TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter One

Context and related works

In this chapter, we first introduce graphs and how they arise from real world networks. Then we present the graph classification problem, along with applications in which it arises. Finally, we proceed to detailing state-of-the-art methods and their limitations, before stating our contribution.

1.1 Graph structures and its presence in real world

The need for graphs and their analysis can be traced back to 1679, when G.W. Leibniz wrote to C. Huygens about the limitations of traditional analysis methods of geometric figures and said that "we need yet another kind of analysis, geometric or linear, which deals directly with position, as algebra deals with magnitude", (**Graph_application**). This lead to graphs, mathematical objects that provide a pictorial and efficient form to represent data with many inter-connections.

Formally, a graph of size v is a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{u_1, ..., u_v\}$ is a set of v graph nodes (or vertices), and $\mathcal{E} \in \mathcal{V} \times \mathcal{V}$ is the set of edges between these nodes, i.e. $(u_i, u_j) \in \mathcal{E}$ means that the graph has an edge between node u_i and node u_j.

Graph structures are used to model a set of objects and their interactions/relations. While the nature of these objects and their interactions vary with the application, the underlying

Network	Nodes	Node features	Edges	Edge features
Transportation System	cities	registered cars	Routes	Length, cost
Banking Network	Account holders	account status	Transactions	Transaction value
Social Network	users	name, country	Interactions	type (like, comment)

Table 1.1 Some real world graphs

modeling paradigm is the same for all applications: objects are represented by nodes, and a relation between two objects is represented by an edge between the corresponding two nodes. **For instance, in a social network like Facebook, nodes are .. and edges are... In a biological network such as the brain, nodes are brain regions and edges are.. In a transportation network such as the subway, nodes are .. and edges are.. (see Table.1.1 for a list of different examples).

These graphs, if not too large, can be visually represented in order to provide an intuitive understanding of the existing interactions. Such an illustration is in Fig. 1.1 in the application of chemical reactions.

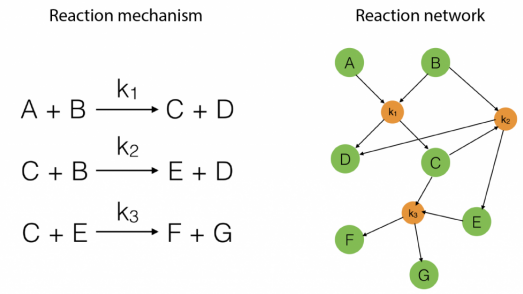

Figure 1.1 Graph structures in representing chemical reactions mechanisms

1.2 Graph classification problem

Graph classification can be understood in several ways. Here, we place ourselves in the context of *supervised learning*, where we suppose we have access to a set of pre-labeled graphs

$(\mathcal{X} = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}, \mathcal{Y} = \{y_1, \ldots, y_n\})$, where each graph $\mathcal{G}_i$ is *a priori* known to belong to the class with label y_i. Stated simply, the graph classification problem we are interested in in this work may be stated as: given this prior information, design a classification algorithm that, given in input any graph (importantly, any graph belonging or not to $\mathcal{X}$), outputs the label of the class to which it belongs.

More formally, consider the set $\mathcal{D}$ of all graphs $\mathcal{G}$ that can occur in some real-world application, a fixed set of classes $\beta = \{\beta_1, \ldots, \beta_l\}$ of finite size l, and a mapping function $f : \mathcal{D} \mapsto \beta$ which maps each graph $\mathcal{G}$ in $\mathcal{D}$ to the class $\beta_{\mathcal{G}}$ it belongs to. Graph classification is the problem of estimating the mapping function f in the case where it is only known on a subset $\mathcal{X} \subset \mathcal{D}$. Formally, we have a dataset $(\mathcal{X} = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}, \mathcal{Y} = \{y_1, \ldots, y_n\})$ of size n such that $\mathcal{X} \in \mathcal{D}^n$ and $\mathcal{Y} \in \beta^n$, where for each graph $\mathcal{G}_i \in \mathcal{X}$ we have that $y_i = f(\mathcal{G}_i)$ is the class of $\mathcal{G}_i$. The classification task is to have a **predictive model** which can predict well, based on some-predefined metric, the class for any new graph $\mathcal{G}$ in $\mathcal{D}$. This prediction functionality of the model is gained using the dataset $(\mathcal{X}, \mathcal{Y})$ to optimize the parameters of the model that is believed to govern the behavior of the mapping function f on $\mathcal{D}$. This optimization completed in this paradigm is called the learning algorithm.

