Graph Kernels and Support Vector Machines for Pattern Recognition

Léo Andéol[*]

Master DAC, Sorbonne Université
Paris, France

Supervised by: Prof. Hichem Sahbi

May 2019

Abstract

The problem of having a framework to classify graphs is becoming increasingly important in the era of data. The issue has been tackled since the beginning of the millennium and there have been significant progress. This report introduces all necessary knowledge to understand the topic and then reviews different graph kernels while focusing on a random walk kernel and its optimization. Some experiments are then conducted to verify the information given by the state of the art, and some attempts are made to accelerate the different methods to compute the kernel.

---

[*]leo.andeol@gmail.com

Contents

1 Introduction

Nowadays the world is shifting towards a new era where data is the most valuable resource, and researchers are working on algorithms to exploit it the best possible way. Most research is currently focused on the so-called "big data", huge quantities of data which are very sparse and of poor quality. However, there also exist structured databases of good quality, in the form of graphs, this different form being the one studied throughout this project. Graphs can be very useful in various fields, but are partly known for their use in biology, as they can be used to represent proteins or other types of molecules.

Classification of such graphs is an important problem of pattern recognition which affects a lot of sectors such as medicine, biology, and more. It was studied since the beginning of the millennium and significant progress has been made : several methods were discovered and gave good results but met a big issue, the complexity of these methods. Indeed, they do not scale well to large graphs as their complexity tends to be several times polynomial. Since then, the main objective of research on this topic has been to either find more computationally efficient algorithms, or to find methods to accelerate the ones already in use.

The first part of this report will introduce all the necessary background knowledge which are graphs as already mentioned, but also Support Vector Machines (SVMs) which is a powerful classification algorithm, and kernels which are transformations of data usually used to increase the accuracy of SVMs. However, graphs being non-vectorial data, kernels have also the purpose of becoming a metric of comparison between two instances of graphs. Thus, the main principles of graph kernels will be introduced as well as their definitions. Afterwards, the different methods used to improve the computation complexity of the kernel will be introduced together with their complexities, advantages and disadvantages.

The second part of this report will be focused on the technical side and experiments conducted during the project. A synthetic graph database made of toy data was required to conduct simple and quick experiments. Indeed, it was made in order to verify the claims made in the main publication studied thanks to easy control on matters such as labeled and unlabeled graphs, and variations of the size of either the database or the graphs inside of it. Different challenges and problems met during the implementation will be explained, as well as their found solutions. Then, the main subject of this part will be discussed : the accuracy and computational time of different methods, estimation of their complexities and different attempts to approximate those methods. Finally, an experiment on real datasets of enzymes and proteins is conducted, and the results analyzed.

2 Methodology

2.1 Background

2.1.1 Graphs

Definition A graph[1] is a type of mathematical structure that represents connections between objects. It is more precisely an ordered pair $G = (V, E)$ of two sets: vertices V (or nodes) and edges E that connect two vertices together.

$$E \subseteq \{(u, v) : (u, v) \in V^2\} \tag{1}$$

Vertices represent objects and are usually depicted as circles or spheres whereas edges link pairs or vertices.
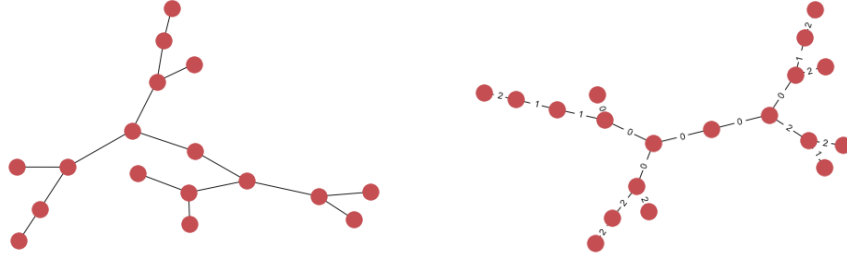

Figure 1: Two "tree" graphs, resp. unlabeled and labeled

Properties Graphs have several different properties that will be used in the rest of this project and will be introduced here

- A graph is undirected if and only if : $\forall (u, v) \in E \implies (v, u) \in E$

- $|V|$ is the number of vertices and $|E|$ the number of edges

- The degree of a vertex $d(v)$ is the number of vertices it is connected to, or the number of distinct edges connected to it.

- A path is a sequence of connected edges linking distinct vertices.

- A cycle is a path where the first and last vertex is the same.

- A graph can have labels (sometimes called colors) either on its vertices or edges (or both). They can take various forms such as integers, more generally elements of a finite or infinite set and even continuous (such as $\in \mathbb{R}$).

- A graph is said to be connected if any vertex is connected, by a path, with any other vertex.

4

- An *undirected* graph is a tree if there is exactly one path between any two vertices.

- A line graph $G' = (V', E')$ of a graph $G = (V, E)$ has a vertex for each edge $e \in E$ in G and an edge $(u, v) \in V'^2$ if and only if there is a vertex in G in common between the two former edges u and v.

- A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is that graph restricted to a subset of nodes $V' \in V$ while keeping all the edges with those vertices $E' = \{(u, v) : (u, v) \in E \land (u, v) \in V'^2\} \subseteq E$

The different methods studied in this report are focused on undirected graphs, either unlabeled or labeled by finite sets. Moreover, only edge-labeled graphs will be studied, thus if a graph is node-labeled, it can be transformed into an edge-labeled graph by computing its line graph.

Adjacency Matrix The adjacency matrix of an undirected graph represents the presence or absence of an edge between two specific vertices. It is defined as the matrix A of dimension $|V| \times |V|$ where entries are given by

$$A_{i,j} = \begin{cases} 1 & \text{if } i \neq j \text{ and } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

History Graphs were first used in their modern form to represent the problem of Seven Bridges of Königsberg, and have been ever since used to represent maps, and thus path-finding algorithms were developed. They have also been used to represent flow problems, scheduling problems, networking routing and many others[1]. Graphs can also be used, with labels, to represent different types of molecules and interactions between them, or more simply to represent molecules by considering atoms as vertices and bonds as edges.
The next part will introduce Support Vector Machines, the algorithm used in this project to classify new unknown graphs into known categories. It has been shown[2] that it can be used effectively on proteins and enzymes.

