UNIVERSITY OF EDINBURGH

MASTER THESIS

Diffusion to Vector – Scalable Representation Learning of Graphs

Benedek András Rózemberczki

Supervisor:
Dr. Rik Sarkar

A thesis submitted in fulfilment of the requirements for the degree of Master of Research in Data Science

in the

Center for Doctoral Training in Data Science
School of Informatics

August 18, 2017

Declaration of Authorship

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Benedek András Rózemberczki)

Signed:

Date:

University of Edinburgh

Abstract

School of Informatics

Master of Research in Data Science

Diffusion to Vector –
Scalable Representation
Learning of Graphs

by Benedek András Rózemberczki

A large number of complex networks has no generic vertex attributes that can be used in network analysis. This means that one has to generate vertex features purely using the topological properties of nodes. One way to extract such features is the creation of a graph embedding. A graph embedding is a representation of the graph in a latent space created by a graph embedding procedure. This representation maintains the distances observed on the graph in the latent space. A type of graph embedding procedures uses features extracted from linear sequences of vertices to create embeddings – these are the so called vertex sequence based embedding procedures. Earlier approaches to sequence based graph embedding are poorly parallelizable, require complicated preprocessing and slow down under some realistic assumptions about graph evolution. In this paper, we propose two diffusion based network embedding algorithms, ED2V and FD2V, that are parallelized, require straightforward pre-processing and their performance is fairly robust to graph evolution. In our experiments we start with showing that our algorithms have consistently good computational performance and they are quite robust to graph densification. After this, we provide evidence that our procedures preserve graph distances and centralities in the embedding space. Finally, in a comprehensive investigation we prove the high quality of representations generated with our approach on a number of downstream machine learning applications. We evaluated features extracted with our embedding methods by performance on multi-label node classification, community detection and edge prediction. Our algorithms are on par with the state of the art sequence based embedding method described by Grover & Leskovec (2016) and on certain tasks and benchmark networks outperform it.

KEYWORDS: social networks, diffusion process, graph embedding, edge prediction, node labeling, community detection, distributed computing

Acknowledgements

This research project was supervised by Dr Rik Sarkar. His pragmatic approach and our discussions helped to considerably refine my ideas about the topic. Moreover, he supported me beyond limits when he corrected my final draft during his vacation and provided detailed feedback on it. I also have to express my gratitude towards fellow MRes and PhD students in the Centre for Doctoral Training in the Data Science. Discussions about related work and possible applications of my method contributed greatly to this project.

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

$\mathcal{G}$	A graph
V	Vertex set of a graph
E	Edge set of a graph
v, w, u	Vertices
(v, w)	Endpoint vertices of an edge
K_0	Null graph
$\mathbf{A}, \mathbf{D}, \mathbf{L}$	Adjacency, degree and Laplacian matrices of a graph
S	Seeder set
P	Vertex sequence
$\widehat{w}$	Window size
d	Feature vector dimensions
α	Learning rate
n	Number of diffusions
k	Number of asynchronous gradient descent epochs
$\mathbf{X}$	Graph embedding matrix
$\mathbf{b}$	Bias vector
$\mathbf{W}$	Weight matrix
$\mathbf{h}$	Hidden representation
$\mathbf{y}^*$	Normalized hitting frequency vector
$\mathbf{y}'$	Estimated hitting frequency vector
λ	Shrinkage parameter
$\mathbf{c}$	Community membership vector
β	Attenuation/addition rates
γ	Minimal neighbourhood overlap
η	Minimal degree
$\mathbf{M}$	Symbolic empty matrix
$\widetilde{\mathbf{X}}$	Edge feature matrix
$\widetilde{\mathbf{y}}$	Edge existence vector
$\deg(\cdot)$	Degree of a vertex
$\mathcal{N}(\cdot)$	Neighbourhood set of a node
$\overline{\cdot}$	Average of a quantity
$\delta(\cdot)$	Delta function
$d(\cdot)$	Shortest path between two nodes
$d'(\cdot)$	Distance between two nodes in latent space
$\lvert \cdot \rvert$	Cardinality of a set
$\mathcal{Q}(\cdot)$	Modularity of a graph clustering
$\mathcal{L}(\cdot)$	Multinomial logarithmic loss
$\sigma(\cdot)$	Identity function
$\Phi(\cdot)$	Softmax function

Chapter 1

Introduction

1.1 Why do we need graph embeddings?

Many real world machine learning applications involve the use of network data. One might want to do tasks such as; classification of users in a social network (Macskassy & Provost, 2003), recommendation of scientific collaborations (Liben-Nowell & Kleinberg, 2007), finding customers who buy similar products (Fouss et al., 2007), or identification of influential members in a community (Kempe et al., 2003). Working with large networks involves two main challenges that one has to face. First, graphs that represent real networks are known to have a low density (Strogatz, 2001; Newman, 2003; Boccaletti et al., 2006). A low density means that only a small fraction of edges exist out of all the potential edges among vertices. The Youtube social network is a prime example for this phenomenon – earlier research shows that in this graph only 0.00046% of the potential edges exist (Yang & Leskovec, 2015). When such sparse graphs are represented by an adjacency matrix, the representation will only have a few non zero values. In addition, even for a fairly small graph the adjacency matrix is considerably large. Second, networks tend to have a high rate of missingness when it comes to the so called generic vertex features (Kossinets, 2006; Huisman, 2009). In case of a social network generic vertex features can be the age, country of origin and interests of users. A high missingness itself means that for a large number of samples we partially have generic vertex features. Moreover, missingness of the generic vertex features tends to be spatially concentrated which introduces further challenges. The problems that these two empirical regularities cause can be solved by *embedding the graph* into a continuous space.

Now let us imagine that we have a *graph embedding procedure* which can generate coordinates for each node of the graph in a low dimensional continuous space solely using the graph topology. Moreover, let us assume that in this space the graph distances among nodes are preserved approximately. Using this representation of the graph would solve the problems we discussed earlier. First, we would have a low dimensional continuous representation of our graph instead of the high dimensional discrete adjacency matrix that is sparse. Second, as we are not using generic vertex features we do not have to worry about missing values. There is a number of graph embedding procedures that solely use the topology to create the embedding. Based on the taxonomy of Goyal & Ferrara (2017) there are three types of methods which can be used for the generation of such embeddings. These are specifically, (i) factorization procedures (Cao et al., 2015; Ou et al., 2016), (ii) vertex sequence based embedding techniques (Perozzi et al., 2014; Grover & Leskovec, 2016) and (iii) deep learning (Defferrard et al., 2016; Kipf & Welling, 2016a,b). Our current work contributes to the literature of vertex sequence based embedding techniques. We are going to demonstrate that diffusion processes can be used to generate node sequences for creating high quality graph embeddings.

1.2 Contributions

In this thesis we introduce diffusion tree based graph embedding algorithms that result in low-dimensional representations of nodes. The diffusion tree that we use to create the embedding could describe the adoption path of a message or the spreading of a disease on a graph from a source node. Using the diffusion trees that we obtain by simulation we create linear sequences of nodes that we utilize to extract features of nodes. Diffusion tree based vertex sequence generation compared to other sequence creation procedures results in node sequences that describe more closely knit local neighbourhoods in the graph. Meaning that the diffusion tree sampled induced subgraphs tend to be dense, clustered and have a low diameter. Because of this, the features that are extracted are more descriptive regarding local structure of the graph than features obtained by using other random processes on the graph (e.g. random walks). We investigate the computational performance of our algorithms. Furthermore, we evaluate the representation quality of embeddings on downstream machine learning tasks such as multi-label node classification, community detection and edge prediction. Specifically our theoretical and empirical contributions are:

THEORETICAL CONTRIBUTIONS

(i) We introduce diffusion tree based node sampling procedures to generate vertex sequences that can be used for learning graph embeddings. In addition, we describe two distinct methods that can create linear sequences of nodes from diffusion trees. Later these vertex sequences are used to create the node embedding.

