Eötvös Loránd University

Faculty of Science

Institute of Mathematics

Computer Science Department

Whole Graph Embedding Methods and Their Performances

Lili Kata Szakács

Applied Mathematics MSc – Computer Science specialization

Internal supervisor:
dr. András Benczúr
Adjunct (ELTE TTK),
Head of Informatics Research
Laboratory, ELKH SZTAKI

External supervisor:
dr. Ferenc Béres
Research Fellow
ELKH SZTAKI

Budapest, 2023

Contents

Acknowledgements

I would like to thank my supervisors for finding a motivating and well-suited topic for my thesis work, as well as their flexibility, guidance and support throughout the process. I am grateful to *Ferenc Béres* for introducing me to the practical aspects of data science and his advices on effective coding. I extend my gratitude to *András Benczúr*; they created a friendly learning environment and provied inspiring perspectives.

The research in this Thesis was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory Grant no RRF-2.3.1-21-2022-00004.

Introduction

Throughout the many years I spent studying mathematics (from grammar school to MSc), graph theory has always held a special place in my heart. When I first encountered the field of network science, I was captivated by its foundation in my favorite topic, as well as its practical applications in real-world scenarios. Artificial Intelligence (AI) became more and more famous in “pop science”, while I could get a glimpse of deep learning and data science at university. This new world introduced a way of thinking and an advanced set of tools that felt liberating.

Graph embedding is a strong tool in data science, especially useful when dealing with datasets composed of networks. Understanding the theoretical background of these methods requires sound knowledge of graph theory. I feel exceptionally fortunate to have had the opportunity to choose a topic for my university project and thesis combining my favorite fields in mathematics.

My thesis is organized as follows. Chapter 1 is an overview of node and graph embedding, concentrating on the theoretical background of these algorithms. In Chapter 2, two datasets are presented for graph classification. In Chapter 3, my measurements and experimental results are discussed with respect to the collections of synthetic and real-world graphs of Chapter 2. Finally, I conclude my results in the Summary.

Chapter 1

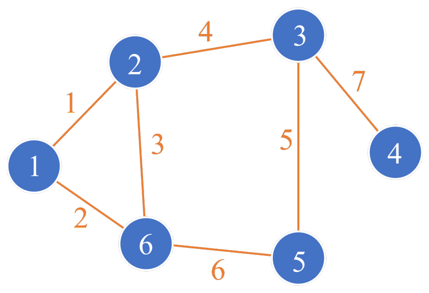
Graph Embedding Methods

1.1 Motivation

Networks are around us everywhere. They appear in various different fields since they provide a suitable way to represent data in many applications, ranging from social to life sciences. Having data represented as a graph or even datasets of graphs is more and more common.

Graph representation in computer science can vary depending on the task or the applied algorithms. The most common structures are adjacency matrices, incidence matrices, edge lists, and adjacency lists (illustrated in Figure 1.1). Matrix representations have a size of $O(n^2)$, where n is the number of nodes. On the other hand, lists have a size of $O(m)$, where m is the number of edges. Therefore, for sparse networks, list representations can save a lot of space compared to matrix representations.

Network datasets often consist of enormous graphs with numerous nodes. However, in many applications, networks are sparse (containing much fewer edges than possible), making the matrix representations filled with zeros providing not much valuable information. A preprocessing step is often required to compress the structural properties of a graph. Additionally, it is crucial to transform networks into a data type compatible with machine learning algorithms, such as vectors.



(a) A graph with 6 nodes and 7 edges

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	1
3	0	1	0	1	1	0
4	0	0	1	0	0	0
5	0	0	1	0	0	1
6	1	1	0	0	1	0

(b) Adjacency matrix

	1	2	3	4	5	6	7
1	1	1	0	0	0	0	0
2	1	0	1	1	0	0	0
3	0	0	0	1	1	0	1
4	0	0	0	0	0	0	1
5	0	0	0	0	1	1	0
6	0	1	1	0	0	1	0

(c) Incidence matrix

1	1	2
2	1	6
3	2	6
4	2	3
5	3	5
6	5	6
7	3	4

(d) Edge list

1	2	6	
2	1	3	6
3	2	4	5
4	3		
5	3	6	
6	1	2	5

(e) Adjacency list

Figure 1.1: Standard Representations of Graphs

1.1.1 Downstream Tasks with Networks

Graphs are ubiquitous data structures in various real-world scenarios, such as social media networks, user interest graphs, citation graphs in research areas, and knowledge graphs. Analyzing these graphs provides valuable insights and has a wide range of applications, including node classification, clustering, recommendation, and link prediction.

With the analysis of graphs based on user interactions in social networks, we can classify users [1], identify communities, recommend friends, or predict whether two users will interact. Examining the transaction graphs of cryptocurrencies, it is possible to profile or deanonymize users raising privacy concerns [2]. The right representation of chemical compounds can be used to predict their properties, such as solubility and anti-cancer activity, or predict their reactions [3, 4]. By analyzing call graphs of programs, malware can be detected [5]. Neuroscientific or gene-based network data prediction or regression can be important to better understand biomarkers of diseases or certain health conditions [3, 6, 4].

1.1.2 Analogous ideas from NLP

In the field of Natural Language Processing (NLP), a fundamental concept is that the meaning and "behavior" of a word are strongly connected with its surrounding context. Hence it can be learned by a computer providing enough appearances of the word in real texts. Additionally, embedding words as tokens into Euclidean spaces simplifies the handling of string-like input in machine learning tasks and allows mathematical operations on words. These ideas led to the famous Word2Vec [7] algorithm based on the Skip-gram model that is illustrated in Figure 1.2.

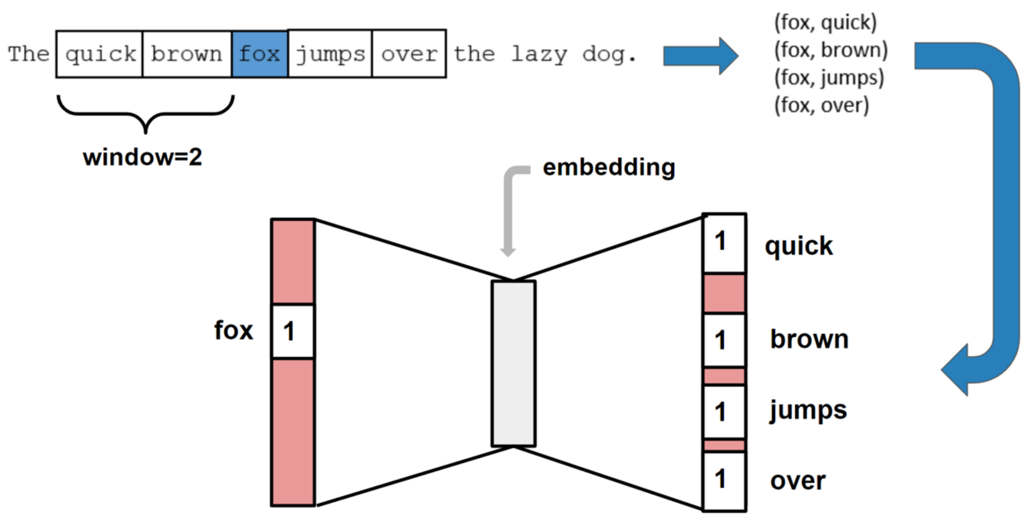


Figure 1.2: Skip-gram model for Word2Vec

First, a One-Hot Encoding of the words in the corpus text is made. Then the "context" of the target word is made by choosing a window size and gathering all the words within that word distance in the text. The objective is to build a shallow neural network that would enable it to predict the context of the target word when provided with its One-Hot encoded version. By context, we mean the summation of the corresponding One-Hot vectors, and this should be reflected in the output of the network. While learning the context and the semantic meaning of words, their embedding appears in the hidden layer.

The Skip-gram model, illustrated in Figure 1.2, can be used just the other way around to predict a word given the context by switching up the input and the output – this approach is called the CBOW (Continuous Bag-of-Words) model.

Analogous to the previous ideas, one could wonder whether a node's properties, function, or role can be learned by looking at its "context", its neighborhood. Several highly effective node embedding algorithms have demonstrated the validity of this parallel concept. Moreover, whole graph embeddings can also benefit from NLP techniques such as Doc2Vec [8], which is the generalization of Word2Vec for documents.

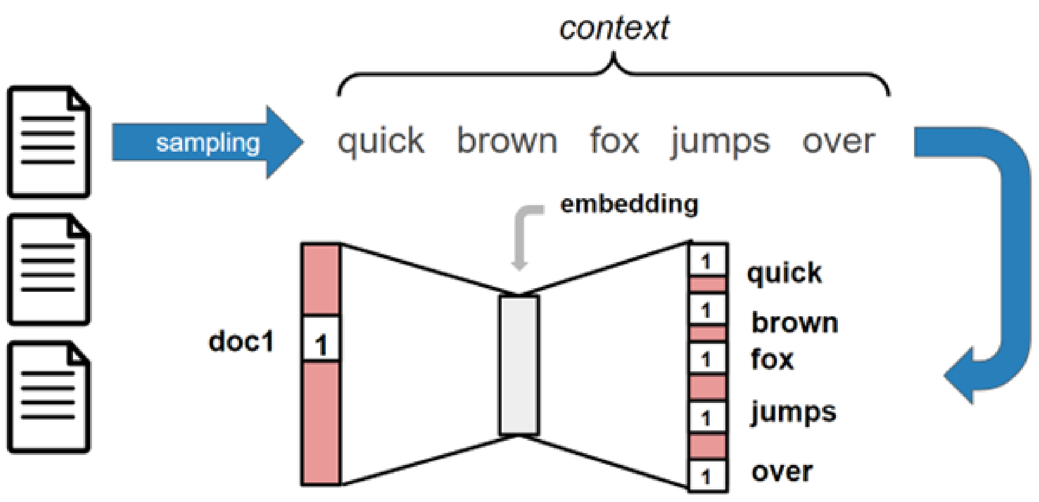


Figure 1.3: Doc2Vec architecture

As illustrated in Figure 1.3, Doc2Vec learns the representation of the document meanwhile solving the task of predicting a word given the other words in its context using Distributed Memory version of Paragraph Vector (PV-DM). It is performed just like in Word2Vec with another feature added to the input, the document ID, which acts as a memory that remembers the document's topic that is missing from the local context. For the training, not just a corpus text but a set of documents is required. A word vector is generated for each word representing its semantic meaning, and a document vector is generated for each document representing its concept.

Just like Doc2Vec used pieces of text (we could say contexts of some words) to embed documents, whole graph embedding can also be done by incorporating information about subgraphs. However, subgraph sampling is a crucial part of this process. Different approaches lead to algorithms suitable for different tasks.

1.2 Node Embedding

As illustrated in Figure 1.4, node embedding algorithms aim to represent network nodes as low-dimensional Euclidean vectors that are the desired input for most Machine Learning (ML) algorithms. This way, one can easily feed network data to ML models trained for various downstream tasks such as link prediction, community detection, or node classification.

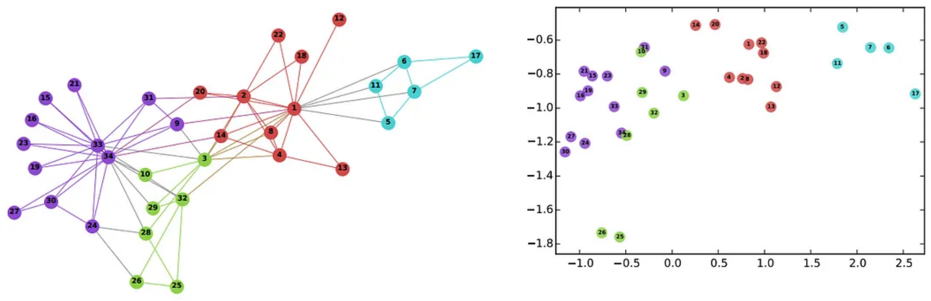


Figure 1.4: Embedding the nodes of the Karate club graph with DeepWalk [9]

1.2.1 Examples

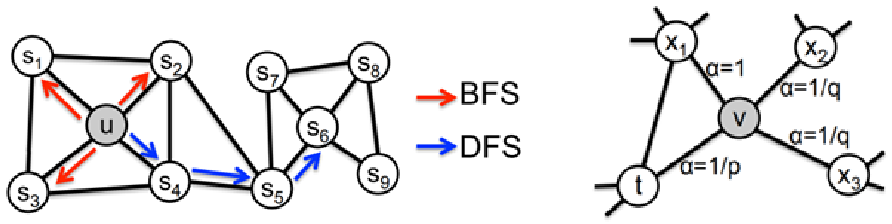
DeepWalk [9]

This algorithm is analogous to Word2Vec. It learns the representation of a given node by feeding its "context", or nodes close by, to the Skip-gram model. Here, multiple random walks with restricted lengths are sampled to extract context nodes to a given target node. Basically, random walks are analogous to sentences in the original NLP setting.

node2vec [10]

The next algorithm works similarly to DeepWalk, but node2vec uses a more sophisticated way to control whether we want to capture local or global descriptors of network nodes.

As illustrated in Figure 1.5(a), two extreme strategies for discovering the graph around a node are Breath First Search (BFS), where the neighborhood is restricted to nodes that are immediate neighbors of the source, and Depth First Search (DFS), where the neighborhood consists of nodes sequentially sampled at increasing distances from the source node. While in DeepWalk, random walk sampling is an uncontrolled process, node2vec has parameters that define whether neighborhood discovery follows a BFS or DFS style or some blend of the two strategies.



(a) Extreme strategies

(b) Parameters

Figure 1.5: Sampling random walks with node2vec [10]

Consider a random walk that has arrived at node v from node t as shown in Figure 1.5(b). The algorithm has two parameters that alter the weight of probabilities regarding the choice of the next node:

- Return parameter p: the weight of returning to node t is $1/p$; setting it to a high value ensures sampling of an already visited node is less likely, while if p is low, it leads to a backtrack step.