2.1.2   Support Vector Machines

Support Vector Machines (SVMs)[3] are a type of machine learning algorithms discovered in the early 90s[4]. It was originally a classification algorithm however it has been expanded since to regression and clustering too.

Classification Classification is one of the two problems of supervised learning that aims to automatically recognize and classify observations, such as recognizing handwritten digits. It can be split into two tasks where the first one is to learn how to separate training data into classes using their attributes and the second is to predict the class of new examples. All of that, while minimizing errors. This problem can be formalized as

Definition 1. Classification is the problem of associating observation data to a class. The training data comes as a set of observations and classes $\{(\mathbf{x_i}, y_i)\}$ of size N where $y_i \in \{-1, +1\}$, and using that data, an algorithm has to predict the class of new test data as accurately as possible.

In other words, the objective is to find the best function $f : \mathbb{R}^d \longrightarrow \{-1, +1\}$ among a set of functions F while minimizing a risk function R

$$f\star = \operatorname{argmin}_f R_n(f) \tag{3}$$

where d is the number of features of the data .

SVMs are especially powerful at this task and widely used for different reasons as explained subsequently.

Margin and support vectors The Support Vector Machines are based on the model of the Perceptron[5], another classification algorithm that tried to find a hyperplane that discriminates the two sets. The main issue with the Perceptron is that it has no formal guarantee to find an optimal hyperplane, and usually would not find it. Moreover, if the data are not linearly separable, the algorithm would not converge.

In order to tackle these issues, the SVM offered three solutions. First, a margin is added in the loss L function in order to not only properly classify the data samples, but also with the maximum certainty, i.e. as far as possible from the decision boundary.

This loss is defined as : $L(y_i, \mathbf{x}_i, w) = \max(0, 1 - y_i(\mathbf{x}_i \cdot \mathbf{w} + w_0))$ (4)

Where $\mathbf{x}$ is the feature vector of an instance, y its class and $\mathbf{w}$ and w_0 respectively the weight vector and bias calculated by training the algorithm. Considering this formulation is not sufficient, as there is an infinity of possible solutions. It is proven that the margin γ between points and the decision boundary is inversely proportional to the norm of the weight vector $\|\mathbf{w}\|$.

$$\gamma_i = \frac{y_i(\mathbf{x}_i \cdot \mathbf{w} + w_0)}{\|\mathbf{w}\|} \tag{5}$$

These two solutions give us the first version the SVM, the hard-margin SVM

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{s.t.} \quad y_i(\mathbf{x}_i \cdot \mathbf{w} + w_0) \geq 1 \quad \forall i \in \{1..n\} \tag{6}$$

However, if the data are not linearly separable the algorithm would not admit a solution since the quadratic programming problem requires all points to be correctly classified. Then, a new term ξ is introduced as an error tolerance, as well as a factor C that determines the balance between error tolerance and minimization of the weight vector norm $\|\mathbf{w}\|$:

$$
\begin{aligned}
&\min_{\mathbf{w}} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i \\
&\text{s.t.} \quad \forall i \in \{1..n\} \quad y_i(\mathbf{x}_i \cdot \mathbf{w} + w_0) \geq 1 - \xi_i
\end{aligned}
\tag{7}
$$

Then, the next section will introduce the second advantage of SVMs.

2.1.3 Kernels

In its dual form, the SVM problem only requires a dot product between the observation vectors.

$$\max_\alpha \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{j=1}^{N} \sum_{i=1}^{N} \alpha_i \alpha_j y_i y_j \mathbf{x_i}^\top \mathbf{x_j}$$
$$\text{s.t.} \quad \forall i \quad 0 \leq \alpha_i \leq C \tag{8}$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

Then, a nonlinear transformation $\phi(x)$ can be used to replace the dot-product $\mathbf{x}_i \cdot \mathbf{x}_j$ by $\phi(\mathbf{x_i}) \cdot \phi(\mathbf{x}_j)$, effectively augmenting the dimensions of vectors from the input space into a higher dimensional feature space and thus the expressiveness of the model by enhancing the separability of the data. However, for complex transformations, such as in infinite dimensions, the function ϕ is not explicitly defined. But it was found[4] that the standard dot products can be replaced by other functions as long as it positive semi definite.

Definition 2. A kernel K is positive semi definite if and only if

$$\sum_{i=1}^{N} \sum_{i=1}^{N} \kappa(\mathbf{x_i}, \mathbf{x_j}) c_i c_j \geq 0 \qquad \forall c_1 \ldots c_N \in \mathbb{R}^* \tag{9}$$

Thus, the dot product can be replaced by a function $K(x_1, x_2)$ which will indirectly map the data to higher dimensions like the polynomial kernel(10), even infinite with the RBF kernel (11) and return the dot product of those transformations, while remaining computable in all cases. This replacement is usually called the Kernel Trick. For example, the polynomial kernel is defined as follows

$$\kappa(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i}^\top \mathbf{x_j} + c)^d \tag{10}$$

where d is the degree of the polynomial and c the constant bias. Another example, the RBF kernel is defined as follows

$$\kappa(\mathbf{x_i}, \mathbf{x_j}) = e^{-\frac{\|\mathbf{x_i} - \mathbf{x_j}\|^2}{2\sigma^2}} \tag{11}$$

where σ is a hyperparameter. Ever since the foundation of SVMs, the kernel trick became a big focus of the machine learning community as it naturally fits in the algorithm and allows supervised learning on very complex data, and enjoying greater accuracy than most algorithms.

The fact that the dot product can be replaced by a function without specifying the map ϕ is the reason why the Kernel Trick became a big focus of research. In particular, it will be possible to make a kernel for non-vectorial data such as graphs, as shown in the next section.

2.2 State of the Art

This section will review the state of the art kernels for graphs. Particular attention will be given to the random walk kernel.

2.2.1 Graph Kernels

Graph Kernels are a type of R-convolution kernels[6] applied to graphs, which are kernels that are based on decompositions of complex objects and comparisons of those decompositions.

Graph Kernels have been studied since Support Vector Machines started getting popular[7]. Since then, a lot of progress has been made, and several types of kernels have been discovered, such as random walk and graphlet[8] kernels, the two main families of kernels (and the ones studied in this section), but not the only ones. There are also graph kernels that use subtrees[9], shortest paths[10] and several others that are unfortunately usually either too complex to compute or not positive semi-definite[11]. However it has been shown[12] that in some cases non positive semi definite kernels can still be used efficiently either by using them as a dissimilarity measure, or by artificially making them positive semi definite.

developper plus les autres kernels?