(ii) We propose a general framework for creating synthetic datasets for testing the representation quality of embeddings on the edge prediction task. Our scheme allows for totally synthetic dataset creation but also for generating more realistic data for representation quality testing.

(iii) We implemented the diffusion tree based embedding procedures and the synthetic edge prediction dataset generators in Python.

EMPIRICAL CONTRIBUTIONS

(i) We investigate the computational performance of our methods compared to other node sequence based embedding methods. We show that our approach has considerable speed advantage in the graph pre-processing and sequence generation phases on real and certain synthetic graphs. In addition, we establish that our performance advantage increases as the size of the graph or its density increases.

(ii) We provide evidence that the distance between nodes on the graph is approximately preserved in the latent embedding space that is created by the mapping procedure. Moreover, our experiments reveal that those nodes which are central in the graph are also central in the embedding space we create. Furthermore, we highlight the visualization capabilities of the embeddings.

(iii) We demonstrate that the quality of learnt representations is competitive with other node-sequence based graph embedding methods when the downstream machine learning task is multi-label node classification. On a number of widely used benchmark datasets we perform similarly to other node sequence based graph embedding methods such as *Node2Vec* and *DeepWalk* (Grover & Leskovec, 2016; Perozzi et al., 2014).

(iv) We present exclusive results on the task of community detection and show that our algorithms create the highest quality embeddings in this regard. Our results are competitive with widely used graph clustering methods and scale to networks with a million of nodes.

(v) We evaluate the sensitivity of representation quality to embedding algorithm parameter changes on the multi-label node classification and community detection tasks.

(vi) We validate that our embeddings can be used effectively to perform edge prediction under the conditions that are widely used to benchmark embedding representation quality. We also support results on this task under a more realistic evaluation regime.

1.3 Thesis outline

Background, related work and the task of sequence based graph embedding are discussed in Chapter 2. The discussion covers sequence generation, feature extraction and representation learning in quite detail. The ideas of parallelized diffusion tree generation and linearisation are outlined in Chapter 3. Computational performance of the reference implementation is benchmarked to implementations of other vertex sequence based graph embedding methods in Chapter 4. Basic properties of the embeddings are presented in Chapter 5 where the visualization capabilities of the proposed algorithms are also highlighted. In Chapter 6 we evaluate the representation quality of the embeddings on downstream machine learning tasks. Section 6.1 is about multi-label node classification, the first downstream machine learning application of the graph representation method introduced by this thesis. The second application, community detection (vertex clustering), is examined in Section 6.3. Results regarding the last application, edge prediction, are presented in Section 6.3. The thesis ends with Chapter 7 which includes the concluding remarks and gives layout for possible extensions of the work in hand.

Chapter 2

Background and related work

In this chapter we are going to introduce basic definitions, discuss the literature and applications of graph embeddings. Finally, we will review node sequence based graph embedding procedures in detail. Definitions related to graph embedding procedures are presented in Section 2.1. The literature of graph embedding methods is discussed in Section 2.2. We review the main types of embedding procedures in Subsection 2.2.1 while possible applications of the embeddings are considered in Subsection 2.2.2. We discuss extensively node sequence based graph embedding methods in Section 2.3, we study feature extraction from sequences in Subsection 2.3.1 and consider learning from the extracted features in Subsection 2.3.2.

2.1 Basic terminology

In this section we overview basic terminology regarding graph embedding procedures. We will introduce a number of definitions and comment on them where needed. Our discussion loosely follows the work of Goyal & Ferrara (2017) and extends on it.

Definition 2.1. *(Graph) A graph $\mathcal{G}(V, E)$ consists a vertex set denoted by V and an edge set noted by E. The edge set is a set of pairs (v, w), where $v, w \in V$.*

We assume that the graph is a mathematical representation of a real world network. We use the vertex-node and edge-link words interchangeably in the whole paper. The cardinality of the vertex and edge sets is denoted by $|V|$ and $|E|$ in the thesis. These cardinalities are respectively the number of nodes and links in the graph.

Definition 2.2. *(Adjacency matrix) An adjacency matrix $\mathbf{A}$ is a $|V| \times |V|$ square matrix with binary values which describes the existence of connections between nodes of a graph. Columns and rows correspond to nodes in an ordered way. An element $\mathbf{A}_{i,j} = 1$ if two nodes i and j are adjacent else $\mathbf{A}_{i,j} = 0$.*

Based on Definition 2.2 one can deduce that the adjacency matrix is a symmetric matrix for non directed graphs. Moreover, it only has non zero elements in the diagonal if self loops of nodes are allowed. It is evident that for a large graph it has a really high dimensionality. It is not necessarily a positive semidefinite matrix.

Definition 2.3. *(Degree matrix) The degree matrix $\mathbf{D}$ is a $|V| \times |V|$ diagonal square matrix obtained by putting the degree of each node to the diagonal elements. An element $\mathbf{D}_{v,w} = \deg(v)$ if $v = w$ else it is 0.*

Definition 2.4. *(Laplacian matrix) The Laplacian matrix $\mathbf{L}$ is a $|V| \times |V|$ square matrix obtained by subtracting the adjacency of a graph by the degree matrix. Simply it is defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$.*

It is important to note that the Laplacian is always a positive semidefinite matrix. This means that it has no negative eigenvalues.

Definition 2.5. *(First-order proximity) The first-order proximity $s(v, u)$ of nodes v and u is the edge weight of (v, u). In an unweighted graph it is 1 when an edge exists between the nodes and it is 0 when an edge does not exist.*

The definition of first-order proximity implies that the adjacency matrix and its weighted variant describe the first-order proximities between pairs of nodes.

Definition 2.6. *(Second-order proximity) The second-order proximity of two nodes v and u is defined by the proximity of their neighbourhoods. Let we define the first-order proximity vector of node v and all other nodes in V as $S_v = [s(v, 1), s(v, 2), \dots, s(v, |V|)]$. Based on this, the second-order proximity of nodes v and u are defined as the similarity of vectors S_v and S_u.*

The way definition 2.6 describes second-order proximity allows for the choice of a similarity measure. For example the neighbourhood overlap and the cardinality of the common neighbourhood of two nodes can be used as second-order proximities.

Definition 2.7. *(First-order distance) The first-order distance of nodes v and u is the distance between nodes $d(v, u)$ defined on graph $\mathcal{G}$ measured by some distance measure.*

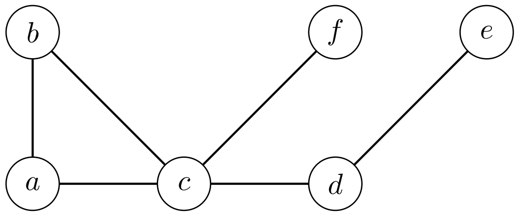
A simple first order distance measure between a pair of nodes is the shortest path distance between v and u. This is the minimal number of hops required to reach node u from v.

Definition 2.8. *(Second-order distance) The second-order distance of two nodes v and w is defined by the distance of their distances from other nodes. Let us define the first-order distance vector of node v and all other nodes in V as $d_v = [d(v, 1), d(v, 2), \dots, d(v, |V|)]$. Based on this, the second-order distance of nodes v and u is defined as the distance of vectors d_v and d_u.*

The intuition behind Definition 2.8 is that if two nodes are approximately at the same distance from other nodes in V and close to each other they have a low second-order distance. Similarly to second-order proximity this definition allows for an arbitrary distance measure between the two distance vectors.

