Computer Science Review 27 (2018) 88–111

Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Review article

The journey of graph kernels through two decades

Swarnendu Ghosh [a], Nibaran Das [a,*], Teresa Gonçalves [b], Paulo Quaresma [b], Mahantapas Kundu [a]

[a] Department of Computer Science, Jadavpur University, India
[b] Department of Informatics, University of Evora, Portugal

ARTICLE INFO

Article history:
Received 11 July 2017
Received in revised form 28 November 2017
Accepted 29 November 2017

Keywords:
Graph kernels
Support vector machines
Graph similarity
Isomorphism

ABSTRACT

In the real world all events are connected. There is a hidden network of dependencies that governs behavior of natural processes. Without much argument it can be said that, of all the known data-structures, graphs are naturally suitable to model such information. But to learn to use graph data structure is a tedious job as most operations on graphs are computationally expensive, so exploring fast machine learning techniques for graph data has been an active area of research and a family of algorithms called kernel based approaches has been famous among researchers of the machine learning domain. With the help of support vector machines, kernel based methods work very well for learning with Gaussian processes. In this survey we will explore various kernels that operate on graph representations. Starting from the basics of kernel based learning we will travel through the history of graph kernels from its first appearance to discussion of current state of the art techniques in practice.

Contents

* Corresponding author.
E-mail addresses: swarbir@gmail.com (S. Ghosh), nibaran.das@jadavpuruniversity.in (N. Das), tcg@di.uevora.pt (T. Gonçalves), pq@di.uevora.pt (P. Quaresma), mahantapas.kundu@jadavpuruniversity.in (M. Kundu).

https://doi.org/10.1016/j.cosrev.2017.11.002
1574-0137/

1. Introduction

Information has always been in the primary focus of researchers in the field of computer science. In our world, most of the available information is represented as networks of meaningfully connected data elements. These connections can signify some sort of interdependence or portray some contextual significance. This relational aspect of information is one of the main challenges for researchers. In this survey will be explored the utility of various graph kernels in this domain of relational information, but before we move on to the details of graph kernels, let us first understand the importance of "graphs" and "kernels" in the field of artificial intelligence. One of the primary tasks is sensible representation of such relational data, so that they could be used to perform machine learning tasks such as classifications, sequence predictions, density estimations and so on.

Information is mainly stored using data structures for computers to process them. While there are many data structures available, the most generic format is a graph. All other data structures are simply some sort of specializations of a graph. As we know, graphs are characterized by their network of nodes connected by edges. Similarly, natural information in general can be broken down to smaller elements that can have some sort of semantic connection hence, this property of graph makes it most suitable for representing relational information. So, the first step of graph based learning is to actually represent the information in the form of a graph. Once that is done the second step is the learning part.

The most straightforward technique for learning is to extract meaningful features from a sample that uniquely predicts its nature. However, that is not always feasible given the dynamic nature of real world problems. Problems can be so complicated that manually extracting features can be really hectic and sometimes humanely impossible. Data in its raw form is not suitable for computational operations. A consistent input space is needed to represent the data in its actual form. The key idea behind finding features is to move the sample from the input space to another dimension where similar samples will be mapped in close proximity while distance between dissimilar samples will be significantly higher. Another branch of machine learning, namely kernel based learning, views the problem from a different perspective. If we can find some metric to map this similarity between samples we can directly map them onto the feature dimension without actually having to learn the features themselves. Another way to explain this is to approximate the nature of the probability distribution of the real world process, also know as the Gaussian process, so that the similar samples stay in close proximity and vice versa. This new dimension is also called an Hilbert space. The entire goal of kernel based learning is to map the available sample space into a suitable Hilbert space. Once we know the Gaussian distribution, also termed as the posterior, it will be much easier to calculate the similarity among samples. Machine learning dived into a new paradigm through the introduction of a special function referred to as a kernel function which can directly map the input space to such feature dimensions. Throughout the next chapters, we will look into details regarding definitions, mathematical concepts and old and modern research works surrounding the application of kernels to the field of graph theory.

As we finish the introductory section we will find our motivation to study more about this domain in next section. Section 3 introduces us to the preliminary concepts of some Gaussian Processes, Kernel based Machine Learning, and Graph Theory. This is absolutely necessary for understanding the concepts of various graph kernels. As we move on to the fourth section, we will discuss the core concepts of graph kernels, starting from the earliest point in the history of research where the first idea of structural kernels was conceived and slowly moving through time to finally analyze a couple of state of the art technologies. Utmost effort has been made to keep all explanation as simple as possible while maintaining enough mathematical formulation to ensure logical clarity.

2. Motivation

Graphs provide one the most generic data structures for representing information. Philosophically speaking a graph represents a network of relationships among objects. All real world phenomena can be interpreted as a system with various components that work in tandem. These relations and interdependence connect these components to form a complex network. Another interpretation may be all real world objects or events can either be described as a network or can be considered to be a part of a larger network. Philosophical arguments have been made in favor of graphs as the most ideal data structure to represent the world in the language of mathematics [1].

In computational terms it has already been mentioned that graph are the most generic form of data structure as all common datatypes can simply be referred to as an instance of a graph. For example, a scalar or a constant can be treated as single node graph, and array or matrix can be seen as a graph where each nodes represent an index in the array and their adjacency is represented by an edge. Stacks and queues have similar structure but with limitation of insertion and deletion property of the nodes. A time series can be modeled by representing time stamps as nodes and connecting each stamp with an edge to the next one.

So, with all this said, the real question is why graphs are not being used as the most common data structure for decades? The simple answer is that handling graphs is complicated. On one hand

graphs provide a lot of flexibility to represent complex data in an efficient way but, the same flexibility stands in the way when computational operations are performed. Normal vectors can be easily represented in a co-ordinate space, hence allowing simple metric like euclidean distance to serve as an excellent choice for vector comparison. However, it is much more difficult to represent a graph in an n-dimensional space hence the difficulty of comparing them. The straightforward or brute force method would be to identify the common parts in both graphs. For this purpose we must find all sub-graphs of the graph. A graph with n nodes will always have 2^n possible sub-graphs. Hence the problem shifts to an exponential search space. As aptly stated by Horst Bunke [2]:

> *"computing the distances of a pair of objects[...] is linear in the number of data items in the case of feature vectors, quadratic in case of strings, and exponential for graphs"*

Hence, to overcome this curse of exponential time complexity, researchers have avoided graph based machine learning for long time before the introduction of stronger computational resources in the last couple of decades. Gradually, analysis revealed that these problems need crucial attention for the sake of progress of research in this field [3].

Most machine learning problems can be reduced to a "simple" task of comparing samples to find their similarity. Specifically, support vector machine is an excellent machine learning technique that make use of this property. The famous kernel trick involves mapping the data samples to a different dimensional space, known as the Hilbert Space, using a kernel function. In the Hilbert Space, closeness of samples represent their similarity. Hence, when we talk about graph based machine learning, our main focus is to be able to compare graphs efficiently. In the last 2 decades lots of contributions have been made with respect to graph based kernels [4–7]. Most graph kernel algorithms are built around a graph comparison technique. For the sake of further contribution and to ignite the interest on graph kernels among a larger group of researchers it is essential to do a collective study of all major contributions to understand the dynamics of graph kernels.

3. Preliminary concepts

As we progress in our journey of rediscovering the domain of graph kernels we must equip ourselves with the proper tools and techniques to ensure proper and clear understanding of the core concepts. This refresher section is divided into three main subsections, namely, Gaussian processes and Covariance function, Kernels and Support Vector Machines, and Graph Theory.

3.1. Gaussian process and co-variance functions

The concept of kernel deals with expressing samples in an alternate feature space where they exhibit some properties which allow similar samples to remain closer. Most processes in the real-world deal with random variables that govern the outcome of the process. In real-world random variables that trigger events are not always completely random but in fact exhibits a probabilistic nature or can be expressed through a probabilistic distribution. Hence the output of the process also exhibits probabilistic properties. We will see one of the most common kind of process called Gaussian process. This Gaussian process can be expressed in terms of a covariance function which computes the relation between different components of the input dimensions. Furthermore, we will see the positive semi definite property for covariance functions which is an essential property to define a kernel. Finally we will associate these kernels to a Hilbert space which is the alternate feature space that we need.

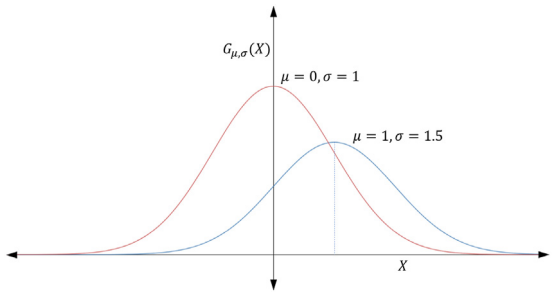

Fig. 1. An uni-dimensional Gaussian process can be represented by a normal distribution. The gray distribution is the standard normal curve with $\mu = 0$ and $\sigma = 1$ and the blue distribution has $\mu = 1$ and $\sigma = 1.5$.

3.1.1. Gaussian process

A Gaussian (or normal) distribution is one of the most commonly used continuous probability distributions in the world of statistics and computer science. This kind of distribution is useful to represent a random variable (X) whose value is centered around a mean value (μ) and which can deviate from the mean by a magnitude depending on the standard deviation (σ). The probability distribution function $f(x)$ for a given value of μ and σ is given by:

$$f(X|\mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}}. \tag{1}$$

Fig. 1 shows how μ and σ affects the Gaussian distribution.

For conceptual clarity we may visualize a single dimensional Gaussian process as a set of observations that depend on a single variable. The expectation of these observations are said to exhibit the nature of a Gaussian distribution. We may notice that we have only talked about a Gaussian distribution that depends only on one random variable (X). Such a distribution is called univariate Gaussian distribution. When we take into consideration more than one random variable we get a multivariate Gaussian distribution which depends on a random vector $(X_1, X_2, \ldots, X_k)$. Any linear combination of these random variables would give us an multi-variate Gaussian distribution. The multivariate Gaussian distribution can be represented in terms of a linear combination of uni-variate distributions as shown in Fig. 2. In this case the probability density function $f_x(x_1, \ldots, x_k)$ for a k-variate Gaussian is given by:

$$f_x(x_1, \ldots, x_k) = \frac{exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu))}{\sqrt{(2\pi)^k |\Sigma|}} \tag{2}$$

where $(x_1, \ldots, x_k)$ is the random vector and μ is a k-dimensional mean vector and Σ is a $k \times k$ covariance matrix given by:

$$\mu = [E[X_1], E[X_2], \ldots, E[X_k]] \tag{3}$$

$$\Sigma = [Cov[X_i, X_j]], i = 1, 2, \ldots, k; j = 1, 2, \ldots, k. \tag{4}$$

Hence the Gaussian process can be expressed in terms of a mean function m and a covariance function K written as $GP(m.K)$, where, $m(i) = E[X_i]$ and $K(i, j) = Cov[X_i, X_j]$. In the next section we will see some properties of covariance functions and examples.

Now if we consider a process which depends on multiple random variables we may be able to define a Gaussian process. Instances of a Gaussian process occur in a continuous domain like time, space etc. Each continuous space shall exhibit properties of a Gaussian distribution and each point will be associated with a random variable. A finite set of such random variables has a multivariate Gaussian distribution. The distribution of the Gaussian process is defined by joint distribution of all those random variables.

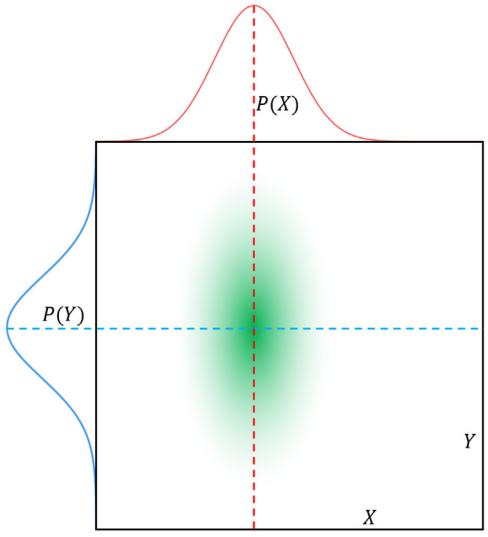

Fig. 2. A multivariate distribution (bi-variate in this case).

Definition 1 (*Gaussian Process*)**.** A Gaussian process is a statistical distribution $X_t, t \in T$, for which any finite linear combination of samples has a joint Gaussian distribution.

Alternatively, time continuous stochastic process is Gaussian if and only if for every finite set of indices $t_1, \ldots, t_k$ in the index set T, $X_t = (X(t_1), X(t_2), \ldots, X(t_k))$ is a multivariate Gaussian Distribution.

Gaussian processes are used in various statistical models. If we can model a random process as a Gaussian process the distributions of various derived quantities can be obtained explicitly. For example, if we have two Gaussian distributions, we can derive the product of these distributions and find the result to be another Gaussian distribution. This can be performed by using simple linear Algebra. Another benefit is that a linear projection of any multivariate Gaussian distribution will give us an uni-variate Gaussian distribution. Hence it is very easy to find the marginal distributions for specific random variables.

In machine learning applications a Gaussian process can use lazy learning to measure similarity between points (the kernel function) from the training data to predict the value for new unseen instance or a test sample. Hence Gaussian processes have a direct connection with machine learning techniques.

3.1.2. Covariance functions

In the previous section, we talked about how a Gaussian distribution can be expressed in terms of a mean function and a covariance matrix. As expressed in (2), the covariance matrix can be found using a covariance function $Cov(x, x')$. A covariance function is crucial in the field of Gaussian process prediction, as it is a latent representation of the function we wish to predict.

Definition 2 (*Covariance Function*)**.** For a random field or stochastic process $Z(x)$ on a domain D, a covariance function $C(x, y)$ gives the covariance of the values of the random field at the two locations x and y.

$$C(x, y) = cov(Z(x), Z(y)). \tag{5}$$

From a machine learning viewpoint and for a supervised learning approach, the concept of similarity between data points is crucial. A basic assumption is that two data points x and y are similar if they are closer to each other in the vector space. Thus, training points which are near to a test point should be informative about its properties. In a Gaussian process covariance function defines the proximity or similarity between data points. A general name for a function k of two arguments mapping a pair of inputs $x, x' \in X \to \mathbb{R}$ is a kernel.

Table 1
Examples of a few commonly used co-variance functions.

Type of covariance function	Formula
Linear	$k_L(x, x') = x^T x'$
Quadratic	$k_Q(x, x') = (x^T x' + c)^2$
Brownian	$k_B(x, x') = min(x, x')$
Squared exponential	$k_{SE}(x, x') = exp\left(-\frac{\|x-x'\|^2}{2l^2}\right)$
Ornstein–Uhlenbeck	$k_{OU}(x, x') = exp\left(-\frac{\|x-x'\|}{l}\right)$
Periodic	$k_P(x, x') = exp\left(-\frac{2sin^2\left(\frac{x-x'}{2}\right)}{l^2}\right)$

Covariance functions may be identified with various properties. A stationary covariance function is a function of $(x - x')$. Thus, it is invariant to translations in the input space. Furthermore, if a covariance function is a function of only $|x-x'|$, it is called isotropic covariance function. Isotropic covariance functions are invariant to all kinds of rigid motion. They are also known as radial basis functions (RBFs). There is another generic form of representation of a covariance function through the dot product $x.x'$. Such functions, which are called dot product covariance functions, are invariant to rotation of the coordinates about the origin, but not translations. Table 1 shows a few examples of commonly used covariance functions.

Definition 3 (*Covariance Matrix*)**.** Given a set of input points $(x_i | i = 1, \ldots, n)$ we can compute the gram matrix K whose entries are $K_{ij} = k(x_i, x_j)$. If k is a covariance function we call the matrix K the covariance matrix.

Definition 4 (*Positive Semidefinite*)**.** A real $n \times n$ matrix K which satisfies $Q(v) = v^T K v \geq 0$ for all vectors $v \in \mathbb{R}$, is called positive semidefinite (PSD).

If $Q(v) = 0$ only when $v = 0$ the matrix is positive definite. $Q(v)$ is called a quadratic form. A symmetric matrix is PSD if and only if all of its eigenvalues are non-negative. A Gram matrix corresponding to a general kernel function need not be PSD, but the Gram matrix corresponding to a covariance function is PSD. We will talk more about positive semidefinite in the next section.

Different covariance functions can generate different kinds of Gaussian surfaces. The machine learning point of view to use this covariance function expressibility is to generate Gaussian functions that fit the train data well. We can see some sample Gaussian surfaces in Fig. 3. In that case information about test data can be achieved by looking at the neighborhood of the data point in the vector space with assumption that it is supposed to exhibit properties similar to nearby training data.

