machine learning, pages 944–951, New York, NY, USA, 2008. ACM.

[130] S. Sonnenburg and V. Franc. COFFIN: A computational framework for linear SVMs. In ICML 2010: Proceedings of the 27th international conference on Machine learning, 2010.

[131] W. N. Street and Y. Kim. A streaming ensemble algorithm (\upshape{SEA}) for large-scale classification. In Proc. of the 7th Internl. Conf. on Knowl. Disc. and Data Mining (KDD'01), pages 377–382, San Francisco, CA, 2001.

[132] P. Su and R. L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In Proceedings of the eleventh annual symposium on Computational geometry - SCG '95, pages 61–70, New York, New York, USA, 1995. ACM Press.

[133] J. Suzuki, T. Hirao, Y. Sasaki, and E. Maeda. Hierarchical directed acyclic graph kernel: Methods for structured natural language data. ...of the 41st Annual Meeting on ..., pages 2–4, 2003.

[134] R. J. Tibshirani and R. Tibshirani. A bias correction for the minimum error rate in cross-validation. The Annals of Applied Statistics, 3(2):822–829, June 2009.

[135] I. TINOCO, O. C. UHLENBECK, and M. D. LEVINE. Estimation of Secondary Structure in Ribonucleic Acids. Nature, 230(5293):362–367, Apr. 1971.

[136] A. M. Turing. Computing Machinery and Intelligence. Mind, 59(236):433–460, 1950.

[137] R. van Deursen and J.-L. Reymond. Chemical space travel. ChemMedChem, 2(5):636–40, May 2007.

[138] V. Vapnik and C. Cortes. Support-Vector Networks. MACHINE LEARNING, 20(3):273–297, 1995.

[139] V. N. Vapnik. An overview of statistical learning theory. IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council, 10(5):988–99, Jan. 1999.

[140] S. V. N. Vishwanathan, K. Borgwardt, and N. Schraudolph. Fast computation of graph kernels. NIPS. Cambridge MA MIT Press, 2006.

[141] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph Kernels. Journal of Machine Learning Research, 11:1201–1242, Apr. 2010.

[142] S. V. N. Vishwanathan and A. A. J. Smola. Fast Kernels for String and Tree Matching. In S. Becker, S. Thrun, and K. Obermayer, editors, NIPS, pages 569–576. MIT Press, 2002.

[143] S. V. N. Vishwanathan and A. J. Smola. Fast kernels for string and tree matching. NIPS, 15, 2003.

[144] G. Wahba. Spline models for observational data, volume 59 of CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1990.

[145] N. Wale, I. Watson, and G. Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. Knowledge and Information Systems, 14(3):347–375, 2008.

[146] Z. Wang, K. Crammer, and S. Vucetic. Breaking the Curse of Kernelization : Budgeted Stochastic Gradient Descent for Large-Scale SVM Training. Journal of Machine Learning Research, 13(10):3103–3131, 2012.

[147] Z. Wang, N. Djuric, S. Vucetic, and K. Crammer. Trading Representability for Scalability : Adaptive Multi-Hyperplane Machine for Nonlinear Classification. In 17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), 2011.

[148] Z. Wang and S. Vucetic. Twin Vector Machines for Online Learning on a Budget. SDM, pages 906–917, 2009.

[149] Z. Wang, S. Vucetic, K. Crammer, and O. Dekel. Online Passive-Aggressive Algorithms on a Budget. Journal of Machine Learning Research - Proceedings Track, 9:908–915, 2010.

[150] K. M. Weeks. Advances in RNA structure analysis by chemical probing. Current opinion in structural biology, 20(3):295–304, June 2010.

[151] O. S. Weislow, R. Kiser, D. L. Fine, J. Bader, R. H. Shoemaker, and M. R. Boyd. New soluble-formazan assay for HIV-1 cytopathic effects: application to high-flux screening of synthetic and natural products for AIDS-antiviral activity. Journal of the National Cancer Institute, 81(8):577–586, 1989.

[152] D. H. Wolpert. THE LACK OFA PRIORI DISTINCTIONS BETWEEN LEARNING ALGORITHMS. Artificial Intelligence, III(1978):392–394, 1995.

[153] X. Y. X. Yan and J. H. J. Han. gSpan: graph-based substructure pattern mining. 2002 IEEE International Conference on Data Mining, 2002. Proceedings., 2002.

[154] H.-f. Yu, C.-j. Hsieh, K.-w. Chang, and C.-j. Lin. Large Linear Classification When Data Cannot Fit in Memory. pages 2777–2782, 2007.

[155] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen. P-packSVM: Parallel Primal grAdient desCent Kernel SVM. 2009 Ninth IEEE International Conference on Data Mining, pages 677–686, Dec. 2009.