Definition 2.9. *(Graph embedding procedure) A graph embedding procedure defined on $\mathcal{G}(V, E)$ is a mapping $f : V \rightarrow \boldsymbol{X} \in \mathbb{R}^{|V| \times d}$ satisfying that $d \ll |V|$ which preserves certain proximity or distance measures defined on $\mathcal{G}(V, E)$*

It is important to see that if we would not condition on having a low dimension compared to the number of vertices, meaning that $d \ll |V|$ does not hold, we would not have a compact representation of the graph. Because having a compressed representation is one of our goals it is a fundamental requirement. As definition 2.9 is quite general we have to point out that when we create an embedding procedure we have the freedom to choose an arbitrary proximity or distance measure that we preserve. The definition of the chosen distance or proximity is not limited by the formal definitions that we introduced. For example earlier approaches to sequence based graph embedding algorithms (Perozzi et al., 2014; Grover & Leskovec, 2016) used approximated random walk proximities (Newman, 2005) between nodes. We have to emphasize that in our work the graph embedding procedures themselves are also referenced as graph embedding algorithms, functions, and methods. These terms have somewhat similar meaning so we use them interchangeably in our work. The embedding procedure described by Definition 2.9 itself can be understood by a simple example. Let us imagine that we have a graph with the vertex and edge sets $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (b, c), (c, a), (a, d), (d, e), (c, f)\}$. Based on the graph $\mathcal{G}(V, E)$ an embedding function can create a visualization like Figure 2.1, where the coordinates of vertices in the 2 dimensional plane are defined by the embedding matrix $\mathbf{X}$.



$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \\ 3 & 1 \\ 2 & 1 \end{bmatrix}$$

Figure 2.1: Example graph embedding. The graph $\mathcal{G}(V, E)$ is defined by vertex set $V = \{a, b, c, d, e, f\}$ and edge set $E = \{(a, b), (b, c), (c, a), (a, d), (d, e), (c, f)\}$. We created the layout with a 2 dimensional embedding that we just randomly made up. The embedding itself is a 6×2 matrix defined by **X**.

2.2 Graph embeddings

In this section first we briefly overview the main types of graph embedding methods and discuss numerous possible applications of these procedures. Embedding methods are in the focus of Subsection 2.2.1 while possible applications are listed in Subsection 2.2.2.

2.2.1 Embedding procedure types

In this subsection we discuss the most well known graph embedding techniques and recent developments regarding them. The discussion involves graph factorization techniques, sequence based embedding methods and deep learning.

Factorization techniques

Graph factorization techniques use a matrix that describe the graph and factorize it in order to create the embedding of the network. The matrix used to represent the graph can be the adjacency matrix itself, the neighbourhood overlap matrix or the graph Laplacian. Based on the properties of the matrix either eigenvalue decomposition or some variant of stochastic gradient descent is used to obtain the graph embedding. These embedding methods all have a weakness, namely that they cannot incorporate generic vertex features. Besides this, their computational complexity is usually $\mathcal{O}(|V|^2)$ – in case of large graphs this complexity is prohibitive.

The *Laplacian Eigenmaps* method by Belkin & Niyogi (2003) creates embedding vectors that can reproduce the weights of the graph Laplacian. Weights of the Laplacian are approximately reconstructed by the inner product of the embedding vectors. *Graph factorization* by Ahmed et al.

(2013) works in a similar fashion, the inner product of the embedding vectors reconstructs the weights of the adjacency matrix. Likewise the procedures introduced by Cao et al. (2015) and Ou et al. (2016) factorize the random walker transmission probability and neighbourhood overlap matrices.

Sequence based embedding methods

Node sequence based graph embedding methods were inspired by word embedding procedures, specifically by the *skip-gram* model (Mikolov et al., 2013a,b). The generation of node sequence based graph embeddings consists three phases. First, the algorithm creates synthetic vertex sequences. The sequence generation itself involves a random process on the graph that creates sequences of nodes. This can be demonstrated by an illustrative example. Let us consider again the graph on Figure 2.1. Moreover, let us assume that the random process on the graph is a random walker that makes truncated random walks with length 4 starting from randomly chosen vertices. A sequence starting from node c could be $c - b - a - c$ while a sequence starting from node f could be $f - c - d - e$. Second, after the sequence generation node specific features are extracted for each of the nodes based on the sequences. The features that are extracted from the synthetic sequences describe the approximated proximities of nodes. Third, finally the embedding itself is learned using the extracted node specific features with a neural network which has a single hidden layer. Rows of the input weight matrix are taken as the node embeddings for each node. The sequence based embedding methods are all characterized by $\mathcal{O}(\log(|V|) \cdot |V|)$ computational complexity as they use certain simplifications to speed up the embedding creation.

Sequence based embedding models originate from the *DeepWalk* model by Perozzi et al. (2014) who used random walks to generate the node sequences. Based on these sequences the extracted features approximate node specific random walk closeness centralities described by Newman (2005) and Brandes & Fleischer (2005). These centralities measure proximity by measuring the frequency that a random walker starting from a given node hits other nodes. The higher the hitting frequency is the closer the source and the other node is. This approach was generalized by Grover & Leskovec (2016) who proposed second-order random walks to generate the vertex sequences and named their method as *Node2Vec*. Second-order random walks alternate between depth-first and breadth-first search on the graph in a random, but somewhat controlled way. The main drawback of this model is that it has parameters that control the alternation between the search strategies and the embedding's representation quality depends on these parameters. Because of this one has to find the optimal parameters with a quite costly grid-search to obtain a high quality graph representation. Recently Pimentel et al. (2017) further refined the sequence based embedding procedures by *Neighborhood Based Node Embedding* which requires less synthetic data to learn the graph representation.

Importantly these models that we listed do not use generic vertex features for the creation of the embedding, so extending them with the inclusion of node metadata was a straightforward extension by Yang et al. (2016). Other related works consider graphs with specific topology such as graphs with signed edge weights (Yuan et al., 2017), rooted subgraphs (Narayanan et al., 2016) and temporal graphs (Rahman & Al Hasan, 2016).

Deep learning

Graph embedding methods that use deep learning are all based on the same idea – they encode a matrix representation of the graph and at the same time they maintain first and higher order proximities or distances. The matrix representation used for creating the embedding is the adjacency matrix in the model created by (Wang et al., 2016). While (Cao et al., 2016) uses the random walk transmission probability matrix. These representations are known to have higher quality than the ones obtained by sequence based graph embedding methods. However, due to complexity constraints (a considerable computational complexity of $\mathcal{O}(|E| \cdot |V|)$) benchmarks are only presented on small graphs by the authors. Importantly, the architectures introduced by Wang et al. (2016) and Cao et al. (2016) did not use node features. Nevertheless, recent developments in graph signal processing allow the use of generic vertex features when an embedding is created. For a detailed introduction to the topic see Shuman et al. (2013). Building on this progress of graph signal processing Defferrard et al. (2016) and (Kipf & Welling, 2016a) introduced semi-supervised graph convolutional deep neural networks that create embeddings that use generic vertex features and at the same time maintain proximities of nodes in the embedding space. They use their method on citation network data with word occurrence features extracted from the papers to classify them into scientific fields. They show that graph convolutional neural networks achieve state of the art results on node labeling if generic vertex features are available. An unsupervised procedure to create embeddings is discussed in Kipf & Welling (2016b). We have to note that these convolutional networks have an appealing computational complexity of $\mathcal{O}(|E|)$ and their memory complexity is the same. Essentially this means that for smaller graphs one can use high performance deep learning libraries with graphical processing unit support to fit these models.

2.2.2 Possible applications

Embedding procedures create low dimensional and continuous representations of input graphs. These learnt representations have a number of useful applications. Some of them is quite evident and widely discussed in the literature. Moreover, some of the possible applications is used to benchmark the representation quality of embedding algorithms. We must note that the fact that embeddings can be used in supervised machine learning tasks such as regression and node labeling originates from the phenomena called homophily and assortative mixing. These terms describe the *birds of a feather* regularity observed in most of the social and technological networks (McPherson et al., 2001). Meaning that nodes with similar features are connecting to each other with a higher probability. As embeddings preserve distance and proximity in the latent space one can exploit homophily and assortativity to do supervised and unsupervised learning.

Visualization

Network visualization is the most evident application of the learnt embeddings. Graph visualizations are powerful exploratory data analysis tools. They can help to identify the possible number of clusters or assess the level of homophily/assortativity when one colours nodes by discrete or continuous generic vertex features. Besides these, one might also use them to get a general understanding of the network – whether bridges are present, are there large communities or small fragmented ones, or how closely knit the communities are. The graph visualization capabilities of sequence based embedding algorithms are emphasized by Perozzi et al. (2014); Grover & Leskovec (2016); Kipf & Welling (2016a) on some small real world networks while Goyal & Ferrara (2017) presents results on synthetic graphs for a number of embedding procedures.