Note that graph classification as considered here, has nothing to do with the more common problem of *node classification* in a graph, in which there exists only one graph and the goal is to separate the node set in a partition of communities. In our work, graphs are classified, not nodes. This being said, the extra information that the nodes and/or edges may have in some applications (gender, age for instance for nodes of a social network; maximum bandwidth, number of channels for instance for edges of a communication network; etc.) could in principle be used along with the graph structure to classify different graphs into different classes. However, as the existence of such extra-information is very application-dependent, we prefer to focus here on the case where nodes and edges do not carry such information: the only information one has access to for classification is the graph structure.

This problem has been addressed in many different fields of research, such as:

- ***Marketing analytics:*** advertisers and marketers are interested in detecting the influential people communities in Social Networks in the sense that addressing their products' advertisements to such groups would be a more rewarding investment. This can be approached with graph classification applied on these networks (**marketing_analytics**).

- ***Banking security:*** graph classification is used to catch unusual patterns of fraudulent transactions (**banking_security**).

- ***Biology and genomics:*** graphs are based on proteins such that nodes correspond to amino acids which compound the protein and a pair of amino acids are linked by an edge if they are less than 6 Angstroms apart. The task is to detect whether a protein is an enzyme or not (**protein_application**), to mention a few.

1.3 State-of-the-art methods for graph classification

We present here existing algorithms for the graph classification problem and discuss their limitations. In general, these algorithms can be classified in four main categories: set based, frequent sub-graph based, kernel based, and graph neural networks based algorithms.

Set based algorithms. This type of algorithms is only applicable to cases where nodes/edges are supplied with features or attributes, as they completely disregard the graph's structure. Based on the provided feature vectors, a distance function of interest between the graphs is computed. The drawback of this method is that it does not take the structure (topology) of the graph itself into consideration. For example, if we just compare how much the edges' features of one graph are similar to the edges' features of another, we can have two graphs with the same set of edge features, which will lead to maximum similarity, even though their graph structures can be arbitrarily different. On the other hand, a strength of these algorithms is their low computations cost that is usually linear or quadratic in the number

of nodes and edges (**graphlet_kernel**).

Frequent sub-graph based algorithms. These algorithms contain two steps. First, the graph dataset $\mathcal{X}$ is analyzed to enumerate the frequent sub-graphs occuring in the different graphs. Then, another analysis is done to choose the most discriminative sub-graphs out of the ones found during the first step. The disadvantage of using this method is the computational cost that grows exponentially with the graph size (**graphlet_kernel**).

Graph kernels based algorithms. It is a middle ground between both previous methodologies, where the graph structure is well considered, and in most cases, these algorithms are designed in a way that the computational time is a polynomial function of the graph size (**graphlet_kernel**). However, some effective and competitive kernels still require exponential time, and this is in short the problem we approach in this work using random features to approximate these kernels or to compete with them in notably lower computational time.

Graph neural networks (GNNs) based algorithms. GNNs compute a representation vector (embedding vector) for every node in a graph, where this vector is recursively computed by aggregating the representation vectors of neighboring nodes. The goal of this aggregation technique is that nodes that are neighbors (or close) to each other in the graph are more likely to have similar representations (with respect to some similarity function) and vice versa. On the graph level, a representation vector is computed by aggregating its nodes' representation vectors. This aggregated vector now representing the graph itself is used as a usual feature vector which can be fed to a typical deep neural network to learn the classification task. Traditional GNNs such as graph convolutional networks (GCNs) and GraphSAGE fail to provide high performance classifying graphs whose node/edges don't include any original feature vectors, and that even applies on graphs with simple topology (**GCN_powerful**). I could not re-write this last sentence: I don't understand it However,

another GNN structure was developed to overcome this weakness point complete failure is more than a "weakness", and it is referred to by Graph Isomorphism Network (GIN). Regarding the computational time, it is mainly a matter of the layers number in the network, since this parameter in reality represents how far from a node we want to go in order to compute its representation vector.