- In-out parameter q: the weight of distancing away from t is $1/q$; if $q > 1$, the random walk is biased towards nodes close to t, hence a BFS manner is achieved while setting $q < 1$, the walk is more inclined to visit further nodes to gain a DFS-like behavior.
- (The weight of moving to a common neighbor of v and t stays 1.)

Role2Vec [11]

Role2Vec differs mainly from the previous techniques, it is not an embedding of the nodes, but of the subsets of nodes having supposedly the same role in the graph – thus the name.

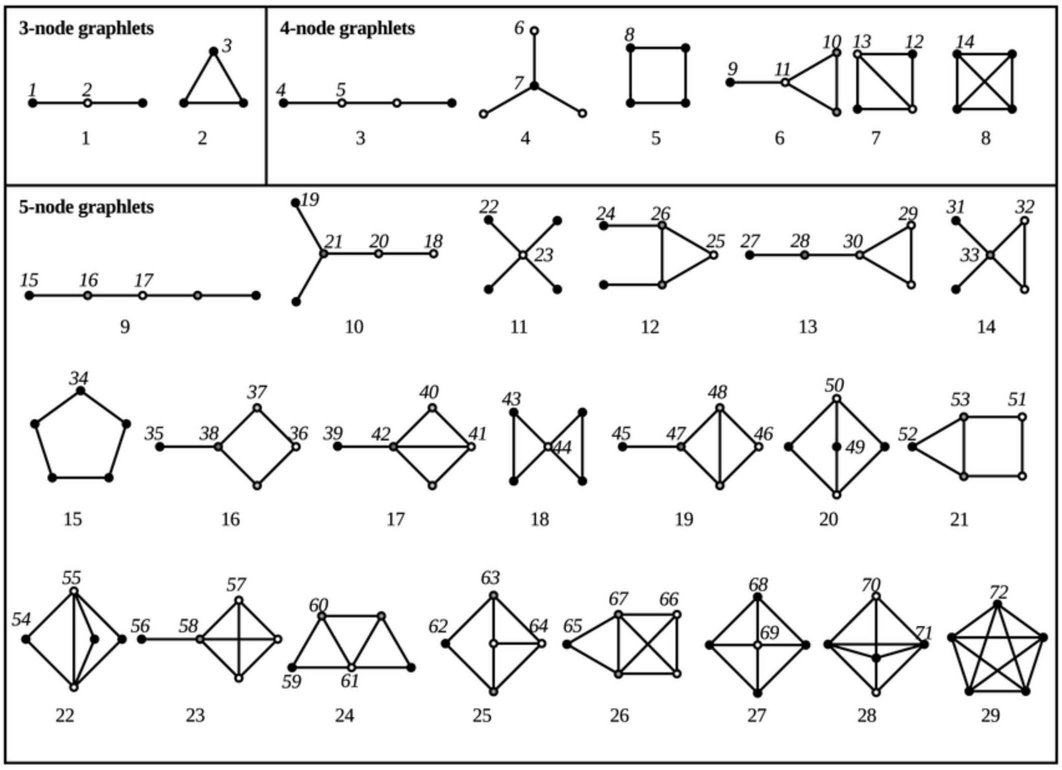


Figure 1.6: Graphlet-based vertex types in Role2Vec [12]

The Role2Vec framework starts by learning a function F that maps the n vertices to a set of m vertex types (where the vertices of one type are similar in terms of attributes and/or structural features) with $m << n$ usually. This function could be learned automatically or defined manually. For example, in Figure 1.6, vertex types defined by graphlets with at most five nodes are shown: there are 29 graphlets and 72 vertex types based on their position in these graphlets. These types are identified for the mapping function F.

Attributed random walks are generated which are sequences of adjacent vertex types: $F(v_0), F(v_1), ...F(v_l)$ where $v_0, v_1, ...v_l$ would be a random walk of length l. Role2Vec uses these attributed walks as contexts for learning the embedding of vertex types (thus the vertices in the same typeset will share an embedding).

This elegant idea grants an approach that is efficient in terms of memory and time achieved by reducing the size of the corpus.

1.3 Whole Graph Embedding

The concept of embedding nodes or whole graphs is rather similar, except that the targets are different. Graph embedding algorithms aim to assign low-dimensional Euclidean vectors to graphs such that the representation of structurally similar graphs would also be close to each other in the embedding space.

There are some challenges that these algorithms have to face:

- The properties of the graphs should be well represented.
- Non-isomorphic graphs should have distinguishable representations
- Permutation-invariance: the embedding should be invariant to the permutation of nodes.
- Scale-adaptivity: it should represent local and global structures enabling the embedded vector for graph comparisons on different levels.
- Size-invariance: being able to represent structural similarity regardless of graph magnitude.
- Preferably, the size of the network should not decrease the speed of the embedding process too much.

- The dimension of the embedded vector should be chosen carefully: large enough to capture structural properties well, but it may cause an increase in runtime at machine learning tasks.

There are numerous different techniques to calculate whole graph representations, but the most prominent approaches are based on graph measures (Section 1.3.2), statistical analysis of some properties (Section 1.3.2), spectral properties (Sections 1.3.4, 1.3.4, 1.3.4, and 1.3.5), or walk-based calculations (Sections 1.3.3, 1.3.3, 1.3.5, and 1.3.5). The following sections of this chapter will provide an overview of graph embedding techniques categorized based on their primary approach to the problem.

In my work, I used the implementations of embedding algorithms from Karate Club [12], an open-source Python framework for unsupervised learning on graphs. This package contains a variety of state-of-the-art graph-mining algorithms for community detection, node embedding, and whole graph embedding. For the algorithms described in this chapter, not every parameter setting was implemented in Karate Club. Thus, in Sections 1.3.2–1.3.5, I highlighted the parameters with *italics* that I could tune during my experiments through the Karate Club Python API[1].

1.3.1 Notations

Let $G = (V, E)$ be an undirected graph, where $|V| = n$ and $|E| = m$. The binary adjacency matrix, $A \in \{0,1\}^{n \times n}$ has a nonzero value at the intersection of a row and column whose respective nodes are connected with an edge. The weighted adjacency matrix denoted by W contains the weights of the edges instead of just ones. A_{ij}^k denote the element in the i^{th} row and j^{th} column of the k^{th} power of A. A_{ij}^k is also the number of k-length walks of G starting from node i and ending in node j.

Let $D = diag(A \cdot \mathbf{1})$ denote the diagonal matrix of size $n \times n$ containing the degrees of nodes $(d(v))$ or the sum of edges ending in the corresponding node in the weighted case $(d_W(v) = \sum_{u=1}^{n} W_{uv})$.

[1]https://karateclub.readthedocs.io/en/latest/

The Laplacian matrix of a graph is defined as $L = D - A$ for the unweighted or $L = D - W$ for the weighted case. L could also be written in the form $L = \Phi \Lambda \Phi^T$ where $\Lambda = diag(\lambda_k)$ is the diagonal matrix with the eigenvalues of L and $\Phi = [\phi_1, \phi_2, ...\phi_n]$ is the orthogonal matrix formed by the corresponding eigenvectors. L is positive semi-definite with eigenvalues denoted by $0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \leq ... \leq \lambda_n$. The algebraic multiplicity of the eigenvalue zero equals the number of connected components [13].

The normalized Laplacian matrix of G is defined as $\mathcal{L} = I - D^{-1/2} A D^{-1/2}$ for the unweighted, or $\mathcal{L} = I - D^{-1/2} W D^{-1/2}$ for the weighted case. It is also positive semi-definite with eigenvalues denoted by $\mu_1 \leq \mu_2 \leq ... \leq \mu_n$ and respective ortogonal eigenvectors $\varphi_1, \varphi_2, ...\varphi_n$. A good property of $\mathcal{L}$ is that $\mu_i \in [0, 2]\ \forall i$. Moreover, the algebraic multiplicity of the eigenvalue zero equals the number of connected components, while the algebraic multiplicity of the eigenvalue 2 is the number of bipartite components [13].

$P = D^{-1}A$ is the transition matrix for the random walk on the graph where P_{uv} (the element in the row and column corresponding to nodes u and v) is the probability of choosing v in a random walk after u (hence the name). It is indeed a probability distribution since $\sum_{v \in V} P_{uv} = \sum_{v \in V} (D^{-1}A)_{uv} = 1$ by definition. The powers of P characterize the random walks on the graph: P^k_{uv} is the probability of a random walk starting from source node u, hitting the target node v in the k^{th} step.

1.3.2 Intuitive and Explainable Methods

Arbitrary Subset of Graph Measures

Calculating multiple graph measures and arranging them in a vector would yield an embedding. Possible features include global descriptors like graph diameter, radius, transitivity, clustering coefficient, number of components, eigenvectors or eigenvalues of the adjacency, or the Laplacian matrix of the given network. Some types of pooling (min, max, mean) for node-level measures like degree can also be considered.

Some measures might not represent the graph well in the given dataset or task, and we have to take into consideration the computational complexity, which could be too high for a dataset of big graphs.

However, relatively small embedded vectors could achieve an explainable method and high accuracy by choosing the measures carefully concerning the task.

Molontay and Nagy [14, 15] have chosen 17 graph metrics that describe various characteristics of networks of four kinds:

- Degree distribution related e.g. interval degree probabilities, average degree, maximum degree (normalized by the size)
- Shortest paths related e.g. diameter and average path length (normalized by the log of the size)
- Centrality related e.g. maximum eigenvector and betweenness centrality
- Clustering related e.g. Global clustering coefficient, average local clustering coefficient

Metrics strongly correlating with the number of nodes and edges were excluded. From the remaining set of graph measures, a selection based on Spearman's rank correlation resulted in an embedding of 8 dimensions.

The previously described embedding was tested for a domain classification task on 482 networks from 6 domains, and more than 90% accuracy was achieved. More details about their result are discussed in Chapter 2.2.

LDP (Local Degree Profile) [16]

Let $G(V, E)$ be a graph, and for each node $v \in V$, let $DN(v) = \{\text{degree}(u) | (u, v) \in E\}$. We take five node features:

$$(\text{degree}(v),\ \min(DN(v)),\ \max(DN(v)),\ \text{mean}(DN(v)),\ \text{std}(DN(v)));$$

they summarize the degree information of node v and its 1-neighborhood. The aggregation of features over different nodes in the same graph is done by performing either a histogram or an empirical distribution function operation. This procedure is repeated for all five node features, and to get the embedded vector of the graph, all the feature vectors are concatenated.

Computational complexity: This approach only requires the calculation of degree for each node and saves the statistics of the 1-size neighborhood for each node; this can be done in $O(E)$ time. Then, to map V numbers into B bins takes $O(V)$ time, making the total complexity $O(E)$.

Parameters:

- histogram or empirical distribution
- *number of bins of histogram* or empirical distribution
- linear or logarithmical scale for the histogram or empirical distribution – the latter is suitable for e.g. networks whose degree distribution follows the power law

1.3.3 Walk-based Methods

graph2vec (Graph to Vector) [17]

The graph2vec embedding method is inspired by Doc2Vec, where each document is considered as a set of words. Similarly, in graph2vec, a graph is represented as a collection of rooted subgraphs. Then, the embeddings of the graphs are obtained by the skip-gram language model with the rooted subgraphs as its vocabulary.

The first part of this algorithm is to create the vocabulary. In each graph, the Weisfeiler-Leman algorithm (a BFS-like recursive algorithm) can extract a rooted subgraph in depth d around each node.

The second part uses the first, target graph – subgraph context (and additionally features) pairs are trained with Stochastic Gradient Descent (SGD) optimizer. A negative sampling strategy is used due to the large size of the vocabulary. The negative samples are sets of k subgraphs, each of which is not present in the target graph, but in the vocabulary of subgraphs. Only the embeddings of the negative samples are updated instead of going over the whole vocabulary.

Computational complexity: For all the graphs, it takes $O(e \cdot g \cdot N \cdot d^2)$, where g is the number of graphs and N is the dominant graph size in the training dataset. Hence, we can say that the average runtime is $O(e \cdot n \cdot d^2)$ per graph.

Parameters:

- *"degree" or depth of rooted subgraphs (d)*
- *dimension of the embedding (δ)*
- *learning epochs (e)*
- *learning rate (α)*
- *whether to erase the graph features if any present*
- *frequency of down sampling*

GL2vec (Graph and Line Graph to Vector) [18]

A major shortcoming of graph2vec is that it cannot incorporate edge features during the learning process. GL2vec overcomes this limitation by using both G and $L(G)$, the line graph (edge-to-vertex dual graph) of G. This algorithm makes the embedding of G and $L(G)$ with graph2vec, so the edge features can simply be used while learning the representation of $L(G)$ as its node features. Then, it concatenates the two vectors to get the final embedding.

Computational complexity: Since graph2vec has complexity $O(e \cdot n \cdot d^2)$ per graph, the second part of GL2vec has a complexity of $O(e \cdot (n+m) \cdot d^2)$. The creation of the line graph has complexity $O(n^2)$, hence the first part is dominant.

Parameters: Same as in graph2vec (1.3.3)

1.3.4 Spectral property Methods

SF (Spectral Features) [19]

As introduced in Section 1.3.1, $\mathcal{L}$ is the normalized Laplacian matrix of a connected, undirected graph G. The k smallest positive eigenvalues of $\mathcal{L}$ in ascending order are chosen to represent the graph as a vector. If the graph has less than k nodes, right zero padding is used to get appropriate dimensions: $X = (\mu_1, ..., \mu_{|V|}, 0, ..., 0)$.

Computational complexity: It is an eigenvalue-problem (eigendecomposition), which has practical and widely used implemetation based on the Lanczos algorithm [20].