Node labeling

Node labeling is a supervised learning task where one wants to classify nodes into different categories. One can predict the topic of scientific papers in citation networks to different topics, pages that users in a social network will like or the function of proteins in a protein-protein interaction network. Nodes might belong to multiple category at the same time – in such situations the task is multi-label node classification. There are algorithms that solve this task without creating an embedding such as the *weighted vote relational neighbour classifier* (Macskassy & Provost, 2003) or using node features extracted with the *label propagation algorithm* (Gregory, 2010). While these algorithms do classification without embedding creation most of them has a low classification accuracy compared to embedding based procedures. The referenced embedding algorithms perform well when features extracted with them are used to do node labeling. Moreover, node labeling is an important task to consider as the quality of the embedding representation is primarily evaluated on this task in the literature.

Results obtained by (Perozzi et al., 2014) using social network data collected from sites such as Flickr, BlogCatalog and Youtube demonstrated that sequence based embedding techniques support good features for node labeling with regards to group membership and interests. In addition, they demonstrated that predictions using features extracted with sequence based embedding are extremely accurate even when only 1% of data is labelled and plain logistic regression is used to classify nodes. Findings by Grover & Leskovec (2016) extend the investigation of (Perozzi et al., 2014) by showing that features that were created by sequence based graph embeddings are valuable for node labeling when protein-protein interaction networks are considered. Findings of Pimentel et al. (2017) gave further evidence to support the capabilities of these methods on the same datasets. Experiments by Kipf & Welling (2016a) had shown that embedding based methods do fairly on citation networks but they are outperformed by graph convolutional networks that can use node features. Importantly, the work of Kipf & Welling (2016a) was limited to node classification on small graphs with thousands of vertices.

Regression

Regression on networks corresponds to predicting continuous (non-categorical) features of nodes. One might try to infer quantities like the age of users in a social network, the amount of traffic that visits a blog or the number of citations a scientific paper will receive (Al Zamal et al., 2012; Peersman et al., 2011; Sarigöl et al., 2014). The simplest model that solves this task on networks is similar to the *weighted vote relational neighbour classifier* (Macskassy & Provost, 2003; Al Zamal et al., 2012). It extracts a neighbourhood of nodes and uses the weighted average of the neighbours' features to predict the quantity of interest. Currently there is little research on the usefulness of node sequence based embedding learnt features for regression tasks. The only work that considers this is an application of the method described by Perozzi et al. (2014). The same authors show that the created embedding can be used to predict the age of users and that their methods outperform a number of simple baselines (Perozzi & Skiena, 2015).

Edge prediction

Edge prediction is essentially a binary classification task – based on a pair of nodes one has to predict whether a link is formed between them or not. This description of the task itself makes it evident that in an optimal scenario one has temporal snapshots of the network. However, in most of the cases it is not possible to obtain temporal data and because of this synthetic data is needed. Classical non embedding based link prediction procedures that solve this task use similarity metrics of the neighbourhood sets of nodes as predictors. These metrics include the *neighbourhood overlap*, *preferential attachment index* or the *SimRank measure* (Liben-Nowell & Kleinberg, 2007).

While Perozzi et al. (2014) does not consider solving this downstream machine learning task Grover & Leskovec (2016) had illustrated that embedding methods create features that are effective to predict link formation among nodes. Results on social and collaboration networks presented by Grover & Leskovec (2016) show that sequence based embeddings outperform other baseline methods on this task. The findings of (Pimentel et al., 2017) support additional evidence that sequence based graph embedding procedures perform well on this task. The edge prediction performance of graph convolutional neural networks is discussed by (Kipf & Welling, 2016b) where deep learning methods are evaluated on citation networks and generic vertex features are used for creating an embedding. We have to point out that most of the embedding methods are evaluated in an unrealistic setting with synthetic data. In our opinion this evaluation questions the link prediction performance of the models that use embedding features.

Community detection

The extraction of communities from a network is essentially the clustering of nodes based on topology. This is one of the simplest unsupervised machine learning tasks that one can do with networks. Standard approaches to community detection do not create latent space embeddings of the nodes to cluster them, but use micro level topological properties of nodes to group them into dense subgraphs (Girvan & Newman, 2002; Clauset et al., 2004; Pascal & Latapy, 2005;

Blondel et al., 2008). As the survey presented in Goyal & Ferrara (2017) points out previous research on sequence based graph embeddings does not consider this task to evaluate the learned representation quality. We are going to fill this gap and show that using features derived with these methods to cluster nodes in latent space gives impressive clustering results on a number of social, technological and biological networks.

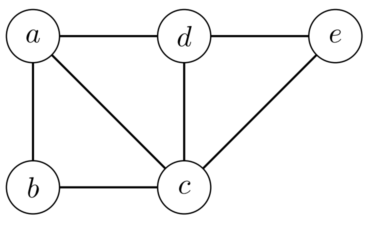
2.3 Creating node sequence based graph embeddings

In this section we will give a general overview on how node sequence based graph embedding procedures work. First, we discuss the extraction of node features from vertex sequences in Subsection 2.3.1. Second, in Subsection 2.3.2 we demonstrate how we learn the embedding based on the extracted features. In both subsections we give examples to illustrate how the mechanics of node sequence based graph embedding procedures work.

2.3.1 Feature generation using node sequences

Let us consider the node sequence generation first. In the following the graph representing the network is again denoted by $\mathcal{G}(V, E)$. The set of vertices is V while the set of edges is E. In the remainder we assume that graph can be either directed or undirected and also that edge weights are not present.

With the framework that node sequence based graph embedding methods use in order to generate node specific features one needs linear sequences of nodes. Moreover, it is required that each of the nodes appears at least in one of the sequences. These sequences of nodes might be present in the data naturally or generated by some artificial process. Examples of non-synthetic node sequences include content sharing patterns on social networks, movement of vehicles between traffic points or series of financial transactions among bank customers. Regarding the synthetic node-sequences the data generation possibilities that can be considered include random walks, second order random walks or diffusion processes. The paper in hand proposes the application of diffusion processes as they have amenable properties regarding the representation quality. The use of diffusion processes to create sequences and description of parallel sequence generation is described in Chapter 3.



$$a - b - c - d - c - d - e - c - d - c - d$$

$$e - d - e - d - c - d - e$$

$$b - a - c - d - a - b - a - c - b - c - d$$

Figure 2.2: Example graph with linear vertex sequences. The graph $\mathcal{G}(V, E)$ is defined by vertex set $V = \{a, b, c, d, e\}$ and edge set $E = \{(a, b), (b, c), (c, a), (c, d), (d, a), (d, e), (c, e)\}$. Three vertex sequences are listed with differing lengths and these sequences are used for feature extraction in our example.

Now let us assume that we have already available node sequences and these sequences were extracted from the graph depicted on Figure 2.2. Specifically, the vertex set contains nodes

a, b, c, d, e where these nodes are indexed respectively from 1 to 5. Furthermore, we assume that we have 3 node sequences available for feature extraction.