2.2.2 Graphlets

A graphlet is a small graph (with few nodes $k \leq 5$). The idea of the graphlet kernel κ is to generate a set S of all possible graphlets of a certain size k and calculate for each of them the frequency of occurrence in a certain graph $G = (V, E)$ to build a vector $\mathbf{f}_G$ of dimension $|S|$. The kernel is then defined as follows

Definition 3. Let G_1 and G_2 be two graphs, $\mathbf{f}_{G_1}$ and $\mathbf{f}_{G_2}$ the frequency vectors of respectively G and G_2, then the kernel κ is defined as

$$\kappa(G_1, G_2) = \mathbf{f}_{G_1}^{\top} \mathbf{f}_{G_2} \tag{12}$$

comment les méthodes operent : differentes sous methodes (ex methode avaec labels, methode avec orienté ou pas) $\implies$ partie déjà assez longue ?

However the complexity of computing this vector $\mathbf{f}_G$ is $O(n^k)$[8] where $n = |V|$ and is thus computationally extremely expensive. In order to address this issue, a sampling method is introduced. Indeed, if enough samples are drawn, the Law of Large numbers indicates that the sampled distribution will converge to the actual (theoretical) distribution. However a bound has been found on the complexity of such sampling[13] allowing better precision.

2.2.3 Random Walks

Graph kernels based on Random Walks have been studied since kernels started to be used on SVMs[7], several of them were discovered and then unified[14]. Random walks are a type of rather intuitive algorithms : the idea is to randomly walk through a graph and then compare it to random walks in another graph. Actually, given two graphs, the random walks are computed at the same time. Moreover, it has also been shown[15] that performing a walk on two separate graphs at the same time is the same as performing a walk on the product graph. The product graph is in simpler the graph obtained by adding a vertex for each unique unordered pair of vertex of each graph, and adding an edge between two vertices if and only if there is an edge between those vertices in both graphs.

The *vec* operator will also be used as it is very linked to the tensor product. It is the vector obtained by stacking all columns of A.

Definition 4. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, the product graph $G_\times = (V_\times, E_\times)$ is then defined as

- $V_\times = \{(v, w) : v \in V_1, w \in V_2\}$

- $E_\times = \{((v_1, w_1), (v_2, w_2)) : v_1, v_2 \in V_1^2, w_1, w_2 \in V_2^2\}$

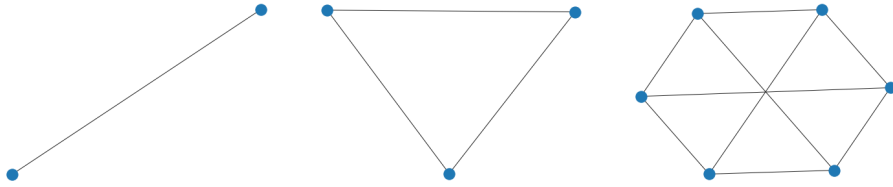For example, on very simple graphs, the following result is obtained

Figure 2: A graph G_1, a graph G_2 and the product graph $G_1 \otimes G_2$

However, technically, the adjacency matrix $W_\times$ of a such graph is the result of the Kronecker (tensor) product of the two adjacency matrices A_1 and A_2 of the two graphs [16]

$$W_\times = A_1 \otimes A_2 \tag{13}$$

In case the graphs are labeled, then the adjacency matrix $W_\times$ is the sum of the Kronecker products of adjacency matrices restricted to each label l

$$W_\times = \sum_{l=1}^{d} A_1^{(l)} \otimes A_2^{(l)} \tag{14}$$

Where d is the size of the label set and $A^{(l)}$ the adjacency matrix of G_x restricted to the label l which is defined as follows

$$A^{(l)} = \begin{cases} 1 & \text{if } i \neq j \text{ and } (i, j) \in E \text{ and } (i, j) \text{ is labeled } l \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

Kernel definition   The idea of the kernel is to perform random walks simultaneously on two graphs in order to compare the walks they have in common. As shown above, this can be achieved by simply performing a random walk on the product graph. By making the adjacency matrix stochastic, and computing the k-th power of the matrix $W_\times$, the probabilities obtained are the ones of walks in common (starting and ending at the same nodes) between the two graphs of length k. By taking into account the start $p_\times$ and end $q_\times$ probabilities, the sum of walks of all possible lengths can be calculated. Moreover using a power series $\mu(k)$ helps de-emphasizing longer walks and makes the sum converge.

9

Definition 5. Let $p_\times$ and $q_\times$ be respectively the start and end probability of each node, and let $W_\times$ be the adjacency matrix of the product graph of G and G', and finally $\mu(k)$ be a convergent function of k, we define the random walk kernel as[14]

$$\kappa(G_1, G_2) = \sum_{k=0}^{\infty} \mu(k) q_\times^\top W_\times^k p_\times \tag{16}$$

This kernel can be computed in $O(kpn^6)$ with k the the number of iterations and p a constant which depends on the complexity of $\mu(k)$.

Inverse Kernel   There is a specific case of the kernel provided that a power series is used. Indeed, by replacing the power series as follows $\mu(k) = \lambda^k$ the following kernel is obtained

$$\kappa(G_1, G_2) = q_\times^\top (I - \lambda W_\times)^{-1} p_\times \tag{17}$$

It can also be computed in $O(n^6)$ however in practice the constant factor is much lower which makes it a better option for very small graphs and databases. Moreover, it will inspire some of the following acceleration methods.