Parameters: *number of eigenvalues desired (k)*

The eigenvalues of the normalized Laplacian matrix lie between 0 and 2 which makes it convenient for the downstream use of a standard classifier. The multiplicity of the eigenvalue 0 corresponds to the number of connected components in the graph, hence the omission of μ_0 since it is always equal to 0. Other eigenvalues are also known to denote the presence of specific structures, e.g. 2 denotes a bipartite graph.

This embedding also has a physical interpretation. Each eigenvalue corresponds to the energy level of a stable configuration of the nodes (the lower the energy, the stabler the configuration) [21] or corresponds to frequencies associated with a Fourier decomposition of any signal living on the vertices of the graph [22].

IGE (Invariant Graph Embedding) [23]

In this method, the embedded vector is a concatenation of three different approaches chosen to have invariance and also a great discriminative power for a large family of graphs. Let G be an attributed, connected graph.

1) Consider the Laplacian matrix L of G with eigenvalues $0 = \lambda_1 < \lambda_2 \leq \lambda_3 \leq ... \leq \lambda_n$. Define a k_1-dimensional vector $F_1(G) = (\lambda_2, ..., \lambda_{k_1+1})$ if

$n \geq k_1+1$ and if $n \leq k_1$, then left zero padding is used to get appropriate dimensions: $F_1(G) = (0, ..., 0, \lambda_2, ..., \lambda_n)$.

2) Let P be the transition matrix for the random walk on the graph. If there are features on the nodes, it can be viewed as a feature vector F; if not, the degrees are used: $F = (d_1, d_2, ...d_n)$. $P^kF = (x_1^k, x_2^k, ...x_n^k)$ can be considered as the aggregation of the features of its k-distance neighbors for each node. For each $k \in \{1, 2, ...k_2\}$, a histogram of t_2 bins is performed on the elements of P^kF. The second embedding $F_2(G)$ is the concatenation of k_2 vectors of length t_2 each.

3) Let Λ denote the diagonal matrix with the eigenvalues of L. $X = \sqrt{\Lambda^+}U^T$ is the pseudo-inverse of Λ. The rows of X define an embedding of the nodes of size n. The matrices $x(i)^Tx(j)$ (for $1 \leq i, j \leq n$) are "flattened" to obtain vectors of size n^2. These vectors are then passed through a histogram of t_3 bins to gain the third representation $F_3(G)$.

The final graph embedding is the concatenation of the above-defined three representations for fixed parameters k_1, k_2 and t_2, t_3:

$$F_{IGE}(G) = (F_1(G), F_2(G), F_3(G)),$$

which is a vector of size $k_1 + t_2k_2 + t_3$, independent of the size of the graph. All three representations are permutation-invariant, so as their concatenation.

Computational complexity: The complexity of this method is dominated by searching for the eigenvalues of L (see 1.3.4).

Parameters:

- *number of eigenvalues (dimensions) desired for the first representation* (k_1)
- size of the neighborhood in consideration for the features in the second representation (k_2)
- number of bins used in the second representation (t_2)
- number of bins used in the third representation (t_3)

FGSD (Family of Graph Spectral Distances) [24]

Consider the weighted, undirected graph $G(V, E, W)$ with its weighted Laplacian matrix $L = \Phi\Lambda\Phi^T$, where $\Lambda = diag(\lambda_k)$ is the diagonal matrix with the eigenvalues of L $(0 \leq \lambda_1 \leq \lambda_2 ... \leq \lambda_n)$ and $\Phi = [\phi_1, \phi_2, ...\phi_n]$ is the orthogonal matrix formed by the corresponding eigenvectors.

Let f be an arbitrary nonnegative (real-analytical) function on $\mathbb{R}^+$ with f(0) = 0 and let $f(L)$ denote $f(L) := \Phi \cdot \text{diag}(f(\lambda_k))\Phi^T$. Here $\phi_k(x)$ denotes the x-entry value of ϕ_k, also, $f(L)_{xy}$ represent xy-entry value in matrix $f(L)$.

The f-spectral distance between $x, y \in V$ is defined as:

$$S_f(x, y) = \sum_{k=0}^{N-1} f(\lambda_k)(\phi_k(x) - \phi_k(y))^2.$$

The function $\{S_f(x, y)|f\}$ is referred to as the family of graph spectral distances, hence the name of the embedding. The representation of G is $R = \{S_f(x, y)|\forall(x, y) \in V\}$, from which the embedding vector is made by a histogram.

If $f(x) = x^p$, where $p \geq 1$, S_f captures the neighbourhood information in distance at most p. On the other hand, for a decreasing function, $S_f(x, y)$ captures global information. For example, $S_f(x, y)$ is called harmonic distance in the case of $f(x) = 1/x$; it is the default function for this embedding method.

Computational complexity: To avoid direct eigenvalue decomposition, an approximation could be performed. However, it is better to use structural properties and sparsity of $f(L)$ for efficient exact computation of S_f. After some alternations not changing the set of solutions, a sparse linear system is achieved, which can be solved by Cholesky factorization and back-substitution, resulting in overall $O(n^2)$ [25].

Parameters:

- the function f
- *number of bins of the histogram forming F and also its range considered*

1.3.5 More Complex Methods

NetLSD [26]

Let G be a graph and $\mathcal{L}$ its normalized Laplacian matrix with spectrum $\{\mu_1, \mu_2, ...\mu_n\}$ and eigenvectors $\{\varphi_1, \varphi_2, ...\varphi_n\}$, respectively.

Consider an associated heat diffusion process on the graph, the initial heat u_0 on one of the vertices with a fixed value. We want to calculate the heat of each vertex at time t denoted by u_t. The heat equation associated with the Laplacian is:

$$\frac{\delta u_t}{\delta t} = -\mathcal{L} u_t.$$

Its closed-form solution is given by the $n \times n$ heat kernel matrix:

$$H_t = e^{-t\mathcal{L}} = \sum_{j=1}^{n} e^{-t\mu_j} \varphi_j \varphi_j^T.$$

where $(H_t)_{ij}$ represents the amount of heat transferred from vertex v_i to vertex v_j at time t. To aggregate this information, the heat trace at a time t is defined:

$$h_t = \sum_{j=1}^{n} e^{-t\mu_j}.$$

The NetLSD representation consists of a heat trace signature, i.e., a collection of heat traces at different time scales: $h(G) = \{h_t\}_{t>0}$.

The wave trace could be used instead or as a complement to the heat trace. The wave equation is:

$$\frac{\delta^2 u_t}{\delta t^2} = -L u_t$$

with a complex solution matrix:

$$W_t = e^{-itL} = \sum_{j=1}^{n} e^{-it\mu_j} \varphi_j \varphi_j^T.$$

from which the wave trace can be defined as:

$$w_t = \sum_{j=1}^{n} e^{-it\mu_j}.$$

Computational complexity: The full eigendecomposition of the normalized Laplacian would take $O(n^3)$ time. Thus, an approximation of the heat trace signatures is used. The second order Taylor-expansion is computationally inexpensive, requiring only $O(m)$ time, and the error stays small until time scale 1. Truncated spectrum approximation for large eigenvalues and linear interpolation for the interloping eigenvalues is used.

Parameters:

- whether to use the heat trace and/or the wave trace
- *number and value of timesteps for evaluating the heat trace*
- *number of eigenvalue approximations*

The heat trace, hence the embedding, is permutation-invariant because isomorphic graphs are isospectral as well. It is also scale-adaptive. For small t, the heat trace depicts local connectivity, while for large t it encodes global connectivity.

GeoScattering [27]

Consider the weighted, undirected graph $G(V, E, W)$ and an $n \times 1$ signal-vector $x : V \mapsto \mathbf{R}$. Three different summation operators will be defined on this vector to gain a permutation invariant embedding.

The simplest summation operator computes the sum of the responses of the signal $\mathbf{x}$ (the unnormalized moments). It can also be thought of as the zero order geometric scattering moments:

$$S\mathbf{x}(q) = \sum_{v \in V} \mathbf{x}(v)^q \text{ for } 1 \leq q \leq Q.$$

Let $P = \frac{1}{2}(I + AD^{-1})$ be a lazy random walk matrix since P^t governs the probability distribution of a lazy random walk after t steps (lazy noting that $v_i = v_{i+1}$, staying at the same vertex is allowed). The $n \times n$ wavelet matrix at the scale 2^j is defined as:

$$\mathbf{\Psi}_j = P^{2^{j-1}} = P^{2^j} = P^{2^{j-1}}(I - P^{2^{j-1}}).$$

The value $\mathbf{\Psi}_j\mathbf{x}(v)$ aggregates the signal information from vertices within 2^j steps of v, but does not average the information like the operator P^{2^j}.

The second summation operators will be the first-order geometric scattering moments:

$$S\mathbf{x}(j,q) = \sum_{v \in V} |\mathbf{\Psi}_j \mathbf{x}(v)|^q \text{ for } 1 \leq j \leq J,\ 1 \leq q \leq Q.$$

Augmentation can be achieved by incorporating second-order geometric scattering moments, obtained by the graph wavelet and absolute value transforms:

$$S\mathbf{x}(j,j',q) = \sum_{v \in V} |\mathbf{\Psi}_{j'} |\mathbf{\Psi}_j \mathbf{x}(v)||^q \text{ for } 1 \leq j \leq j' \leq J,\ 1 \leq q \leq Q.$$

The transformation can be iterated additional times, leading to higher order features, and thus has the general structure of a graph convolutional network.

The collection of graph scattering moments

$$S\mathbf{x} = \{S\mathbf{x}(q), S\mathbf{x}(j,q), S\mathbf{x}(j,j',q)\}$$

provides a rich set of multiscale invariants of the graph G.

Computational complexity: The matrix powers of P have a crutial part in the complexity of this embedding. With the right implementation, calculating these can be done in around $O(n^3 + nj)$ via diagonalization.

Parameters:

- graph features to be use as signals $\mathbf{x}$
- *wavelet matrix scales (J will result in maximal scale 2^J)*
- *unnormalized, or normalized moments considered (Q)*
- order of geometric scattering moments incorporate in the representation

This embedding method is stable in a sense that there is an upper bound on distances between similar graphs that only differ by modifications that can be treated as noise.

FeatherGraph [28]

The core idea of this embedding method is to capture node feature distributions in a neighborhood with characteristic functions.

Let $G = (V, E)$ be an attributed and undirected graph. In the case of unattributed graph input, the logarithm of degrees and clustering coefficients of the nodes can be used as features. A feature of the nodes can be described by a random variable X or as the feature vector $\mathbf{x} \in \mathbb{R}^{|V|}$, where $\mathbf{x}_v$ denotes the feature value for node $v \in V$.

The characteristic function of X for source node $u \in V$ at evaluation point $\theta \in \mathbb{R}$ is:

$$\varphi_X(\theta) = E[e^{i\theta X}|G, u] = \sum_{v \in V} P_{uv} \cdot e^{i\theta \mathbf{x}_v}$$

where P is the transition matrix of a random walk on G. Using Euler's identity, the real and imaginary parts of the characteristic function can be obtained:

$$Re(\varphi_X(\theta)) = \sum_{v \in V} P_{uv} \cdot cos(\theta \mathbf{x}_v), \qquad Im(\varphi_X(\theta)) = \sum_{v \in V} P_{uv} \cdot sin(\theta \mathbf{x}_v).$$

It can be assumed that the neighborhood of u at scale r consists of nodes that can be reached by a random walk with at most r steps from the source node. The distribution of a feature in the neighborhood of u at scale r can be described using the real and imaginary parts of its characteristic function. The probability of arriving at node v from source node u with a random walk in exactly r steps is P^r_{uv}. Hence, the r-scale random walk weighted characteristic function of feature X can be defined as:

$$\varphi_X(\theta, r) = E[e^{i\theta X}|G, u, r] = \sum_{v \in V} P^r_{uv} \cdot e^{i\theta \mathbf{x}_v}.$$

Its real and imaginary parts can be expressed as:

$$Re(\varphi_X(\theta, r)) = \sum_{v \in V} P^r_{uv} \cdot cos(\theta \mathbf{x}_v), \qquad Im(\varphi_X(\theta, r)) = \sum_{v \in V} P^r_{uv} \cdot sin(\theta \mathbf{x}_v).$$

The evaluation of each node's characteristic function on the whole domain would not be necessary. Only d points will be sampled from the domain, those are described by the evaluation point vector $\Theta \in \mathbb{R}^d$. The characteristic function of a node will be evaluated at these points, and the real and imaginary parts of these complex values will be used for its $2d$-dimensional representation per scale and feature. These vectors are concatenated for each scale in $\{1, 2, ...r\}$ and feature. The final representation has dimension $2 \cdot d \cdot r \cdot k$, where k is the number of features in consideration.