In the following we will consider a simple example to illustrate how node sequence based embedding algorithms work. To generate features from the sequences at hand we need to choose a sliding window size denoted by $\widehat{w}$. The window size that we choose limits the maximal graph proximity among nodes that we are going to approximate. In this specific case we consider that $\widehat{w} = 2$. We calculate the co-occurrence frequencies for node c as follows – we count how many times other nodes appeared at given positions before and after node c limited by the windows size. In this toy example it means positions at maximal 2 steps before or after c in the sequence. Counts at different positions are stored in separate vectors for each node. The resulting frequency vectors at different positions are as follows:

$$\mathbf{y}_{c,-2} = \begin{bmatrix}1\\2\\3\\1\\1\end{bmatrix} \quad \mathbf{y}_{c,-1} = \begin{bmatrix}2\\2\\0\\3\\1\end{bmatrix} \quad \mathbf{y}_{c,+1} = \begin{bmatrix}0\\1\\0\\7\\0\end{bmatrix} \quad \mathbf{y}_{c,+2} = \begin{bmatrix}1\\0\\3\\0\\2\end{bmatrix}$$

The components of these vectors have a well defined meaning. For example, the 2^{nd} component of $\mathbf{y}_{c,-2}$ is the number of times node b appeared 2 steps before node c in the processed sequences. Similarly, the 4^{th} component of $\mathbf{y}_{c,+1}$ is the number of times node d appeared 1 step after node c in our sequences. These feature vectors can form a vector with $2 \cdot \widehat{w} \cdot |V|$ components if they are concatenated in increasing order of the positions. We coined this vector as the hitting frequency vector of node c and denote it by $\mathbf{y}^*_c$. The general notation of this hitting frequency vector specific to a given node node v is $\mathbf{y}^*_v$. Importantly this vector is a vector of features for node c that we just generated. Later this vector is normalized by the sum of components so the hitting frequencies sum up to zero. By applying the sliding window on each sequence for each of the nodes we obtain a hitting frequency vector for each node – we extract features for all of them. This means that altogether after processing each sequence we are going to have $|V|$ feature vectors that each have a size of $2 \cdot \widehat{w} \cdot |V|$.

Components of the vectors can be interpreted in a less formal way as noisy proximities in the graph. The higher a given component of the hitting frequency vector is, the closer the respective node is to the node which the hitting frequency vector corresponds to. These vectors characterize the location of the node in the graph quite well but at the same time they have a high dimensionality. Later when we learn the representation of nodes with the embedding function we essentially reduce the dimension of these hitting frequency vectors.

2.3.2 Learning the embedding

In this section we consider the learning of an embedding as we already discussed feature generation in the previous subsection. We defined the embedding function as a mapping between

$\mathcal{G}(V, E)$ and $\mathbf{X} \in \mathbb{R}^{|V| \times d}$ in Section 2.1. The hitting frequency vectors themselves that we just created are already a representation of the network, but not in $\mathbb{R}^{|V| \times d}$ as they describe the graph in $\mathbb{R}^{|V| \times 2 \cdot \widehat{w} \cdot |V|}$. In order to resolve this Perozzi et al. (2014) proposes based on the ideas of Mikolov et al. (2013a,b) the use of neural networks to compress the node-graph representation. Their solution is that the embedding itself is created by fitting a fully connected neural network with a single hidden layer. We will adopt this idea and a schematic of the architecture that we will use, specific to the toy example investigated in this section, is on Figure 2.3.

$$\mathbf{y}'_v = \Phi(\mathbf{W}_{out}^T \mathbf{h}_v + \mathbf{b}_{out})$$

$$\mathbf{h}_v = \sigma(\mathbf{W}_{in}^T \mathbf{x}_v + \mathbf{b}_{in})$$

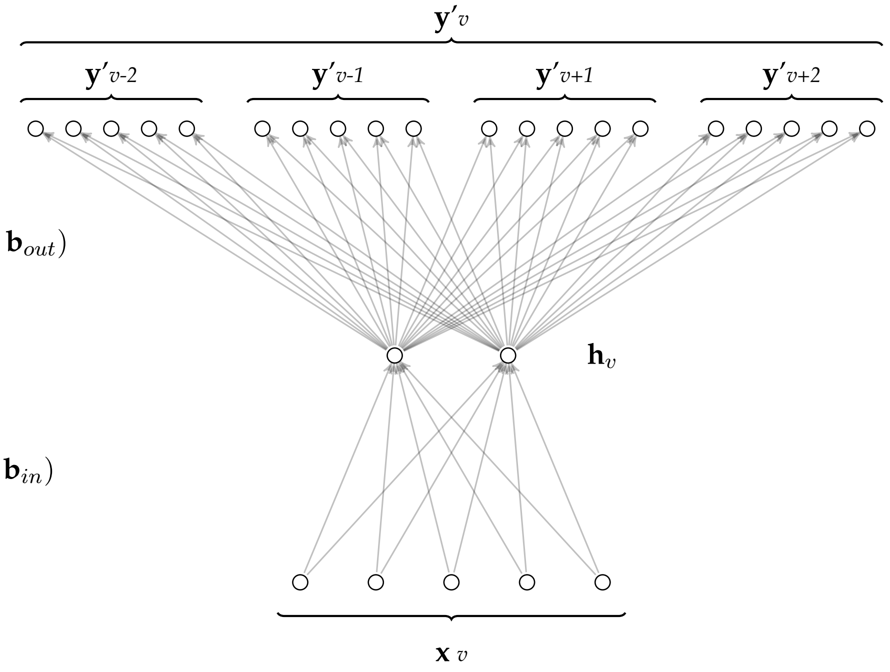


Figure 2.3: Example architecture of a neural network used for creating a graph embedding. The binary input vector has $|V|$ components and in this specific example we create the embedding of a graph with 5 nodes and we use a window size of 2. The embedding weight matrix $\mathbf{W}_{in}$ is a $|V| \times d$ matrix – in our example we have 2 neurons (number of embedding dimensions) in the hidden layer. So the input weight matrix is 5×2. The input bias $\mathbf{b}_{in}$ vector has d components – again in this instance this means 2 components. Our goal is to approximate the hitting frequency vector by $\mathbf{y}'$ based on the hidden layer values, the output weight matrix and bias vector. The hitting frequency vector has $2 \cdot \widehat{w} \cdot |V|$ components, thus in our example it has $2 \cdot 2 \cdot 5 = 20$ components. The output weight matrix $\mathbf{W}_{out}$ is a $d \times (2 \cdot \widehat{w} \cdot |V|)$ matrix so in line with this in our illustration it would be 2×20. Finally, the output bias vector $\mathbf{b}_{out}$ has $2 \cdot \widehat{w} \cdot |V|$ components hence in our example it has 20 components.

Now let us assume that each v in V has a corresponding binary vector denoted by $\mathbf{x}_v$. Every binary vector has $|V|$ components and every component is zero except for the v^{th}. Later this vector is referenced as the hot-one encoded vector. We will predict the hitting frequency vector of nodes with hot-one encoded vectors. The approximated hitting frequency vector is outputted by the network and the hot-encoded one is imputed to the network on Figure 2.3. Between the output and input layers we have a single hidden layer. The vector $\mathbf{h}_v$ is the hidden representation of node v in d dimensions. We obtain this hidden representation by Equation 2.1 where the weight matrix $\mathbf{W}_{in}$ is a $|V| \times d$ matrix, $\mathbf{W}_{in}$ is the bias vector with d components and σ is an elementwise function. In our model $\mathbf{W}_{in}$ and $\mathbf{b}_{in}$ are trainable weights. In the actual implementation the function σ is chosen to be the identity function. There is a simple justification for this – if σ is the identity function the $\mathbf{W}_{in}^T \mathbf{x}_v$ multiplication is simplified to taking the v^{th} row of the input weight matrix. In the definition of a graph embedding function we postulated that $d \ll |V|$ meaning that we should have a low dimensional representation of nodes compared to the number of nodes. In order to achieve this we will have a low number of neurons in the hidden layer – in other terms we will form a so called bottleneck.

$$\mathbf{h}_v = \sigma(\mathbf{W}_{in}^T \mathbf{x}_v + \mathbf{b}_{in}) \tag{2.1}$$

Based on the hidden representation of v one wants to approximate the normalized vector of hitting frequencies from v – this is defined by Equation 2.2. The rational behind this is simple – the hidden layer vector is a compressed representation of the hitting frequency vector. Furthermore, because $d \ll |V|$ we can only approximate the real hitting frequency vector. The approximated hitting frequency vector $\mathbf{y}_v'$ of v is on the left hand side, the weight matrix $\mathbf{W}_{out}$ is a $d \times (2 \cdot \widehat{\boldsymbol{w}} \cdot |V|)$ matrix while the bias column vector has $2 \cdot \widehat{\boldsymbol{w}} \cdot |V|$ components. Finally the non-linear function Φ is chosen as the softmax function.