Sylvester Equation   A Sylvester equation, also sometimes called Lyapunov equation is a matrix equation of the following shape :

Definition 6. Let A, B, and C be matrices of compatible dimensions, then the discrete-time Sylvester Equation is

$$AXB + C = X \tag{18}$$

Which can be generalized as

$$\sum_{i=0}^{d} A_i X B_i + C = X \tag{19}$$

These equations will be referred to as Sylvester Equation and Generalized Sylvester Equation. These equations are solved usually using Schur decompositions in $O(n^3)$ for the basic Sylvester equation and in unknown time for the generalized version[14].
This equation can be used to solve our kernel faster assuming a geometric function $\mu(k) = \lambda^k$ with λ an hyperparameter. It can be achieved by replacing the A and B matrices by our adjacency matrices A_1 and A_2, adding λ, and C by our start probabilities $p_\times$ while vectorizing the whole equation thus obtaining

$$vec(X) = vec(\lambda A_1 X A_2) + p_\times \tag{20}$$

From which the following can be obtained (where I is the identity matrix)

$$q_\times^\top vec(X) = q_\times^\top (I - \lambda W_\times)^{-1} p_\times \tag{21}$$

10

Thus, having calculated the inverse kernel by an alternative method. The advantage of this method is that it is very fast and theoretically adapted to both labeled and unlabeled graphs. Moreover, compared to the others, this option is very simple and does not require specific parameters, but like many others, suffers when it encounters singular matrices.

Conjugate Gradient Method The Conjugate Gradient Method[17] is an optimization algorithm used to find approximate solutions of linear systems that have a positive semi definite matrix. As its name suggests, it is a gradient based iterative algorithm. The main idea is that normal gradient descent only takes into account the gradient at the current step of the algorithm, and depending on the step size may cancel progress made in the previous step. The
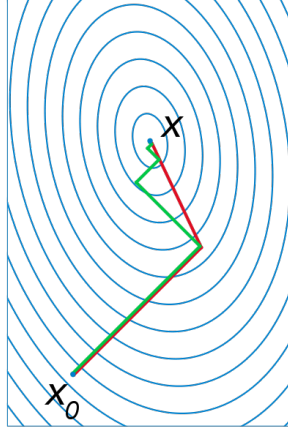

Figure 3: Conjugate gradient (red) compared to Gradient Descent (green)

Conjugate Gradient method is introduced as a fix for that issue. The idea of the algorithm is to keep former gradients in order improve the convergence speed by making the new gradient orthogonal to all former ones through the Gram-Schmidt process, at each step. Thus, unless there are approximation errors, the algorithm guarantees convergence in r steps, being the number of dimensions of the problem, here the size of our graph.

However, this algorithm has limitations, it can only be used to solve linear systems of equations as $Ax = b$ under the constraint that A is symmetric positive definite. In our case, this algorithm can be used to solve the following problem

$$(I - \lambda W_\times)x = p_\times \tag{22}$$

And the kernel value can be obtained by computing $q_\times^\top x$. The complexity of that method is $O(rn^3)$ for unlabeled graphs and $O(rdn^3)$ for labeled graphs where r is the effective rank of the matrix $W_\times$. It is important to note that this kernel also assumes a geometric $\mu(k)$ function.

This method is thus slower than the Sylvester Equation, however it is easily applied to labeled graphs, which is its main advantage.

Fixed Point Iterations   A fixed point is a value for which a function, returns that same value

Definition 7. Let S be a set and f a map $f : S \to S$,
$x \in S$ is a fixed point of f if and only if $x = f(x)$

Fixed point iterations is a method of computing a fixed point of a function by applying repeatedly the following equation until $\|x_{n+1} - x_n\| < \epsilon$ where ϵ is the acceptable level of error.

$$x_{n+1} = f(x_n) \tag{23}$$

There is also a guarantee of convergence

Theorem 1. *In order for the fixed-point iterations to converge, it is necessary that $\lambda < |\xi|^{-1}$ where ξ is the largest eigenvalue of $W_\times$*

Since this method requires the computation of $W_\times$, the kernel value can be computed for any type of labeling. For unlabeled graphs, the complexity is $O(kn^3)$ and $O(kdn^3)$ for labeled graphs, where d is the number of labels and k the number of iterations which can be estimated by the following assuming that the contraction factor is $\lambda\xi_1$ with ξ_1 the largest eigenvalue of $W_\times$:

$$k = O\left(\frac{\ln \epsilon}{\ln \lambda + \ln|\xi_1|}\right) \tag{24}$$

Finally, the fixed point iterations is more demanding than the others since it has a condition on the value of λ and thus assumes as geometric $\mu(k)$ function, and the number of iterations can easily vary a lot. The only advantage of this method is that it is very simple while still being better than the raw kernel.

Spectral Decomposition   The Spectral Decomposition, usually called the eigendecomposition of a matrix M is a factorization in the form VDV^{-1} where D is the diagonal matrix of eigenvalues and V the matrix of eigenvectors. This decomposition can reduce the computation time of the kernel by making most of the matrix operations trivial

$$\kappa(G, G_2) = \sum_{k=0}^{\infty} \mu(k) q_\times^\top (V_\times D_\times V_\times^{-1})^k p_\times = q_\times^\top V_\times \left(\sum_{k=0}^{\infty} \mu(k) D_\times^k\right) V_\times^{-1} p_\times \tag{25}$$

Indeed, the power now only applies to a diagonal matrix which can be computed much faster. Moreover, this method unlike the others preserves the prior $\mu(k)$ without any conditions. If $\mu(k) = \lambda^k$ is chosen then the kernel becomes

$$\kappa(G, G_2) = q_\times^\top V_\times (I - \lambda D_\times)^{-1} V_\times^{-1} p_\times \tag{26}$$

12

That kernel is even more trivial to compute as the inverse of a diagonal matrix is simply the inverse of each of its entries and is thus computed in linear time. Finally, by using $\mu(k) = \frac{\lambda^k}{k!}$ as prior, the following new kernel is obtained

$$\kappa(G, G_2) = q_\times^\top V_\times e^{\lambda D_\times} V_\times^{-1} p_\times \tag{27}$$

However, the complexity of the computation of the eigendecomposition is as large as the raw kernel $O(pn^6)$, but, for unlabeled graphs, a big improvement is possible by simply calculating the eigendecompositions of the original adjacency matrices A_1 and A_2 thus cutting the time to $O(pn^3)$, thanks to a property of the Kronecker product

$$A_1 \otimes A_2 = (V_1 D_1 V_1^{-1}) \otimes (V_2 D_2 V_2^{-1}) = (V_1 \otimes V_2)(D_1 \otimes D_2)(V_1 \otimes V_2)^{-1} \tag{28}$$

By rewriting the spectral decomposition kernel taking advantage of this property, the following new kernel is obtained

$$\kappa(G_1, G_2) = (q_1^\top V_1 \otimes q_2^\top V_2)(\sum_{k=0}^{\infty} \mu(k)(D_1 \otimes D_2)^k)(V_1^{-1} p_1^\top \otimes V_2^{-1} p_2^\top) \tag{29}$$

Which can also be altered by choosing specific $\mu(k)$ as described above. This method ends up being the most efficient of all for unlabeled graphs as it is computed in $O(pn^2)$ where p is the computation time of the power series μ. Moreover this complexity can further improve the computation time of the gram matrix because the eigendecomposition of an adjacency matrix only needs to be computed once. However, this method cannot be expanded to labeled graphs as it encounters difficulties applying the property mentioned above.