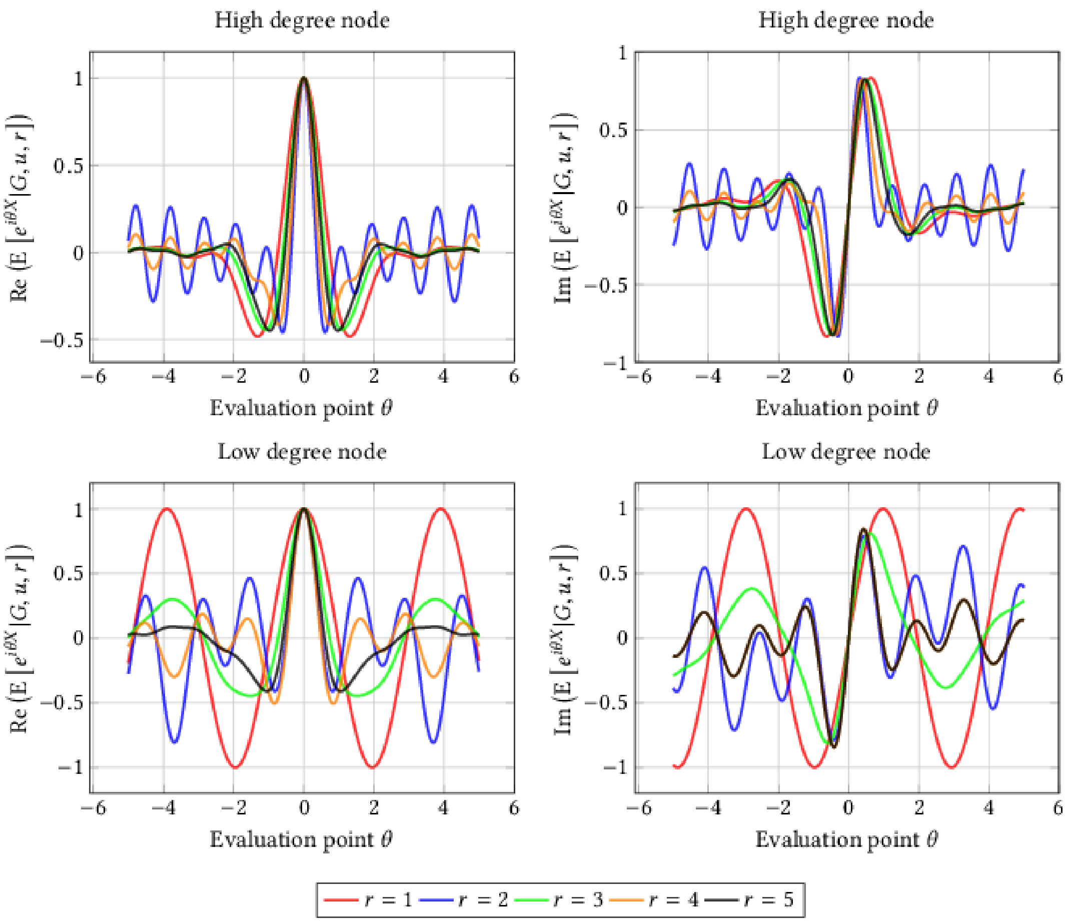


Figure 1.7: Example of the real and imaginary parts of the r-scale random walk weighted characteristic function [28]

Until now, this method can be used as a node embedding. These representation vectors can be pooled by permutation invariant functions to output the whole graph embedding.

In the parametric model variant, the evaluation points are learned in a semi-supervised fashion to make them the most discriminative concerning a downstream classification task.

In the implementation of KarateClub, the maximal evaluation point value (θ_{max}) and the number of evaluation points (d) can be set. The evaluation points will be evenly distributed in the $[0.01, \theta_{max}]$ interval. Since it uses the logarithm of degrees and clustering coefficients of the nodes as features, the outputs are $(4 \cdot d \cdot r)$-dimensional embedded vectors.

Computational complexity: The r-scale random walk weighted characteristic functions for the whole graph and one feature can be written as a $\mathbb{C}^{|V| \times d}$ matrix. Evaluating on Θ will become a matrix multiplication. Calculating the complex matrix and evaluation has a time complexity of $O(|E| \cdot d \cdot r)$. If it needs to be done for all scales up to r and k features, the time complexity becomes $O(|E| \cdot d \cdot r^2 \cdot k)$.

Parameters:

- *size of neighborhood in consideration (the number of scales, r)*
- *the number of evaluation points for the k-scale random walk weighted characteristic functions (d)*
- whether to use the parametric variant; how to learn the evaluation points
- whether to choose the evaluation points and how
- *pooling function for node embeddings*

This embedding of the graph is not only permutation-invariant but also robust to noise in the data. If the feature vector changes on one node, the absolute changes of the r-scale random walk weighted characteristic functions stay low.

Using characteristic functions seems to be useful because of a few important properties (illustrated in Figure 1.7):

- the real part is an even function while the imaginary part is odd;
- the range of both parts is in the $[-1, 1]$ interval;
- nodes with different structural roles have different characteristic functions.

Chapter 2

Datasets for Classification Tasks

Classification is an essential concept in the field of supervised machine learning. At its core, it involves categorizing or grouping data points into predetermined classes or categories based on their distinguishing attributes. The ultimate objective is to develop a model that can accurately assign new data points to the correct classes or categories.

When it comes to the classification of tabular data, where data points are represented as vectors, it's important to ensure that the input vectors are of a manageable size and can be processed in a reasonable amount of time. This may involve preprocessing the input data to reduce its size and complexity which sometimes might even improve the performance of the classification.

In this chapter, I demonstrate two datasets prepared for graph classification. The related measurements and results are detailed in Chapter 3.

2.1 Artificial Dataset

The first classification problem is focused on distinguishing between graphs generated using different random network generators.

Our motivation for constructing this synthetic dataset was that we could generate graphs of arbitrary size with random graph generators, while many real-life graph classification dataset mostly contains networks with at most a few hundred nodes. Building models that reproduce the properties of real networks have great importance. To accurately model a network domain using random graphs, it's essential to carefully select the random graph models and suitably adjust their parameters.

Alternatively, we can view various random graph generators with different parameters as models for different network domains. By differentiating between these generators, we aim to predict the performance of a classification pipeline on real-life networks. However, classifying an artificial dataset might be an overly simplified version of network domain prediction and, therefore, an easier task. Still, we might be able to draw some conclusions, like, embedding algorithms that perform poorly in the synthetic task may not be robust tools for more complex problems. Hence, they should be excluded first, especially if they are computationally intensive.

2.1.1 Random Graph Generation

While generating this dataset, I included various random graph models and degree distributions to better represent the variety of real-life networks.

The first generating algorithm was the Random Regular graph[2], which outputs a d-regular graph of n nodes. It has a degree distribution asymptotically uniform. These graphs are not likely in real life due to their high symmetry.

The graphs generated by the Erdős-Rényi model[3] have n nodes and all possible edges are chosen to be present with a fixed probability p. These graphs have binomial, asymptotically Poisson degree distribution, resulting in most of the degrees close to average with small deviation. They're not common in real life either due to their degree distribution and their lack of hubs but have small-world property.

[2]`https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.random_regular_graph.html`

[3]`https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html`

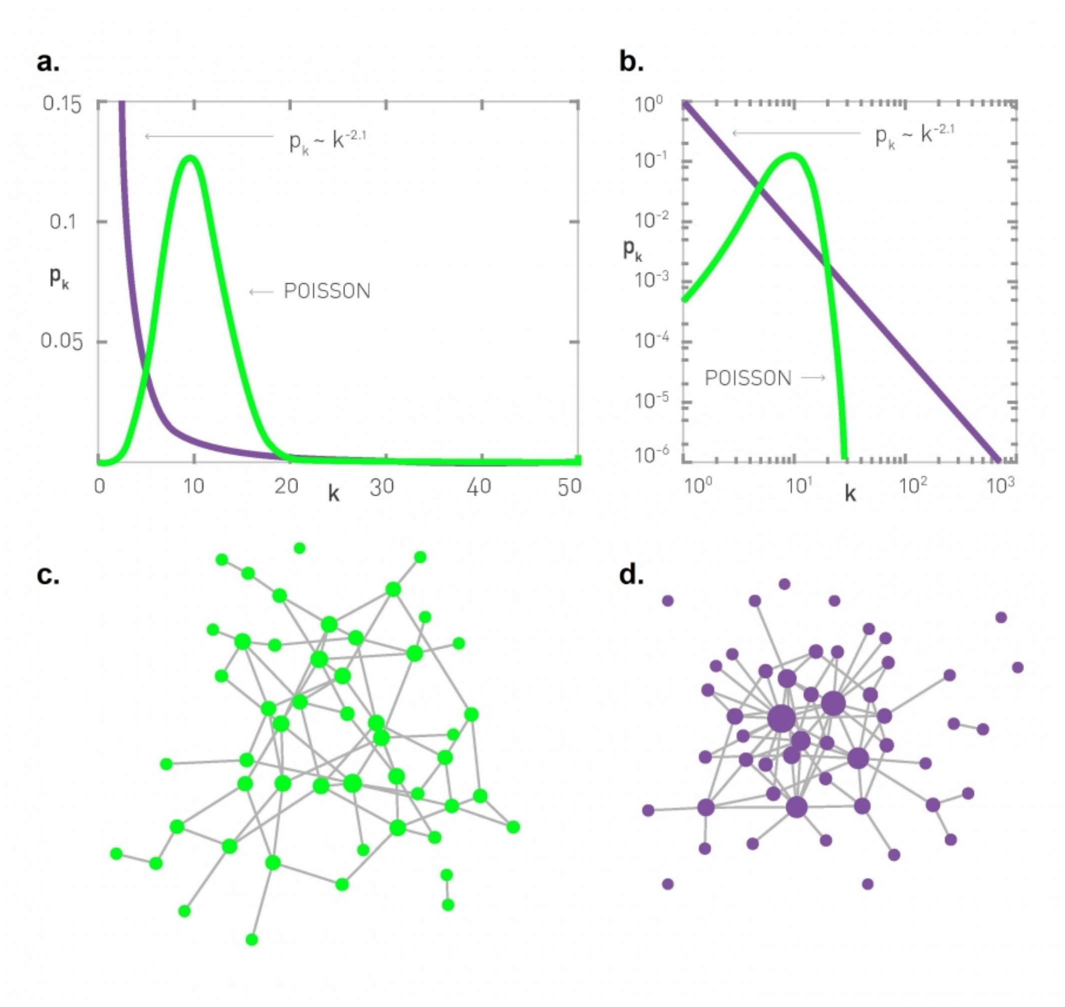


Figure 2.1: Graphs with Poisson (green) and Scale-free (purple) Degree Distribution [29]

The final category of random graphs used is scale-free graphs meaning that their degree distribution follows a power law: $p_k = k^{-\gamma}$, where p_k denotes the fraction of nodes in the graph having degree k. As a result, most of the nodes have a small degree and very few nodes are hubs. The difference from Erdős-Rényi graphs are illustrated in Figure 2.1. These graphs are characterized by having small-worldness and a clustering coefficient dependent only on the average degree and not the size. Due to these properties, real-life networks are commonly modeled with scale-free graphs [29].

Several models based on the concept of preferential attachment exist, resulting in slightly different values of γ or even modified equations but significant differences in the graph structure. I used three of these:

- Barabási-Albert model[4]: a graph of n nodes is grown by attaching new nodes one by one, each with m new edges that are preferentially attached to existing nodes, meaning that the probability of choosing an existing node as a neighbor is proportional to their degrees. The initial graph is a star on $m + 1$ nodes here.
- Dual Barabási-Albert model[5]: Barabási-Albert model but a new node is preferentially attached with either m_1 edges with probability p, or m_2 edges with probability $1-p$. The initial graph is a star on $max(m_1, m_2)+1$ nodes here.
- Algorithm of Holme & Kim [30] for Copying model (Power Cluster Graph model)[6]: Barabási-Albert model but when a new node is preferentially attached with m new edges, each random edge is followed by a chance of making an extra edge to one of the endpoint's neighbors too with probability p, thus forming a triangle. This algorithm enables a higher average clustering.

I employed the random graph models as implemented in NetworkX [31] in my study. The generated graphs were of size 1000 to 1100 nodes, and they were connected to ensure that the implementation of the embedding algorithms could be applicable. The parameters of the models and the number of graphs generated can be seen in Table 2.1.

[4]`https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.barabasi_albert_graph.html`

[5]`https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.dual_barabasi_albert_graph.html`

[6]`https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.powerlaw_cluster_graph.html`

Graph generating model	**Parameter(s)**	**Number of graphs generated**
Random Regular (RR)[3]	$d = 4; 6; 8; 10; 12$	87 in total
Erdős-Rényi (ER)[??]	$p = 0.1; 0.2$ $p = 0.05; 0.075; 0.125; 0.15; 0.25$	20 – 20 10 – 10 (90 in total)
Barabási-Albert (BA)[4]	$m = 1; 2; 3;$ $m = 4; 5; 6$	18 – 18 10 – 10 (84 in total)
Dual Barabási-Albert (DB)[5]	(m_1, m_2, p) = (1,3,0.25); (1,3,0.50); (1,3,0.75); (3,4,0.25); (3,4,0.50); (3,4,0.75); (4,6,0.25); (4,6,0.5); (4.6,0.75)	81 in total
Holme & Kim (HK)[6]	$(m, p) = (1, 0.2); (1, 0.4); (1, 0.6); (2, 0.2); (2, 0.4); (2, 0.6); (3, 0.2); (3, 0.4); (3, 0.6)$	86 in total

Table 2.1: Generated Graphs for the Artificial Dataset

2.2 Real-life Dataset

Several datasets with network-like data points, such as those found in chemical, bioinformatic, infrastructural, and social network databases, may require differentiation within groups (e.g., protein functions) or between groups. To assess the performance of graph embedding algorithms on a real-world dataset, I performed a domain classification task on the dataset of M. Nagy and R. Molontay [14].

This dataset consists of 500 graphs, gathered one by one from online databases, whose self-loops were removed and treated as undirected, unweighted graphs. These networks belong to six domains, as shown in Table 2.2, and their sizes cover a wide range from small graphs to networks with thousands of nodes.