$$\mathbf{y}_v' = \Phi(\mathbf{W}_{out}^T \mathbf{h}_v + \mathbf{b}_{out}) \tag{2.2}$$

As our goal is the approximation of $\mathbf{y}_v^*$ we have to define the loss of approximation. The loss is a function of $\mathbf{y}_v^*$ and $\mathbf{y}_v'$ that is described by Equation (2.3). The loss function $\mathcal{L}$ is chosen as the cross entropy of $\mathbf{y}_v^*$ and $\mathbf{y}_v'$.

$$\mathcal{L}\left(\mathbf{y}_v^*, \mathbf{y}_v'\right) = \mathcal{L}\left(\mathbf{y}_v^*, \Phi(\mathbf{W}_{out}^T \mathbf{h}_v + \mathbf{b}_{out})\right) \tag{2.3}$$

As a next step we have to calculate the loss for all of the vertices in V and have to sum it up. This sum is described by Equation (2.4). Our goal is to minimize this sum. We are searching for the weight matrices $\mathbf{W}_{in}, \mathbf{W}_{out}$ and vectors $\mathbf{b}_{in}, \mathbf{b}_{out}$ that minimize Equation (2.4).

$$\sum_{v \in V} \mathcal{L}\left(\mathbf{y}_v^*, \Phi(\mathbf{W}_{out}^T \mathbf{h}_v + \mathbf{b}_{out})\right) \tag{2.4}$$

Essentially the weight matrix $\mathbf{W}_{in}$ is the embedding itself – for each v in V we have a d dimensional representation in a latent space. The weight matrix is used to approximately reconstruct the hitting frequencies of a node. If two nodes have similar hitting frequency vectors, meaning that their proximity is high, they will also have a similar latent space representation. We can assume that two nodes with similar hitting frequency vectors are going to be close on the graph itself (Newman, 2005).

In order to find the optimal weights of the proposed neural network we can use stochastic gradient descent (Bottou, 1991). As our goal originally is the efficient and scalable learning of the embedding we can use asynchronous gradient descent – henceforth ASGD – to learn the embedding in a parallel way (Recht et al., 2011). Earlier work regarding sequence based graph embedding has already proven that parallelized learning with ASGD does not deteriorate the representation quality (Perozzi et al., 2014; Grover & Leskovec, 2016). During the training of the network we will always linearly decrease the learning rate α from its starting value to zero. We must emphasize that using parallelization does not solve the problem that passing a single sample through the neural network has $\mathcal{O}(|V|)$ computational complexity. On account of this a single training epoch with all of the nodes has a computational complexity of $\mathcal{O}(|V|^2)$. This complexity would render our model useless for larger graphs.

The creators of the skip-gram model had two propositions to improve upon this quadratic computational complexity (Mikolov et al., 2013b). One of the propositions is the use of *negative sampling* (Gutmann & Hyvärinen, 2010) to reduce the number of weights that they have to update during training. Another solution is the use of *hierarchical softmax activations* instead of the softmax activations in the output layer. As Perozzi et al. (2014), Grover & Leskovec (2016) and Pimentel et al. (2017) all choose the use a hierarchical softmax activations we also decided to use it. With this modelling choice the computational complexity of a single training epoch is reduced to $\mathcal{O}(\log(|V|) \cdot |V|)$. Considering that $\log 10^7 \approx 16$ this complexity essentially means that for most of the practical applications we have a scalable (near linear complexity) training of our neural network and we can possibly learn embeddings of graphs with millions of nodes. We have to note that sequence generation and feature extraction is independent from the model chosen for the dimension reduction. One could use other dimensionality reduction methods such as principal component analysis to create an embedding. However, most of the other dimension reduction methods have a computational complexity that is worth than $\mathcal{O}(\log(|V|) \cdot |V|)$.

It is evident that the earlier described feature extraction mechanism and the model used to create the embedding allows for free choice of parameters. However, as the literature on sequence based embedding procedures is already quite extensive we will evaluate our models in most of the experiments with parameters that are used as standards in other works (Perozzi et al., 2014; Grover & Leskovec, 2016; Yang et al., 2016; Pimentel et al., 2017). These parameters that we reference as the standard parameter settings are $\widehat{w} = 10$, $d = 128$, $\alpha = 0.025$ and k the number of training epochs is set to be 1. We emphasize in our work when we deviate from these settings.

Chapter 3

Learning from diffusion trees

This chapter gives an overview on how the neural network described in the end of Chapter 2 is able to use features extracted with diffusion trees. We discuss the sampling of diffusion trees and learning from them in different sections. The notion of diffusion trees and sequence sampling methods are outlined in subsections of Section 3.1 with examples. While the application of the sampling methods themselves and parallelization possibilities are overviewed in Section 3.2.

3.1 Diffusion tree sampling

In this section we will describe how can we sample linear sequences of nodes with diffusion trees that necessarily include an arbitrarily chosen node. This is needed as we will want to sample linear sequences of nodes for all of the nodes in the vertex set. The two proposed node sequence sampling procedures both use diffusion processes to sample vertices from the graph. In generic terms a *diffusion process* on a network describes the spreading of something on a graph. This thing that spreads can be a disease, a technology or a piece of information. A *diffusion tree* is a directed graph which describes the way the spreading phenomenon proceeds on the graph. In our examples it would show who infected whom, who adopted the technology from whom or who told the message to whom. A simple illustrative exemplar graph is depicted on Figure 3.1. The graph is defined by vertex set $V = \{a, b, c, d, e, f\}$ and edge set $E = \{(a, b), (b, c), (c, a), (a, d), (d, e), (c, f), (b, f), (f, e)\}$. The graph on Subfigure 3.1a can represent a group of friends and the diffusion process can be the spread of a rumour. We plotted a diffusion tree on Subfigure 3.1b where the seeder (the node which started the diffusion) was node c and we assumed that all of the nodes received the rumour.

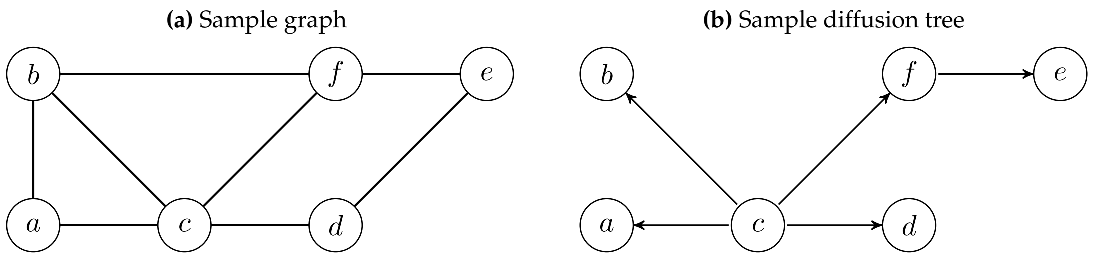


Figure 3.1: Example diffusion tree. The graph $\mathcal{G}(V, E)$ is defined by vertex set $V = \{a, b, c, d, e, f\}$ and the edge set $E = \{(a, b), (b, c), (c, a), (a, d), (d, e), (c, f), (b, f), (f, e)\}$. The original graph is on the left and the diffusion tree originating from c is on the right. At the end of the process all of the nodes adopted the rumour.

A diffusion tree is not a linear sequence of nodes. Because of this we have to define a traceback procedure that generates linear sequences of nodes based on the sampled diffusion tree. In this specific example an arbitrarily defined linear sequence of nodes that does not consider the direction of edges based on the tree could be the following one:

$$c - d - c - a - c - b - c - d - c - f - e$$

Before describing the sampling and traceback procedures we have to make a remark regarding how we store the graph on which we will initiate the diffusion process. In order to generate vertex sequences we need a connected graph denoted by $\mathcal{G}(V, E)$. In our implementation this graph is stored as an edge list hash table. This data structure uses the nodes as keys and the corresponding values are lists of the nodes' first order neighbours. Choosing this data structure is useful because looking up the first order neighbours of node v and retrieving them as a list takes $\mathcal{O}(\mathcal{N}_{\mathcal{G}}(v))$. Moreover, the memory complexity of this data structure is $\mathcal{O}(|V| + |E|)$ which is considerably better than keeping an adjacency matrix in memory. Consequently, simulating diffusion processes on the graph becomes easier with an edge list hash table as looking up first order neighbours of a node is a fundamental operation of the diffusion process simulation.