3.1.3. Gaussian process regression

Gaussian process regression can be considered an interpolation method such that interpolated samples from Gaussian processes are constrained by prior covariances as per the training data points. The Gaussian process itself can be explained through different perspectives; the simplest representation may be as an infinite dimensional Gaussian random variable with a specified co-variance structure. While this process demonstrates its properties it is not useful for practical models. A weight-space view represents Gaussian processes as weighted averages of training target values. Another interpretation views Gaussian processes as distribution over functions: finite dimensional Gaussian processes are distributions

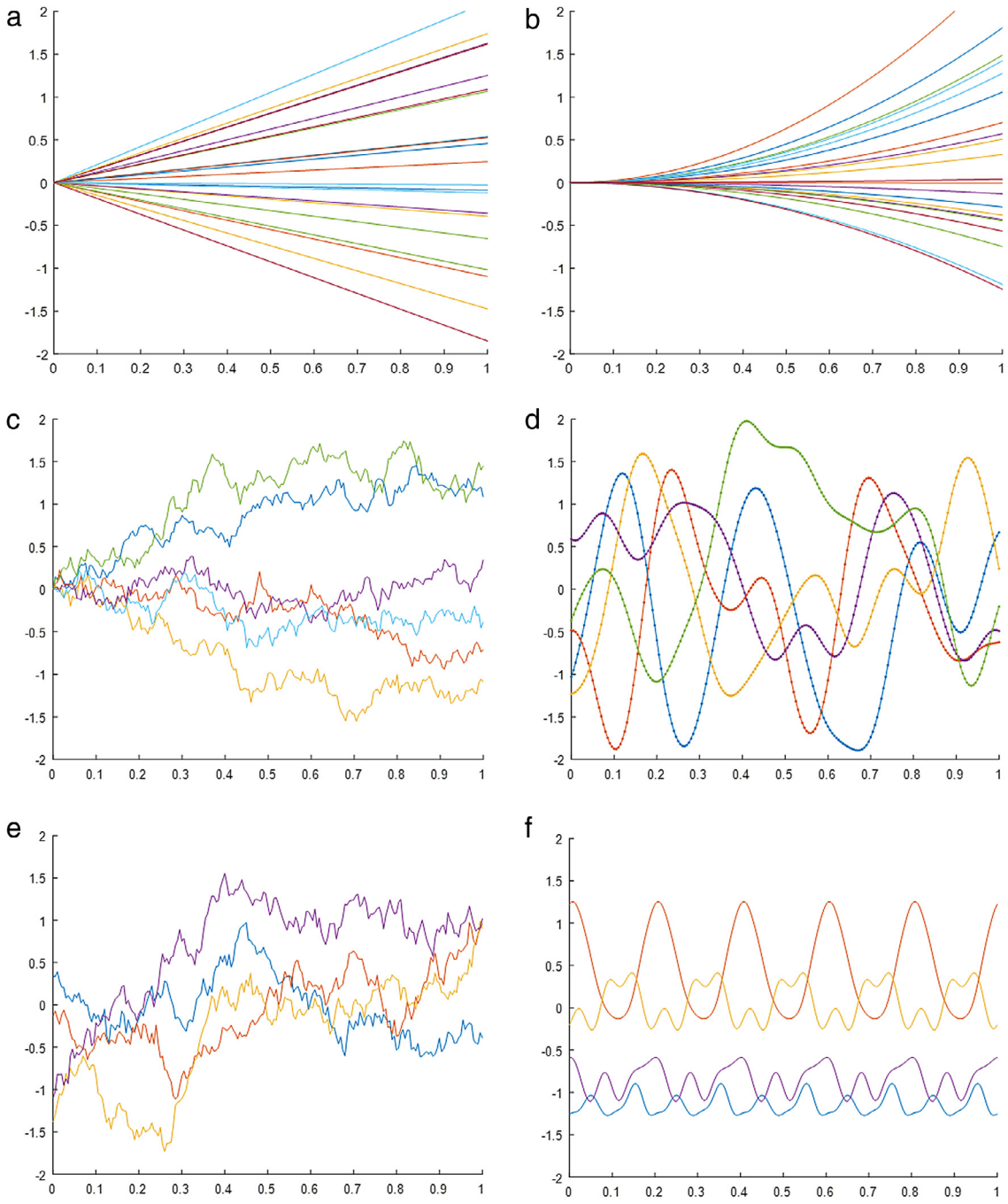

Fig. 3. Gaussian Processes generated using some covariance functions: (a) Linear (b) Polynomial, (c) Brownian, (d) Squared Exponential, (e) Ornstein–Uhlenbeck, (f) Periodic covariance functions or kernels.

over finite dimensional vectors, while infinite dimensional Gaussian Processes are distributions over infinite dimensional vectors or functions.

A Gaussian process prior offers unconstrained samples. Data observations force the Gaussian process prior to pass through these points and hence result in a Gaussian process posterior. Samples generated from this Gaussian process posterior can be used to predict properties of unseen samples with a certain degree of uncertainty as demonstrated in Fig. 4.

3.2. Kernels and support vector machines

For researchers more interested in the machine learning aspects it is more important to focus on the co-variance functions or the kernel functions. We have seen in the previous section that kernels corresponding to the covariance are positive semi-definite. From another angle, we can say that if we create a similarity function for the training data points such that the similarity matrix is positive semi-definite, it can be used as a kernel function. In this section, we define the properties of positive semi-definite matrices. Let C be a symmetric matrix such that $C(i, j) = K(x_i, x_j)$,where $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is a symmetric function or similarity function. K can be proved to be a positive semidefinite kernel on $\mathcal{X}$ by satisfying any of the following criteria:

1. iff $\forall i,\ \lambda_i \geq 0$, where λ_i is the ith eigenvalue of the matrix C
2. iff $u^T C u \geq 0, \forall u \in \mathbb{R}^n$

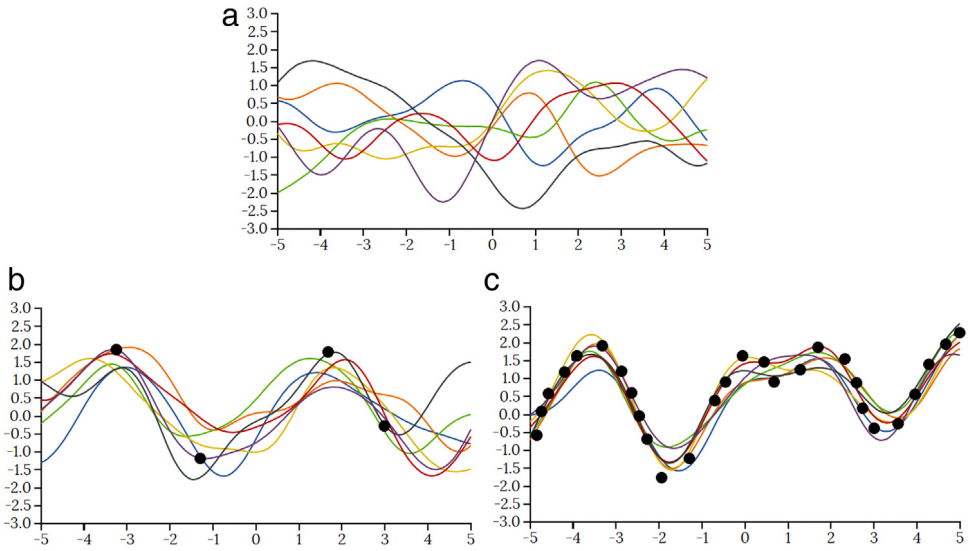

Fig. 4. Gaussian process regression. (a) The prior distributions generated by squared exponential kernels. (b) Posterior distribution after introduction of data samples, (c) With more number of data samples the degree of uncertainty decreases.

3. $\sum_{i,j=1}^{n} c_i c_j K(x_i, x_j) \geq 0$ for any $n \in \mathbb{N}$; $x_1 \ldots x_n \in \mathcal{X}$; $c_1, \ldots, c_n \in \mathbb{R}$.

Another way of checking if a symmetric matrix C is positive semidefinite is to check whether we can factorize C into AA^T. In that case,

$$u^T C u = u^T A A^T u = (A^T u)^T (A^T u) = \|A^T u\| 2 \geq 0. \tag{6}$$

Hence C would be positive semidefinite.

3.2.1. Inner products and PSD kernels

An inner product is, in some sense, a canonical form of positive semidefinite kernel. Inner products can be seen as a generalized version of the dot product which is mapped to a more general form of vector space called Hilbert Space. The mathematical concept of a Hilbert space, named after David Hilbert, generalizes the notion of Euclidean space. It extends the methods of vector algebra and calculus from the two-dimensional Euclidean plane and three-dimensional space to spaces with any finite or infinite number of dimensions.

Definition 5 (*Hilbert Spaces*)**.** A Hilbert space is an abstract vector space of infinite dimensions possessing the structure of an inner product that allows length and angle to be measured.

A dot product kernel for a d-dimensional vectors can be written as $k(x, x') = x^T x'$, where $x, x' \in S = \mathbb{R}^d$. A further generalization of the dot product kernel can be taken in terms of a mapping function $\phi : S \to \mathbb{R}^d$, which can be written as $k(x, x') = \phi(x)^T \phi(x')$. Finally if we consider the function $\phi : S \to \mathcal{H}$, to map the values of x and x' in a Hilbert Space $\mathcal{H}$, we can rewrite the kernel function as $k(x, x') = (\phi(x), \phi(x'))$, where the output given by the kernel function is simply a inner product. The Hilbert space associated with a kernel is referred to as a Reproducing Kernel Hilbert Space (RKHS) [8]. It can be shown, by means of functional analysis, that every kernel function is associated with a RKHS and that every RKHS is associated with a kernel function.

3.2.2. Examples of positive semidefinite kernels

Some examples of positive semidefinite kernels are:

- Linear Kernel. $K_L(x, x') = x^T x'$, where $x, x' \in \mathbb{R}^d$.
- Polynomial Kernel. $K_P(x, x') = (x^T x' + r)^n$, *where* $x, x' \in \mathbb{R}^d$, $r > 0$.
- Gaussian Kernel (RBF Kernel). $K_{RBF}(x, x') = e^{-\frac{\|x-x'\|^2}{2\sigma^2}}$, *where* $x, x' \in \mathbb{R}^d$, $\sigma > 0$.
- Laplacian Kernel. $K_{Lap}(x, x') = e^{-\alpha \|x-x'\|}$, *where* $x, x' \in \mathbb{R}^d$, $\alpha > 0$.
- Abel Kernel. $K_{Abel}(x, x') = e^{-\alpha |x-x'|}$, *where* $x, x' \in \mathbb{R}^d$, $\alpha > 0$.

3.2.3. Kernels in support vector machines

In the previous section we studied the use of Kernel Functions to compute kernel matrices. However, to understand how we can use these kernel matrices for machine learning, we must uncover the concepts of support vector machines. We will explore the early designs of kernels from the basics of the Support Vector Machine to the underlying principle of a big family of learning techniques. An in-depth study on topic was performed by [9], and the references therein.

Traditionally, Support Vector Machines (SVMs) address a binary classification problem, although Multiclass-SVMs have come to practice over consequent years [10]. The problem can be summarized as, given a set of training objects along with class labels $\{x_i, y_i\}_{i=1}^m$ with $x_i \in X = \mathbb{R}^d$, $d \in N$, $y_i \in Y = \{\pm 1\}$, we need to train a classifier $f : X \to Y$ that predicts the labels of unseen data samples.

3.2.4. Maximizing the margin

The most common technique to solve this problem is to use Large margin methods. It involves introducing a hyperplane between class $y = 1$ and class $y = -1$. If we imagine a straight line in 2-Dimensional space or 2-D Plane in 3-dimensional space we can use them to separate the corresponding co-ordinate space into two regions as shown in Fig. 5.

An hyperplane is an equivalent of that in an n-dimensional space. Depending on the position of x_i with respect to the hyperplane, y_i is predicted to be 1 or -1. If we assume that such a hyperplane exists which correctly separates both classes, then infinite such hyperplanes exist, which may be parameterized by (w, b) with $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ and can be written as $\langle w, x \rangle + b = 0$, where $\langle w, x \rangle$ denotes the dot product between vectors w and x. These hyperplanes satisfy

$$y_i(\langle w, x_i \rangle + b > 0), \forall i \in \{1, 2, \ldots, m\} \tag{7}$$

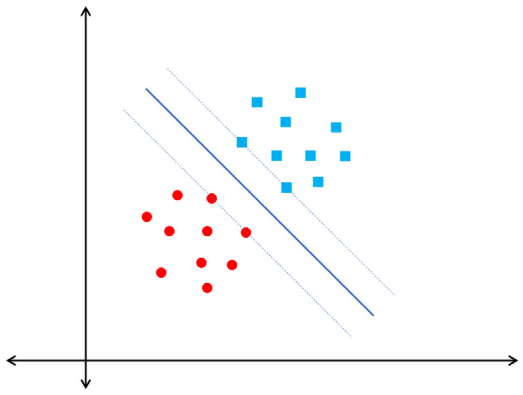

Fig. 5. Maximizing the margin.

and these hyperplanes correspond to decision functions represented by

$$f(x) = sgn(\langle w, x_i \rangle + b) \tag{8}$$

where $f(x)$ is the (predicted) class label for data sample x. Among these hyperplanes, an optimal hyperplane may be chosen that maximizes the margin i.e., the minimum distance between the nearest data points from both classes and the hyperplane itself [11].

3.2.5. Moving the problem to feature space or the kernel trick

Even with soft-margin classifiers we cannot solve all kinds of classification problem. Let us take into account the following 2-D example where all the positive data points lie within a circle and all negative data points forms a concentric circle outside it. It is impossible to find a linear hyperplane which correctly classifies them. In other terms, as long as the input data points are mapped like this, we cannot draw an hyperplane. Hence, the obvious step would be to map these input points in way that they are linearly separable. The trick to overcome this sort of problem is to map the input points into an alternate dimension $\mathcal{H}$, generally higher than the dimension of input space. The basic idea is to find some non-linear mapping $\phi : \mathbb{R}^d \rightarrow \mathcal{H}$ such that, in the Hilbert Space $\mathcal{H}$, we can use the SVM formulation as mentioned before, simply by replacing $\langle x_i, x_j \rangle$ with $\langle \phi(x_i), \phi(x_j) \rangle$. If we define a kernel function k as

$$k(x, x') = \phi(x_i), \phi(x_j) \tag{9}$$

we obtain decision functions in the form of

$$\begin{aligned} f(x) &= sgn(\int_{i=1}^{m} y_i \alpha_i . \phi(x_i), \phi(x_j). + b) \\ &= sgn(\int_{i=1}^{m} y_i \alpha_i k(x, x') + b) \end{aligned} \tag{10}$$

and the following quadratic problem (for the hard-margin case)

$$\begin{aligned} &\underset{\alpha \in \mathbb{R}^m}{maximize} \quad W(\alpha) = \int_{i=1}^{m} \alpha_i - \frac{1}{2} \int_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ &subject\ to \quad \alpha_i \geq 0,\ \forall i = 1, \ldots, m,\ and \int_{i=1}^{m} \alpha_i y_i = 0. \end{aligned} \tag{11}$$

This means that we simply move our classification problem into a higher dimensional space $\mathcal{H}$ and solve it even without explicitly computing the mapping φ to $\mathcal{H}$. This is commonly known as the famous kernel trick which is demonstrated in Fig. 6.

3.2.6. Practical implementations

In practical implementations we often find ourselves with a training sample along with its features and labels. In general cases these features fail to map the data sample in a manner that is linearly separable, so we make use of a kernel function $k(x, x')$ that computes the kernel matrix $\mathcal{K}_{Train}$. So, if we have M training examples, we get a $M \times M$ kernel matrix, where $\mathcal{K}_{Train}(i, j) = k(i, j)$ for training samples i and j; each training sample is then mapped to a new M-dimensional space and unseen test instances once mapped on to same space can be classified. The key to good performance is obviously the choice of the kernel function $k(x, x')$, such that it is able to create a mapping that is linearly separable in the Hilbert Space or the M-dimensional space.

3.3. Graph theory

To understand the importance of graph kernels we first need to be clear about basic graph concepts. The following sections deals with various nomenclatures associated with graph based learning and also show some of the most commonly used graph comparison techniques.

3.3.1. Graph nomenclature

In its most general form, a graph is a set of nodes connected by edges. In general, $V(G)$ refers to the set of nodes of graph G, and $E(G)$ refers to the edges of graph$G$.

Definition 6 (*Graph*)**.** A graph is a pair $G = (V, E)$ of sets of nodes (or vertices) V and edges E, where each edge connects a pair of nodes, i.e., $E \subseteq V \times V$.

Directed, Undirected and Labeled Graphs. Graphs can have many variations depending on properties of nodes and edges.

Definition 7 (*Labeled Graph*)**.** A labeled graph is a triple $G(V, E, \mathcal{L})$ where (V, E) is a graph, and $\mathcal{L} : V \cup E \rightarrow Z$ is a mapping from the set of nodes V and edges E to the set of node and edge labels Z.

A graph with labels on its nodes is called node-labeled, whereas a graph with labels on edges is called edge-labeled. Sometimes labels and labeled graph are also referred to as attributes and attributed graphs, respectively. Depending on whether we assign directions to edges, the resulting graph is directed or undirected.

Definition 8 (*Directed and Undirected Graph*)**.** Given a graph $G = (V, E)$, if we assign directions to edges such that edge $(v_i, v_j) \neq edge(v_j, v_i)$ for $v_i, v_j \in V$, then G is called a directed graph. G is an undirected graph if $\forall v_i, v_j \in V, (v_i, v_j) = edge(v_j, v_i)$.

The size of the graph is generally defined by the number of nodes. It is generally written as $|V|$ or $|V(G)|$; if $|V|$ is finite, then the graph is finite. Graph G is said to be larger than G' if $|V(G)| > |V(G')|$ and vice-versa. $|E|$ or $|E(G)|$ denotes the number of edges.

Neighborhood of a Graph. In a graph G, two nodes v_i and v_j are adjacent if $(v_i, v_j) \in E$; two edges $e_i \neq e_j$ are adjacent if they share a common node. If all the nodes of G are pairwise adjacent, then G is complete.

Definition 9 (*Adjacency Matrix*)**.** The adjacency matrix $A = (A_{ij})_{n \times n}$ of graph $G = (V, E)$ is defined by

$$A_{ij} := \begin{cases} 1 & if (v_i, v_j) \in E, \\ 0 & otherwise \end{cases} \tag{12}$$

where v_i and v_j are nodes from G.

The number of neighbors of a node is defined by its degree.

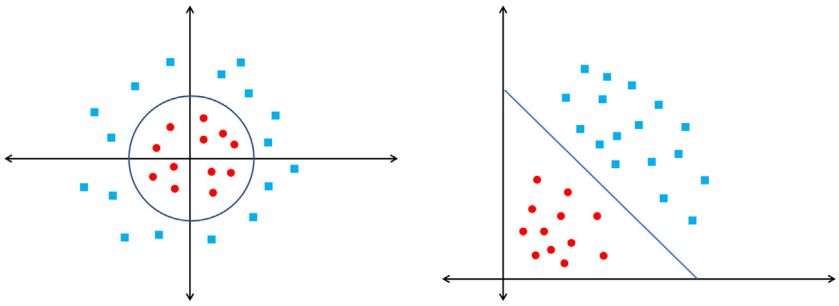

Fig. 6. Toy example illustrating kernel trick: Mapping a circle into feature space: data point distribution in input space (Left) and feature space (Right).

Definition 10 (*Degree of Node*)**.** The degree $d_G(v_i)$ of a node v_i in $G = (V, E)$ is the number of edges at v_i:

$$d_G(v_i) := |v_j : (v_i, v_j) \in E. \tag{13}$$

For an undirected graph, the degree of a node i is equal to the number of neighbors of v_i, where $\delta(vi) := \{v_j|(v_i, v_j) \in E\}$ is the set of neighbors of node v_i. A node without neighbors is isolated from the other part of the graph. The number $\Delta_{min}(G) := \min\{d_G(v)|v \in V\}$ denotes the minimum degree of G, whereas the number $\Delta_{max}(G) := \max\{d_G(v)|v \in V\}$ denotes its maximum degree. If all the nodes of G have the same degree k, then G is k-regular, or simply regular.

$$d_G(G) := \frac{1}{|V|}\int_{v\in V} d_G(v). \tag{14}$$

Pairs of nodes which are not adjacent are called independent. A self-loop is an edge (v, v) with two identical ends. A graph contains multiple edges if there may be more than one edge between two nodes v_i and v_j .