Nearest Kronecker Product Approximation This method is more particular as it is not a new way to compute the kernel, but rather a way to generalize all the other methods. Indeed the Nearest Kronecker Product Approximation[18] is used to approximate two matrices A and B as if their Kronecker product if equal to the Kronecker product of two unlabeled graphs' adjacency matrices $W_\times \approx A \otimes B$. This methods allows using any of the methods mentioned above since it returns only two matrices and thus brings back the problem to unlabeled graphs. This method tries to minimize the Frobenius norm of the difference $\|W_\times - A \otimes B\|_F$. It can be computed in $O(dn^2)$ since $W_\times$ is a sum of Kronecker products, but since it is used in conjunction with other methods that are all in cubic time complexity, their overall complexity remains unchanged.

However this method also brings its own set of drawbacks. Since it required the Kronecker product of each couple of adjacency matrices to be computed, the previous advantage of the Spectral Decomposition (only computing once the eigendecomposition for each adjacency matrix) is lost as the obtained matrices are unique for each $W_\times$.

2.2.4 Digest

Several acceleration methods were introduced with various advantages and disadvantages. The following table sums up the most important attributes of each acceleration method. The next section will be focused on experiments. It

Method	Labels		$\mu(k)$		Complexity	
	None	Finite Set	λ^k	Any	Unlabeled	Labeled
Sylvester Equation	$\times$	$\times$[1]	$\times$		$O(n^3)$	Unknown
Conjugate Gradient	$\times$	$\times$	$\times$		$O(rn^3)$	$O(rdn^3)$
Fixed Point Iterations	$\times$	$\times$	$\times$		$O(kn^3)$	$O(kdn^3)$
Spectral Decomposition	$\times$		$\times$	$\times$	$O(pn^3)$	

Table 1: Digest of the different properties of the aforementioned methods

starts by introducing the framework used for this project and then analyzes the performances of those methods.

---

[1]Theoretically[21]

3 Experiments

A big part of this project consisted in implementing the different methods used to compute the random walk kernel[14], verifying the results obtained in that article, and making new attempts to accelerate those algorithms while keeping the best accuracy possible.

3.1 Databases and Metrics

3.1.1 A synthetic graph database

In order to conduct experiments and verify claims made in the article, toy data is necessary. The first thing that had to be done is to create a graph database generator, of very simple and standard graphs that had predictable behaviors and could thus be used for standard tests. Those graphs would also be altered using simple transformations, in order to create different graphs belonging to the same class. For simplicity, it was decided to only use 3 classes. Focus is put
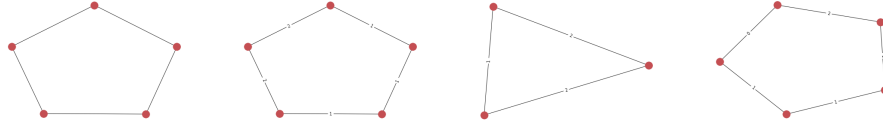
Figure 4: Ring graphs, resp. unlabelled, labelled, altered structured, altered labels

on 3 standard types of graphs : the ring, the star and the tree. A ring graph is a connected graph where each vertex is connected to exactly two vertices, it thus forms one big cycle. A star graph is a particular tree-structured graph composed of a central node, and of several other nodes that are only connected to the central node, thus looking like stars in cartoon representations. Finally, a tree is a famous type of graph : it is a connected graph without any cycle, but here a special type of tree will be studied, since each vertex has a maximum degree of 3. Thus a synthetic graph database generator is created. For each
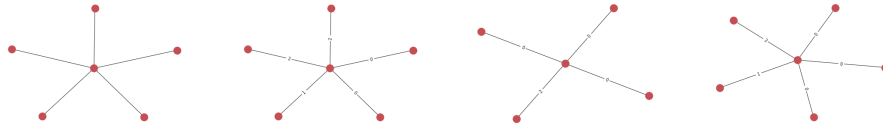

Figure 5: Star graphs, resp. unlabeled, labeled, altered structured, altered labels

type of graph, it creates a graph of predetermined size with randomly generated

edge labels in a given interval. Then, for each graph, it alters it a predefined number of times, and each time creates a new graph that has been both altered on structure and on labels. Alteration on structure involves removing or adding a predefined number of nodes with respect of the type of the graph thus staying in the same class as the source.

1. Ring graphs are simply regenerated slightly longer or shorter randomly.

2. Star graphs are also regenerated either with more or less nodes, randomly.

3. Tree graphs are either expanded by adding randomly leaves anywhere in the graph, or reduced by randomly removing leaves.

Alteration on labels will randomly switch a predefined number of edge labels. Once this is done, the generator will create two databases out of the set of previously generated graphs, one will be made of the adjacency matrix of each graph without looking at labels, and the second one will be an array of adjacency matrices taken from graphs induced by selecting only edges with a specific label, and that for all labels in the label set.
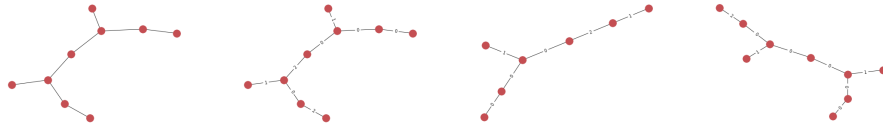

Figure 6: Tree graphs, resp. unlabeled, labeled, altered structured, altered labels

3.1.2 Real databases

After testing the different methods on synthetic data, they will be applied to two real datasets. The two selected datasets are Enzymes and MUTAG, respectively a dataset of protein structures and a dataset of chemical compounds. The objective for Enzymes is to classify the proteins into one of the 6 top-level classes, and for MUTAG to classify the chemical compounds based on their mutagenic effect on bacterium. on va analayser le temps de calcul

Dataset	size	classes	# of vertices	# of edges
Enzymes	600	6	33	124
MUTAG	188	2	18	39

Table 2: Statistics on the real datasets

16

3.1.3 Metrics

In the following experiments only two simple metrics will be used to compare the different methods. First, the accuracy is evaluated in a given set X simply using the 0-1 loss function as follows

$$L(X, \mathbf{y}) = \frac{1}{|X|} \sum_{i=1}^{|X|} \begin{cases} 1 & \text{if } f(\mathbf{x}_i) \neq y_i \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

This function gives the rate of prediction errors on test data in X since the classes are balanced.