The sophisticated Eulerian circuit diffusion graph traceback algorithm is discussed in Subsection 3.1.1, while the fast endpoint traceback method is presented in Subsection 3.1.2. In both subsections we provide examples to illustrate how the traceback procedures work.

3.1.1 Eulerian circuit traceback on diffusion graphs

In this subsection we will overview how can we generate a single diffusion tree starting from node v and how can we linearise it with an Eulerian circuit. The basic idea is that one can generate a connected graph from a seeding node that has an Eulerian circuit by design. The obtained Eulerian circuit is a linear sequence of nodes which can be used for feature extraction. This method requires the before mentioned strongly connected graph, a starting node v and a fixed number of vertices to be sampled l. We have to make sure that $|V| \leq l$ holds, namely that we do not want to sample more nodes than the number of vertices in the graph.

In order to generate a sequence we will initiate a probabilistic diffusion process on the graph. The initial seeder is v and the set of seeders S is initialized as v. The diffusion graph $\widetilde{\mathcal{G}}$ is the null graph K_0, and its initial vertex set is $\{v\}$. Until we do not reach the number of desired vertices in the seeder set we repeat the following process:

(i) We sample a random node u from S.

(ii) From the first order neighbours of u we take a random node w.

(iii) If the node w is not in S we know that it is not in the seeder set but it is connected to a node in the seeder set. We add the node to the set of seeders and to the vertex set of the diffusion tree. Finally, we also add the directed edges (u, w) and (w, u) to the edge set of the diffusion tree.

When the while loop terminates we have a graph with l nodes and it contains the start node v. This graph is not a tree in the graph theory terms as it has cycles. Moreover, it is not necessarily a linear sequence of nodes. We coin the subgraph generated by Algorithm 1 as the bidirectional diffusion graph. To help with the understanding of the procedure we described it with pseudocode in Algorithm 1.

```
Data: $\mathcal{G}$ – Graph object – (hash table with node keys and edge lists as values).
      $l$ – Number of vertices to be sampled.
      $v$ – Starting node .
Result: $P$ – Eulerian sequence traceback of diffusion tree from node $v$ on graph $\mathcal{G}$ containing $l$ unique nodes.
$S \leftarrow \{v\}$
$\widetilde{\mathcal{G}} \leftarrow K_0$
$V_{\widetilde{\mathcal{G}}} \leftarrow S$
while $|S| < l$ do
    $w \leftarrow$ Random Sample($S$)
    $u \leftarrow$ Random Sample($N_{\mathcal{G}}(w)$)
    if $u \notin$ S then
        S $\leftarrow$ S $\cup \{u\}$
        $V_{\widetilde{\mathcal{G}}} \leftarrow V_{\widetilde{\mathcal{G}}} \cup \{u\}$
        $E_{\widetilde{\mathcal{G}}} \leftarrow E_{\widetilde{\mathcal{G}}} \cup \{(u,w),(w,u)\}$
    end
end
$P \leftarrow$ Random Eulerian Circuit($\widetilde{\mathcal{G}}, v$)
```

Algorithm 1: Directed diffusion tree generation and Eulerian sequence traceback

Claim 3.1. *The bidirectional diffusion graph is Eulerian.*

Proof. According to Biggs et al. (1976) a directed graph has an Eulerian circuit if it has the following properties:

(i) It has a single strongly connected component.

(ii) All other vertices have an equal in and out degree.

First, our sampling procedure ensures that the graph has a single strongly connected component. Second, if the vertex set is singleton in the end of the diffusion process the seeder node's in-degree and out-degree are both zero. Third, in the iterative process the edges are added in a manner that the in-degree and out-degree of nodes is always the same. ■

This means that starting from node v in the bidirectional diffusion graph one can traverse the edges of the graph in a way that every directed edge is visited once. To put it simply, starting from node v one can initiate a random Eulerian circuit on $\widetilde{\mathcal{G}}$ which generates a linear sequence of nodes. Earlier we demonstrated that a linear sequence of nodes can be used to extract features for learning. Generation of this Eulerian circuit has time complexity $\mathcal{O}(l)$ if the method described in (Edmonds & Johnson, 1973) is used. The number of non-unique nodes in the sequence generated by the circuit will be $2 \cdot l - 1$ (Biggs et al., 1976). Intuitively the starting node will appear at least twice in the sequence (starting and end node). Moreover, the diffusion graph and the traceback itself is random. From a given starting node possibly a number of unique sample graphs and traceback sequences are obtainable. The outline of the idea is summarized with pseudo-code by Algorithm 1.

To demonstrate the idea of Eulerian diffusion graph traceback let us look at the example graph on Subfigure 3.2a. The number of nodes in this graph is 12, it is undirected and connected. Now let us consider sampling a sequence with $l = 6$ nodes starting from node b. A possible diffusion graph is depicted on Subfigure 3.2b. Two randomly chosen examples of Eulerian tracebacks from b on this specific diffusion tree are below.

$$b - d - e - d - a - d - b - c - b - i - b$$

$$b - i - b - c - b - d - e - d - a - d - b$$

Another diffusion graph that can be obtained by a random spreading process is on Subfigure 3.2c. On this tree two examples of tracebacks are down.

$$b - c - e - c - f - g - l - g - f - c - b$$

$$b - c - f - g - l - g - f - c - e - c - b$$

This simple example demonstrates that from the same starting node and graph quite heterogeneous vertex sequences might originate. It also pinpoints the fact that the same diffusion graph is a possible source for multiple sequences.

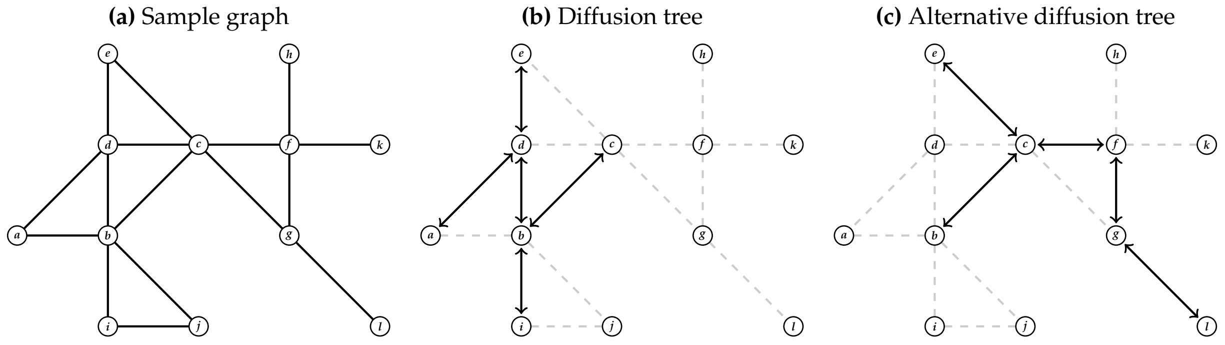


Figure 3.2: Eulerian trace backs of diffusion trees – illustrative example. The graph itself is either directed or undirected and we must sample a set of nodes that is smaller than $|V|$. From a given starting node one might sample multiple diffusion trees. The resulting diffusion trees can have a number of distinct tracebacks with the methods that we describe.

3.1.2 Fast endpoint traceback on diffusion trees

The previously outlined method traces back the vertices in a way that does not rely on the way that the diffusion process spreads on the network. Another traceback method can use the already existing paths among vertices on the tree that were generated by the diffusion process itself. Simply, we might want to have a traceback procedure that is faster than the Eulerian method.