Definition 11 (*Walk, Path and Cycle*)**.** A walk w (of length $k - 1$) in a graph G is a nonempty alternating sequence $(v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k)$ of nodes and edges in G such that $e_i = \{v_i, v_{i+1}\}$ for all $1 \leq i \leq k - 1$. If $v_1 = v_k$, the walk is closed. If the nodes in w are all distinct, it defines a path p in G, denoted $(v_1, v_2, \ldots, v_k)$. If $v_1 = v_k$ then p is a cycle.

A *Hamilton path* is a path that visits every node in a graph exactly once. An *Euler path* is a path that visits every edge in a graph exactly once. A graph G is called connected if any two of its nodes are linked by a path in G, otherwise G is referred to as 'not connected' or 'disconnected'.

Definition 12 (*Edge Walk and Edge Path*)**.** Given a graph $G = (V, E)$ with $\{e_1, \ldots, e_l\} \subset E$ and $\{v_{i_1}, v_{i_2}, v_{j_1}, v_{j_2}\} \subset V$, an edge walk $w = (e_1, e_2, \ldots, e_l)$ is defined as a sequence of edges e_1 to e_l where e_i with $1 \leq i \leq l$ is a neighbor of $e_{i+1} = e_j$, i.e. $e_i = (v_{i_1}, v_{i_2})$ and $e_j = (v_{j_1}, v_{j_2})$ are neighbors if $v_{i_2} = v_{j_1}$. An edge path p is defined as an edge walk without repetitions of the same edge.

Graph Isomorphism and Subgraph Isomorphism. For graph based machine learning it is often necessary to find the similarity between two graphs. For this purpose, the use of graph isomorphism or subgraph isomorphism concepts are used. It is not always possible to check directly from their adjacency matrix if two graphs are similar as this check is proportional to the order of the vertices of the graphs. Hence, we need to compute isomorphism separately.

Definition 13 (*Isomorphism*)**.** Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' isomorphic, and write $G \simeq G'$, if there exists a bijection $f : V \to V'$ with $(v, v') \in E \to (f(v), f(v')) \in E'$ for all $v, v' \in V$. Such a map f is called an isomorphism.

Even though graphs may not be identical, we will see that partial similarity can also carry important information. Hence along with graph isomorphism, subgraph isomorphism also proves to be very useful.

Definition 14 (*Subgraph, Induced Subgraph, Clique*)**.** Graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq ((V' \times V') \cap E)$, and is denoted by $G' \sqsubseteq G$. G is then a supergraph of G'. If $|V(G')| < |V(G)|$ or $|E(G')| < |E(G)|$, then G' is a strict subgraph of G and is denoted by $G' \sqsubset G$. If additionally, $E' = ((V' \times V') \cap E)$ then G' is called an induced subgraph of G. A complete subgraph is referred to as a clique.

Instead of matching whole graph we may also perform subgraph isomorphism, i.e isomorphism of a graph with subgraph of another graph. Many times we will see graphs being vectorized for such matching problems; such a vector representation of graphs are called graph invariants.

Definition 15 (*Graph Invariant*)**.** Let $\sigma : \mathcal{G} \to \mathbb{R}^d$ with $d \geq 1$ be a mapping from the space of graphs $\mathcal{G}$ to $\mathbb{R}^d$. If $G \simeq G' \implies \sigma(G) = \sigma(G')$, then σ is called a graph invariant with d dimensions.

3.3.2. Graph comparison

Earlier we have mentioned how similarity between data samples can be extended to kernel based learning so, for graph type data, it is essential to study graph comparison problems to find graph based similarities.

Definition 16 (*Graph Comparison Problem*)**.** Given two graphs G and G' from the space of graphs G, the graph comparison problem aims to find a function $s : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ such that $s(G, G')$ quantifies the similarity (or dissimilarity) between G and G' .

Graph comparison has been an active domain of research in computer science [12,13]. Here we will summarize and review a few traditional algorithmic approaches of graph comparison. This field of research can be divided into three main categories namely, similarity techniques based on graph isomorphism, topological descriptors and inexact matching algorithm.

Exact Matching Techniques. Exact graph matching techniques are applicable to scenarios where similarity of graphs are valid only when either the graphs match exactly or one of the graphs is fully contained within the second graph.

Graph Isomorphism. An obvious similarity measure for graphs is to check them for topological similarity, i.e., for isomorphism. This can give us a basic similarity value, which may be 1 for isomorphic graphs, and 0 for non-isomorphic graphs. Unfortunately, there is no polynomial runtime algorithm known for this problem of graph isomorphism [14]. We should note, that graph isomorphism is obviously an NP Problem, but it is still a mystery whether it P

or NP-complete. Intuitively, to verify isomorphism of two graphs G and G', one must consider all permutations of nodes in G' to check if any one of them is identical to G.

All invariants of two graphs must be identical for the two graphs to be isomorphic. Therefore, sometimes a simple test is often enough to prove the fact that two graphs are not isomorphic. For example, if two graphs do not have the same number of nodes and edges, they can never be isomorphic. But, if both of them are of identical size, we have to resort to graph invariants. These are computationally more expensive, like the shortest path length which has a cubic runtime complexity in terms of nodes. As a matter of fact, the most efficient way to find if two graphs are not isomorphic would be to compute multiple graph invariants in the increasing order of their computational complexity: if the graphs differ in any one invariant, they cannot possibly be isomorphic any more. "Nauty" [15], which is the fastest isomorphism testing program, is based on exactly this approach. However, it is still very difficult to decide isomorphism for two graphs which are very similar.

Subgraph Isomorphism. If two graphs G and G' are of different sizes, they are definitely not isomorphic, but there exists a possibility that the smaller graph is isomorphic to a subgraph of the larger graph. Hence, it is not enough to solely rely on exact matching. Even subgraph level isomorphism is also important. Unfortunately, this problem is also known to be NP-complete [14], and hence, it is not practically feasible for large graphs.

Incidentally, this problem is also harder than graph isomorphism. Because in this case, we not only have to check whether G' is isomorphic to G as before, but also we have to find if G' is isomorphic to any of the subgraphs of G. In short, for isomorphism, we have to take into account all permutations of G', while for subgraph isomorphism, we must consider all permutations of G' along with all subsets of G (of the size of G'). Hence, we can say that the isomorphism problem is only one instance of the subgraph isomorphism problem, where $|V(G)| = |V(G')|$ and $|E(G)| = |E(G')|$.

Partial Matching Techniques. A major disadvantage of using either graph or subgraph isomorphism is that they do not take into account partial similarities of two graphs. Graphs must either be topologically equal, or contained inside each other, to be considered similar. This is an important limitation of isomorphism-based measures of graph similarity. We will see that for many problems it is almost impossible to find exact matches, however partial matches can reveal important relations.

Maximum Common Subgraph. To overcome the above mentioned limitation of isomorphism based methods failing to catch partial similarity, we can use the concept of a maximum common subgraph [16].

Definition 17 (*Maximum Common Subgraph, mcs*)**.** Let G and G' be graphs. A graph G_{sub} is called a common subgraph of G and G' if G_{sub} is a subgraph of G and of G' . G_{sub} is a maximum common subgraph (mcs) if there exists no other common subgraph of G and G' with more nodes.

In general, the maximum common subgraph may not be unique, i.e., there can be more than one maximum common subgraph. Intuitively, maximum common subgraphs indicate inherent similarity among two graphs. The larger the Maximum Common Subgraph, the more similar they are. Another partial matching technique can be understood by turning the idea of using the maximum common subgraph upside-down: two graphs G and G' are similar if they are both part of a "small" supergraph G_{super}. The smaller the size of G_{super}, the more equivalent G and G' are.

Definition 18 (*Minimum Common Supergraph, MCS*)**.** Let G and G' be graphs. A graph G_{super} is called common supergraph of G and G' if there exists a subgraph isomorphism from G to G_{super} and from G' to G_{super}. A common supergraph of G and G' is called minimum common supergraph (MCS) if there exists no other common supergraph of G and G' with fewer nodes than G_{super}.

As we can see in [12], the computation of the minimum common supergraph may be reduced to the computation of a maximum common subgraph. While the size of the maximum common subgraph and the minimum common supergraph can indicate the measure of similarity, they can also define distances on graphs. For example, [17] define a distance based on the ratio of the size of the maximum common subgraph with respect to that of the larger of the two graphs:

$$d_1(G, G') = 1 - \frac{|mcs(G, G')|}{max(|G|, |G'|)}. \tag{15}$$

Another approach shows the difference of the sizes of the minimum common supergraph and the maximum common subgraph can also be used as a distance metric [18]:

$$d_2(G,\ G') = |MCS(G,\ G')| - |mcs(G,\ G')|. \tag{16}$$

Maximal Common Subgraphs. Maximum common subgraph is not always a proper measure of similarity. There may be some graphs that share many small subgraphs, but which do not have even one significantly large common subgraph. Such graphs would be marked as dissimilar by a similarity measure based on the size of maximum common subgraph. However, an approach that accounts for such high frequency local similarities, could be counting the number such maximal common subgraphs. Obviously, this procedure is also NP-hard, as it requires repetition of subgraph isomorphism verification. Nonetheless, efficient algorithms have been proposed to address this problem, which reduce the problem of finding maximum common subgraphs to find all cliques in the corresponding product graph [19]. Bron and Kerbosch [20] applied the classic branch-and-bound algorithm to enumerate all cliques in the product graph. While this is a widely-used technique for graph comparison in fields like bioinformatics [21] it faces immense runtime problems when the size of the product graph becomes huge. For example, to compare two graphs of size 24, the resultant product graph has roughly 600 nodes. Ina Koch [19] reported that Bron–Kerbosch algorithm on a product graph of this size required more than 3 h on hardware of that era.

Inexact Matching Techniques. Another family of graph matching algorithms does not enforce absolute matching of graphs and subgraphs. These inexact matching techniques measure the dissimilarity between two graphs in terms of some cost functions. Sometimes edit distances that transform one graph into the other are also used. From a practical point of view, these error-tolerant matching techniques seem attractive, as real-world objects are many times corrupted by noise. Therefore, it is important to integrate some sort of error tolerance into the graph similarity algorithms. The most famous concept within the class of error-tolerant graph matching techniques is the graph edit distance [22,23]. In simple terms, a graph edit operation is either a insertion, deletion or substitution (i.e., label alteration). Edit operations can be applied to both nodes as well as edges. By means of edit operations differences between two graphs can be modeled. In order to improve the modeling capabilities, often costs are assigned to every edit operation. The costs are mostly real nonnegative numbers. They must be chosen based on the domain information. Typically, more likely a certain distortion can occur, the lower its cost of operation. The edit distance, written as $d(G, G')$, of two graphs is defined as the minimum cost c accumulated over all possible sequences S

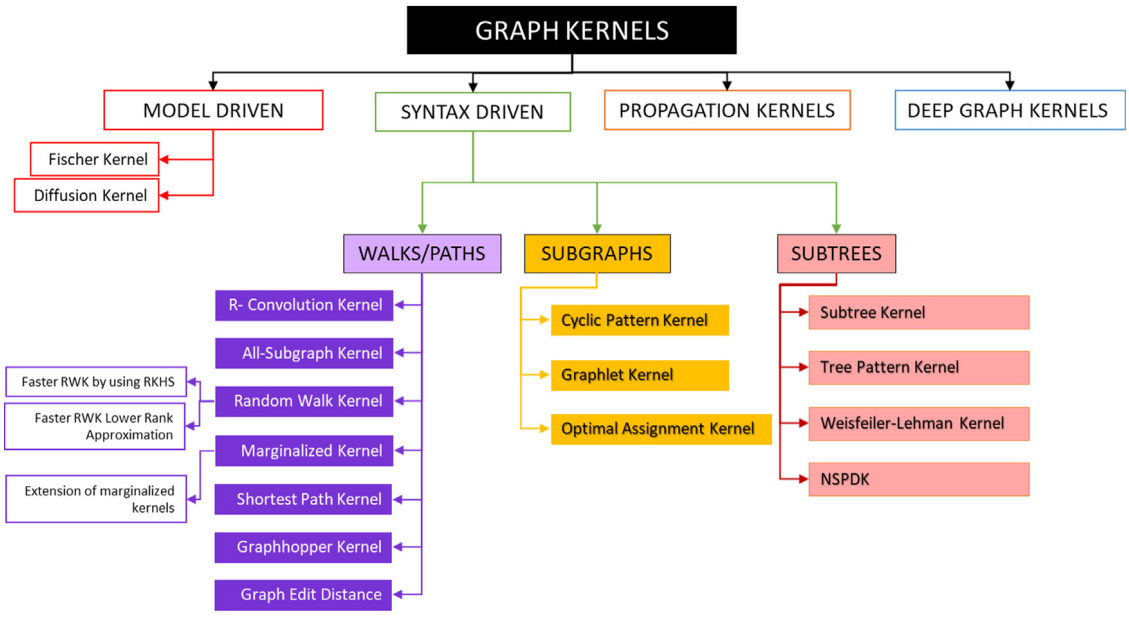

Fig. 7. Various kernels as classified in this survey.

of edit operations that successfully transform the graph G into G'. Formally,

$$d(G, G') = \min_{S} .c(S) |.S. \tag{17}$$

Obviously, if $G = G'$, then $d(G, G') = 0$, and the more G and G' are different, the larger is $d(G, G')$.

Similarity Measures based on Topological Descriptors. A major reason why graph matching, graph based learning, and graph mining are difficult and expensive lies within the complex structure of graphs which does not help in obtaining a simple feature vector representation scheme. The third family of similarity measures for graph matching techniques aim at finding efficient feature vector representations of the graphs. These feature vector descriptions of graph structures are often called topological descriptors. The goal is to device an efficient vector-representations of graphs such that by comparing these vectors we get a good indication of their similarity. One of these schemes is based on the spectral graph theory [24].

The roots of such graph encoding schemes lie in the domain of chemo-informatics. A major challenge in this area was to address queries on large databases of molecular graphs. For this purpose, many different molecular (topological) descriptors were invented, as mentioned in the extensive work of Todeschini [25]. A simple and elegant example among them is the Wiener Index [26], defined as the sum of all shortest paths in a graph.

Definition 19 (*Wiener Index*)**.** Let $G = (V, E)$ be a graph. Then the Wiener Index $W(G)$ of G is defined as

$$W(G) = \int_{v_i \in G} \int_{v_j \in G} d(v_i, v_j) \tag{18}$$

where $d(v_i, v_j)$ is defined as the length of the shortest path between nodes v_i and v_j from G.

Clearly, this index is exactly the same for isomorphic graphs. Hence, the Wiener Index, and all such other topological descriptors (that exclude node labels), represent a family of graph invariants (see Definition 15). Nonetheless, the reverse hypothesis that identical topological descriptors indicate isomorphism is not always true (but if that is the case we call such a topological descriptor, a complete graph invariant [27]). However, all known complete graph invariants have an exponential runtime complexity, as their computation is somewhat equivalent to solving the graph isomorphism problem itself.

Recent Trends in Graph Comparison. Due to the above discussed problems in traditional approaches to graph matching, researchers have started to take new roads to handle these problems. Some notable contribution has been made in regards to automatic learning of edit distance parameters [28,29] and the usage of graphical models for graph matching [30]. Other alternative strategies include efficient branch and bound algorithms that have been developed to enumerate frequent common subgraphs [31–33].

While new approaches like these show promising results in applications, none of these methods can completely avoid the same problems encountered in the classical approaches like increasing runtime for larger graphs or resorting to simplified representations of graphs that ignore significant parts of topological information.

However, graph kernels are one of the latest approaches to graph comparison. Interestingly, graph kernels employ concepts from all the above discussed traditional branches of graph comparison. Most of them measure similarity in terms of isomorphic substructures but they also allow for inexact matching of graphs and, as the primary benefit of kernel based learning, they also treat graphs as vectors in the Hilbert space. Though the earliest graph kernel can date back to almost 20 years [34], major contributions have been made in the last decade which renders this domain as one of the most elite domains of research.

4. Graph kernels

Before we move on to study graph kernels we need to go through a brief summarization of different classes of kernels on structured data. Kernel methods and support vector machines especially have succeeded in various learning problems on data represented as a single table. But most of the 'real-world' data is structured, i.e., it has no default representation in a equation format. Generally, to apply such kernel methods to 'real-world' data, we need extensive pre-processing to map the data into a real vector space and therefore into a single table. Most of the datasets used can be found either from links provided by the respective

authors in the papers or this website which has a good collection of Graph Kernel datasets.[1] Fig. 7 shows the various types of kernels that will be taken into account in the consequent chapters.

Graph Kernels can be broadly categorized into two major branches based on the principal driving force of their definition, namely, model based kernels and syntax based kernels.

4.1. Model driven kernels

Model driven kernels rely on some kind of knowledge about the sample space, i.e., about the relationships among data. There are principally two subcategories in this branch namely, generative models and transformative models. While parameters of generative models are treated as features for comparison, transformative models study the ability of the graphs to transform them in certain way as per the problem domain. These transformed graphs can be seen as a model of the instance space; while each edge only contains local information about neighboring vertices during the process of transformation, the set of all edges contain information about the global structure of the sample space.

4.1.1. Kernels from generative models

Generative models like hidden Markov models [35], are widely used in computer science research. One motivation behind designing kernels for generative models is to be able to apply kernel based techniques to sequential data. Sequences naturally occur all around us, for example, spoken words are sequences of phonemes; proteins are sequences of amino acids and genes are sequences of nucleic acids. Another motivation is to get better classification performance using generative models.

One of the earliest and most famous generative model kernel function is the Fisher kernel [36,37,34]. The main idea is to make use of the gradient of log-likelihood with respect to the parameters of a generative model as the features for a discriminative classifier. The reason to use this feature space instead of the sequence itself is that the gradient of the log-likelihood defines the generative process of the sequences rather than the corresponding posterior probabilities.

Let U_x be the gradient of the log-likelihood with respect to the parameters of the generative model $P(x|\theta)$ at x,

$$U_x = \nabla_\theta log P(x|\theta). \tag{19}$$

Moreover, let I be the Fisher information matrix, i.e ., the value of the outer product $U_x U_x^T$ over $P(x|\theta)$. The Fisher kernel is then defined as

$$k(x, x') = U_x^T I^{-1} U_{x'}. \tag{20}$$

More recently, a more generalized framework for defining kernel functions on generative models has been developed [38]. The so-called 'Marginalized kernels' present the Fisher kernel as one special case. The paper compares the Fisher kernel with other marginalized kernels and argues that these have a few advantages over the Fisher kernels. Marginalized Kernels are discussed later in details in under Section 4.2.2.

4.1.2. Kernels from transformations

This family of kernels are based on knowledge about properties of instances as well as possible transformations between instances. The most well-known kernel in this class is the diffusion kernel. The core idea behind diffusion kernels [39] is that it is easier to describe the locality of an instance rather than describe the entire structure of the instance space. The same applies when we compute similarity between instances; the neighborhood of an instance can be defined as a set of all instances that differ from this one only due to the presence or absence of one particular attribute. For example, when working with molecules, such properties might be some functional groups or some specific kinds of bonds. The neighborhood relation though local in nature, induces the global information about the mechanics of the instance space. The approach taken by the diffusion kernel algorithm is to try to capture this global information in a simple kernel function while depending on the local neighborhood descriptions only.