1.4 Our contribution

One of the methods of the kernel-based algorithms (the third out of the four categories listed), called graphlet kernel, has proven to be competitive for graph classification. Theoretically and empirically, it was shown that a desired performance or a required amount of information to be preserved from the original graph can be reached with sufficiently large k. **hold your horses! we need at least a few sentences actually *explaining* what is the graphlet kernel you are talking about. For instance, we have yet no clue what k is.** However, the computational cost becomes prohibitive as k (the graphlet size) and/or v (the size of the graph) become too large. Thus it cannot be applied on large-scale graph datasets.

The advent of Optical Processing Units (OPUs) opened a new horizon solving this problem, since it can apply enormous number of *Random Projections* in light speed. **hold your horses! The reader has no clue *why* making random projections in light speed is actually useful for your problem. You need at least one sentence explaining what you mean by random projections

In this work, we did the sufficient mathematical analysis to prove that OPUs' light-speed random feature projections compete the k-graphlet kernel with respect to *Maximum Mean Discrepancy (MMD)* Euclidean metric. Moreover, we empirically tested this hypothesis and made sure that the the theoretical MMD error is aligned with the empirical one with respect to the parameters introduced in the problem (sampling technique, number of sampled subgraphs, number of random features, etc). Instead of this last paragraph, I invite you to write

"Our contributions are the following:" followed by an "enumerate" environment to make a clear and thorough list of your achievements. After that enumeration, I invite you to write "On top of these contributions, we have also:" followed by an "enumerate" environment to make a list of things you did that we cannot call "contributions" yet as they either did not work or are work in progress.

Chapter Two

Background

In this chapter we present the necessary background that the reader should have in order to proceed through the next two chapters, which are the core of this work. After a brief overview of kernel methods in machine learning, we will present random features and graph kernels. In the next chapter, these different notions will be combined in the proposed algorithm.

2.1 Kernel methods and random features

We first start by an overview of kernel methods.

2.1.1 Kernel methods and kernel trick

Kernel methods is a family of classic algorithms in machine learning that learn models as a combination of similarity functions between data points (x, x'), defined by positive semi-definite (psd) *kernels* $\mathcal{K}(x, x')$. Denote by $\mathcal{D}$ the set of all possible data points. A symmetric function $\mathcal{K} : \mathcal{D} \times \mathcal{D} \mapsto \mathbb{R}$ is said to be a positive semi-definite kernel if:

$$\forall n \in \mathbb{N},\ \forall \alpha_1, \ldots, \alpha_n \in \mathbb{R},\ \forall x_1, \ldots, x_n \in \mathcal{D},\quad \sum_{i,j}^{n} \alpha_i \alpha_j \mathcal{K}(x_i, x_j) \geq 0\,. \tag{2.1}$$

Can we *not* use mathcal for a function? I suggest κ

Let us now illustrate how kernels can be incorporated to learning models and how that is useful to learn more complex functions. To do that let us consider a simple classification

problem: take $\mathcal{D} = \mathbb{R}^d; d \in \mathbb{N}$, and let $(\mathbf{X}, \mathbf{Y})$ be a labeled dataset of size n with datapoints $\mathbf{X} \in \mathbb{R}^{n \times d}$ and labels $\mathbf{Y} \in \{-1, 1\}^n$. and here X and Y should be mathcal right? Ok, I'll stop here for notation comments as I recall you were saying that you had not changed notations yet. Also I guess you'll need to define the vector $\mathbf{y}$ where y_i is the label of $\mathbf{x}_i$ Many of the learning models designed to solve this problem, like Support Vector Machine (SVM) add ref and Perceptron binary classifier add ref, rely on the inner product as a measure of similarity between data points: during the training, they only use inner products $\mathbf{x}_i^T \mathbf{x}_j$, and then produce models with the following form (**inner_product**):

$$\hat{y}(\mathbf{x}) = \text{sign}\left\{\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}\right\} \text{ with } \alpha_i \in \mathbb{R} \tag{2.2}$$