The second metric used is the computation time, expressed in seconds.

3.2 Evaluation Protocols

Implementing code from the main paper is a challenge in itself. Several problems aren't really documented and finding functions to solve them, or even explanations on how to solve them is sometimes difficult.

Support Vector Machines The Support Vector Machines algorithm is not implemented since it is not the focus of this project. Instead, the famous library libsvm[19] is used through the python library scikit-learn[20] because of its efficiency and renown. Moreover, there are some hyperparameters to optimize, namely the penalty coefficient on classification mistakes C. Several experiments on various databases shower that the penalty coefficient C had virtually no effect on the accuracy, thus was kept at a value of 1.

Gram Matrix The SVM library we used allows us to precompute the kernel and feeding it to the algorithm. This kernel matrix is called a gram matrix where its entries $M_{i,j} \in [0, 1]$ represents the similarity between G_i and G_j. Thus this matrix is obviously symmetric and it is only required to compute approximately half of the entries (precisely $\frac{n^2+n}{2}$).

Raw Kernel The Raw Kernel is very simple to implement however encounters an issue : a sum to infinity cannot be computed as is, and thus a choice had to be made. It is decided that the raw kernel would be computed over T iterations where T would be the number of vertices of the largest graph. Moreover it was also decided to use the aforementioned power series $\mu(k) = \lambda^k$ to be able to compare results with the other methods restricted to it. The advantage is that all graph kernel values would be computed with the same number of iterations and the number of iterations would be kept small while all vertices would be reachable from any other vertex (the shortest path between two vertices in a connected graph $G = (V, E)$ is at most $|V|$ long).

Inverse Kernel This kernel is also very simple to implement since it only required computing $I - \lambda W_\times$ and its inverse, which is handled exclusively by numpy (part of the scipy project).

Sylvester Equation The Sylvester Equation is a good example of poorly documented problems (compared to the others). There are very few resources on the subject, and even less implementation of solvers. We could theoretically use the Sylvester Equation on both unlabeled and discrete-labeled graphs which are respectively represented as Sylvester and Generalized Sylvester Equations (involving a sum of matrices), however, the only library available so far solves the first one, but not the second one. There seems to be confusing terminology since there is a solver for the Generalized Sylvester Equation, but a different one than the one described in the studied source[14]. The same source also mentions a way to solve the generalized version[21], however, this option is not explored because of a lack of time.

Conjugate Gradient Method The Conjugate Gradient Method is however very well documented[17] and there are several libraries implementing this algorithm. Since the main library in python[22] only implemented this method on sparse matrices, it gave very poor computational time efficiency on our matrices. It was then decided to implement it directly and gave much better results. As said previously, the algorithm guarantees convergence in r step where r is the effective rank of the matrix $I - \lambda W_\times$. However, by grid search on the max-

Figure 7: Accuracy and computation time depending on number of maximum iterations

imum number of iterations allowed for the algorithm, it was discovered that on synthetic data the effective rank of the matrices are very low and the algorithm converges very quickly. This can be cause by the fact that the synthetic database is too simple and the adjacency matrices too sparse, making the problem too easy to solve.

Fixed Point Iterations   The Fixed Point Iterations method is very easy to implement since the algorithm only consists in repeating $x_{t+1} = p_\times + \lambda W_\times x_t$ until either convergence is obtained or the maximum number of iterations reached. It rapidly gave good results thus being used as a tool of comparison with other kernels that struggled more at the beginning. However there is a constraint on
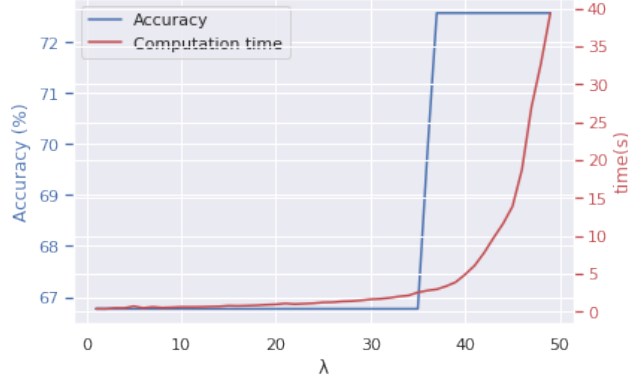

Figure 8: Accuracy and computation time depending on chosen λ

the lambda (the geometric factor) used in the kernel as proved above. Experiments showed that a lambda very close to the constraint increases significantly the time of computation to convergence, as well as the density of the adjacency matrix to a lower extent.

Spectral Decomposition   The Spectral Decomposition method is also very well documented since it is simply another name for the eigendecomposition, one of the main problems of linear algebra. Because of that it is very simple to implement since the method only requires Kronecker products and inverses. However, the fact that several inversions are required can lead to an accumulation of numerical imprecision. Moreover, in some other cases the left eigenvectors matrix is singular and is not invertible. In order to fix that, an ϵ was added to all the diagonal of the adjacency matrices such that it wouldn't alter the results (and did not) while allowing all kernels to be computed. Finally, the chosen $\mu(k)$ function is the power series to keep the method comparable to others.

Nearest Kronecker Product   The Nearest Kronecker Product Approximation (NKP)[18] is used in conjunction with others methods in order to make them compatible with labeled graphs, it is especially useful with the spectral decomposition which is theoretically the fastest method available. There is some documentation of the topic however it is relatively hard to get a good grasp on the topic. Thus, a simple version of the NKP was implemented that does not take into account the special property of our product graph being composed of

19

a sum of Kronecker products, making the overall complexity $O(n^3)$ instead of $O(dn^2)$.