The main mathematical principle used in diffusion kernels is matrix exponentiation. The exponential of a square matrix H is defined as

$$e^{\beta H} = \lim_{n\to\infty} \int_{i=0}^{n} \frac{(\beta H)^i}{i!}. \tag{21}$$

An efficient method to compute $e^{\beta H}$ is to convert H into a diagonal matrix D such that $H = T^{-1}DT$ and, thereby, compute

$$e^{\beta H} = T^{-1} e^{\beta D} T. \tag{22}$$

While, H is called the "generator", the kernel matrix $k(x, x')$ refers to the exponential of the generator i.e. $k(x, x') = e^{\beta H}$.

In case of a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the negative Laplacian of the graph G is the generator H, which can be written as

$$[H]_{i,j} = \begin{cases} 1 & v_i, v_j \in E \\ -|\delta(v_i)| & v_i = v_j \\ 0 & otherwise. \end{cases} \tag{23}$$

4.2. Syntax driven kernels

Syntax based models focus on the semantics of the data. It comprises of the largest family of kernels, namely the convolutional kernels. Most of the algorithms deal with manipulation of syntactic elements like walk, paths, cycles, subgraphs, subtrees and so on. We will see that lots of these kernels preserve both local and global features for comparison and hence are applicable in various domains.

4.2.1. R-convolution kernels

One of the earliest and most generic class of kernels on structured data was defined by Haussler [40] in the name of R-Convolution Kernels. Let $x, x' \in \mathcal{X}$ be the objects and $\overrightarrow{x}, \overrightarrow{x}' \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_D$ be tuples of parts of these objects. Given the relation $R : (\mathcal{X}_1 \times \cdots \times \mathcal{X}_D) \times \mathcal{X}$ we can define the decomposition $R^{-1}(x) = \{\overrightarrow{x} : R(\overrightarrow{x}, x)\}$. Then, the convolution kernel is defined as

$$k_{conv}(x, x') = \int_{\widehat{x}\in R^{-1}(x), \widehat{x'}\in R^{-1}(x')} \int_{d=1}^{D} k_d(x_d, x'_d). \tag{24}$$

The basic idea of the convolution kernel is to capture the semantics of composite objects using a relation R between an object and its parts; for graphs, parts may be considered to be subgraphs. The idea of a convolutional kernel is very generic hence it deserves a separate section. The following section will focus mainly on graphs henceforth.

4.2.2. Graph kernels based on walks and paths

We saw R-convolution kernels as a generic kernel for structured data. It uses similarity measurement techniques by breaking down a structured data into smaller parts. The most obvious implementation of the technique on graph data is be to break down the graph into all possible subgraphs and check for isomorphism; thus, this subgraph kernel may be defined as

$$k_{subgraph}(G, G') = \int_{S\sqsubseteq G} \int_{S'\sqsubseteq G'} k_{isomorphism}(S, S') \tag{25}$$

[1] https://ls11-www.cs.uni-dortmund.de/staff/morris/graphkerneldatasets.

where,

$$k_{isomorphism}(S, S') = \begin{cases} 1, & if\ S \simeq S' \\ 0, & otherwise. \end{cases} \quad (26)$$

During his early works on graph kernels, Gärtner showed that the computation of this all-subgraph kernel is NP-hard [41]. The proof of this is simple: to solve $k_{subgraph}(G, G')$ we need to compute $k_{isomorphism}(S, S')$ and it is already a proven fact that graph isomorphism is an NP-hard problem.

As alternative two classes of random walk based kernels were proposed in the coming years, viz. the product graph kernel or commonly known as Random Walk Kernel [42] and the marginalized graph kernel [38]. The Random Walk Kernel had various extensions in consequent years like the graph edit distance based kernel [16] and other speedups were proposed by [5,43].

Marginalized Kernels also saw some upgrade through the work of [44]. When studying walk based kernels, it is interesting to note that all walks are represented as sequence of nodes. Borgwardt [45] proposed a family of path based algorithms based on edge sequences.

Random Walk Kernel. Gärtner [42] proposed a random walk kernel counting common walks in two graphs; it uses a special graph product kernel called direct product graph [46].

Definition 20 (*Direct Product Graph*)**.** The direct product of two labeled graphs $G = (V, E, \mathcal{L})$ and $G' = (V', E', \mathcal{L}')$, denoted as $G_\times(V_\times, E_\times, \mathcal{L}_\times) = G \times G'$, can be defined as

$$\begin{aligned} V_\times &= \{(v_i, v'_{i'}) : v_i \in V \wedge v'_{i'} \in V' \wedge \mathcal{L}(v_i) = \mathcal{L}(v'_{i'})\} \\ E_\times &= \{((v_i, v'_{i'}), (v_j, v'_{j'})) \in V_\times \times V_\times \\ &(v_i, v_j) \in E \wedge (v'_{i'}, v'_{j'}) \in E' \wedge \mathcal{L}(v_i, v_j) = \mathcal{L}(v'_{i'}, v'_{j'})\}. \end{aligned} \quad (27)$$

Using this product graph the random walk kernel is defined as

Definition 21 (*Random Walk Kernel*)**.** Let G and G' be two graphs, let $A_\times$ denote the adjacency matrix of their product graph $G_\times$, and let $V_\times$ denote the node set of the product graph $G_\times$. With a sequence of weights $\lambda = \lambda_0, \lambda_1, \ldots (\lambda_i \in \mathbb{R}; \lambda_i \geq 0, \forall i \in \mathbb{N})$, the product graph kernel is defined as

$$k_\times(G, G') = \int_{i,j=1}^{|V_\times|} \left[\int_{k=0}^{\infty} \lambda_k A_\times^k \right]_{ij} \quad (28)$$

if the limit exists.

The limit can be easily computed for two particular assignment choices of λ_k, first using the geometric series and second using the exponential series.

Setting $\lambda_k = \lambda^k$, i.e., to a geometric series, we obtain the geometric random walk kernel which is defined as

$$k_\times(G, G') = \int_{i,j=1}^{|V_\times|} [\int_{k=0}^{\infty} \lambda^k A_\times^k]_{ij} = \int_{i,j=1}^{|V_\times|} [(I - \lambda A_\times)^{-1}]_{ij}. \quad (29)$$

For this equation a matrix inversion of $A_\times$ is needed which is of the size $n^2 \times n^2$. Matrix inversion has a cubic time complexity, cranking up the overall computational complexity to $O(n^6)$.

Similarly, setting $\lambda_k = \frac{\beta^k}{k!}$, i.e., to an exponential series, we obtain the exponential random walk kernel defined as

$$k_\times(G, G') = \int_{i,j=1}^{|V_\times|} [\int_{k=0}^{\infty} \frac{(\beta A_\times)^k}{k!}]_{ij} = \int_{i,j=1}^{|V_\times|} [e^{\beta A_\times}]_{ij}. \quad (30)$$

In this case, the matrix diagonalization of the $n^2 \times n^2$ matrix $A_\times$ is necessary to compute $\beta A_\times$, which is again an operation with cubic runtime in the size of the matrix. Hence, the total complexity goes up to $O(n^6)$.

Faster computation of random walk kernels by extending concepts of linear algebra to RKHS. To handle extreme complexity issues Vishwanathan [5] extended concepts of linear algebra to RKHS(Reproducing Kernel Hilbert spaces). The product graph kernel as shown in (35) is rewritten as

$$k_\times(G, G') = \int_{k=0}^{\infty} \lambda_k q_\times^T W_\times^k p_\times \quad (31)$$

where, $W_\times^k$ is a weight matrix that represents the similarity between simultaneous k length random walks on G and G' and $p_\times$ and $q_\times$ denote the initial and stopping probability distributions.

The special case where $\lambda_k = \lambda^k$ as defined in (36) can be rewritten as

$$k_\times(G, G') = \int_{k=0}^{\infty} \lambda^k\, q_\times^T\, W_\times^k\, p_\times \ = q_\times^T (I - \lambda W_\times)^{-1} p_\times. \quad (32)$$

The weight matrix may be defined as

$$W_\times = \int_{l=1}^{d} A_l \otimes A'_l \quad (33)$$

where, A_l is the adjacency matrix showing entries corresponding to label l only and $\otimes$ is the Kronecker Product. The weight matrix may be solved using various methods like

1. **Sylvester Equation Method**. Converting the problem to solving the Sylvester equation (39) can be rewritten as

   $$k_\times(G, G') = q_\times^T (I - \lambda W_\times)^{-1} p_\times = q_\times^T vec(X) \quad (34)$$

   where X is defined by a Sylvester equation as

   $$X = \int_i A'_i \lambda X A_i^T + X_0 \quad (35)$$

   where, $X_0 = p_\times$, i.e. the initial probability and A_i was previously defined as the adjacency matrix filtered by label i. Now, with the given parameters, we can solve for X and hence for $k_\times(G, G')$ in $O(n^3)$ runtime using an available Matlab code called "dylap".[2]
2. **Conjugate gradient**. For a matrix M and a vector b we can use Conjugate Gradient (CG) methods to efficiently solve the system of equations given by $Mx = b$ [47]. While, these methods are designed specifically for symmetric positive semi-definite matrices, they can also be used to solve other types of linear systems with efficiency. They are particularly efficient when the matrix is rank deficient or, in other terms, has a small effective rank, i.e., a significantly smaller number of distinct eigenvalues. Moreover, if computing matrix–vector products is not expensive (because of some reason like M being sparse), the Conjugate Gradient Method can be sped up significantly [47]; specifically, computing Mv for any vector v requires $O(k)$ time and the effective rank of the matrix M is given by m, so the algorithm requires only $O(mk)$ time to solve $Mx = b$.
   The graph kernel of Eq. (39) can be computed in two-steps: first solving the linear system $(I - \lambda W_\times)x = p_\times$ for x, then calculating $q_\times^T x$.
   Now we focus on efficient ways to solve (43) using conjugate gradient method. To be noted, if G and G' contain n nodes each, then $W_\times$ is a square matrix of the order n^2. Direct computation of the matrix–vector product $W_\times r$, needs $O(n^4)$ time. Key to their speed-ups is the ability to extend concepts of linear algebra to Hilbert spaces [48]; specifically, by extending concepts of Kronecker Product to Reproducing Kernel Hilbert Spaces, they established

   $$W_\times r = (\Phi(L) \otimes \Phi(L'))vec(R) = vec(\Phi(L')R\Phi(L)^T). \quad (36)$$

[2] http://www.mathworks.com/help/control/ref/dlyap.html.

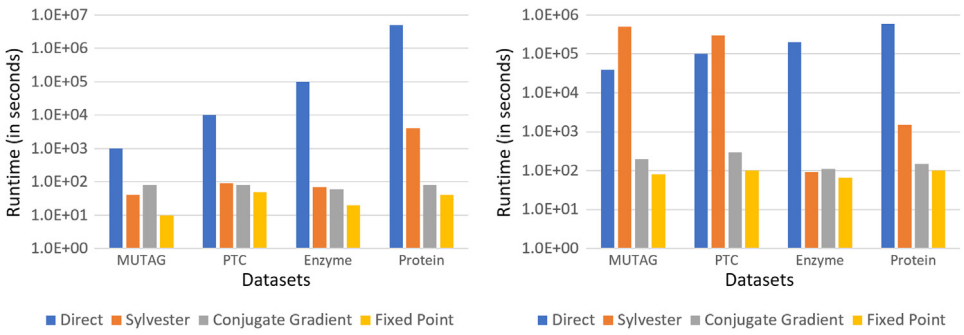

Fig. 8. Time (in seconds on a log-scale) to compute 100 × 100 kernel matrix for unlabeled (left) resp. labeled (right) graphs from several datasets. Compare the conventional direct method (black) to our fast Sylvester equation, conjugate gradient (CG), and fixed-point iteration (FP) approaches. Image Courtesy: [5].

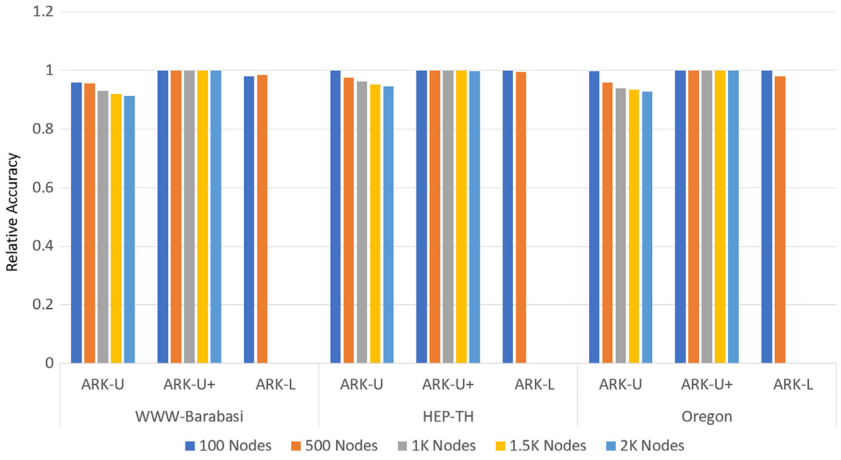

Fig. 9. Relative accuracy with respect to exact techniques.

If $\Phi \in \mathbb{R}^s$ for some s then the above matrix–vector product can be computed in $O(n^3 s)$ time. If $\Phi(L)$ and $\Phi(L')$ are sparse then $\Phi(L')R\Phi(L)^T$ can be computed yet more efficiently—if there are $O(n)$ non-ϵ entries in $\Phi(L)$ and $\Phi(L')$, then computing (43) requires only $O(n^2)$ time.

3. **Fixed point Iterations method**. Fixed-point methods can be defined by rewriting (39) as

$$x = p_\times + \lambda W_\times. \tag{37}$$

Solving for x is equivalent to find a fixed point during the above iteration [47]. If x_t denotes the value of x at iteration t, we set $x_0 := p_\times$, and compute

$$x_{t+1} = p_\times + \lambda W_\times x_t \tag{38}$$

iteratively until $\|x_{t+1} - x_t\| < \varepsilon$, where $\| \|$ denotes the Euclidean norm and ε some predefined tolerance. This is guaranteed to converge if all the eigenvalues of $\lambda W_\times$ lie inside the unit distance which can be ensured by setting $\lambda < 1/\xi_{max}$, where ξ_{max} is the largest eigenvalue of $W_\times$.
The above method is closely related to the power method used to compute the largest eigenvalue of a matrix [49]. Efficient pre-conditioners may also be used to speed up convergence as shown by [49]; since each iteration of (46) involves the calculation of the matrix–vector product $W_\times x_t$ all kinds of speed-ups for the computation of the matrix–vector product discussed before for conjugate gradient methods are also applicable here. For example, by taking $W_\times$ as a sum of Kronecker products the worst-case time complexity is reduced to $O(n^3)$ [42], in contrast to [38] who calculated the matrix–vector product explicitly.

A statistical analysis of the above mentioned techniques can be seen in Fig. 8.

Faster computation considering lower rank approximation. A significant contribution to random walk kernel has been made recently by Kang [43]. Their main contribution was to consider a lower rank approximation $\hat{W}_\times$ of $W_\times$ in (38).

Definition 22 (*r-Approximation of a Matrix*)**.** Given a matrix A, the r-Approximation of A is a matrix $\hat{A}$ satisfying the following equation

$$\|A - \hat{A}\|_F \leq \min_{z|rank(Z)=r} \|A - Z\|_F \tag{39}$$

meaning that $\hat{A}$ provides an equal or better approximation than the best rank-r approximation.

Using this r-rank approximation $\hat{W}_\times$ may be substituted in (38) as

$$k_\times(G, G') = q_\times^T (I - \lambda \hat{W}_\times)^{-1} p_\times. \tag{40}$$

The most well-known way of computing r-rank approximation of a matrix is through a rank-r Singular Value Decomposition.

Definition 23 (*Singular Value Decomposition*)**.** For a any matrix $M^{m\times n}$ matrix, there exists a factorization called singular value decomposition of M, of the form $M = U\Sigma V^T$, where $U^{m\times m}$ is a unitary matrix, $\Sigma^{m\times n}$ is a diagonal matrix and V^T is a conjugate transpose of $V^{n\times n}$ unitary matrix.

Instead of explicitly computing $W_\times$, SVD may be directly applied on $W_\times^1$ and $W_\times^2$. The algorithm may be expressed as follows

Algorithm 1 Approximate random walk kernel for unlabeled nodes and asymmetric W (ARK-U)

Input:
Adjacency matrix A_1 and A_2 of a graphs G_1 and G_2 respectively
Starting and ending probabilities (p_1, q_1) for G_1 and (p_2, q_2) for G_2 respectively,
decay factor c.

Output: Approx. random walk kernel $\hat{k}(G_1, G_2)$

1. $W_1 \leftarrow A_1 D_1^{-1}$; // row normalized A_1
2. $W_2 \leftarrow A_2 D_2^{-1}$; // row normalized A_2
3. $U_1 \Sigma_1 V_1^T \leftarrow W_1^T$ //SVD on W_1^T
4. $U_2 \Sigma_2 V_2^T \leftarrow W_2^T$ //SVD on W_2^T
5. $\widehat{\Lambda} \leftarrow ((\Lambda_1 \otimes \Lambda_2)^{-1} - c(V_1^T \otimes V_2^T)(U_1 \otimes U_2))^{-1}$
6. $L \leftarrow (q_1^T U_1 \otimes q_2^T U_2)$
7. $R \leftarrow (V_1^T p_1 \otimes V_2^T p_2)$
8. $\hat{k}(G_1, G_2) \leftarrow (q_1^T p_1)(q_2^T p_2) + cL\widehat{\Lambda}R$

For some further speedups (ARK-U+) on symmetric $W_\times$, it was proposed that Eigen Value Decomposition may be used instead of Singular value decomposition as the only difference is in the signs which does not matter for symmetric matrices. The eigenvector computation is simplified as $\widehat{\Lambda} \leftarrow ((\Lambda_1 \otimes \Lambda_2)^{-1} - cI)^{-1}$.

For labeled graphs, $W_\times = \tilde{L}(W_1^T \otimes W_2^T)$, where $\tilde{L} = \sum_{j=1}^{d_n} L_1^{(j)} \otimes L_2^{(j)}$, where $L_k^{(j)}$ is a diagonal label matrix whose ith element is 1 if the node i of the graph G_k has the label j, or 0 otherwise.