Domain	**Description**	**Range of network size**	**Number of networks**
Brain	Human and animal connectomes	50 - 2,995 (avg: 946)	100
Cheminformatics	Protein-protein (enzyme) interaction networks	44 - 125 (avg: 55)	100
Food	What-eats-what, consumer-resource networks	19 - 1,500 (avg: 118)	100
Infrastructural	Transportation (metro, bus, road, airline) and distribution networks (power and water)	39 - 40K (avg: 4,562)	68
Social	Facebook, Twitter and collaboration networks	85 - 34K (avg: 5,183)	118
Web	Pieces of the World Wide Web	146 - 16K (avg: 4,488)	14

Table 2.2: Real-world network dataset for graph classification [14]

In their previous work [32], *M. Nagy* and *R. Molontay* managed to perform 0.912 accuracy in a graph domain classification task for this dataset with only eight carefully selected topological features extracted from degree distribution, shortest path, centrality, and clustering-related metrics (details in Section 2.3.1). The beauty of this approach is the explainability of these features, but calculating some of them is computationally costly. For example, the average path length divided by the logarithm of the size or the maximum eigenvector centrality is computable in $O(n^3)$.

They used graph2vec (1.3.3) and AttentionWalk ([33]) as baseline methods and made the classification from different graph representations with kNN, Random Forest, and Decision Tree. Their results can be seen in Table 2.3:

Approach	**kNN**	**Decision tree**	**Random Forest**
Topological features	78.5%	85.2%	89.3%
Selected features	87.0%	85.5%	91.2%
graph2vec ($\delta = 8$)	52.3 %	51.2%	55.8%
graph2vec ($\delta = 200$)	79.2 %	75.4%	82.1%
graph2vec ($\delta = 1100$)	84.2 %	74.1%	84.5%
AttentionWalk ($\delta = 128$)	27.0%	43.0%	54.0%
AttentionWalk ($\delta = 256$)	28.4%	37.9%	54.7%

Table 2.3: Results of *R. Molontay* and *M. Nagy* [14]

The main objective of my research was to try out additional whole graph embedding methods. My measurements, in Chapter 3 not only show that graph2vec and AttentionWalk can be outperformed with state-of-the-art models, but it highlights that there are good performing algorithms with small embedding dimensions.

Chapter 3

Measurements

In this chapter, I will present measurements of graph classification on the two datasets introduced in Chapter 2. The primary focus will be on the performance of various embedding algorithms combined with classifiers to evaluate their effectiveness in solving classification tasks. The running time of the embedding algorithms and the dimension of the embedded vectors are also discussed with consideration of their parameters.

For the measurements, I used some of the whole graph embedding algorithms implemented in the Karate Club Python API [12]. If a graph could not be embedded with a specific algorithm, it was substituted with the zero vector of respective size. This happens, for instance, whenever the graph is too small and the embbedding algorithms cannot build the vocabulary.

3.1 Classification on Artificial Dataset

The artificial dataset of this measurement is presented in detail in Section 2.1.

3.1.1 Tasks

The evaluation of the performance of embedding algorithms was conducted using two distinct types of classification tasks:

- distinguishing between graphs generated by random graph models;

- differentiating between graphs created by the same random graph model but with varying parameters.

More precisely, the six defined tasks were:

- **Barabási-Albert types**: distinguishing between the different scale-free models of preferential attachment; the three categories are Barabási-Albert, Dual Barabási-Albert and Holme & Kim.
- **Barabási-Albert parameters**: determining the parameter of the Barabási-Albert model (the number of new edges of the new node); the categories are integers from 1 to 6.
- **Dual Barabási-Albert parameters**: determining the parameters of the Dual Barabási-Albert model (the expected value (rounded down) of the numbers of new edges of the new node); the categories are integers from 2 to 6.
- **Barabási-Albert types' parameters**: determining the expected value of the number of new edges of the new node in graphs generated by both Barabási-Albert and Dual Barabási-Albert models; the categories are integers from 1 to 6.
- **Different types of models**: distinguishing between the Barabási-Albert model, Erdős-Rényi random graphs, and random regular graphs, three models with different degree distribution, hence significantly different properties; the three categories are the above-mentioned three models.
- **All models**: distinguishing between all the five random graph models used; the five categories are Barabási-Albert, Dual Barabási-Albert, Holme & Kim, Erdős-Rényi and Random Regular graph models

The number of generated graphs in each category of the above-defined tasks can be seen in Table 3.1, also presenting the approximate number of graphs in the train and test dataset.

Task Name	**Classes**	**#graphs in train set**	**#graphs in test set**
Barabási-Albert types	Barabási-Albert	63	21
	Dual Barabási-Albert	61	20
	Holme & Kim	64	22
Barabási-Albert parameters	1 – 6 (the value of parameter)	14 or 8 per class	4 or 2 per class
Dual Barabási-Albert parameters	2 – 6 (expected value of parameter)	7 to 21 per class	3 to 7 per class
Barabási-Albert types' parameters	1 – 6 (expected value of parameter)	15 to 35 per class	5 to 10 per class
Different types of models	Barabási-Albert	63	21
	Erdős-Rényi	67	23
	Random Regular	66	21
All models	Barabási-Albert	63	21
	Dual Barabási-Albert	61	20
	Holme & Kim	64	22
	Erdős-Rényi	67	22
	Random Regular	66	21

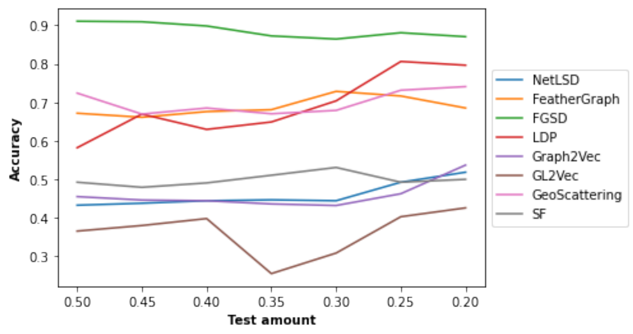
Table 3.1: Classification Tasks for the Artificial Dataset

3.1.2 Results

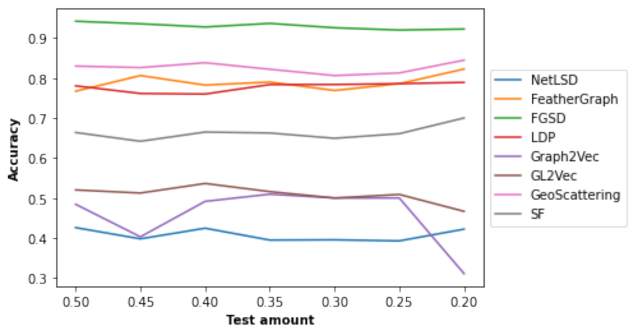
In these classification tasks, I used various whole graph embedding algorithms from the Karakteclub [12] Python package with their default parameter settings (detailed in Section 1.3).

The dataset of the embedded vectors was split into train-test datasets of proportions $0.75 - 0.25$. Logistic Regression Classifier[7] (from the scikit-learn [34] Python package) was used with the default settings except for the solver, which was Newton CG (Newton conjugate gradient method) optimizer.

After varying the train-test ratio to identify the minimal size of the training dataset that does not significantly impact performance, it became apparent that there was no substantial deviation from the original 25% test ratio. The top-performing embedding algorithm exhibited near-perfect performance across all ratios. The results of the first and the last classification tasks regarding the train-test ratio can be seen in Figure 3.1.



(a) Barabási-Albert types



(b) All models

Figure 3.1: Performance of graph embedding methods on two classification tasks with varying test-ratio

The performances of the eight embedding algorithms on the six tasks of Section 3.1.1 are summarized in Figure 3.2.

As anticipated, "Different types of models" proved to be the easiest classification task, indicated by five out of seven models with almost perfect accuracy. The reason behind this is that the different degree distributions are easier to distinguish from each other. On the other hand, the most challenging task was distinguishing between the scale-free models, likely because they share

[7]`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

relatively similar characteristics. This task was also part of the classification between all the models and might explain why it was the second most difficult.

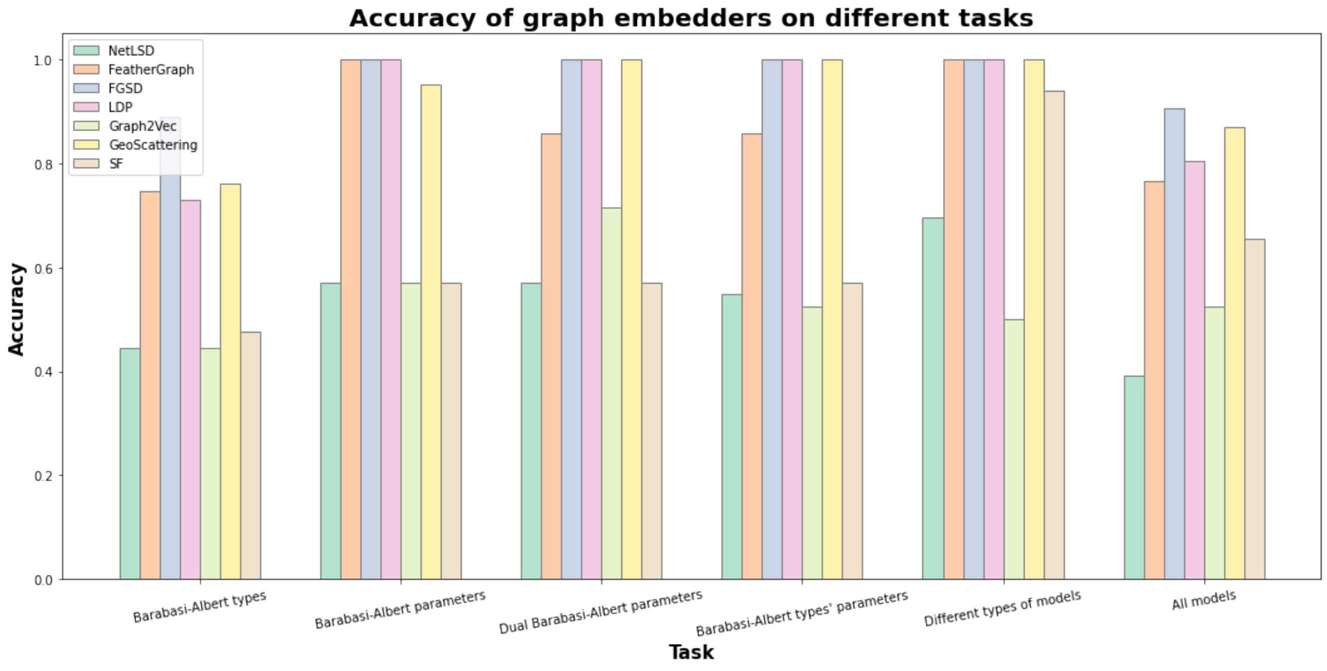


Figure 3.2: Performance of graph embedding methods on various classification tasks

In my experiments, four embedding algorithms, FeatherGraph, FGSD, LDP, and GeoScattering, stand out with almost perfect performance in multiple tasks. The most surprising of them was LDP since it is a rather simple and very fast algorithm. Its performance might be due to the strong connection to the degrees, which is particularly useful in these tasks.

It is important to consider that for larger graphs, the execution times of the embedding algorithms can become intolerably slow, or, in some cases, the algorithms may not be completed at all. Figure 3.3 shows the results of a previous study, where my objective was to analyze the scaling of runtimes of various embedding algorithms. The input data consisted of randomly generated Barabási-Albert graphs with magnitudes of orders 10 to 10^5.

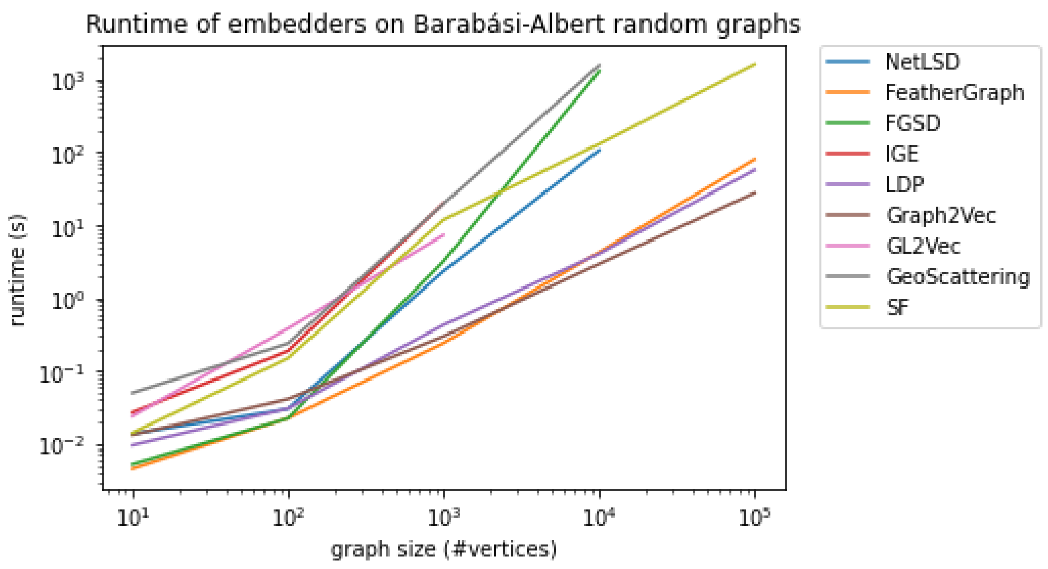


Figure 3.3: Runtime of embedding algorithms on different scales

FGSD and GeoScattering showed great promise in terms of their performance on artificial datasets. However, their runtimes can pose significant challenges when applied to datasets with large networks.

3.2 Classification of real-life networks

To solve the real-life domain classification task, introduced in Section 2.2, again, I use whole graph embedding algorithms from the Karakteclub [12] Python package with a wide range of manually selected parameter settings shown in the Appendix A. Here, by learning embedding vectors of different sizes, I explore the trade-off between dimensionality and accuracy (See more details in Section 3.2.3.)