3.3 Performance

First of all the different methods will be tested with synthetic data to verify claims made in the report. Then, the methods will be applied to real world problems.

3.3.1 Performance on synthetic data

Gram Matrices The algorithms approximate the kernel using very different techniques, thus the gram matrices obtained from the kernels were similar in appearance, but had significant differences of scale. It was then decided to normalize very simply the gram matrices so they could be more easily compared.

$$M = (M - min(M))/(max(M) - min(M)) \tag{31}$$

It can be seen that the two matrices are indeed very similar, but locally there can

Figure 9: A gram matrix computed with the raw method, another with the fixed point method, and the difference between the two

be big differences. The difference matrix shows that the fixed point kernel has higher values on the three diagonal blocks in average, and in average also, less in the outer parts of the matrix. Since in this example the data is ordered by class, it shows that the fixed point method gives in average higher values to graphs of the same class and smaller values to graphs of different classes than the raw method. It can be conjectured that the fixed point method will give a better accuracy, which is verified empirically in another paragraph. Afterwards, in order to verify all the algorithms gave similar results, it was decided to compute the Frobenius norm of the difference of two gram matrices (divided by the size of the matrix, thus giving the mean squared error of any entry). The following results were obtained and were satisfying. Indeed, the mean squared error is small in most cases so that the different acceleration methods are probably going to give approximately correct results.

	Raw kernel	Inverse Kernel	Sylvester Equation	Conjugate Gradients	Fixed points	Spectral Decomp.
Raw.	0	1.1e-4	9.8e-5	8.9e-5	1.0e-4	1.0e-04
Inv.	-	0	2.1e-5	7.9e-5	4.0e-6	6.8e-6
Syl.	-	-	0	8.0e-5	1.7e-5	1.4e-5
Con.	-	-	-	0	7.9e-5	7.9e-5
Fix.	-	-	-	-	0	2.8e-6
Spe.	-	-	-	-	-	0

Table 3: Mean squared error of matrix entries

Label Use   The synthetic database generates at the same time adjacency matrices of each graph, and the list of adjacency matrices filtered by each label. Thus, we can compare the scores obtained for each of our metrics, computed using the fixed point kernel since it is both fast and allows the use of labels. The accuracy has a lot of variation and thus it is hard to compare it. It can
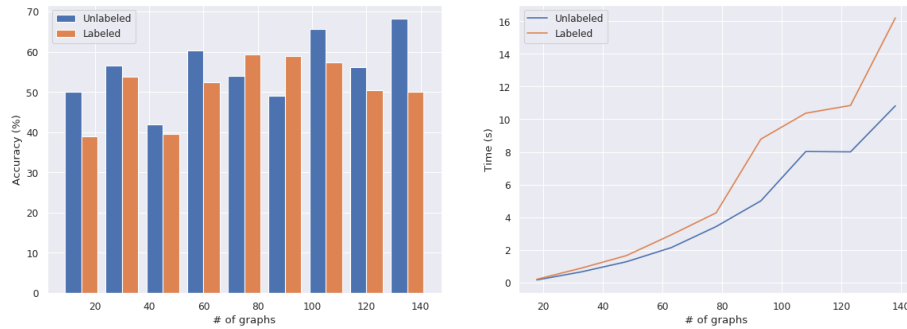
Figure 10: Accuracy and computation time of learning for unlabeled and labeled graphs

probably be explained by the fact that we are working on graphs that are not big enough, and thus a small change in labels could have a big effect on the kernel, for example on the central edge of a tree graph. It is conjectured that provided a large graph and connected enough, a small change in label does not alter much its list of adjacency matrices, labels will have a positive impact on the accuracy. Concerning the computation time, the difference is quite small though by increasing the size of the database the difference can be better observed. There is a small factor of difference during the computation of the tensor product.

refaire cette experience en changeant quoi ?

Comparison of methods depending on number of graphs   Pas de sens, grandit à la meme vitesse = taille de bd au carré ?
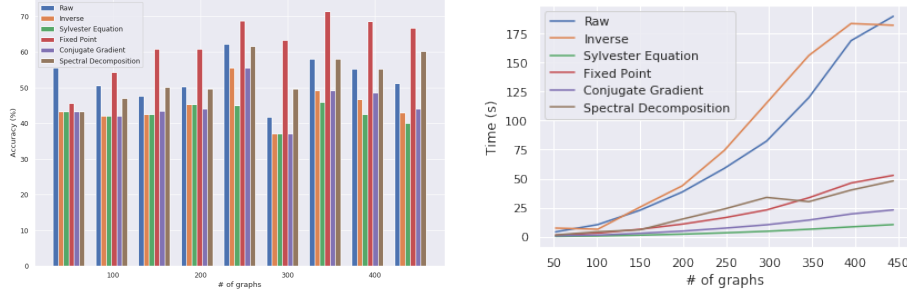
faire ça

Figure 11: Accuracy and Computation time of different methods depending on the size of the database

Comparison of methods depending on number of nodes In order to verify the complexities of the acceleration methods introduce above and compare their efficiency their accuracy and computation time was calculated on synthetic databases with graphs of varying sizes. The databases were generated independently, thus the experiments of different sizes should not be compared in absolute values but relative rather. However, being generated following the same precise rules, the deviation is not too large. Concerning the accuracy, the
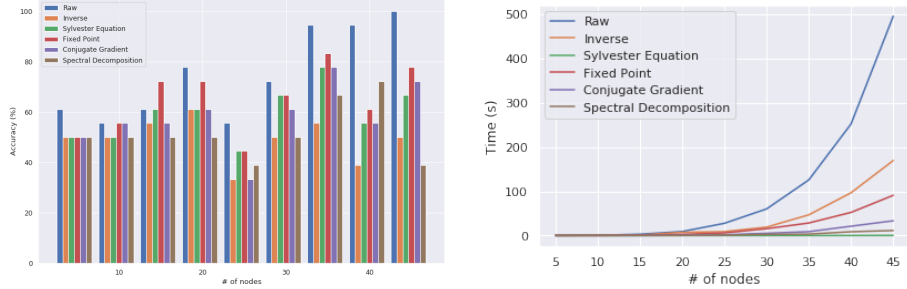
Figure 12: Accuracy and Computation time of different methods depending on the size of graphs

conjecture on fixed point accuracy made earlier is verified in most cases : the fixed point method gives the best approximation of the raw kernel, albeit at the cost of higher computational time. It can be also noted that the raw kernel method is rarely beaten, only once by the fixed point kernel only. Surprisingly the inverse method performs in average worst, all the other methods approximating it giving better results. Finally, the Spectral Decomposition method gives disappointing results, which can be explained by the accumulation of numerical imprecision.