where the values $[\alpha_i]_{i=1}^n$ in Eq. 2.2 are optimized based on the dataset $(\mathbf{X}, \mathbf{Y})$ by the learning algorithm. The intuition behind Eq. 2.2 is that the output class for every new data point x is expected to be the same class of nearby points in the input set $\mathcal{D}$. This is achieved by introducing the inner product $\mathbf{x}_i^T \mathbf{x}$ to control how much the class y_i contributes in the output $\hat{y}(\mathbf{x})$. The parameters $[\alpha_i]_{i=1}^n$ controls how strongly the data point x_i can affect other neighboring points. They mainly depend on how both classes are distributed in the input set $\mathcal{D}$, and on how much the dataset $(\mathbf{X}, \mathbf{Y})$ is noisy.

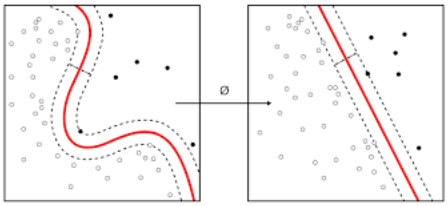

Figure 2.1 The left figure shows a case where the input data in their original space are not separable by a linear boundary. The right figure shows the same data transformed to a new space using a lifting function ø, and we can see that different classes are now separable using linear boundary.

A *kernel method* consists in replacing every inner products $\mathbf{x}_i^T \mathbf{x}_j$ by a psd kernel $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$ during training, and similarly $\mathbf{x}_i^T \mathbf{x}$ by $\mathcal{K}(\mathbf{x}_i, \mathbf{x})$ during prediction. Let us now explain the

intuition behind this, starting by rewriting Eq. 2.2 as

$$\hat{y}(\mathbf{x}) = \text{sign}\{\mathbf{x}^T(\mathbf{X}^T \; diag([\alpha]_{i=1}^n) \; \mathbf{Y})\},$$

where $diag([\alpha]_{i=1}^n)$ is the diagonal matrix with values $[\alpha]_{i=1}^n$. **As $\mathbf{Y}$ is not really well defined, the reader, as it is, does not really understand the algebraic calculus at stake here**

To get the decision boundary of such models we solve $\mathbf{x}^T\mathbf{q} = 0$, where $\mathbf{q} = (\mathbf{X}^T \; diag([\alpha]_{i=1}^n) \; \mathbf{Y}) \in \mathbb{R}^d$. It is the equation of a hyper-plane in the input space $\mathbb{R}^d$, also referred to as a *linear* decision boundary. The question here is: what if the the two classes are not separable by a hyper-plane (Fig. 2.1)?

One common solution to this problem is to map the data points from $\mathbb{R}^d$ to another space $\mathbb{R}^m$ through a proper mapping function ϕ such **NK: Not fond of ø. How about φ ? I** agree: φ, ψ? that the two classes become separable with a linear decision boundary in $\mathbb{R}^m$. Then we can apply the same learning models specified in Eq. 2.2 but on the transformed data. Let us consider for example the dataset shown in Fig. 2.2, we can use the following mapping function ϕ: $(x_1, x_2) \mapsto (\sqrt{2}x_1x_2, x_1^2, x_2^2)$ to move from $\mathbb{R}^2$ on the left, where data are not linearly separable, to $\mathbb{R}^3$ on the right, where they are.

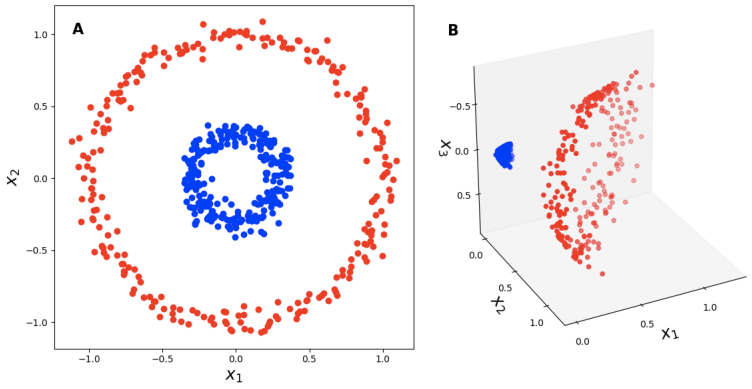

Figure 2.2 Using the mapping function ϕ: $(x_1, x_2) \mapsto (\sqrt{2}x_1x_2, x_1^2, x_2^2)$ to map the data on the left in $\mathbb{R}^2$ to $\mathbb{R}^3$ where they are linearly separable

Learning such a function ϕ is what is typically done by neural networks using complex optimization methods. Kernel methods are much simpler (and elegant) methods to perform this mapping. They are justified by the following key theorem.