Due to the increased size of real-life networks, I only experimented with NetLSD, LDP, FeatherGraph, graph2vec, and SF that proved to be the five algorithms with the lowest runtime for large graphs in Figure3.3. Four out of five of these algorithms use some randomness (mainly for initialization), with the exception of LDP. Thus, for the four non-deterministic algorithms, I used 10 different random seeds, and I present the mean model performance in Section 3.2.2.

3.2.1 Classifiers

To classify the embedded vectors, two different methods from the scikit-learn [34] Python package were used: Logistic Regression and Random Forest.

For each set of embedded vectors (obtained by applying the same algorithm and parameters to all networks), 10 train-test splits were made to evaluate the performance of the classifier on these inputs. The resulting performances were then averaged to obtain mean performances for each embedding parameter setup. This approach was repeated for each random seed wherever applicable (the exception is LDP) and averaged once again.

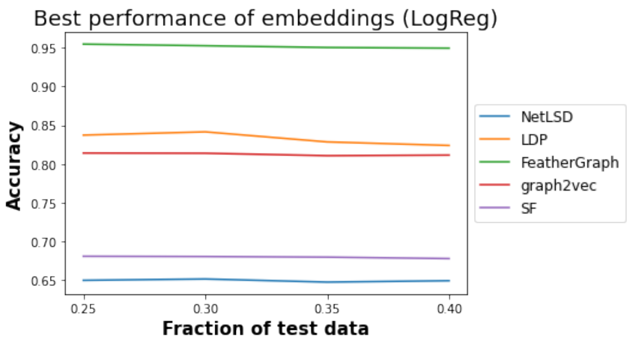
- **Logistic Regression[8]:** The classifier was used with the default settings except for the solver, which was *Newton CG (Newton conjugate gradient method)* optimizer. Four ratios of the test dataset were tried: 0.25, 0.30, 0.35, and 0.40 (see Figure 3.4).
- **Random Forest[9]:** There were two different measurements with this classifier:
  - **Varying test ratio:** Like for Logistic Regression, four ratios of the test dataset were tried: 0.25, 0.30, 0.35, and 0.40 (see Figure 3.4). The number of trees (e) was set to 15 with a maximal tree depth (d) of 7. The other settings of the classifier were the default.
  - **Parameter optimization:** The ratio of the test dataset was set to 0.3 for these measurements, and the best setting of the number of trees (e) and the maximal tree depth (d) was optimized with grid search on the parameter space: $(e, d) \in \{5, 9, 13, 17, 21\} \times \{5, 7, 9, 11\}$.

[8]`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

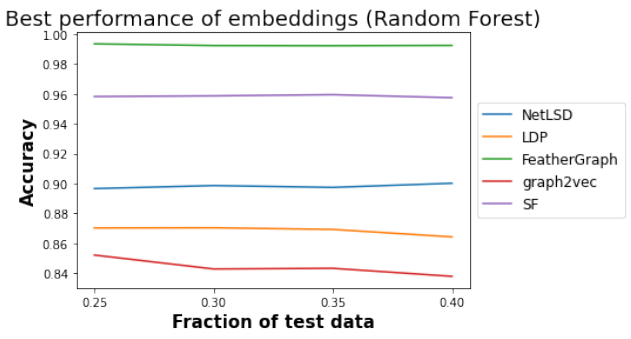
[9]`https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`

3.2.2 Results

In Figure 3.4, the accuracy of the five tested embedding algorithms is shown with respect to the fraction of test data for both classifiers Logistic Regression and Random Forest ($e = 15$, $d = 7$). The best performance here means the highest accuracy for the given test with some parameter setup of the embedder, so it might be that the points of a curve correspond to different parameters of the same embedder.



(a) Logistic Regression Classifier



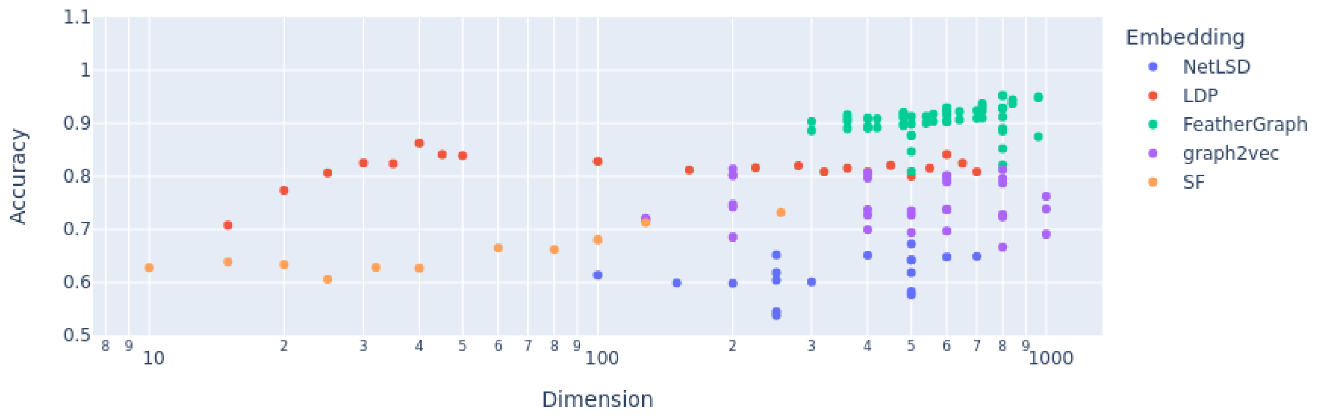
(b) Random Forest Classifier

Figure 3.4: Accuracy of different embedding methods as a function of test data fraction

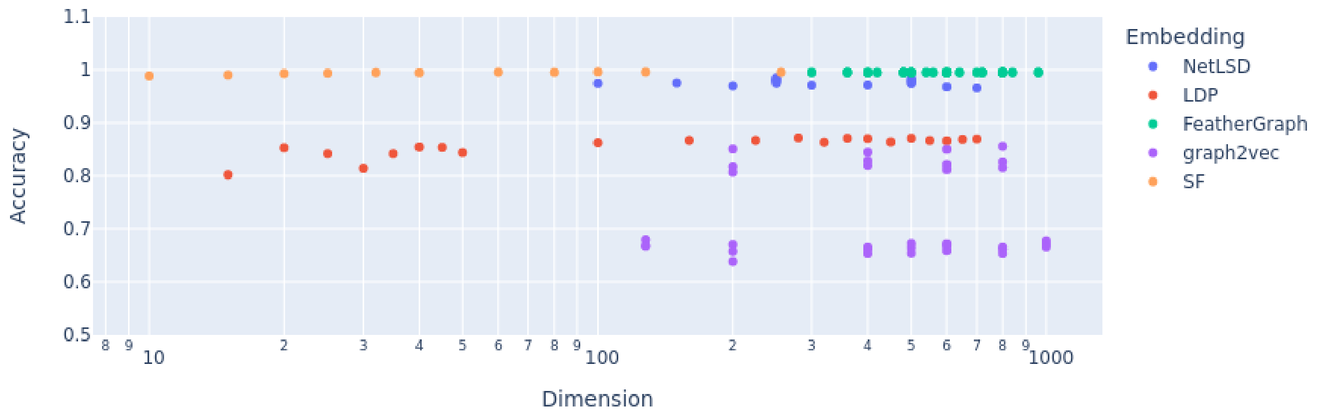
Logistic Regression and Random Forest have similar results to the extent that the modification of the test ratio will not make a significant effect. It indicates that only 25% of the data (125 graphs) being used as the training dataset yields the same performance as 40% (200 graphs) for both classifiers. However, there is a difference between the performance of some algorithms with the different classifiers.

The ranking of the algorithms is clearly visible in both cases. The most powerful embedder was Feather Graph for both classifiers with accuracy from 0.95 to 0.98. In general, graph embedding models had better performance with the Random Forest than with Logistic Regression as the downstream classifier. SF and NetLSD are notably improved with Random Forest, whereas the accuracy of LDP and graph2vec are just slightly modified. It is important to note that all of these models demonstrate particularly strong performance with the Random Forest Classifier, achieving accuracies exceeding 0.85.

In Figure 3.5 the accuracy can be seen as a function of dimension on a log-linear scale for both Logistic Regression and Random Forest. Every point on the diagram corresponds to a different parameter setup of the respective algorithm (denoted by its color). For Random Forest, the illustrated performance was achieved with grid search (see Section 3.2.1).



(a) Logistic Regression Classifier



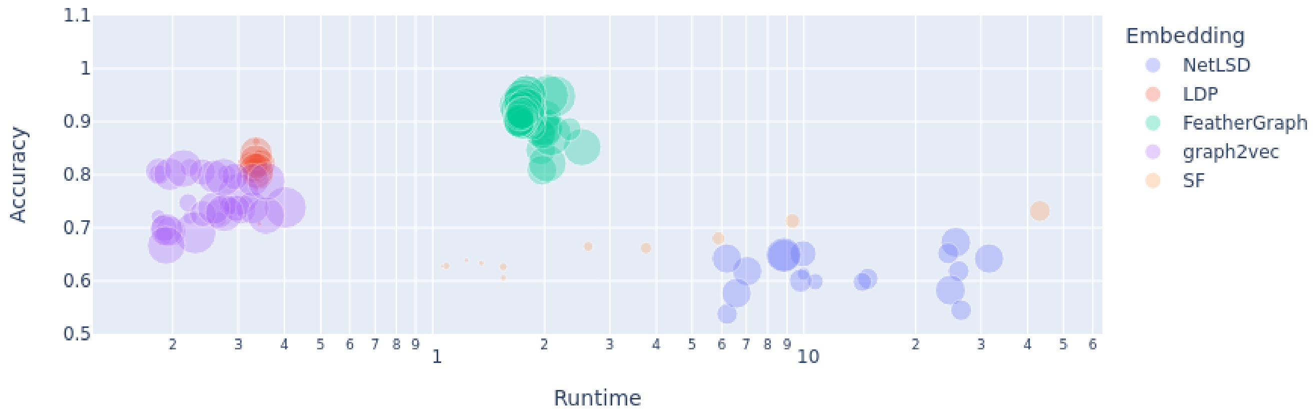
(b) Random Forest Classifier

Figure 3.5: Dimension – Accuracy trade-off of embedding algorithms with different classifiers[10]

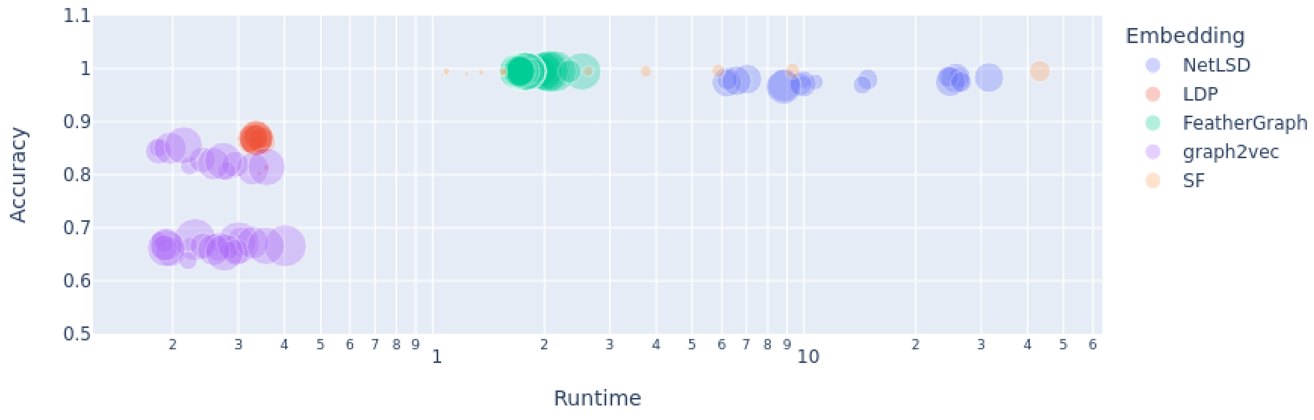
[10]Interactive figures at `https://info.ilab.sztaki.hu/~lilikata/msc_thesis/`

Surprisingly, the low dimensional embedding of LDP and SF has competitive performance when Random Forest is chosen as a downstream classifier. For higher dimensions, FeatherGraph is the best option for both classifiers. NetLSD is only competitive in performance with Random Forest. The different parameters of graph2vec have a relatively wide range of accuracy; two distinct groups can be seen with Random Forest, and better performance is achieved by tuning down the learning rate with one order.

In Figure 3.6, the Accuracy as a function of the average running time of the embedding (per graph) is shown on a log-linear scale, where the bubble size is proportional to the dimension of the embedded vector.



(a) Logistic Regression Classifier



(b) Random Forest Classifier

Figure 3.6: Average runtime (per graph) – Accuracy trade-off of embedding algorithms with different classifiers[10]

In general, graph2vec is the quickest algorithm. However, it has the worst performance with these parameters for Random Forest and also outperformed with Logistic Regression. LDP is the second, and FeatherGraph is the third algorithm with the least runtime, and the latter has a very promising accuracy as well in Figure 3.5. The runtime of NetLSD and SF is dependent on their parameters, but they are slower than the other three algorithms.

SF with a parameter value of 100 (number of eigenvalues) achieved the best overall performance, with an accuracy of 0.996266. The 100-dimensional embedded vectors were fed into Random Forest Classifier with a tree number of 9 and maximal depth of 11 for optimal performance. The average computation time for the embedding was $5.88s$ per graph, significantly slower than $1.28s$, the average computational time of the slightly less accurate FeatherGraph. It achieved an accuracy of 0.995367 with the following parameters: 3 as the highest matrix power or distance, 2.5 as the maximal evaluation point value, 50 as the number of evaluation points, and *min* function as pooling. This parameter setup resulted in 600-dimensional embedded vectors, which were best classified using a Random Forest with a tree number of 9 and maximal depth of 9.