The algorithm for the computation of the approximate random walk kernel for labeled nodes may be written as

Algorithm 2 Approximate random walk kernel for labeled nodes (ARK-L)

Input:
Weight matrix W_1 and W_2 for graph G_1 and G_2 respectively
Label Matrices $L_1^{(1)}$ to $L_1^{(d_n)}$ of G_1 and $L_2^{(1)}$ to $L_2^{(d_n)}$ of G_2
Starting and ending probabilities (p_1, q_1) for G_1 and (p_2, q_2) for G_2 respectively,
decay factor c.

Output: Approx. random walk kernel $\hat{k}(G_1, G_2)$

1. $U_1 \Sigma_1 V_1^T \leftarrow W_1^T$ //SVD on W_1^T
2. $U_2 \Sigma_2 V_2^T \leftarrow W_2^T$ //SVD on W_2^T
3. $\widehat{\Lambda} \leftarrow ((\Lambda_1 \otimes \Lambda_2)^{-1} - c(\sum_{j=1}^{d_n} V_1^T L_1^{(j)} U_1 \otimes V_2^T L_2^{(j)} U_2))^{-1}$
4. $L \leftarrow \sum_{j=1}^{d_n} q_1^T L_1^{(j)} U_1 \otimes q_2^T L_2^{(j)} U_2$
5. $R \leftarrow \sum_{j=1}^{d_n} V_1^T L_1^{(j)} p_1 \otimes V_2^T L_2^{(j)} p_2$
6. $\hat{k}(G_1, G_2) \leftarrow \sum_{j=1}^{d_n} (q_1^T L_1^{(j)} p_1)(q_2^T L_2^{(j)} p_2) + cL\widehat{\Lambda}R$

Kang [43] performed extensive experiments to show impressive speedups over traditional approaches as shown in Table 2, whereas, Fig. 9 shows the loss in relative accuracy with respect to exact techniques.

Graph Edit Distance Based Kernels. Another extension to the random walk kernel was proposed by Neuhaus [16]. Their work showed that the random walk kernel worked well in some datasets, specifically when local neighborhood played a significant role in the similarity analysis. On the other hand, when global matching constraints were taken into account, it performed significantly worse than graph matching techniques like the graph edit distance. Hence, to take the best of both worlds, they proposed to consider only a subset of random walks which was a part of the optimal edit path from G to G'. If $S = \{v_1 \to v_1', v_2 \to v_2' \ldots\}$ then the adjacency matrix of the direct product graph $G \times G'$ is given by

$$[A_\times]_{(u,u'),(v,v')} = \begin{cases} k((u,u'),(v,v')) & \text{if } ((u,u'),(v,v')) \in E_\times \text{ and} \\ & u \to u' \in S \text{ and } v \to v' \in S \\ 0 & \text{otherwise.} \end{cases} \quad (41)$$

This adjacency matrix was then used with the standard random walk kernel as defined in (35).

Another explicit graph based kernel was also proposed by [16] which explicitly uses edit distance for kernel computations. Two kernels were defined

$$k_I^+(x, x') = \int_{x_0 \in I} k_{x_0}(x, x')$$
$$k_I^*(x, x') = \prod_{x_0 \in I} k_{x_0}(x, x') \quad (42)$$

where $k_{x_0}(x, x') = \frac{1}{2}(d(x, x_0)^2 + d(x_0, x')^2 - d(x, x')^2)$ and $d(.,.)$ is the (non-negative and symmetric) edit-distance of two patterns. It can be seen that the kernel is computed considering x_0 as the origin of the pattern space, hence x_0 is also called zero-graph and I is a set of zero graphs from the training set. Also, to be noted that this kernel is designed not only for graphs but for any kind of structured data which has a non-negative and symmetric edit-distance measure; a common example other than the graph is the string. Fig. 10 shows how this kernel faired against various datasets.

Marginalized Graph Kernels. Though motivated differently, the marginalized graph kernels of [38] are closely related to other walk based kernels. Their kernel is defined as the expectation of a kernel over all pairs of label sequences from two graphs. For extracting features from graph $G = (V, E, \mathcal{L})$, a set of label sequences is produced by performing a random walk. At the first step, $v_1 \in V$ is sampled from an initial probability distribution $p_s(v_1)$ over all nodes in V; subsequently, at the ith step, the next node $v_i \in V$ is sampled subject to a transition probability $p_t(v_i|v_i - 1)$ and the random walk ends with probability $p_q(v_i - 1)$

$$\int_{v_i=1}^{|V|} p_t(v_i|v_i - 1) + p_q(v_i - 1) = 1. \quad (43)$$

Each random walk generates a sequence of nodes $w = (v_1, v_2, \ldots, v_l)$ where l is the length of w (possibly infinite). The probability for the walk w is described as

$$p(w|G) = p_s(v_1) \int_{i=2}^{l} p_t(v_i|v_{i-1}) p_q(v_l). \quad (44)$$

Associated with a walk w, we obtain a sequence of labels which is an alternating label sequence of node labels and edge labels from the space of labels $\mathcal{Z}$

$$h_w = (h_1, h_2, \ldots, h_{2l-1}) \in \mathcal{Z}^{2l-1}. \quad (45)$$

The probability of a label sequence h is equal to the sum of the probabilities of all walks w emitting a label sequence h_w identical to h as

$$p(h|G) = \int_w \delta(h = h_w).p_s(v_1) \int_{i=2}^{l} p_t(v_i|v_{i-1} p_q(v_l)) \quad (46)$$

Table 2
Speedups obtained by [43] with respect to exact methods.

	ARK-U	ARK-U+	ARK-L
WWW	8×	522×	46 075×
HEP-TH	6×	389×	695×
Oregon	11×	422×	97 865×

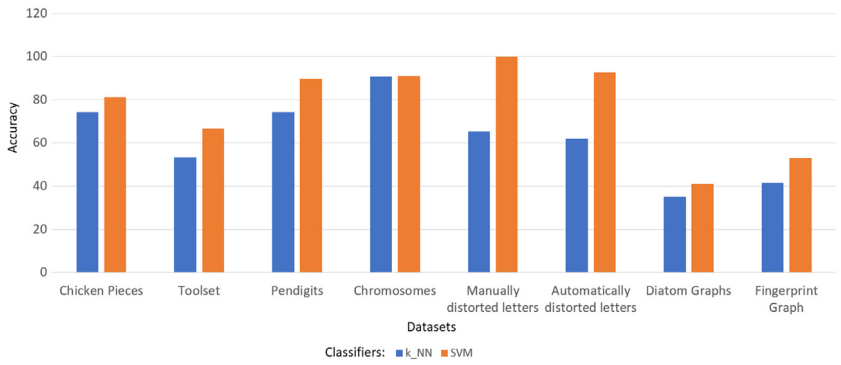

Fig. 10. Performance of Graph Edit Distance based Kernel on various datasets.

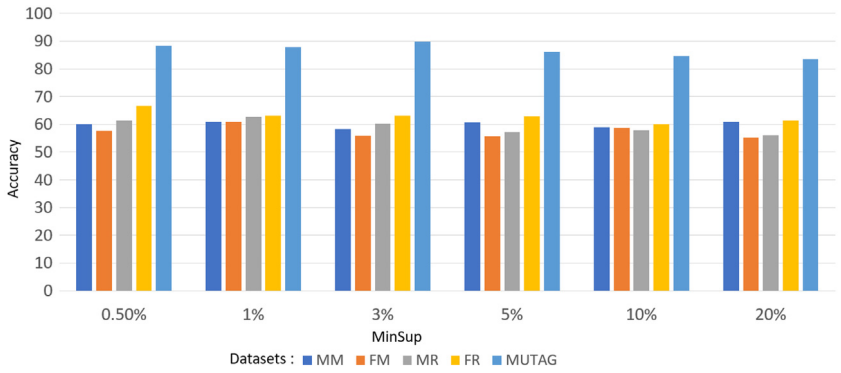

Fig. 11. Classification accuracies (%) of the pattern discovery method. *X* axis shows the ratio of the minimum support parameter to the number of compounds.

where δ is a function that returns 1 if its argument holds or 0 otherwise.

[38] then define a kernel k_z between two label sequences h and h'. Assuming that k_v is a nonnegative kernel on nodes and k_e is a nonnegative kernel on edges, then the kernel for label sequences is defined as the product of label kernels when the lengths of two sequences are identical ($l = l'$):

$$k_z(h, h') = k_v(h_1, h'_1) = \int_{i=2}^{l} k_e(h_{2i-2}, h'_{2i-2}) k_v(h_{2i-1},\ h'_{2i-1}). \quad (47)$$

The label sequence graph kernel is then defined as the expectation of k_z over all possible h and h' as

$$k(G, G') = \int_h \int_{h'} k_z(h, h') p(h|G) p(h'|G'). \quad (48)$$

As shown by [38], the marginal graph kernel is also of complexity $O(n^6)$ because after converting to matrix notation, we need to find the inverse of a $n^2 \times n^2$ matrix.

Other than issues of time complexity, graph kernels face another major issue called tottering. Walks allow for repetitions of nodes and edges, which means that the same node or edge is counted repeatedly in a similarity measure based on walks. In an undirected graph, a random walk may even start tottering between the same two nodes in the product graph, leading to an artificially high similarity score, which is caused by one single common edge in two graphs. Furthermore, a random walk on any cycle in the graph can in principle be infinitely long, and drastically increase the similarity score, although the structural similarity between the two graphs is minor. They performed experiments on the PTC dataset which further subdivided according to category of animals (Male Mouse (MM), Female Mouse (FM), Male Rat (MR) and Female Rat (FR)). Other than that, they also used the MUTAG dataset (see Fig. 11).

Extensions of Marginalized Graph Kernels. An extension of marginalized graph kernel was proposed by Mahé [44] to overcome the problem of tottering and computational complexity. The computational complexity was handled by increasing the specificity of the labels through the inclusion of contextual information about the vertex. They applied a simple method which involved numbering vertices with respect to the number of adjacent nodes; starting by numbering all vertices as 1, an iterative process renumbered every vertex by the sum of its adjacent nodes in every iteration. This increase in specificity reduced number of common label paths between graphs which shortened the computation time. Secondly, tottering was handled by modifying the probability function of the walks as previously described in (35). The new probability function takes into account only first order walks and can be expressed as

$$p(w|G) = p_s(v_1) p_t(v_2|v_1) \int_{i=2}^{l} p_t(v_i|v_{i-2}, v_{i-1}) p_q(v_l). \quad (49)$$

Shortest Path Kernel. While studying walk based kernels it is interesting to note that all walks are represented as sequence of nodes. Borgwardt [45] proposed a family of algorithms based on edge sequences. After defining edge walks and edge path in Section 3.3.1, the all-path kernel can be defined as

Definition 24 (*All-paths Kernel*)**.** Given two graphs G_1 and G_2, let $P(G_i)$ be the set of all paths in graph G_i where $i \in \{1, 2\}$ and k_{path} be a positive definite kernel on two paths, defined as the product of kernels on edges and nodes along the paths. The all-paths kernel $k_{all\text{-}paths}$ can be defined as

$$k_{all\text{-}paths}(G_1, G_2) = \int_{p_1 \in P(G_1)} \int_{p_2 \in P(G_2)} k_{path}(p_1, p_2). \quad (50)$$

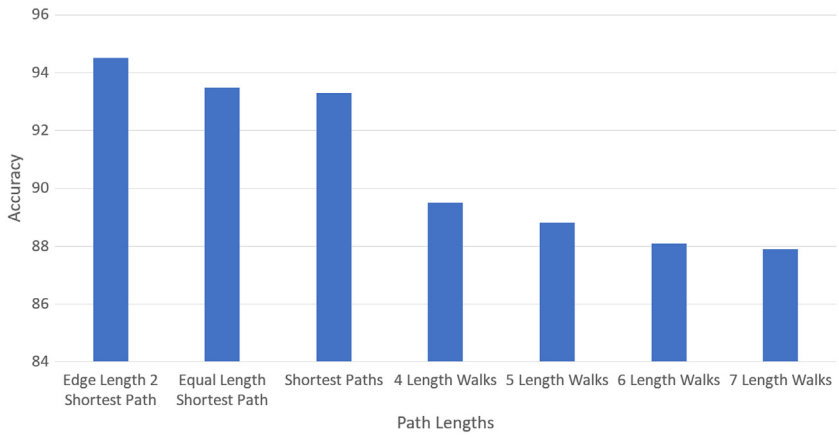

Fig. 12. Walk kernel vs. shortest-path kernel. Prediction accuracy on 540 proteins from 6 EC classes in 10-fold cross-validation.

The all-path kernel is obviously an NP-Hard problem hence not suitable for computational purpose. Hence, the shortest path kernel was proposed by [45]; their method involved converting a graph to its corresponding shortest path graph. Given a graph $G(V_G, E_G)$ a shortest-path graph S is defined by (V_s, E_s) where $V_s\{v_s^1, v_s^2, \ldots, v_s^n\} = V_g\{v_g^1, v_g^2, \ldots, v_g^n\}$ and $e(v_s^1, v_s^2) \in E_s$ if $\exists$ a shortest-path p from v_g^1 to v_g^2 and $label(e(v_s^1, v_s^2)) = cost(p)$. Given the shortest path graph, the shortest path kernel is defined as

Definition 25 (*Shortest-path Graph Kernel*)**.** Let G_1 and G_2 be two graphs Floyd-transformed into S_1 and S_2. We can then define our shortest-path graph kernel on $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ as

$$k_{shortest\text{-}paths}(S_1, S_2) = \int_{e_1 \in E_1} \int_{e_2 \in E_2} k_{walk}^{(1)}(p_1, p_2) \tag{51}$$

where $k_{walk}^{(1)}$ is a positive definite kernel on edge walks of length 1.

They also proposed further label enrichment along with the cost by including other information in edge labels like number of edges in the path or average cost of the edges. This kind of information can be used for restricting the computation only to paths which have matching number of edges. Another modification to increase accuracy was to consider k-shortest paths instead of the shortest path; even k-shortest walks and not paths were also considered for the sake of comparison. The comparison can be seen in Fig. 12 .

Graph Hopper Kernel. An alternative approach to path based kernels was proposed in 2013 by Feragen et al. [50]. Instead of comparing paths through products of kernels on their lengths and ending points, they compared paths through node kernels for the nodes encountered during hopping along the path. In this way the path kernel could be decomposed into a sum of node kernels.

The Graph Hopper kernel is defined as a sum of path kernels over the family of shortest paths P and P'

$$k(G, G') = \sum_{\pi \in P, \pi' \in P'} k_p(\pi, \pi') \tag{52}$$

where k_p can be defined as a sum of node kernels k_n on nodes simultaneously encountered while hopping along paths π and π' in G and G'

$$k_p(\pi, \pi') = \begin{cases} \sum_{j=1}^{|\pi|} k_n(\pi(j), \pi'(j)), & if\ |\pi| = |\pi'| \\ 0 & otherwise. \end{cases} \tag{53}$$

Experiments on various datasets revealed the performance and speed of this kernel on numerous large datasets. Table 3 shows the corresponding statistics.

4.2.3. Graph kernels based on limited-size subgraphs

Cyclic Pattern Kernels. In 2004 Horváth et al. [51] provided a technique that decompose a graph into cyclic patterns, then count the number of common cyclic patterns which occur in both graphs. Their kernel is plagued by computational issues; in fact they show that computing the cyclic pattern kernel on a general graph is NP-hard. They consequently restrict their attention to practical problem classes where the number of simple cycles is bounded. They defined the kernel as

$$k_{CP}(G_i, G_j) = |C(G_i) \cap C(G_j)| + |T(G_i) \cap T(G_j)| \tag{54}$$

where $C(G_i)$ refers to a set of minimum cost cycles of graph G_i and $T(G_i)$ refers to a set of minimum cost trees of graph G_i.

Graphlet Kernels for large graph comparison. A very elegant subgraph based kernel was defined by Shervashidze [52]. If $\mathcal{G} = \{graphlet(1), \ldots, graphlet(N_k)\}$ is the set of k-size graphlets and G is a graph of size n, a vector f_G of length N_k is defined whose ith component corresponds to the frequency of occurrence of graphlet(i) in G, $\#(graphlet(i) \sqsubseteq G)$; f_G was also called the k-spectrum of G. Using these statistics, the graphlet kernel was defined as

$$k_{GK}(G, G') = f_G^T f_{G'}. \tag{55}$$

We can check the performance and speedup of various variation of graphlet kernels on MUTAG, PTC, Enzymes and D&D Datasets in Table 4.

Optimal Assignment Kernels. In the aforementioned graph kernels, R-convolution often boils down to an all-pairs comparison of substructures from two composite objects. Intuitively, finding a best match, an optimal assignment between the substructures from G and G' would be more attractive than an all-pairs comparison. In this spirit, Frohlich [53] defined an optimal assignment kernel on composite objects that include graphs as a special case.

Definition 26 (*Optimal Assignment Kernel*)**.** Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be some non-negative, symmetric and positive definite kernel. Assume that x and x' are two composite objects that have been decomposed into their parts $x := (x_1, x_2, \ldots, x_{|x|})$ and $x' := (x'_1, x'_2, \ldots, x'_{|x|})$. Let $\Pi(x)$ denote all possible permutations of x, and analogously $\Pi(x')$ all possible permutations of x'. Then, $k_A : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is given by

$$k_A(x, x') = \begin{matrix} \max_{\pi \in \Pi(x')} \int_{i=1}^{|x|} k(x_i, x'_{\pi(i)}) & if\ |x'| > |x| \\ \max_{\pi \in \Pi(x)} \int_{i=1}^{|x'|} k(x_{\pi(j)}, x'_j) & otherwise \end{matrix} \tag{56}$$

and is called the optimal assignment kernel.

Table 3
Mean classification accuracies with standard deviation for all experiments, significantly best accuracies in bold. OUT OF MEMORY means that 100 GB memory was not enough. OUT OF TIME indicates that the kernel computation did not finish within 30 days. Runtimes are given in parentheses; x' y'' means x minutes, y seconds.

KERNEL	ENZYMES	PROTEINS	AIRWAYS	SYNTHETIC
GraphHopper	69.6 ± 1.3 (12′10″)	74.1 ± 0.5 (2.8 h)	66.8 ± 0.5 (1 d 7 h)	86.6 ± 1.0 (12′10″)
PROP-diff	37.2 ± 2.2 (13″)	73.3 ± 0.4 (26″)	63.5 ± 0.5 (4′12″)	46.1 ± 1.9 (1′21″)
PROP-WL	48.5 ± 1.3 (1′9″)	73.1 ± 0.8 (2′40″)	61.5 ± 0.6 (8′17″)	44.5 ± 1.2 (1′52″)
SP	71.0 ± 1.3 (3 d)	75.5 ± 0.8 (7.7 d)	OUT OF TIME	85.4 ± 2.1 (3.4 d)
CSM	69.4 ± 0.8	OUT OF MEMORY	OUT OF MEMORY	OUT OF TIME
WL	48.0 ± 0.9 (18″)	75.6 ± 0.5 (2′51″)	62.0 ± 0.6 (7′43″)	43.3 ± 2.3 (2′8″)

Table 4
Classification accuracy (in % ± standard error) and Runtime on unlabeled graph benchmark datasets. RW is the random walk kernel, while SP is the shortest-path kernel. GK Ak m denotes our graphlet kernel computed using m samples of size k graphlets. GK Cn denotes the graphlet kernel computed using all connected graphlets of size k. ">1 day" means computation did not finish within 24 h.