Theorem 1 (Mercer theorem)**.** *To every positive semi-definite kernel* $\mathcal{K} : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$*, there exists a Hilbert space* $\mathbb{H}$ *and a feature map* $\phi : \mathbb{R}^d \mapsto \mathbb{H}$ *such that for all* $x, x' \in \mathbb{R}^d$ *we have:*

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') =< \phi(\mathbf{x}), \phi(\mathbf{x}') >_{\mathbb{H}} \tag{2.3}$$

where $< \phi(\mathbf{x}), \phi(\mathbf{x}') >_{\mathbb{H}}$ *is the inner product defined in* $\mathbb{H}$*.*

This theorem states that replacing the inner product $\mathbf{x}_i^T\mathbf{x}$ in Eq. 2.2 by a positive semi-definite kernel $\mathcal{K}(\mathbf{x}_i, \mathbf{x})$ is equivalent to implicitly map the data from the original input space $\mathcal{D}$ to another feature space $\mathbb{H}$ and then apply the classical inner product. Therefore, one *does not need to know explicitely the mapping* ϕ nor the new feature space $\mathbb{H}$, instead, it is sufficient to evaluate the kernel $\mathcal{K}$ for pairs of data points in the original input space $\mathcal{D}$. This main feature of kernel methods is known as the *kernel trick*. It has two main advantages:

- Kernels allow us to transform data to a new Hilbert space of very high or even infinite dimensionality, which can make the learning model able to represent more complex functions.

- Kernels are often computationally cheaper, since they save the time required to compute the explicit co-ordinates of the data in the new feature space by directly calculating the inner product between the transformed data.

To better illustrate these benefits, we take the Gaussian kernel as an example, which is one of the most classical kernels in $\mathbb{R}^d$, defined as:

$$\mathcal{K}_G(\mathbf{x}, \mathbf{x}') = \exp^{-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2}} \tag{2.4}$$

where $\sigma > 0$ is called the bandwidth parameter of the kernel. The lifting function ϕ_G of this kernel is located in a Hilbert space of infinite dimension, but the kernel can be easily evaluated for any pair $(\mathbf{x}, \mathbf{x}') \in \mathcal{D} = \mathbb{R}^d$.

Despite their advantages, kernel methods still have some drawbacks:

- Since for most kernels we need to evaluate the kernel on each pairs of data points, for a dataset $(\mathbf{X}, \mathbf{X})$? of size n, we need $O(n^2)$ memory entries to compute what is called a Gram matrix, whose $(i,j)_{th}$ entry equals the kernel between points $\mathcal{K}(\mathbf{x_i}, \mathbf{x_j})$.

- Even if most kernels are designed so that they can be evaluated in polynomial time in the dimensionality of the input space $\mathcal{D}$ (for instance computing $\mathcal{K}_G(\mathbf{x}, \mathbf{x}')$ for two vectors in $\mathbb{R}^d$ costs d operations), some kernels (especially on graphs) are computationally expensive (**graphlet_kernel**).

To overcome these disadvantages, random feature projections is a technique developed to approximate kernels with less computational time and less memory storage. do RFs really overcome both of these problems?