3.2.3 Effect of Parameters

In this section, some properties are detailed about the embedding algorithms as the parameters are varied (illustrated in Figures 3.5 and 3.6).

LDP *(Local Degree Profile)* (Section 1.3.2)

Dimension: 5× the number of the bins of the histogram

LDP has a peak accuracy around dimension 40 and for higher dimensions, it stays nearly the same. There's no significant difference in runtime either, calculating the histogram is a fast process. When considering Logistic Regression as a classifier, LDP may be a preferable choice for small dimensions over SF due to its superior performance and faster runtime. However, when using Random Forest, SF outperforms LDP. Nonetheless, LDP remains a noteworthy option with significantly faster computation time (around 0.64 seconds/graph) compared to SF or FeatherGraph.

graph2vec (Section 1.3.3)

Dimension: can be set as a parameter

The computation time of graph2vec is surprisingly quick, the fastest embedding out of the five algorithms that I analyzed in detail ($0.2 - 0.4$ seconds/graph). Only LDP demonstrated weak competition with graph2vec in terms of runtime. It slows down with the increasing number of training epochs and dimensions. In general, higher dimensions tend to result in slightly better performance. However, increasing the number of training epochs does not improve the method significantly. The highest accuracy (0.8558621) is attained with 800 dimensions and only 10 learning epochs. Still, FeatherGraph is the superior option in the same dimension range.

SF *(Spectral Features)* (Section 1.3.4)

Dimension: number of eigenvalues

The runtime of SF quickly becomes very slow with the increase of dimension (or parameter) due to the complexity of the eigenvalue search. However, the performance just slightly gets better. It has an accuracy of 0.994575 when the dimension is 56, which is almost as good as the peak performance (0.996266) achieved by 100-dimensional vectors. Meanwhile, the runtime of the former is just 1.55 seconds instead of 5.88 – the potential gains may not outweigh the wait. There is no need to exceed $150 - 160$ eigenvalues since FeatherGraph outperforms SF with faster runtime in that dimension range.

NetLSD *(Network Laplacian Spectral Descriptor)* (1.3.5)

Dimension: number of timesteps for evaluating the heat trace

NetLSD gets slower for a higher number of eigenvalue approximation steps, while the performance increases slightly. This proves that the approximation algorithms used by this method are performing well for this task. The best accuracy with 400 approximation steps is 0.985154, while an accuracy of 0.980392 can be achieved with just 200 steps as well. Considering the better performance and faster runtime of FeatherGraph within the same dimension range, utilizing NetLSD may not be worthwhile.

FeatherGraph (Section 1.3.5)

Dimension: $2\times$ the number of node features (2 in this case) $\times$ the maximal distance to be considered $\times$ number of evaluation points of the characteristic function

The runtime of FeatherGraph stays almost the same for every tested parameter setup. The significance of parameters that compose the same dimensional vectors is uncertain, as it is unclear whether higher matrix powers or a greater number of evaluation points yield better performance. However, it can be seen[10] that the worst pooling method is *mean*, it is always worse than *max* or *min*, but the relation of the latter two is also not apparent. Considering runtime as well as performance for both classifiers, FeatherGraph may be the optimal choice among these five embedding algorithms.

3.2.4 Parameters of Random Forest

In the heatmaps of Figures 3.7–3.9, we present Random Forest performance (accuracy) for various parameter settings. The test ratio was set to 0.3 and the parameters, number of trees (e) and maximal depth of trees (d), were explored for the grid $(e, d) \in \{5, 9, 13, 17, 21\} \times \{5, 7, 9, 11\}$. Lighter color indicates better accuracy. Note that the same color does not necessarily denote the same performance as different heatmaps might be on different color scales.

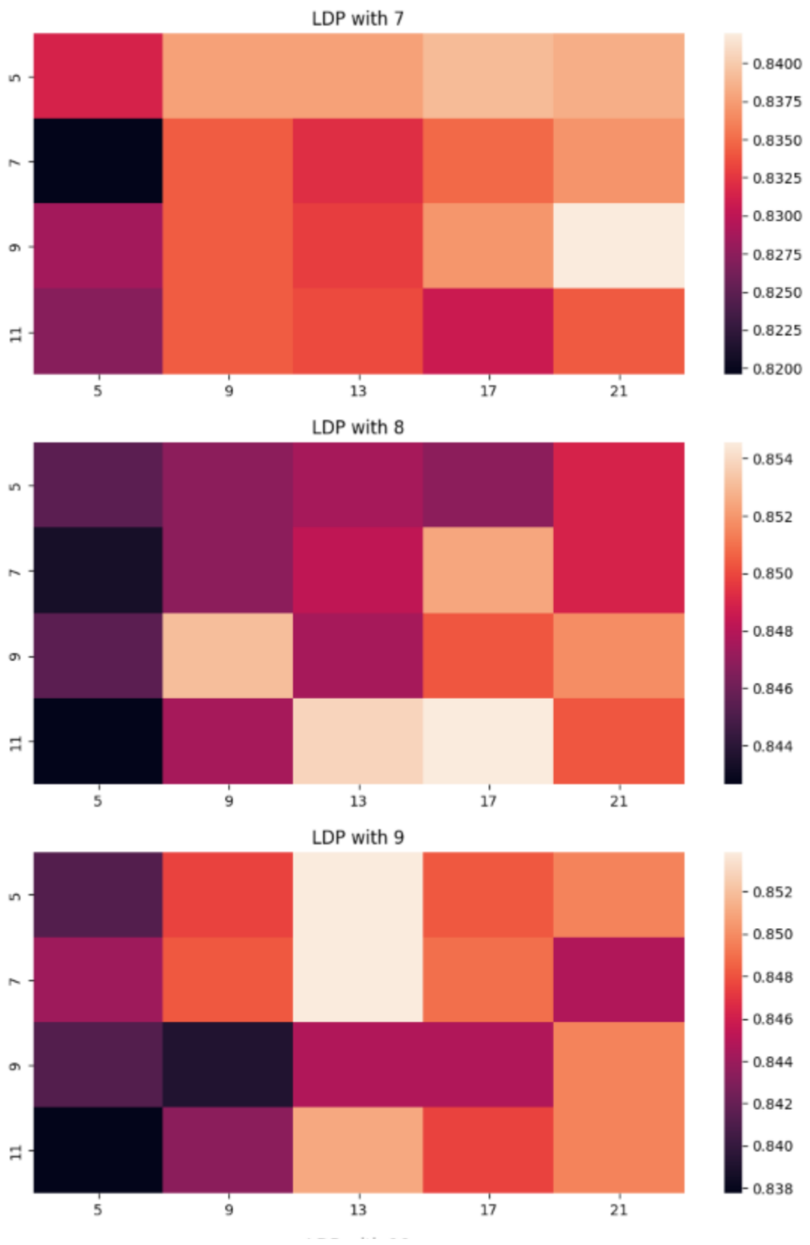


Figure 3.7: LDP accuracy of embedding methods with different parameters of Random Forest Classifier: number of trees (e) on the horizontal, while the maximal depth of trees (d) on the vertical axis.

LDP does not seem "compatible" with this classifier, having no particular pattern. Moreover, it is the only one of the five studied embeddings to perform worse with Random Forest than with Logistic Regression Classifier.

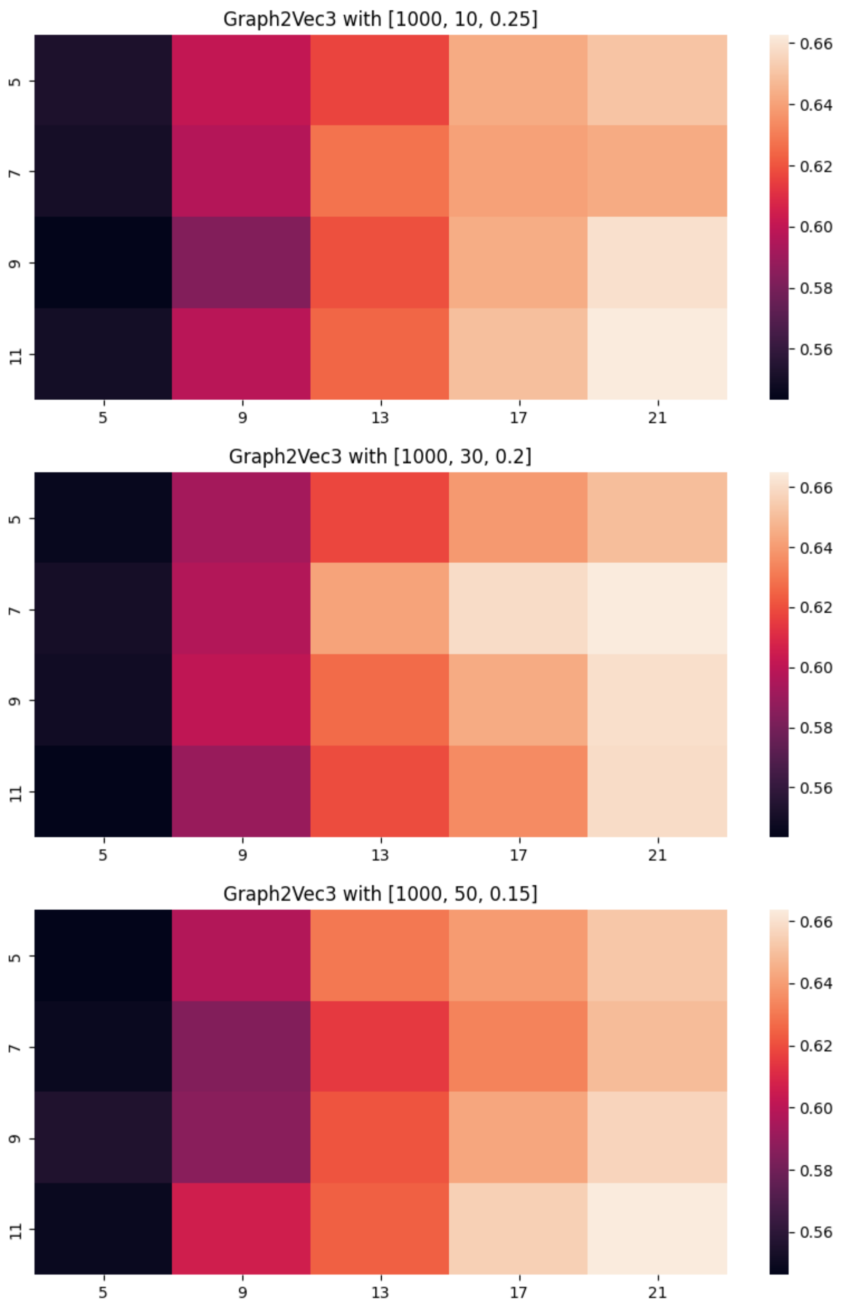


Figure 3.8: Graph2vec ccuracy of embedding methods with different parameters of Random Forest Classifier: number of trees (e) on the horizontal, while the maximal depth of trees (d) on the vertical axis.

An interesting pattern could be noticed with graph2vec: it favored more trees, but there were no significant differences with the higher depth of them.

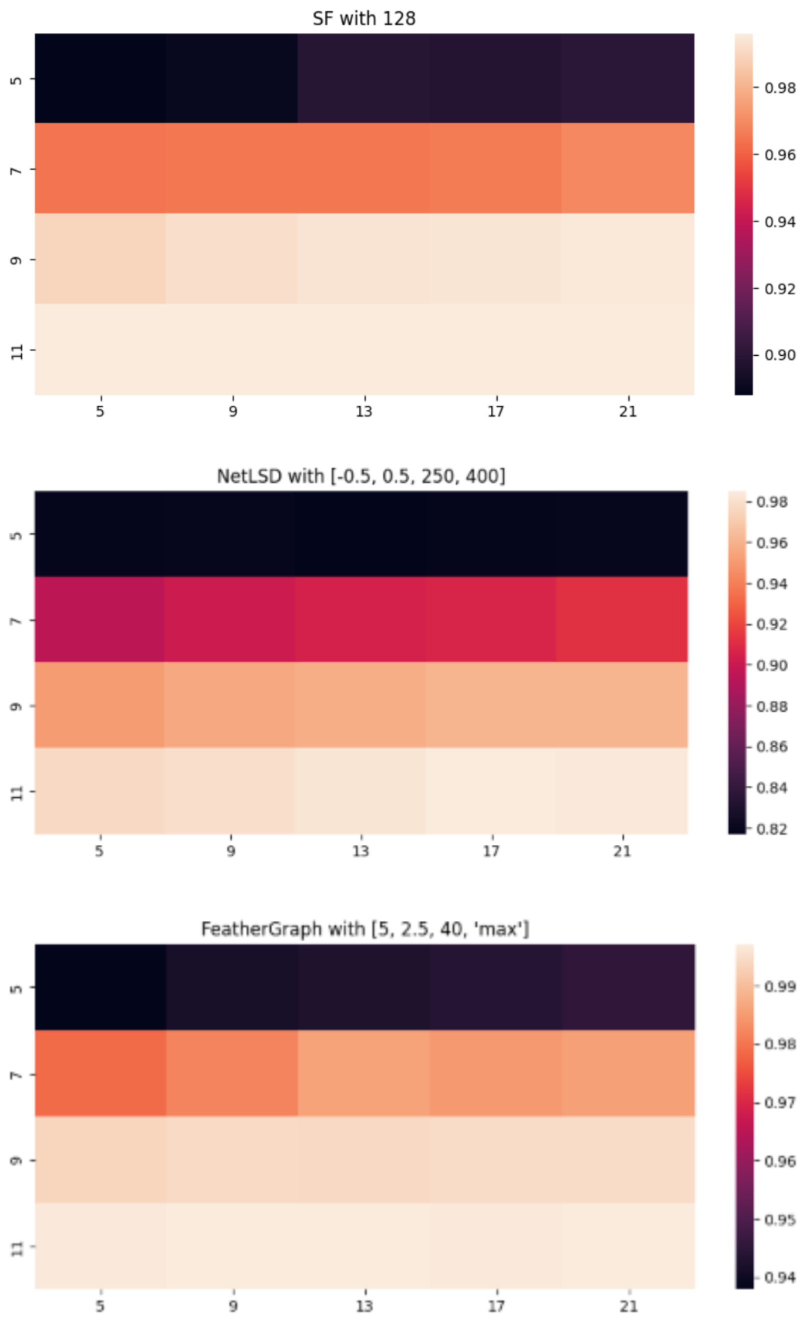


Figure 3.9: SF, NetLSD and FeatherGraph accuracy of embedding methods with different parameters of Random Forest Classifier: number of trees (e) on the horizontal, while the maximal depth of trees (d) on the vertical axis.