KERNEL	ACCURACY				RUNTIME			
	MUTAG	PTC	Enzymes	D & D	MUTAG	PTC	Enzymes	D & D
RW	71.89 ± 0.66	55.44 ± 0.15	14.97 ± 0.28	>1 day	00:42:03	00:02:39	00:10:45	>1 day
SP	81.28 ± 0.45	55.44 ± 0.61	27.53 ± 0.29	>1 day	00:23:02	00:02:35	00:05:01	>1 day
GK A3 1016	79.70 ± 0.43	55.34 ± 0.33	23.07 ± 0.38	74.92 ± 0.12	00:21:05	00:29:07	00:00:39	00:02:09
GK A3 1154	79.54 ± 0.41	54.90 ± 0.42	24.22 ± 0.37	75.05 ± 0.10	00:23:01	00:042:6	00:00:48	00:02:19
GK A3 4061	80.91 ± 0.40	55.21 ± 0.34	25.45 ± 0.70	75.23 ± 0.13	00:01:18	00:02:39	00:01:51	00:06:35
GK AS 4615	80.21 ± 0.37	55.13 ± 0.37	24.85 ± 0.79	75.44 ± 0.13	00:01:38	00:03:01	00:02:51	00:05:58
GK A3 all	82.11 ± 0.62	55.26 ± 0.39	25.8 ± 0.23	75.41 ± 0.13	00:00:35	00:00:09	00:00:03	00:02:34
GK C3	66.55 ± 0.83	57.08 ± 0.43	19.8 ± 0.21	74.14 ± 0.12	00:00:14	00:00:36	00:00:13	00:02:14
GK A4 1986	79.80 ± 0.38	58.88 ± 0.50	26.93 ± 0.57	74.47 ± 0.17	00:01:39	00:03:02	00:04:20	00:11:35
GK A4 2125	80.69 ± 0.31	58.86 ± 0.53	27.25 ± 0.44	74.50 ± 0.14	00:01:46	00:03:16	00:04:36	00:12:21
GK A4 7942	81.74 ± 0.44	59.25 ± 0.50	27.96 ± 0.40	74.53 ± 0.16	00:06:33	00:12:03	00:16:35	00:42:45
GK A4 8497	81.48 ± 0.38	59.39 ± 0.57	28.06 ± 0.45	74.59 ± 0.16	00:06:57	00:12:49	00:17:38	00:45:36
GK A4 all	82.17 ± 0.58	59.65 ± 0.31	28.95 ± 0.50	74.62 ± 0.12	00:04:38	00:10:08	00:00:49	02:44:59
GK C4	69.00 ± 0.74	58.62 & 0.41	23.61 ± 0.22	75.90 ± 0.10	00:00:26	00:00:09	00:00:04	00:35:22
GK A5 5174	81.62 ± 0.69	58.26 ± 0.47	29.54 ± 0.42	75.11 ± 0.14	00:03:14	00:08:01	00:16:57	0l:29:54
GK A5 5313	81.93 ± 0.71	58.29 ± 0.42	29.52 ± 0.25	75.14 ± 0.14	00:03:18	00:08:06	00:17:03	0l:01:54
GK A5 20696	82.64 ± 0.66	57.88 ± 0.54	29.96 ± 0.52	75.52 ± 0.12	00:08:56	00:18:28	00:42:02	0l:30:18
CK A5 21251	83.27 ± 0.79	58.03 ± 0.42	30.48 ± 0.51	75.35 ± 0.10	00:09:05	00:18:04	00:00:27	02:06:45
CK A5 all	83.50 ± 0.60	58.65 ± 0.40	30.64 ± 0.26	>1 day	00:07:17	16:02:16	20:26:08	>1 day
GK C5	70.94 ± 0.76	56.06 ± 0.46	26.66 ± 0.23	>1 day	00:01:19	00:02:01	00:40:07	>1 day

While based on a nice idea, the optimal assignment kernel is unfortunately not positive definite [5], seriously limiting its use in SVMs and other kernel methods.

Fast neighborhood subgraph pairwise distance kernel. Another simple and elegant idea based on a similar concept was introduced by Costa [54]; they called it Neighborhood Subgraph Pairwise Distance Kernel (NSPDK). A relation $R_{r,d}(A^v, B^u, G)$ between two graphs A^v and B^u rooted at v and u and a graph G were defined; $R_{r,d}$ is true iff both A^v and B^u are isomorphic to $\mathcal{N}_r$, where $\mathcal{N}_r$ refers to a neighborhood of G with a radius r and $\mathcal{D}(v, u) = d$. The kernel is then defined as

$$k_{r,d}(G, G') = \int_{\substack{A_v, B_u \in R^{-1}_{r,d}(G) \\ A'_{v'}, B'_{u'} \in R^{-1}_{r,d}(G')}} \delta(A_v, A'_{v'})\delta(B_u, B'_{u'}) \tag{57}$$

where the exact matching kernel $\delta(x, y)$ is 1 if $x \simeq y$ (i.e. if the graph x is isomorphic to y) and 0 otherwise.

They have weighted their performance and speed against various kernels with similar approaches such as the Graph Fragment Kernel (GFK) introduced in [55], the Weighted Decomposition Kernel (WDK) [56], the Pairwise Maximum Common Subgraphs Kernel (PMCSK) introduced in [57], the Neighborhood Subgraph Kernel (NSK) similar in spirit to the fast kernel presented in [52] and the Pairwise Distance Kernel (PDK), similar to the Equal Length Shortest-Path Kernel [45].

4.2.4. Graph kernels based on subtree patterns

Subtree Pattern Kernels. As an alternative to walk kernels on graphs, graph kernels comparing subtree-patterns were defined in [58]. Intuitively, this kernel considers all pairs of nodes V from G and V' from G' and iteratively compares their neighborhoods. "Subtree-pattern" refers to the fact that this kernel counts subtree-like structures in the two graphs. In contrast to the strict definition of trees, subtree-patterns may include several copies of the same node or edge, hence they are not necessarily isomorphic to subgraphs of G or G' let alone subtrees of G and G' . To be able to regard these patterns as trees, [58] treat copies of identical nodes and edges as if they were distinct.

More formally, let $G(V, E)$ and $G'(V', E')$ be two graphs. The idea of the subtree-pattern kernel $k_{v,v',h}$ is to count pairs of identical subtree-patterns in G and G' with height less than or equal to h, with the first one rooted at $v \in V(G)$ and the second one rooted at $v' \in V(G')$. Now, if $h = 1$ and $L(v) = L'(v')$ we have $k_{v,v',h=1}$; if $h = 1$ and $L(v) \neq L'(v')$ we have $k_{v,v',h=0}$. For $h > 1$, one can compute $k_{v,v',h}$ as follows

1. Let $M_{v,v'}$ be the set of all matchings from the set $\delta(v)$ of neighbors of v to the set $\delta(v')$ of neighbors of v' , i.e.,
$$M_{v,v'} = R \subseteq \delta(v) \times \delta(v') | \forall (v_i, v'_i), (v_j, v'_j) \in R : (v_i = v'_i) \rightarrow (v_j = v'_j) \wedge (\forall (v_k, v'_k) \in R : \mathcal{L}(v_k) = \mathcal{L}'(v'_k)). \tag{58}$$
2. Compute
$$k_{v,v',h} = \lambda_v \lambda_{v'} \int_{R \in M_{v,v'}} \int_{(v,v') \in R} k_{v,v',h-1}. \tag{59}$$

Here λ_v and $\lambda_{v'}$ are positive values smaller than 1 to cause higher trees to have a smaller weight in the overall sum.

Given two graphs $G(V, E)$, $G'(V', E')$, then the subtree-pattern kernel of G and G' is given by

$$k_{tree,h} = \int_{v \in V} \int_{v' \in V'} k_{v,v'}. \tag{60}$$

As the walk kernel, the subtree pattern kernel suffers from tottering. Due to the complex patterns it examines, its runtime is even worse than that of the random walk kernel. It grows exponentially with the height h of the subtree-patterns considered.

Tree Pattern Kernels. While talking about sub-tree based kernels two of the most important things to consider is the size of the trees and how much they branch out. These two factors are heavily connected with the actual performance of any subtree based kernels. Mahé [59] defined two kernels specifically keeping this in mind; graphs were represented using a dictionary of smaller tree patterns whose occurrence in the graph defined the kernel.

Definition 27 (*Tree-pattern*)**.** Let a graph $G = (\mathcal{V}_G, \mathcal{E}_G)$ and a tree $t = (\mathcal{V}_T, \mathcal{E}_T)$, with $\mathcal{V}_t = (n_1, n_2, \ldots, n_{|t|})$. A $|t|$-tuple of vertices $(v_1, \ldots, v_{|t|}) \in \mathcal{V}_G^{|t|}$ is a tree-pattern of G with respect to t, denoted by $(v_1, \ldots, v_{|t|}) = pattern(t)$, iff the following holds

$$\begin{aligned} &\forall i \in [1, |t|], && label(v_i) = label(n_i) \\ &\forall (n_i, n_j) \in \mathcal{E}_t, && (v_i, v_j) \in \mathcal{E}_G \wedge label(v_i, v_j) = label(n_i, n_j) \\ &\forall (n_i, n_j), && (n_i, n_j) \in \mathcal{E}_t, j \neq k \iff v_j \neq v_k. \end{aligned} \tag{61}$$

In other words, a tree-pattern is a combination of graph vertices that can be arranged in a particular tree structure, according to the labels and the connectivity properties of the graph. Note from this definition that vertices of the graph are allowed to appear several times in a tree-pattern, under the condition that sibling nodes of the corresponding tree are associated to distinct vertices of the graphs. We now introduce a function to count occurrences of these patterns.

Definition 28 (*Tree-pattern Counting Function*)**.** A tree-pattern counting function returning the number of times a tree-pattern occurs in a graph is defined for the tree t and the graph $G = (\mathcal{V}_G, \mathcal{E}_G) with vertices \mathcal{V}_G = (v_1, v_2, \ldots, v_{|G|})$ as

$$\begin{aligned} \psi_t(G) &= |(\alpha_1, \ldots, \alpha_{|t|}) \in [1, |\mathcal{V}_G|]^{|t|} : (v_{\alpha_1}, \ldots, v_{\alpha_{|t|}}) \\ &= pattern(t)|. \end{aligned} \tag{62}$$

A restriction to ψ_t to patterns rooted in a specified vertex v is given by

$$\begin{aligned} \psi_t^{(v)}(G) &= |(\alpha_1, \ldots, \alpha_{|t|}) \in [1, |\mathcal{V}_G|]^{|t|} : (v_{\alpha_1}, \ldots, v_{\alpha_{|t|}}) \\ &= pattern(t) \wedge v_{\alpha_1} = v|. \end{aligned} \tag{63}$$

Definition 29 (*Tree-pattern Graph Kernel*)**.** The tree-pattern graph kernel K for the graphs G_1 and G_2 is given by

$$K(G_1, G_2) = \int_{t \in \mathcal{T}} w(t) \psi_t(G_1) \psi_t(G_2) \tag{64}$$

where $\mathcal{T}$ is a set of trees, $w : \mathcal{T} \to \mathbb{R}$ is a tree weighting function and ψ_t is the tree-pattern counting function as shown in the previous definition.

Based on this kernel definition two cases were studied; one is a perfectly depth-balanced tree of order h, i.e. a tree where the depth of each leaf node is h; the other is based on branching cardinality $branch(t)$ which is defined by the number of leaf nodes minus 1.

Definition 30 (*Size-based Balanced Tree-pattern Kernel*)**.** For the pair of graphs G_1 and G_2, the size based balanced tree-pattern kernel of order h is defined as

$$K_{Size}^h(G_1, G_2) = \int_{t \in \mathcal{B}_h} \lambda^{|t|-h} \psi_t(G_1) \psi_t(G_2). \tag{65}$$

Definition 31 (*Branching-based Balanced Tree-pattern Kernel*)**.** For the pair of graphs G_1 and G_2, the branching-based balanced tree-pattern kernel of order h is defined as

$$K_{Size}^h(G_1, G_2) = \int_{t \in \mathcal{B}_h} \lambda^{branch(t)} \psi_t(G_1) \psi_t(G_2). \tag{66}$$

Two extensions were proposed: one by simply removing tottering during subtree computation using both size and branch based formulas, another is until-N extensions to the branch based computation which, instead of taking balanced trees of order h, takes trees till a depth of 10.

Weisfeiler–Lehman graph kernels. The Weisfeiler–Lehman Subtree graph kernel, proposed by Shervashidze [60] is a state-of-the-art, efficient kernel for graph comparison introduced in [61] and elaborated upon in [60]. The kernel computes the number of subtrees shared between two graphs using the Weisfeiler–Lehman test of graph isomorphism; the rewriting procedure underlying the Weisfeiler–Lehman kernel is given in Algorithm 1, which is taken from [60].

Algorithm 3 Weisfeiler–Lehman Relabeling

Input:
graphs $G = (V, E, l)$, $G' = (V', E', l)$ and number of iterations h
Output:
label functions l_0 to l_h
Comments:
$M_n(v)$ are sets of labels for a vertex v and $N(v)$ is the neighborhood of v
– **for** n = 0 **to** h **do**
1. Multiset-label determination
– **for** each $v \in V$ **do**
– **if** $n = 0, M_n(v) = l_0(v) = l(v)$
– **if** $n > 0, M_n(v) = \{l_{n-1}(u) | u \in N(v)\}$
2. Sorting each multiset
– **for** each $M_n(v)$ **do**
– **sort** the elements in $M_n(v)$, in ascending order and **concatenate** them into a string $s_n(v)$
– **for** each $s_n(v)$, **if** $n > 0$, **add** $l_{(n-1)}(v)$ as a prefix to $s_n(v)$
3. Label compression
– **for** each $s_n(v)$**do**
– **sor**t all strings $s_n(v)$ together in ascending order
– **map** each string $s_n(v)$to a new compressed label, using a function $f : \Sigma^* \to \Sigma$, such that $f(s_n(v)) = f(s_n(v'))$ *iff* $s_n(v) = sn(v')$
4. Relabeling
– **for** each $s_n(v)$, **do**
set $l_n(v) = f(s_n(v))$

The idea of the rewriting process is that for each vertex a label multiset based on the labels of the neighbors of the vertex is created. This multiset is sorted and together with the original label concatenated into a string, which is the new label; for each unique string, a new (shorter) label is introduced and this replaces the original vertex label. Note that the sorting of the strings in step 3 is not necessary, but provides a simple way to create the label dictionary f. The rewriting process can be efficiently implemented using counting sort, for which details are given in [60].

Definition 32 (*Weisfeiler–Lehman Graph Kernel*)**.** Let $G_n = (V, E, l_n)$ and $G'_n = (V', E', l_n)$ be the nth iteration rewriting of the graphs G and G' using the Algorithm 3 and h the number of iterations. Then the Weisfeiler–Lehman kernel is defined as

$$k_{WL}^h(G, G') = \int_{n=0}^{h} k_\delta(G_n, G'_n), \tag{67}$$

Table 5
Prediction accuracy (± standard deviation) on graph classification benchmark datasets.

Method/DataSet	MUTAG	NCI 1	NCI 109	ENZYMES	D & D
WL subtree	82.05 (±0.36)	82.19 (±0.18)	82.46 (±0.24)	52.22 (±1.26)	79.78 (±0.36)
WL edge	81.06 (±1.95)	84.37 (±0.30)	84.49 (±0.20)	53.17 (±2.04)	77.95 (±0.70)
WL shortest path	83.78 (±1.46)	84.55 (±0.36)	83.53 (±0.30)	59.05 (±1.05)	79.43 (±0.55)
Ramon & Gartner	85.72 (±0.49)	61.86 (±0.27)	61.67 (±0.21)	13.35 (±0.87)	57.27 (±0.07)
p-random walk	79.19 (±1.09)	58.66 (±0.28)	58.36(±0.94)	27.67 (±0.95)	66.64 (±0.83)
Random walk	80.72 (±0.38)	64.34 (±0.27)	63.51 (±0.18)	21.68 (±0.94)	71.70 (±0.47)
Graphlet count	75.61 (±0.49)	66.00 (±0.07)	66.59 (±0.08)	32.70 (±1.20)	78.59 (±0.12)
Shortest path	87.28 (±0.55)	73.47 (±0.11)	73.07 (±0.11)	41.68 (±1.79)	78.45 (±0.26)

where,

$$k_\delta((V, E, L), (V', E', l')) = \int_{v=V} \int_{v' \in V'} \delta(l(v), l'(v')). \tag{68}$$

Here δ is the Dirac kernel, which tests for equality: 1 if its arguments are equal and 0 otherwise.

Variations to the kernel can be obtained by introducing variations to the base Dirac kernel. Two variations are discussed in their work using edge kernel and shortest path kernel as the base kernel.

They have performed experiments on five datasets namely MUTAG, NCI1, NCI109, ENZYMES, D&D as show in Table 5.

4.3. State of the art approaches

4.3.1. Propagation kernels

In most of the previous kernels we have dealt with completely labeled graphs. However in modern research with increasing size of data we often encounter partially labeled graphs or attributed graphs. Even the mere size of graphs can be overwhelming for the previously mentioned graph kernel algorithms. Hence, to address all this issues Neumann [62] introduced the concept of propagation kernels. Propagation kernels are based on the pattern of how information spreads through a set of graphs. They make use of early-stage distributions from propagation schemes like random walks to model structural information embedded in node labels, attributes and edge properties. This has a couple of benefits; firstly, traditional propagation schemes can be directly used to naturally create kernels for many types of graphs like unlabeled, partially labeled, labeled, directed, as well as attributed graphs; secondly, by exploiting existing efficient and informative propagation schemes, propagation kernels can be significantly faster than modern approaches without sacrificing their predictive power. It was also shown that if the graphs in question had a regular structure, for example, with image or video data, this regularity can be exploited to scale the kernel computation to large databases of graphs with very high number of nodes.

Before defining a propagation kernel let us first define a general similarity function (a positive semidefinite co-variance function) $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, for graph instances $G^{(i)} = (V^{(i)}, E^{(i)}, L^{(i)}) \in \mathcal{X}$, where $V^{(i)}, E^{(i)}, L^{(i)}$ represents a set of nodes, edges and a label distribution for graph i respectively. Also, graph G_i has n_i nodes labeled with one of the k labels. Propagation kernels can be computed through an iterative procedure as follows.