2.1.2 Random features

Random features (RF) (**rahimi2008random**) is an approach developed to approximate kernel methods with reduced computational time. The idea is that, instead of considering the true lifting function ϕ in Eq. 2.3, we explicitly map the data points using an appropriate randomized feature map $\varphi : \mathcal{D} \to \mathbb{C}^m$, such that the kernel evaluated for two data points x, x' is approximated by the inner product of their random features with high probability:

$$\mathcal{K}(x, x') =< \phi(x), \phi(x') >_{\mathbb{H}} \approx \varphi(x)^* \varphi(x') \tag{2.5}$$

Considering this approximation, we can transform the input with φ and then apply a linear learning method as in Eq. 2.2 to have a similar learning power as the original non-linear kernel machine, while avoiding the $\mathcal{O}(n^2)$ cost of constructing the Gram matrix whatever the embedding, computing a Gram matrix between n elements takes $\mathcal{O}(n^2)$ What are you trying to say?. Note that with RF we do not use the kernel trick anymore, but construct an explicit mapping φ to approximate the kernel $\mathcal{K}$.

Most RF constructions are known as Random *Fourier* Features (RFF), and are based on the following theorem.

Theorem 2 (Bochner's theorem)**.** *A continuous and shift-invariant kernel* $\mathcal{K}(x, x') = \mathcal{K}(x - x')$ *on* $\mathbb{R}^d$ *is positive definite if and only if* $\mathcal{K}$ *is the Fourier transform of a non-negative measure.*

As a direct consequence, we can easily scale any shift-invariant kernel to obtain $\mathcal{K}(0) = \int p = 1$, so that its Fourier transform $p(w)$ is a correct probability distribution. We obtain that any shift-invariant psd kernel is of the form:

$$\mathcal{K}(x - x') = \int_{\mathbb{R}^d} p(w) e^{jw^T(x-x')} dw = E_w[\xi_w(x)^* \xi_w(x')] \tag{2.6}$$

where $\xi_w(x) = e^{-jw^T x}$, where the expectation E_w is over the appropriate probability distribution $p(w)$. Note that, since $\mathcal{K}$ is a real-valued function, from Eq. 2.6 one can also prove that:

$$\mathcal{K}(x - x') = \int_{\mathbb{R}^d} p(w) cos(w^T(x - x')) dw = E_w[\tilde{\xi}_w(x)\tilde{\xi}_w(x')] \tag{2.7}$$

where $\tilde{\xi}_w(x) = \sqrt{2} cos(w^T x + b)$ such that w is drawn from $p(w)$ and b is drawn uniformly from $[0, 2\pi]$, so we can use real-valued mapping if desired.

As a result, for w a random variable drawn from $p(w)$, $\xi_w(x)\xi_w(x')$ is an unbiased estimate of $\mathcal{K}(x, x')$. The RF methodology consists in averaging m instances of the estimator with different random frequencies w_j drawn identically and independently (iid) from $p(w)$, that is, define

$$\varphi(x) = \frac{1}{\sqrt{m}} (\xi_{w_j}(x))_{j=1}^m$$

make it clearer that this is a vector in dimension m. For instance use bold. such that $\varphi(x)^* \varphi(x') = \frac{1}{m} \sum_{j=1}^m \xi_{w_j}(x)^* \xi_{w_j}(x')$, which converges to $\mathcal{K}(x, x')$ by the law of large numbers. Moreover, Hoeffding's inequality guarantees exponentially fast convergence in m between $\varphi(x)^* \varphi(x')$ and the kernel true value:

$$\forall \epsilon > 0 \qquad Pr(|\varphi(x)^* \varphi(x') - \mathcal{K}(x, x')| \geq \epsilon) \leq 2e^{\frac{-m\epsilon^2}{4}}, \tag{2.8}$$

that is, for any error ϵ, the probability that the estimation is off by more than ϵ is controlled by an exponentially decaying term.

I would add here the theorem of the form (the union bound on the previous Hoeffding):

Theorem 3. *Let $\epsilon \in (0,1)$ and $\delta \in (0,1)$. Consider a dataset $\mathcal{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ of n elements, and a psd shift-invariant kernel κ. The random embedding $\varphi(x) \in \mathbb{R}^m$ enables a controlled approximation of* all *the elements of the Gram matrix with probability larger than $1-\delta$,* i.e.

$$Pr\left(\forall(\mathbf{x},\mathbf{x}') \in \mathcal{X}^2 \quad |\varphi(x)^*\varphi(x') - \mathcal{K}(x,x')| \leq \epsilon\right) \geq 1-\delta$$

provided that

$$m \geq \mathcal{O}\left(\frac{1}{\epsilon^2}\log\frac{n}{\delta}\right).$$

I find this version useful as we clearly see that in fact random embedding is classically useful when $\log n \leq d$. If d is too small, random embedding is in general useless.