On the contrary, the other three embedding algorithms (SF, NetLSD, and FeatherGraph) performed better, with deeper trees almost disregarding their number.

Summary

In my thesis, I extensively studied various whole graph embedding methods, ranging from simple baseline algorithms to state-of-the-art solutions.

To begin with the theoretical background, I read about the methods and compared their approaches alongside complexity for deeper comprehension (see Chapter 1).

I employed two different classification measures to compare carefully selected methods. An important consideration was the runtime efficiency, especially for handling large graphs as they occur in real-life applications.

The first type of classification was performed on an artificial dataset of randomly generated graphs (see Section 2.1). The tasks were to tell apart different random models or their parameters (see Section 3.1). Four embedding algorithms consistently demonstrated excellent performance: GeoScattering, FGSD, FeatherGraph, and LDP. Notably, LDP being a rather simple and quick approach, was a pleasant surprise. However, due to their runtime limitations, GeoScattering and FGSD were not viable options for subsequent tasks.

The second classification task involved a real-life dataset comprising networks from six different domains (see Section 2.2 and 3.2). This second dataset contained graphs of size over 10^5 nodes. Therefore the consideration of the scaling of the runtime became essential. The choice of the classifier proved crucial in this domain classification task, as Random Forest consistently outperformed Logistic Regression. When evaluating the methods, three main objectives were taken into account: accuracy, runtime, and the dimension of the embedded vector. SF is a suitable algorithm for low-dimensional embeddings, although its runtime can be a limiting factor. Among the analyzed methods, FeatherGraph was overall the best method, with a remarkable performance in a reasonable runtime. This state-of-the-art algorithm excelled in both classification tasks, establishing itself as the best-performing method of this study.

Appendix A

Parameter Setups

The following tables present the hyperparameter setups of embedding algorithms utilized in the measurements discussed in Section 3.2.

Name of Embedding	**Parameter description**	**Parameter setups**
NetLSD (1.3.5)	• beginning and ending of timescale interval • number of timescale steps • number of eigenvalue approximation steps	[-0.2,0.2,250,200], [-0.2,0.2,500,400], [-0.2,0.2,250,400], [-0.2,0.2,500,200], [-0.5,0.5,250,200], [-0.5,0.5,500,400], [-0.5,0.5,250,400], [-0.5,0.5,500,200], [-0.1,0.1,250,200], [-0.1,0.1,500,400], [-0.1,0.1,250,400], [-0.1,0.1,500,200], [-0.5,0.5,100,200], [-0.5,0.5,150,200], [-0.5,0.5,200,200], [-0.5,0.5,300,200], [-0.5,0.5,400,200], [-0.5,0.5,600,200], [-0.5,0.5,700,200]
LDP (1.3.2)	• number of bins of the histogram	3, 4, 5, 6, 7, 8, 9, 10, 20, 32, 45, 56, 64, 72, 80, 90, 100, 110, 120, 130, 140

Table A.1: Parameter Setups of the Embedding Algorithms for Measurements (NetLSD, LDP)

Name of Embedding	**Parameter description**	**Parameter setups**
FeatherGraph (1.3.5)	• maximal distance to be considered in the characteristic function, or highest adjacency matrix power • maximal evaluation point value • number of evaluation points of the characteristic functions • pooling function	[5,2.5,25,"mean"], [5,2.5,25,"min"], [5,2.5,25,"max"], [5,2.5,40,"mean"], [5,2.5,40,"min"], [5,2.5,40,"max"], [5,1.5,25,"mean"], [5,1.5,25,"min"], [5,1.5,25,"max"], [5,1.5,40,"mean"], [5,1.5,40,"min"], [5,1.5,40,"max"], [6,4.0,40,"mean"], [6,4.0,40,"min"], [6,4.0,40,"max"], [5,2.5,15,"min"], [5,2.5,15,"max"], [5,2.5,20,"min"], [5,2.5,20,"max"], [5,2.5,30,"min"], [5,2.5,30,"max"], [5,2.5,35,"min"], [5,2.5,35,"max"], [5,2.5,40,"min"], [5,2.5,40,"max"], [6,3.0,15,"min"], [6,3.0,15,"max"], [6,3.0,20,"min"], [6,3.0,20,"max"], [6,3.0,25,"min"], [6,3.0,25,"max"], [6,3.0,30,"min"], [6,3.0,30,"max"], [6,3.0,35,"min"], [6,3.0,35,"max"], [4,2.5,25,"min"], [4,2.5,25,"max"], [4,2.5,30,"min"], [4,2.5,30,"max"], [4,2.5,35,"min"], [4,2.5,35,"max"], [4,2.5,40,"min"], [4,2.5,40,"max"], [4,2.5,45,"min"], [4,2.5,45,"max"], [4,2.5,50,"min"], [4,2.5,50,"max"], [3,2.5,30,"min"], [3,2.5,30,"max"], [3,2.5,35,"min"], [3,2.5,35,"max"], [3,2.5,40,"min"], [3,2.5,40,"max"], [3,2.5,45,"min"], [3,2.5,45,"max"], [3,2.5,50,"min"], [3,2.5,50,"max"]

Table A.2: Parameter Setups of the Embedding Algorithms for Measurements (FeatherGraph)

Name of Embedding	**Parameter description**	**Parameter setups**
graph2vec (1.3.3)	• dimension of the embedded vector • number of learning epochs • learning rate	[128,10,0.25], [128,30,0.2], [128,50,0.15], [200,10,0.25], [200,30,0.2], [200,50,0.15], [200,10,0.025], [200,30,0.02], [200,50,0.015], [400,10,0.25], [400,30,0.2], [400,50,0.15], [400,10,0.025], [400,30,0.02], [400,50,0.015], [500,10,0.25], [500,30,0.2], [500,50,0.15], [600,10,0.25], [600,30,0.2], [600,50,0.15], [600,10,0.025], [600,30,0.02], [600,50,0.015], [800,10,0.25], [800,30,0.2], [800,50,0.15], [800,10,0.025], [800,30,0.02], [800,50,0.015], [1000,10,0.25], [1000,30,0.2], [1000,50,0.15]
SF (1.3.4)	• number of eigenvalues desired	10, 15, 20, 25, 32, 40, 60, 80, 100, 128, 256

Table A.3: Parameter Setups of the Embedding Algorithms for Measurements (graph2vec, SF)

Bibliography

[1] Ferenc Béres et al. *Vaccine skepticism detection by network embedding.* 2021. arXiv: `2110.13619 [cs.SI]`.

[2] Ferenc Béres et al. *Blockchain is Watching You: Profiling and Deanonymizing Ethereum Users.* 2020. arXiv: `2005.14051 [cs.CR]`.

[3] Pinar Yanardag and S.V.N. Vishwanathan. "Deep Graph Kernels". In: KDD '15. Sydney, NSW, Australia: Association for Computing Machinery, 2015, 1365–1374. ISBN: 9781450336642. DOI: `10.1145/2783258.2783417`. URL: `https://doi.org/10.1145/2783258.2783417`.

[4] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. "Learning Convolutional Neural Networks for Graphs". In: *Proceedings of The 33rd International Conference on Machine Learning.* Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 2014–2023. URL: `https://proceedings.mlr.press/v48/niepert16.html`.

[5] Annamalai Narayanan et al. "subgraph2vec: Learning Distributed Representations of Rooted Sub-graphs from Large Graphs". In: (June 2016).

[6] Jianzong Du et al. "Graph Embedding Based Novel Gene Discovery Associated With Diabetes Mellitus". In: *Frontiers in Genetics* 12 (2021). ISSN: 1664-8021. DOI: `10.3389/fgene.2021.779186`. URL: `https://www.frontiersin.org/articles/10.3389/fgene.2021.779186`.

[7] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: `1301.3781 [cs.CL]`.

[8] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents*. 2014. arXiv: `1405.4053 [cs.CL]`.

[9] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk: Online Learning of Social Representations”. In: *CoRR* abs/1403.6652 (2014). arXiv: `1403.6652`. URL: `http://arxiv.org/abs/1403.6652`.

[10] Aditya Grover and Jure Leskovec. “node2vec: Scalable Feature Learning for Networks”. In: *CoRR* abs/1607.00653 (2016). arXiv: `1607.00653`. URL: `http://arxiv.org/abs/1607.00653`.

[11] Nesreen K. Ahmed et al. *Learning Role-based Graph Embeddings*. 2018. arXiv: `1802.02896 [stat.ML]`.

[12] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. “An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs”. In: *CoRR* abs/2003.04819 (2020). arXiv: `2003.04819`. URL: `https://arxiv.org/abs/2003.04819`.

[13] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[14] Marcell Nagy and Roland Molontay. “Network classification-based structural analysis of real networks and their model-generated counterparts”. In: *Network Science* (2022), pp. 1–24.

[15] Marcell Nagy and Roland Molontay. “On the Structural Properties of Social Networks and Their Measurement-Calibrated Synthetic Counterparts”. In: *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ASONAM ’19. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2020, 584–588. ISBN: 9781450368681. DOI: `10.1145/3341161.3343686`. URL: `https://doi.org/10.1145/3341161.3343686`.

[16] Chen Cai and Yusu Wang. “A simple yet effective baseline for non-attribute graph classification”. In: *CoRR* abs/1811.03508 (2018). arXiv: `1811.03508`. URL: `http://arxiv.org/abs/1811.03508`.

[17] Annamalai Narayanan et al. “graph2vec: Learning Distributed Representations of Graphs”. In: *CoRR* abs/1707.05005 (2017). arXiv: `1707.05005`. URL: `http://arxiv.org/abs/1707.05005`.

[18] Hong Chen and Hisashi Koga. “GL2vec: Graph Embedding Enriched by Line Graphs with Edge Features”. In: *Neural Information Processing*. Ed. by Tom Gedeon, Kok Wai Wong, and Minho Lee. Cham: Springer International Publishing, 2019, pp. 3–14. ISBN: 978-3-030-36718-3.

[19] Nathan de Lara and Edouard Pineau. “A Simple Baseline Algorithm for Graph Classification”. In: *CoRR* abs/1810.09155 (2018). arXiv: `1810.09155`. URL: `http://arxiv.org/abs/1810.09155`.

[20] Guanghan Xu and Thomas Kailath. “Fast estimation of principal eigenspace using Lanczos algorithm”. In: *SIAM Journal on Matrix Analysis and Applications* 15.3 (1994), pp. 974–994.

[21] Thomas Bonald, Alexandre Hollocou, and Marc Lelarge. *Weighted Spectral Embedding of Graphs*. 2018. arXiv: `1809.11115 [cs.LG]`.

[22] David I Shuman, Benjamin Ricaud, and Pierre Vandergheynst. *Vertex-Frequency Analysis on Graphs*. 2013. arXiv: `1307.5708 [math.FA]`.

[23] Alexis Galland and Marc Lelarge. “Invariant embedding for graph classification”. In: *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data*. Long Beach, United States, June 2019. URL: `https://hal.archives-ouvertes.fr/hal-02947290`.

[24] Saurabh Verma and Zhi-Li Zhang. “Hunt For The Unique, Stable, Sparse And Fast Feature Learning On Graphs”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper_files/paper/2017/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf`.

[25] Mario Botsch, David Bommes, and Leif Kobbelt. “Efficient Linear System Solvers for Mesh Processing”. In: *Proceedings of the 11th IMA International Conference on Mathematics of Surfaces*. IMA’05. Loughborough, UK: Springer-Verlag, 2005, 62–83. ISBN: 3540282254.

DOI: 10.1007/11537908_5. URL: https://doi.org/10.1007/11537908_5.

[26] Anton Tsitsulin et al. "NetLSD: Hearing the Shape of a Graph". In: *CoRR* abs/1805.10712 (2018). arXiv: 1805.10712. URL: http://arxiv.org/abs/1805.10712.

[27] Feng Gao, Guy Wolf, and Matthew Hirn. "Geometric Scattering for Graph Data Analysis". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 2122–2131. URL: https://proceedings.mlr.press/v97/gao19e.html.

[28] Benedek Rozemberczki and Rik Sarkar. *Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models*. 2020. arXiv: 2005.07959 [cs.LG].

[29] Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge: Cambridge University Press, 2016. ISBN: 9781107076266 1107076269. URL: http://barabasi.com/networksciencebook/.

[30] Petter Holme and Beom Jun Kim. "Growing scale-free networks with tunable clustering". In: *Physical review E* 65.2 (2002), p. 026107.

[31] *NetworkX package*. https://networkx.org/documentation/stable/reference/generators.html.

[32] *Structural analysis of real networks and their model-generated counterparts – Supplementary data*. https://github.com/marcessz/Complex-Networks.

[33] Sami Abu-El-Haija et al. *Watch Your Step: Learning Node Embeddings via Graph Attention*. 2018. arXiv: 1710.09599 [cs.LG].

[34] *scikit-learn Python package*. https://scikit-learn.org/stable/index.html.

List of Figures

List of Tables