1. Counting common node label distribution. First, let us generate feature vectors $\phi(G_t^{(i)})$ for every graph by counting common label distributions induced over the nodes among the respective graphs. Therefore, each node in every graph is placed into one of the "bins", each one which collecting similar label distributions, and these vectors count the nodes in each of those bins for each graph.
2. Calculating current kernel contribution. Given these vectors, for each pair of graphs $G^{(i)}$ and $G^{(j)}$, we calculate

$$k(G_t^{(i)}, G_t^{(j)}) = \phi(G_t^{(i)}), \phi(G_t^{(j)}) \tag{69}$$

where $\langle .,. \rangle$ signifies an arbitrary base kernel. This value will contribute to the final kernel value between these graphs. Hence, the T-iteration propagation kernel between two graphs $G^{(i)}$ and $G^{(j)}$ can be defined as

$$K_T(G^{(i)}, G^{(j)}) = \int_{t=0}^{T} k(G_t^{(i)}, G_t^{(j)}). \tag{70}$$

3. Propagating node label distributions. Finally, iteratively update the node label distributions.

$$L_t^{(i)} \to L_{t+1}^{(i)}. \tag{71}$$

The exact choice of scheme for this update results in different types of propagation kernels.

One of the most common techniques for label propagation was the process of diffusion which stated

$$L_t^{(i)} \to T^{(i)} L_{t+1}^{(i)} \tag{72}$$

where $T^{(i)}$ is a transition matrix, for which $T_{mn}^{(i)}$ contains a weight value showing the probability of label transference from n to m. For partially labeled graphs, unlabeled nodes can be initialized through a uniform label distribution. One drawback of such propagation scheme is that the originally labeled node loses its purity over time. To tackle the issue originally labeled data points may be reset after each iteration thus ensuring the provided labels are pushed back during each iteration of the diffusion process.

As shown below in Table 6, experimentations were performed on the MUTAG, NCI, NCI109 and D&D dataset. They competed with Weisfeiler–Lehman, Shortest Path and Graph-hopper kernels.

Further analysis, as demonstrated in Table 7, was made using partially labeled data which was obtained by omitting labels from MSRC9 and MSRC21 Dataset.

4.3.2. Deep graph kernels

R-convolution is the generic framework for handling discrete objects. The key idea is to recursively decompose structured objects into their "atomic" sub-structures and to define valid local kernels among them. In the case of graphs, for a graph G, let $\phi(G)$ denote a vector which contains counts of such atomic substructures or graphlets, and $\langle .,. \rangle_{\mathcal{H}}$ denote a dot product in a Hilbert space $\mathcal{H}$. Then, the kernel between two graphs G and G' can be written as

$$k(G, G') = \langle \phi(G), \phi(G') \rangle. \tag{73}$$

We have previously discussed how kernel functions behave like similarity functions, hence diagonal dominance in a kernel function is a desirable quality. However, the above definition fails to consider some important factors like interdependence among graphlets, or exponential increase in the number of graphlets with size of the graph. More importantly it does not show diagonal dominance in the kernel matrix thus proving that it fails to interpret the similarity within a class and the dissimilarity across them. An alternative kernel was proposed Yanardag [63] which modified the

Table 6
Performance analysis of propagation kernels: Accuracy and standard deviation (average runtime for each of the 10-folds of cross validation).

METHOD	MUTAG	NCII	NCII09	D&D
PK	84.5 ± 0.6 (0.2″)	84.5 ± 0.1 (4.5′)	83.5 ± 0.1 (4.4′)	78.8 ± 0.2 (3.6′)
WL	84.0 ± 0.4 (0.2″)	85.9 ± 0.1 (5.6′)	85.9 *0.1 (7.4′)	79.0 ± 0.2 (6.7′)
SP	85.8 ± 0.2 (0.2″)	74.4 ± 0.1 (21.3″)	73.7 ± 0.0 (19.3″)	OUT OF TIME
GH	85.4 ± 0.5 (1.0′)	73.2 ± 0.1 (13.0 h)	72.6 ± 0.1 (22.1 h)	68.9 ± 0.2 (69.1 h)

Table 7
Performance analysis of propagation kernels: Accuracy and standard deviation with respect to missing labels.

Dataset	Method	Labels missing			
		20%	40%	60%	80%
MSRC9	PL	90.0 ± 0.4	88.7 ± 0.3	86.6 ± 0.4	80.4 ± 0.6
	LP+WL	90.0 ± 0.2	87.9 ± 0.6	83.2 ± 0.6	77.9 ±1.0
	WL	89.2 ± 0.5	88.1 ± 0.5	85.7 ± 0.6	78.5 ± 0.9
MSRC21	PL	86.9 ± 0.3	84.7 ± 0.3	79.5 ± 0.3	69.3 ± 0.3
	LP+WL	85.8 ± 0.2	81.5 ± 0.2	74.5 ± 0.3	64.0 ± 0.4
	WL	85.4 ± 0.4	81.9 ± 0.4	76.0 ± 0.3	63.7 ± 0.4

above equation (79) by introducing a similarity matrix $\mathcal{M}$ in the equation

$$k(G, G') = \phi(G)^T \mathcal{M} \phi(G') \tag{74}$$

where, $\mathcal{M}$ is a positive semi-definite matrix that encodes the relationships between sub-structures. These similarity values may be calculated using simple techniques like graph edit distances or can even be learned through standard clustering techniques. The learning technique proposed by Yanardag [63] was based on neural language models such as continuous bag of words or skip gram modeling. They are typical word embedding models that are used for word prediction algorithms. The analogy that fuels this idea is that sub-structures or graphlets can be treated in the same way as words that make up the entire graph like a sequence of words make a sentence.

Experiments were performed by comparing existing graphlet kernels with their deep variants on MUTAG, PTC, ENZYMES, PROTEINS, NCI1, NCI109 datasets. Their performance can be seen in Table 8 where we can see that the deep variants of the kernels worked better in every case.

4.4. Choosing the right kernel

Throughout the last two decades various kernels surfaced in the field of graph based algorithms as it can be seen in Fig. 13. However, the application of a graph kernel to a specific domain requires the understanding of the advantages and disadvantages of the various available techniques. While earlier versions of random walk kernels prove to be quite efficient for simple graphs, as the size and complexity increases we need to shift to other modern techniques. Normally complexity of random walk kernels are in the order of $O(n^6)$, however faster computations up to $O(n^3)$ were obtained by extending concepts of linear algebra to Reproducing Kernel Hilbert Spaces [47]; even further speedups up to $O(n^2)$ were obtained by Kang [43] considering rank approximation of the adjacency matrix.

Graph Edit Distance Kernels [16] come quite handy when its easy to compute the minimum edit distance between two graphs. Walk based Kernel can also be visualized through the perspective of probability of node sequences through marginalized kernels [38]. One of the major issues of walk based kernels is tottering; tottering refers to repeated walks through same nodes and tt can drastically affect the similarity values. Extensions of marginalized kernels [44] can handle tottering to great extent; another way to avoid tottering is to incorporate path based kernels. Paths by definition do not pass through the same node more than once.

With the increase in size and complexity of graphs it is necessary to move onto newer techniques. Sub-graph [51,53,61] and Sub-tree kernels [58–60] deal specifically with large complex graphs. The similarity computation is based on the assumption that graphs with similar sub-structures will tend to have greater similarity in properties. Before using such kernels one needs to ensure that semantic information do not propagate over large distances throughout the graph. In other words, if nodes that are far apart in the graph have huge semantic significance, subgraph or subtree based kernels might not be able to capture the similarity accurately.

Many of the above mentioned kernels depend on labels assigned to nodes and edges. However, sometimes graphs may not possess all the labels. Propagation Kernels [62] work wonderfully for such problems. Their label propagations schemes can actually handle missing labels. However, labels in such cases should have adjacent dependencies; if adjacent labels change drastically then label propagation schemes might not be able to predict the missing labels properly.

Kernel Functions by nature demonstrate the similarity between two samples. Hence the diagonal dominance of the kernel matrix is a very desirable quality. Deep Graph Kernels [63] significantly boost the diagonal dominance by introducing a similarity matrix in the computation of traditional kernels.

From the previous paragraphs we can understand that various graph kernels address various problems; it is not easy to pick the best performing graph kernel without taking into account the type of the graph we are dealing with. Various properties of graphs can be observed to decide the appropriate kernel for the problem. Among them most notable properties are, size of the graph, type of labels, amount of information propagation across large lengths, chance of tottering, etc.

4.5. Future of graph kernels

Graph kernels show considerable amount of promise in the future. As discussed before, many real world scenarios or events can be conceptualized through graphs. Most scientific fields show increasing use of structured data. From nuclear studies in CERN,[3] to decoding our own genome,[4] we have been generating tremendous amount of structured information. Through internet we have access to an immensely large network connecting almost 3.78 billion[5] people across the world. These huge sources of structured information must be processed to extract what is essential for the progress of science. Structured data provides a different level of challenge to traditional artificial approaches because of the complexity of information. But with graph kernels it is possible handle such challenges. One of the most significant progress in machine learning is the onset of deep learning [64]. Unlike previous methods, deep learning algorithms can work with raw data

[3] home.cern/.

[4] https://www.genome.gov/12011238/an-overview-of-the-human-genome-project/.

[5] www.internetlivestats.com/.

Table 8
Comparison of classification accuracy (± standard deviation) of the graphlet kernel (GK), shortest-path kernel (SP), Weisfeiler–Lehman kernel (WL) to their deep variants on the datasets.

Dataset	GK	Deep GK	SP	Deep SP	WL	Deep WL
MUTAG	81.66 ± 2.11	82.66 ± 1.45	85.22 ± 2.43	87.44 ± 2.72	80.72 ± 3.00	82.94 ± 2.68
FTC	57.26 ± 1.41	57.32 ± 1.13	58.24 ± 2.44	60.08 ± 235	56.97 ± 2.01	59.17 ± 1.56
ENZYMES	26.61 ± 0.99	27.08 ± 0.79	40.10 ± 1.50	41.65 ± 1.57	53.15 ± 1.14	53.43 ± 0.91
PROTEINS	71.67 ± 0.55	71.68 ± 0.50	75.07 ± 0.54	75.68 ± 0.54	72.92 ± 0.56	73.30 ± 0.82
NCI I	62.28 ± 0.29	62.48 ± 0.25	73.00 ± 0.24	73.55 ± 0.51	80.13 ± 0.50	80.31 ± 0.46
NCII09	62.60 ± 0.19	62.69 ± 0.23	73.00 ± 021	73.26 ± 0.26	80.22 ± 0.34	80.32 ± 0.33

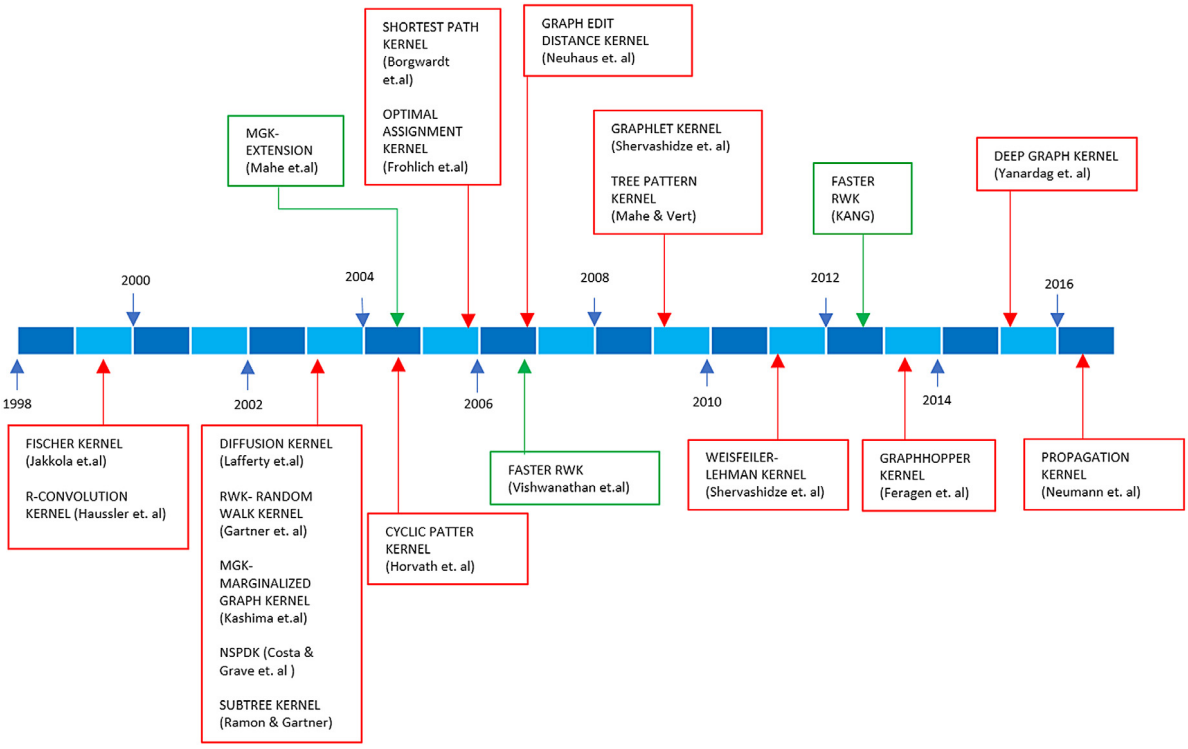

Fig. 13. A time-line of all discussed graph kernels, Red Box denotes first occurrence of the kernel while green boxes denote upgrade on existing kernels. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

rather than manually extracted features. Though it has proved quite efficient for classification or regression problems, yet its only starting to scratch the surface of structured or relational information [65,66]. Deep Learning algorithm thrives on the availability of huge amount of raw data, while graph kernels can process relational or structured information quite efficiently. A combination of these strengths can really boost the progress in research related to structured data. Just like inference models, graph based knowledge representation models [67] can also use these techniques to affect various domains. When Google developed their knowledge graph [68] they set a new standard for search engine optimization. While deep learning is promising in non-structured or raw data, graph kernels can equally contribute in the relational aspect of artificial intelligence. A comprehensive topics that currently use graph kernels is summarized in Section 4.6. Recent advances in hardwares, specifically GPUs(Graphics processing units) have allowed the slow algorithms of deep learning to run at a lightning space. GPUs provide thousands of computational cores to enable parallel computation. However GPUs are mostly limited to matrix based operations. But graph kernels mostly deal with matrix operations as well, and hence can benefit more by using the CUDA computational cores to its full potential. We definitely need more sophisticated APIs that allows us to integrate graph kernels into CUDA based operations. Also to aid in the process of boosting graph kernels to its full potential we have a plethora of graph representational formats[6] that come with their own set of pros and cons. Overall graph kernel can prove to be the one of the major steps in the domain of machine learning with structured data.

4.6. Applications

Having studied the theoretical aspects of graph kernel, it is essential to know about the possible domains where they can be applied. In the subsections below, some of the most prominent application areas will concisely discussed.

4.6.1. Chemoinformatics

Traditionally, graphs have been used to model molecular compounds in chemistry [69]. Chemoinformatics aims at predicting characteristics of molecules from their graph structures, e.g. toxicity, or effectiveness as a drug. Most traditional benchmark datasets for graph mining algorithms originate from this domain, including MUTAG [70] and PTC [71].

[6] en.wikipedia.org/wiki/Graph_database#List_of_graph_databases.

4.6.2. Bioinformatics

A major reason for the growing interest in graph-structured data is the advent of large volumes of structured data in molecular biology. This structured data comprises graph models of molecular structures, from RNA to proteins [72], and of networks which include protein–protein interaction networks [73], metabolic networks [74], regulatory networks [75], and phylogenetic networks [76]. Bioinformatics seeks to establish the function of these networks and structures. Currently, the most successful approach towards function prediction of structures is based on similarity search among structures with known function. For instance, if we want to predict the function of a new protein structure, we compare its structure to a database of functionally annotated protein structures. The protein is then predicted to exert the function of the (group of) protein(s) which it is most similar to. This concept is supported by models of evolution: proteins that have similar topological structures are more likely to share a common ancestor, and are more likely to carry out the same biochemical function [77].

4.6.3. Social network analysis

Another important source of graph structured data is social network analysis [78]. In social networks, nodes represent individuals and edges represent interaction between them. The analysis of these networks is both of scientific and commercial interest. On the one hand, psychologists want to study the complex social dynamics between humans, and biologists want to uncover the social rules in a group of animals. On the other hand, industries want to analyze these networks for marketing purposes. Detecting influential individuals in a group of people, often referred to as 'key-players' or 'trend-setters', is relevant for marketing, as companies could then focus their advertising efforts on persons known to influence the behavior of a larger group of people. In addition, telecommunication and Internet surfing logs provide a vast source of social networks, which can be used for mining tasks ranging from telecommunication network optimization to automated recommender systems.

4.6.4. Internet, HTML, XML

A fourth application area for graph models is the Internet which is a network and hence a graph itself. HTML documents are nodes in this network, and hyperlinks connect these nodes. In fact, Google exploits this link structure of the Internet in its famous PageRank algorithm [79] for ranking websites. Furthermore, semi-structured data in form of XML documents is becoming very popular in the database community and in industry. The natural mathematical structure to describe semi-structured data is a graph. As the W3 Consortium puts it "The main structure of an XML document is tree-like, and most of the lexical structure is devoted to defining that tree, but there is also a way to make connections between arbitrary nodes in a tree".[7] Consequently, XML documents should be regarded as graphs.

Various tasks of data manipulation and data analysis can be performed on this graph representation, ranging from basic operations such as querying [80] to advanced problems such as duplicate detection [81].

4.6.5. Natural language processing

Language in general possesses a very generic entity relationship structure, hence graph based algorithms have always been used in this field [82–84]. Tree kernels are particularly common [85] in NLP because of parse tree representations of the language which are very precise to deal with common concepts.

[7] https://www.w3.org/News/2005.

4.6.6. Image processing

Images can also be treated as graphs if we consider similar components in an image as node and their semantic relations as edges. A lot of work has been seen in this area: earlier, image classification was performed using methods like graph edit distance [16] or marginalized kernels [86] later much sophisticated approaches that involved point clouds [87] or segmentation graphs [88] has also been seen.

5. Conclusion

Graph Kernels is a beautiful concept that has immense future prospect. It can deal with problems of numerous domains and unlike many other learning techniques, it excels in learning relational models. Information around us, no matter from which source, can somehow be expressed in terms of graphs through their inherent semantic dependencies, thus making graph kernels a powerful tool for researchers.