As an illustration, consider the Gaussian kernel in Eq. 2.4 as an example. This kernel is shift-invariant and known to be positive semi-definite. It is already correctly normalized since $\mathcal{K}(0) = 1$, and its Fourier transform is a Gaussian probability distribution with inverted variance:

$$p(w) = FT\big(\mathcal{K}_G\big)(w) = \left(\frac{\sigma^2}{2\pi}\right)^{\frac{d}{2}} e^{-\frac{\sigma^2\|w\|^2}{2}}$$

Thus, in practice, in order to approximate the Gram matrix of $\mathcal{K}_G$ on a dataset $\mathcal{X}$ of size n, one i/ draws m iid frequencies from this probability distribution, with m as in Theorem 3; ii/ uses these frequencies to associate to each element $x \in \mathcal{X}$ its associated random feature vector $\varphi(x) \in \mathbb{R}^m$; iii/ uses $\varphi(x)^*\varphi(x')$ as an approximation of $\kappa_G(\mathbf{x},\mathbf{x}')$ where necessary in any kernel-based learning algorithms.

2.2 Graph kernels

We first introduce necessary notations related to graph definition and graph kernels.

$F = (\mathcal{V}_F, \mathcal{E}_F)$ is said to be a subgraph (graphlet) of $\mathcal{G}$ and we write $F \sqsubseteq \mathcal{G}$, if and only if there exist an injective function $\mathcal{M} : \mathcal{V}_F \to \mathcal{V}$ such that $(u,u') \in \mathcal{E}_F \Leftrightarrow (\mathcal{M}(u), \mathcal{M}(u')) \in \mathcal{E}$.

Any edge (u_i, u_i) is called a self loop. In a general graph two vertices u_i and u_j may be connected by more than one edge. A simple graph is a graph with no self loops or multiple edges. Here we always consider simple graphs.

A simple graph can equivalently be represented by an adjacency matrix $\mathbf{A}$ of size $v \times v$. The $(i,j) - th$ entry of $\mathbf{A}$ is 1 if an edge (u_i, u_j) exists and zero otherwise.

Two graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ are isomorphic and we write $\mathcal{G}' \cong \mathcal{G}$ if there exists a bijective function $\mathcal{M} : \mathcal{V} \to \mathcal{V}'$ such that $(u_i, u_j) \in \mathcal{E}$ iff $(\mathcal{M}(u_i), \mathcal{M}(u_j)) \in \mathcal{E}'$.

2.2.1 Convolutional graph kernels

Traditional kernel machines approach problems with vector-valued input data, where they compare different data points $(x, x' \in \mathcal{R}^d)$ using the difference between correspondent pairs of vector entries. Based on that, these kernels are imperfect to be used with graphs, since the graph structure is permutation invariant, i.e. isomorphic graphs have different adjacency matrices but they represent the same structure. So in this case distance-kernels between graph representation vectors (adjacency matrices for example) are not a good choice. As a result it is necessary to measure distance between graphs in ways that are permutation invariant as well. Here the concept of isomorphism is critical in learning algorithms on graphs, not only because there is no known polynomial-time algorithm for testing graph isomorphism (except for graphs with specific structures), but isomorphism is also too strict for learning in a similar way to learning with equality operator (**kriege_graph_kernels**). Most of graph kernels developed for graph learning problems are convolution kernels, where given two graphs, the trick is to divide each into smaller subgraphs and then to pairwise compute the kernel between the resulted subgraphs.