On the other hand, computation time. It is important to remind that in this case some methods (fixed point, conjugate gradient) were implemented entirely for this project, while the Spectral Decomposition and Inverse methods used tools to compute eigenvalues and inverses, and finally the Sylvester Equation solver was taken from a library based in Fortran. This library is a lot more optimized than my personal code and thus creates a bias in computation time. The difference of complexity between the raw method, the inverse method and others is very noticeable and thus the acceleration methods are very time efficient. The most particular case is again the fixed point method which has a complexity increasing much faster than other methods, which could be explained by the number of iterations being much higher than the conjugate gradient for example, and thus computing the matrix vector products several more times.

Nearest Kronecker Product   This acce

3.3.2   Performance on real datasets

This implementation was also tested on two real databases being Proteins and Enzymes

Method	Enzymes		MUTAG	
	Accuracy	Comp. time	Accuracy	Comp. time
Sylvester Equation	18.2 ± 0.83	3'27"	83.5 ± 2.8	5"
Conjugate Gradient	18.2 ± 0.83	1h32'44"	83.5 ± 2.8	18"
Fixed Point Iterations	18.2 ± 0.83	3h14'59"	83.5 ± 2.8	47"
Spectral Decomposition	18.0 ± 1.66	2h15'23"	82.4 ± 1.3	43"

Table 4: Classification accuracy in percent % with standard deviation and computation time for each acceleration method on two datasets

4   Conclusion and Future Work

todo

Acknowledgments

A   Proofs for Section 2 (Methodology)

Theorem 1. *In order for the fixed-point iterations to converge, it is necessary that $\lambda < |\xi|^{-1}$ where ξ is the largest eigenvalue of $W_\times$*

Proof. Let $x_0 = p_\times$ and $t >> 0$

$$
\begin{aligned}
x_{t+1} - x_t &= p_\times + \lambda W_\times x_t - x_t \\
&= p_\times + (\lambda W_\times - 1)(p_\times + \lambda W_\times x_{t-1}) \\
&= p_\times - p_\times + \lambda W_\times p_\times + (\lambda W_\times)^2 x_{t-1} - \lambda W_\times x_{t-1} \\
&= \lambda W_\times (p_\times - x_{t-1} + \lambda W_\times x_{t-1}) \\
&= \lambda W_\times (p_\times - \lambda W_\times x_{t-2} - p_\times + \lambda W_\times (\lambda W_\times x_{t-2} + p_\times)) \\
&= \lambda W_\times (\lambda W_\times (\lambda W_\times x_{t-2} + p_\times - x_{t-2})) \\
&= (\lambda W_\times)^2 (\lambda W_\times x_{t-2} + p_\times - x_{t-2})
\end{aligned}
$$

A pattern is found

$$
\implies x_{t+1} - x_t = (\lambda W_\times)^t (\lambda W_\times x_0 + p_\times - x_0)
$$

And because $x_0 = p_\times$

$$
= (\lambda W_\times)^{t+1} p_\times
$$

Which decreases only when

$$
\xi_1 < 1
$$

The largest magnitude eigenvalue of $\lambda W_\times$

$$
\implies \lambda < |\xi_1|^{-1} \qquad \qquad \square
$$

List of Figures

List of Tables

References

[1] J. A. Bondy, U. S. R. Murty, *et al.*, *Graph theory with applications*, vol. 290. Citeseer, 1976.

[2] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics (Oxford, England)*, vol. 21 Suppl 1, pp. i47–56, June 2005.

[3] C. J. C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Min. Knowl. Discov.*, vol. 2, pp. 121–167, June 1998.

[4] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, Sept. 1995.

[5] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Machine learning*, vol. 37, no. 3, pp. 277–296, 1999.

[6] D. Haussler, "Convolution kernels on discrete structures," Technical Report UCS-CRL-99-10, University of California at Santa Cruz, Santa Cruz, CA, USA, 1999.

[7] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pp. 321–328, AAAI Press, 2003.

[8] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," in *Artificial Intelligence and Statistics*, pp. 488–495, Apr. 2009.

[9] J. Ramon and T. Gärtner, "Expressivity versus efficiency of graph kernels," Citeseer.

[10] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Fifth IEEE international conference on data mining (ICDM'05)*, pp. 8–pp, IEEE, 2005.

[11] N. Shervashidze, *Scalable graph kernels*. PhD thesis, Universität Tübingen, 2012.

[12] B. Schlkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, Mass: The MIT Press, 1st edition ed., Dec. 2001.

[13] T. Weissman, E. Ordentlich, G. Seroussi, S. Verdu, and M. J. Weinberger, "Inequalities for the l1 deviation of the empirical distribution," *Hewlett-Packard Labs, Tech. Rep*, 2003.

[14] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph Kernels," *Journal of Machine Learning Research*, vol. 11, no. Apr, pp. 1201–1242, 2010.

[15] W. Imrich and S. Klavzar, *Product graphs: structure and recognition*. Wiley, 2000.

[16] P. M. Weichsel, "The kronecker product of graphs," *Proceedings of the American mathematical society*, vol. 13, no. 1, pp. 47–52, 1962.

[17] Y. Nesterov, *Lectures on Convex Optimization*. Springer Optimization and Its Applications, Springer International Publishing, 2 ed., 2018.

[18] C. F. Van Loan and N. Pitsianis, "Approximation with kronecker products," in *Linear algebra for large scale and real-time applications*, pp. 293–314, Springer, 1993.

[19] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[21] L. De Lathauwer, B. De Moor, and J. Vandewalle, "Computation of the canonical decomposition by means of a simultaneous generalized schur decomposition," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 2, pp. 295–327, 2004.

[22] E. Jones, T. Oliphant, P. Peterson, *et al.*, "Scipy: Open source scientific tools for python, 2001," 2016.