We started from the absolute basics of Linear algebra and graph theory, and slowly climbed up the ladder to learn how kernels operate in the field of machine learning. Starting from the historical aspects of the domain, we saw that graph kernels can be computed through various walk-based, path-based, subgraph-based or subtree based techniques. Finally, we saw how modern researchers have been successful in creating very fast techniques that can be used for larger graphs. With all this research, graph kernel is at its pinnacle of development. Now is the time to dive into even more challenging problems with these techniques.

Acknowledgments

This work is partially supported by the project entitled "Development of knowledge graph from images using deep learning" sponsored by SERB (Government of India, order no. SB/S3/EECE/054/2016) (dated 25/11/2016), and carried out at the Centre for Microprocessor Application for Training Education and Research, CSE Department, Jadavpur University.

References

[1] R.R. Dipert, The mathematical structure of the world: The world as graph, J. Philos. 94 (7) (1997) 329–358.
[2] H. Bunke, Graph-based tools for data mining and machine learning, in: International Workshop on Machine Learning and Data Mining in Pattern Recognition, Springer, 2003, pp. 7–19.
[3] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, Int. J. Pattern Recognit. Artif. Intell. 18 (03) (2004) 265–298.
[4] T. Gärtner, A survey of kernels for structured data, ACM SIGKDD Explor. Newslett. 5 (1) (2003) 49–58.
[5] S.V.N. Vishwanathan, N.N. Schraudolph, R. Kondor, K.M. Borgwardt, Graph kernels, J. Mach. Learn. Res. 11 (Apr) (2010) 1201–1242.
[6] N. Shervashidze, Scalable Graph Kernels (Ph.D. thesis), Universität Tübingen, 2012.
[7] K.M. Borgwardt, Graph Kernels (Ph.D. thesis), lmu, 2007.
[8] N. Aronszajn, Theory of reproducing kernels, Trans. Amer. Math. Soc. 68 (3) (1950) 337–404.
[9] B. Scholkopf, A.J. Smola, Learning With Kernels: Support Vector Machines, Regularization, Optimization, and Beyond, 2002, arXiv: arXiv:1011.1669v3, http://dx.doi.org/10.1198/jasa.2003.s269.
[10] I. Tsochantaridis, T. Joachims, T. Hofmann, Y. Altun, A.-C. Org, Large margin methods for structured and interdependent output variables, J. Mach. Learn. Res. 6 (2005) 1453–1484. http://dx.doi.org/10.1007/s10994-008-5071-9.
[11] V. Vapnik, a. Lerner, Pattern recognition using generalized portrait method, Autom. Remote Control 24 (1963) 774–780 doi:citeulike-article-id:619639.
[12] H. Bunke, Graph matching : Theoretical foundations, algorithms, and applications, Algorithmica 2000 (2) (2000) 82–88.
[13] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, M. Vento, A comparison of algorithms for maximum common subgraph on randomly connected graphs, in: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), Springer, 2002, pp. 123–132.

[14] M.R. Garey, D.S. Johnson, A Guide to the Theory of NP-Completeness, 1979, http://dx.doi.org/10.1137/1024022.
[15] B. McKay, Nauty User's Guide (version 2.4), Computer Science Dept., Australian National University, 2007, pp. 1–70.
[16] M. Neuhaus, H. Bunke, Edit distance-based kernel functions for structural pattern classification, Pattern Recognit. 39 (10) (2006) 1852–1863. http://dx.doi.org/10.1016/j.patcog.2006.04.012.
[17] H. Bunke, K. Shearer, A graph distance metric based on the maximal common subgraph, Pattern Recognit. Lett. 19 (3) (1998) 255–259.
[18] M.L. Fernández, G. Valiente, A graph distance metric combining maximum common subgraph and minimum common supergraph, Pattern Recognit. Lett. 22 (6–7) (2001) 753–758. http://dx.doi.org/10.1016/S0167-8655(01)00017-4.
[19] I. Koch, Enumerating all connected maximal common subgraphs in two graphs, Theoret. Comput. Sci. 250 (12) (2001) 1–30. http://dx.doi.org/10.1016/S0304-3975(00)00286-3.
[20] C. Bron, J. Kerbosch, Algorithm 457: Finding all cliques of an undirected graph, Commun. ACM 16 (9) (1973) 575–577. http://dx.doi.org/10.1145/362342.362367. arXiv:citation.cfm?doid=362342.362367.
[21] H. Liang, W.F. Ward, PGC-1alpha: a key regulator of energy metabolism, Adv. Physiol. Educ. 30 (4) (2006) 145–151. http://dx.doi.org/10.1152/advan.00052.2006.
[22] H. Bunke, G. Allermann, Inexact graph matching for structural pattern recognition, Pattern Recognit. Lett. 1 (4) (1983) 245–253. http://dx.doi.org/10.1016/0167-8655(83)90033-8.
[23] H. Bunke, Recognition of cursive roman handwriting - past, present and future, in: Proceedings of the International Conference on Document Analysis and Recognition, ICDAR, Vol. 2003-Janua, 2003, pp. 448–459, http://dx.doi.org/10.1109/ICDAR.2003.1227707.
[24] F.R.K. Chung, F. Chung-Graham, F.R.K. Chung, F. Chung-Graham, F.R.K. Chung, Spectral graph theory, ACM SIGACT News 92 (92) (1997) 14. http://dx.doi.org/10.1145/568547.568553.
[25] R. Todeschini, V. Consonni, Handbook of Molecular Descriptors, New York, Vol. 11, 2000, p. 688, http://dx.doi.org/10.1002/9783527613106.
[26] H. Wiener, Correlation of heats of isomerization, and differences in heats of vaporization of isomers, among the paraffin hydrocarbons, J. Amer. Chem. Soc. 69 (11) (1947) 2636–2638.
[27] J. Köbler, On Graph Isomorphism for Restricted Graph Classes, Springer, Berlin Heidelberg, 2006, pp. 241–256.
[28] M. Neuhaus, H. Bunke, Self-organizing maps for learning the edit costs in graph matching, IEEE Trans. Syst. Man Cybern. B 35 (3) (2005) 503–514. http://dx.doi.org/10.1109/TSMCB.2005.846635.
[29] M. Neuhaus, H. Bunke, Automatic learning of cost functions for graph edit distance, Inform. Sci. 177 (1) (2007) 239–247. http://dx.doi.org/10.1016/j.ins.2006.02.013.
[30] T. Caelli, T. Caetano, Graphical models for graph matching: Approximate models and optimal algorithms, Pattern Recognit. Lett. 26 (3) (2005) 339–346. http://dx.doi.org/10.1016/j.patrec.2004.10.022.
[31] S. Kramer, L. De Raedt, C. Helma, Molecular feature mining in HIV data, in: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '01, 2001, pp. 136–143, http://dx.doi.org/10.1145/502512.502533.
[32] M. Deshpande, M. Kuramochi, N. Wale, G. Karypis, Frequent substructure-based approaches for classifying chemical compounds, IEEE Trans. Knowl. Data Eng. 17 (8) (2005) 1036–1050. http://dx.doi.org/10.1109/TKDE.2005.127.
[33] H. Cheng, X. Yan, J. Han, C.-W. Hsu, Discriminative frequent pattern analysis for effective classification, in: Proceedings of the 23rd IEEE International Conference on Data Engineering, ICDE 2007, 2007, pp. 716–725, http://dx.doi.org/10.1109/ICDE.2007.367917.
[34] T. Jaakkola, M. Diekhans, D. Haussler, Using the Fisher kernel method to detect remote protein homologies, in: International Conference on Intelligent Systems for Molecular Biology, ISMB, 1999, pp. 149–158.
[35] S.R. Eddy, Hidden markov models, Curr. Opin. Struct. Biol. 6 (3) (1996) 361–365.
[36] T. Jaakkola, D. Haussler, Probabilistic kernel regression models, in: Proceedings of the 1999 Conference on AI and Statistics, 1999, p. 9.
[37] T.S. Jaakkola, D. Haussler, et al., Exploiting generative models in discriminative classifiers, Adv. Neural Inf. Process. Syst. (1999) 487–493.
[38] A. Kashima, Hisashi Tsunda, Koji Inokuchi, Marginalized Kernels between labeled graphs, in: ICML, (2002), 2003, pp. 321–328.
[39] J. Lafferty, J. Lafferty, G. Lebanon, G. Lebanon, Information diffusion kernels, Adv. Neural Inf. Process. Syst. (2003) 391–398.
[40] D. Haussler, Convolution Kernels on discrete structures, in: Technical Report UCSCRL9910 UC, 23 (1), 1999, 1–38, arXiv: arXiv:1403.8129v1, http://dx.doi.org/10.1007/s10863-011-9338-7.
[41] T. Gärtner, A survey of kernels for structured data, ACM SIGKDD Explor. Newslett. 5 (1) (2003) 49. http://dx.doi.org/10.1145/959242.959248.
[42] T. Gärtner, P. Flach, S. Wrobel, T. Gärtner, On graph Kernels: Hardness results and efficient alternatives, in: Proceedings of the 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop, 2003, pp. 129–143, http://dx.doi.org/10.1007/b12006.
[43] U. Kang, H. Tong, J. Sun, Fast random walk graph Kernel, in: Proceedings of the 2012 SIAM International Conference on Data Mining, 2012, pp. 828–838, http://dx.doi.org/10.1137/1.9781611972825.71.
[44] P. Mahé, N. Ueda, T. Akutsu, J.-L. Perret, J.-P. Vert, Extensions of marginalized graph kernels, in: Proceedings, Twenty-First International Conference on Machine Learning, ICML 2004, 2004, pp. 552–559, arXiv: arXiv:1011.1669v3, http://dx.doi.org/10.1145/1015330.1015446.
[45] K.M. Borgwardt, H.P. Kriegel, Shortest-path kernels on graphs, in: Proceedings - IEEE International Conference on Data Mining, ICDM, 2005, pp. 74–81, http://dx.doi.org/10.1109/ICDM.2005.132.
[46] W. Imrich, S. Klavzar, Product Graphs, Wiley, 2000.
[47] J. Nocedal, S.J. Wright, Numerical Optimization, Vol. 43, 1999, pp. 164–175. http://dx.doi.org/10.1002/lsm.21040. arXiv:NIHMS150003.
[48] S. Vishwanathan, N. Schraudolph, R. Kondor, K. Borgwardt, Graph Kenrels, J. Mach. Learn. Res. 11 (2010) 1201–1242. arXiv:0807.0093.
[49] G.H. Golub, C.F. Van Loan, Matrix Computations, 1996. http://dx.doi.org/10.1063/1.3060478. arXiv: arXiv:1011.1669v3.
[50] A. Feragen, N. Kasenburg, J. Petersen, M. de Bruijne, K. Borgwardt, Scalable kernels for graphs with continuous attributes, in: Advances in Neural Information Processing Systems, 2013, pp. 216–224.
[51] T. Horváth, T. Gärtner, 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, 2004, p. 158, http://dx.doi.org/10.1145/1014052.1014072.
[52] N. Shervashidze, S.V.N. Vishwanathan, T.H. Petri, K. Mehlhorn, K.M. Borgwardt, Efficient graphlet Kernels for large graph comparison, in: Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, 2009, pp. 488–495, http://dx.doi.org/10.1.1.165.7842.
[53] H. Frohlich, J.K. Wegner, F. Sieker, A. Zell, H. Fröhlich, J.K. Wegner, F. Sieker, A. Zell, Optimal assignment Kernels for attributed molecular graphs, in: Proceedings of the 22nd International Conference on Machine Learning, 2005, pp. 225–232, http://dx.doi.org/10.1145/1102351.1102380.
[54] F. Costa, K.D. Grave, Fast neighborhood subgraph pairwise distance Kernel, Computer 54 (v) (2003) 255–262. http://dx.doi.org/10.1016/j.neuropharm.2007.07.003.
[55] N. Wale, G. Karypis, Comparison of descriptor spaces for chemical compound retrieval and classification, in: Proceedings - IEEE International Conference on Data Mining, ICDM, 2006, pp. 678–689, http://dx.doi.org/10.1109/ICDM.2006.39.
[56] S. Menchetti, F. Costa, P. Frasconi, Weighted decomposition kernels, in: Proceedings of the 22nd International Conference on Machine Learning, 2005, pp. 585–592, http://dx.doi.org/10.1145/1102351.1102425.
[57] L. Schietgat, J. Ramon, M. Bruynooghe, H. Blockeel, An Efficiently Computable Graph-Based Metric for the Classification of Small Molecules, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 5255, 2008, pp. 197–209. http://dx.doi.org/10.1007/978-3-540-88411-8-20. LNAI.
[58] J. Ramon, T. Gärtner, Expressivity versus efficiency of graph kernels, in: Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences, 2003, pp. 65–74.
[59] P. Mahé, J.P. Vert, Graph kernels based on tree patterns for molecules, Mach. Learn. 75 (1) (2009) 3–35. http://dx.doi.org/10.1007/s10994-008-5086-2. arXiv:0609024v1.
[60] N. Shervashidze, P. Schweitzer, V. Leeuwen, E. Jan, K. Mehlhorn, K. Borgwardt, Weisfeiler-Lehman graph kernels, J. Mach. Learn. Res. 12 (2011) 2539–2561.
[61] N. Shervashidze, K.M. Borgwardt, Fast subtree kernels on graphs, in: 23rd Annual Conference on Neural Information Processing Systems, 2009, pp. 1660–1668.
[62] M. Neumann, R. Garnett, C. Bauckhage, K. Kersting, Propagation kernels: efficient graph kernels from propagated information, Mach. Learn. 102 (2) (2016) 209–245. http://dx.doi.org/10.1007/s10994-015-5517-9.
[63] P. Yanardag, W. Lafayette, S.V.N. Vishwanathan, Deep graph kernels, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2015, pp. 1365–1374, http://dx.doi.org/10.1145/2783258.2783417.
[64] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Netw. 61 (2015) 85–117.
[65] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, S. Jaiswal, graph2vec: Learning distributed representations of graphs, 2017, arXiv preprint arXiv:1707.05005.
[66] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 2016, arXiv preprint arXiv:1609.02907.
[67] J.F. Sowa, Principles of Semantic Networks: Explorations in the Representation of Knowledge, Morgan Kaufmann, 2014.
[68] A. Singhal, Introducing the knowledge graph: things, not strings, Official Google Blog, 2012.
[69] J. Gasteiger, T. Engel, Chemoinformatics: A textbook, in: Computational, 2003, p. 680. http://dx.doi.org/10.1002/3527601643.

[70] A.K. Debnath, R.L. Lopez de Compadre, G. Debnath, A.J. Shusterman, C. Hansch, Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity., J. Med. Chem. 34 (2) (1991) 786–797. http://dx.doi.org/10.1021/jm00106a046.

[71] H. Toivonen, A. Srinivasan, R.D. King, S. Kramer, C. Helma, Statistical evaluation of the predictive toxicology challenge 2000-2001, Bioinformatics 19 (10) (2003) 1183–1193. http://dx.doi.org/10.1093/bioinformatics/btg130.

[72] H.M. Berman, The protein data bank, Nucleic Acids Res. 28 (1) (2000) 235–242. http://dx.doi.org/10.1093/nar/28.1.235.

[73] I. Xenarios, L. Salwinski, X.J. Duan, P. Higney, S.-M. Kim, D. Eisenberg, DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions, Nucleic Acids Res. 30 (1) (2002) 303–305.

[74] M. Kanehisa, Chapter: The KEGG database, Nature Biotechnol. 22 (8) (2004) 1017–1019. http://dx.doi.org/10.1038/nbt991.

[75] E.H. Davidson, J.P. Rast, P. Oliveri, A. Ransick, C. Calestani, C.-H. Yuh, T. Minokawa, G. Amore, V. Hinman, C. Arenas-Mena, O. Otim, C.T. Brown, C.B. Livi, P.Y. Lee, R. Revilla, A.G. Rust, Z.J. Pan, M.J. Schilstra, P.J.C. Clarke, M.I. Arnone, L. Rowen, R.A. Cameron, D.R. McClay, L. Hood, H. Bolouri, A genomic regulatory network for development, Science 295 (5560) (2002) 1669–1678. http://dx.doi.org/10.1126/science.1069883.

[76] D.H. Huson, D. Bryant, Application of phylogenetic networks in evolutionary studies, Mol. Biol. Evol. 23 (2) (2006) 254–267. http://dx.doi.org/10.1093/molbev/msj030.

[77] J.C. Whisstock, A.M. Lesk, Prediction of protein function from protein sequence and structure, Q. Rev. Biophys. 36 (3) (2003) 307–340. http://dx.doi.org/10.1017/S0033583503003901.

[78] S. Wasserman, K. Faust, Social network analysis : methods and applications, American Ethnologist, Vol. 24, 1994, pp. 219–220. http://dx.doi.org/10.1525/ae.1997.24.1.219. arXiv: arXiv:1011.1669v3.

[79] L. Page, Method for node ranking in a linked database, US Patent 7058628, 1 (12), 1998, pp. 1–13, arXiv: arXiv:1208.5721, http://dx.doi.org/10.1016/j.(73).

[80] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, A query language for XML, Comput. Netw. 31 (11–16) (1999) 1155–1169. http://dx.doi.org/10.1016/S1389-1286(99)00020-1.

[81] M. Weis, F. Naumann, Detecting duplicates in complex XML data, in: Proceedings - International Conference on Data Engineering, Vol. 2006, 2006, p. 109, http://dx.doi.org/10.1109/ICDE.2006.49.

[82] M. Collins, N. Duffy, Convolution Kernels for natural language, in: Proceedings of Neural Information Processing Systems 2001 - Advances in Neural Information Processing Systems 14, 2002, pp. 625–632, http://dx.doi.org/10.1.1.19.8980.

[83] N. Das, S. Ghosh, T. Gonçalves, P. Quaresma, Comparison of different graph distance metrics for semantic text based classification, Polibits (49) (2014) 51–58.

[84] N. Das, S. Ghosh, T. Gonçalves, P. Quaresma, Using graphs and semantic information to improve text classifiers, in: International Conference on Natural Language Processing, Springer, 2014, pp. 324–336.

[85] D. Beck, T. Cohn, C. Hardmeier, L. Specia, Learning structural kernels for natural language processing, Trans. Assoc. Comput. Linguist. (ISSN: 2307-387X) 3 (2015) 461–473. https://transacl.org/ojs/index.php/tacl/article/view/584.

[86] E. Aldea, J. Atif, I. Bloch, Image classification using marginalized kernels for graphs, in: Graph-Based Representations in Pattern Recognition, 2007, pp. 103–113.

[87] F.R. Bach, Graph kernels between point clouds, in: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 25–32, arXiv: arXiv:0712.3402v1, http://dx.doi.org/10.1145/1390156.1390160.

[88] Z. Harchaoui, F. Bach, Image classification with segmentation graph kernels, in: IEEE Conference on Computer Vision and Pattern Recognition, 2007, CVPR'07, IEEE, 2007, pp. 1–8.