Definition 1 (Convolution Kernel)**.** *let $\mathcal{R} = \mathcal{R}_1 \times ... \times \mathcal{R}_d$ denote a space of components such that a composite object $X \in \mathcal{X}$ decomposes into elements of $\mathcal{R}$. Let $R : \mathcal{R} \to \mathcal{X}$ denote the mapping from components to objects, such that $R(x) = X$ iff the components $x \in \mathcal{R}$*

make up the object $X \in \mathcal{X}$*, and let* $R^{-1}(X) = \{x \in \mathcal{R} : R(x) = X\}$*. then, the R-convolution kernel is:*

$$K_{CV}(X, Y) = \sum_{x \in R^{-1}(X)} \sum_{y \in R^{-1}(Y)} \underbrace{\prod_{i=1}^{d} k_i(x_i, y_i)}_{k(x,y)} \tag{2.9}$$

with k_i *is a kernel on* $\mathcal{R}$ *for* $i \in \{1, ..., d\}$*.*

Applying this definition on graphs, $R^{-1}(\mathcal{G} = (\mathcal{V}, \mathcal{E}))$ includes all the components in graph $\mathcal{G}$ that we want to compare with the components $R^{-1}(\mathcal{G}' = (\mathcal{V}', \mathcal{E}'))$ in graph $\mathcal{G}'$. One example of these kernels is the node label kernel, where for two graphs $\mathcal{G}, \mathcal{G}'$, the mapping function R maps the features $x_u \in \mathcal{R}$ of each node $u \in \mathcal{V} \cup \mathcal{V}'$ to the graph that u is a member of. Another example that is mainly related to our work is the k-graphlet kernel, where R here maps the subgraphs of size k to the graph in which it occur. The advantage of using convolution kernel framework with graphs is that kernels are permutation invariant on the graphs level as long as they are permutation invariant on the components level. As a drawback, the sum in Eq. 2.9 iterates over every possible pair of components. As a result, when we choose our components to be more specific such that the kernel value is high between a component and itself while it is low between two different components, each graph becomes drastically similar to itself but distant from any other graph. Thus, a set of weights is usually added so this problem is resolved.

2.2.2 Graphlet Kernel

In general, we have two sets that can be referred to as size-k graphlets. The first set $\mathcal{H} = \{H(1), ..., H(N_k)\}$ contains all graphs of size k and treat isomorphic graphs as different graphs, thus we have here $N_k = 2^{k(k-1)/2}$ different graphlets. The other set $\mathcal{H} = \{H(1), ..., H(\bar{N}_k)\}$ treat all isomorphic graphs of size k as one graph, so we have here $\bar{N}_k < N_k$ but it is still exponential in k. For either set, we define for a graph G the vector $f_G \in \mathcal{R}^{N_k}$, the i-th entry equals the normalized-number of occurrences of $graphlet(i)$ in G

$(\#(graphlet(i) \sqsubseteq G))$. f_G is usually referred to by the k-spectrum of G, and this vector is the key idea behind graphlet kernel.

Definition 2 (Graphlet Kernel)**.** *Given two graphs $\mathcal{G}$ and $\mathcal{G}'$ of size $v_{\mathcal{G}}, v_{\mathcal{G}'} \geq k$, the graphlet kernel $\mathcal{K}_g$ is defined as (***graphlet_kernel***):*

$$\mathcal{K}_g(G, H) = f_{\mathcal{G}}^T f_{\mathcal{G}'}. \tag{2.10}$$

Which naturally involves an associated Euclidean metric $d_{\mathcal{K}}(\mathcal{G}, \mathcal{G}') = \|f_{\mathcal{G}} - f_{\mathcal{G}'}\|_2$. The drawback of this kernel is that computing the k-spectrum vector costs huge computational time, where regardless the cost of isomorphism test if it is required, there are $O\binom{v}{k}$ subgraphs of size k in a graph $\mathcal{G}'$ of size v. As a result, there is a trade off between a more accurate representation of the graph (larger value of k) and the computational cost. However, some techniques are used in order to resolve this limitation as sampling from graph technique (section 2.2.3).

2.2.3 Graph sampling to approximate k-graphlet spectrum

The problem of graph sampling arises when we deal with a large-scale graph and the task is to pick a small-size sample subgraphs that would be similar to the original graph with respect to some important properties.

Sampling from graph techniques are used to resolve the processing cost limitation of graphlet kernel, and it can be done by directly sample s sugraphs $\{F_1, ..., F_s\}$ of size k, and then estimate the k-spectrum vector empirically: $f_{\mathcal{G}}(i) = \frac{1}{s}\Sigma_{j=1}^{s}\mathbb{1}[f_j = H_i]$

Uniform sampling Random walk sampling

to be continued . . .