The algorithm that we propose needs the original graph represented by an edge list hash table, a fixed number of unique vertices to be sampled and a starting node for the diffusion process. As part of the initialization one has to create a counter hash table – this contains the number of nodes that a given node is connected to in the tree. In case of the initial seeder its value is set to be zero. The shortest paths on the tree are stored in a list of lists. The seeder set S initially contains node v. Until the required number of seeder nodes is reached the following is repeated:

(i) From the shortest paths list the first sublist is taken. From this sublist the last vertex u is extracted.

(ii) From the original graph w a random neighbour of u is extracted.

(iii) If this random neighbour is not in the set of seeders the number of infected nodes is incremented. A new sublist is created by appending w to the chosen sublist. This new sublist is appended to the list of lists containing the diffusion paths. Consequently, w is added to the seeders set. The degree counter hash is modified by setting the value corresponding to w as 1. Moreover, the value corresponding to the key u is incremented by 1.

(iv) The diffusion paths list of lists is shuffled so in the next iteration a random diffusion path will be the first path in the list.

When the iteration terminates we have sampled l nodes. Based on the counter hash one can tell which are those nodes that have only one neighbour and one can easily select these endpoints from the tree. Based on the *Node Degrees* hash, the *Shortest Paths* list and v one can select diffusion paths that start in the original seeder node and end in a node with degree 1. The final vertex sequence *Traceback* is initialized with the start node. Finally, in an iterative process the vertex sequence to be created is augmented with the diffusion paths and their reverts without allowing neighbouring duplicates in the sequence. We summarized the graph sampling and sequence traceback with pseudo-code by Algorithm 2.

Similarly to the other vertex sequence generation algorithm the idea behind the method can be truly understood by an illustrative example. Let us consider again the graph on Subfigure 3.2a. Diffusion trees obtained are again the same and they are once again Subfigures 3.2b and 3.2c respectively. In case of Subfigure 3.2b the end nodes are c, e, and i. Two randomly chosen endpoint tracebacks of this tree are just below.

$$b - i - b - c - b - d - e - d - b - d - a - d - b$$
$$b - d - e - d - b - c - b - d - a - d - b - i - b$$

Looking at Subfigure 3.2c one notices that the end nodes are e and l. Using these nodes two possible tracebacks from the endpoints are below.

$$b - c - e - c - b - c - f - g - l - g - f - c - b$$
$$b - c - f - g - l - g - f - c - b - c - e - c - b$$

This example shows that this method is biased towards nodes with a high centrality in the diffusion tree. In the first example b appears 5 times in the sequence, in the second one c appears 4 times. Moreover, the size of the resulting sequence is not a direct function of l. Let us consider that c is the seeder in case of Subfigure 3.2b – this would result in vertex sequences with a total length of 17. The sequences that we described earlier only had lengths of 13.

```
Data: $\mathcal{G}$ – Graph object – (hash table with node keys and edge lists as values).
      $l$ – Number of vertices to be sampled.
      $v$ – Starting node .
Result: Traceback – Endpoint sequence traceback of diffusion tree from node $v$ of graph $\mathcal{G}$ containing $l$ unique
        nodes.
1  Node Degrees $\leftarrow \{\}$
2  Node Degrees$_v \leftarrow 0$
3  Shortest Paths $\leftarrow [[v]]$
4  $S \leftarrow \{v\}$
5  while $|S| < l$ do
6      $u \leftarrow$ Take Last Vertex of Sequence(Shortest Paths$_0$)
7      $w \leftarrow$ Random Sample($N_{\mathcal{G}}(u)$)
8      if $w \notin S$ then
9          Append(Shortest Paths$_0$, $[w]$)
10         $S \leftarrow S \cup \{v\}$
11         Node Degrees$_w \leftarrow 1$
12         Node Degrees$_u \leftarrow$ Node Degrees$_u + 1$
13     end
14     Random Shuffle(Shortest Paths)
15 end
16 Endpoints $\leftarrow$ Select Endpoint Vertices(Node Degrees)
17 Shortest Paths $\leftarrow$ Select Endpoint Diffusion Paths(Shortest Paths, $v$, Endpoints)
18 Traceback $\leftarrow [v]$
19 for Shortest Path in Shortest Paths do
20     Append Without Repetition(Traceback,Shortest Path)
21     Append Without Repetition(Traceback,Revert(Shortest Path))
22 end
```

Algorithm 2: Diffusion tree generation and endpoint trace back

3.2 Parallelized graph embedding based on linear node sequences

The sequence generation methods that we just described in Section 3.1 can generate a linear sequence of nodes from a single starting node. Our goal is to ensure that we have a set of sequences in which every node appears at least one of the sequences. A simple procedure that ensures that every node appears at least in one sequence is obtained by generating trees from every vertex. Later on we will extract features from the created sequences and learn the embedding itself. In this section we introduce a framework to generate the sequences in a way that multiple workers can be used to do the sampling. These workers using a connected graph, a vertex set cardinality and a starting node create linear sequences of nodes. The idea is outlined with pseudo-code by Algorithm 3.

Now let us look at what are the input parameters and data that our method requires. The graph embedding method first needs a source for the graph that it will embed. From each node in the graph we will generate n sample sequences. The sampling procedure needs a parameter l which sets the unique number of nodes sampled. It is the length of the random walks or in our sampling methods it is the number of unique nodes in the sequence. During the feature extraction we use the sliding window size parameter $\widehat{w}$. The neural network that we fit has d neurons in the hidden layer – this is also the number of embedding dimensions. Lastly, the optimization procedure used for finding the optimal weights of the neural network requires the number of epochs k and an initial learning rate α.

```
Data: Source – Path to the edge list used.
      $n$ – Number of sequence samples per node.
      $l$ – Number of unique nodes parameter.
      $d$ – Dimension of vector representation.
      $k$ – Number of epochs in optimization procedure.
      $\widehat{w}$ – Size of sliding window.
      $\alpha$ – Learning rate.
Result: X – Embedding of nodes from graph $\mathcal{G}$ in $d$ dimensions.
1  $\mathcal{G} \leftarrow$ Read Graph(Source)
2  $\mathcal{G}_1, \mathcal{G}_2, \dots \mathcal{G}_S \leftarrow$ Connected Component Extraction and Sorting($\mathcal{G}$)
3  Path Samples $\leftarrow$ []
4  for $i$ in $1 : n$ do
5      Paths $\leftarrow \{\}$
6      $l' \leftarrow l$
7      for $j$ in $1{:}|\{\mathcal{G}_1, \mathcal{G}_2, \dots \mathcal{G}_S\}|$ do
8          $V \leftarrow V_{\mathcal{G}_j}$
9          if $|V| < l'$ then
10             $l' \leftarrow |V|$
11         end
12         for $v$ in $V$ do
13             $P \leftarrow$ Diffusion Tree Generation and Traceback($\mathcal{G}_j, v, l'$)
14             Paths($v$) $\leftarrow P$
15         end
16     end
17     Path Samples($i$) $\leftarrow$ Paths
18 end
19 X $\leftarrow$ Learn Embedding(Path Samples, $d, \widehat{w}, \alpha, k$)
```

Algorithm 3: Learning from node sequences – generalized algorithm description

The way the we create the sequences is the following. After obtaining the graph it is decomposed into a set of strongly connected components. The connected subgraphs are ordered by the number of nodes they contain – each subgraph is an edge list hash table. We need this ordering as we want to know what is the maximal number of nodes that one can sample from the subgraph. For each subgraph we do the following:

(i) We create a vertex set from the shuffled vertices of the subgraph.

(ii) If the cardinality of the subgraph is lower than l (the unique nodes parameter) we replace it with the cardinality of the subgraph.

(iii) We generate for each of the nodes in the subgraph a vertex sequence sample which is added to the *Paths* hashtable.

We store the obtained samples in the *Path Samples* list. For each node we create n samples. After the n^{th} iteration the vertex sequences are stored in the hash table *Paths*. When all of the nodes have a corresponding sample in *Paths* we add it to the list *Path Samples*. Based on the samples and hyperparameters we can extract the hitting frequency features and we can learn an embedding with the neural network using the obtained features.

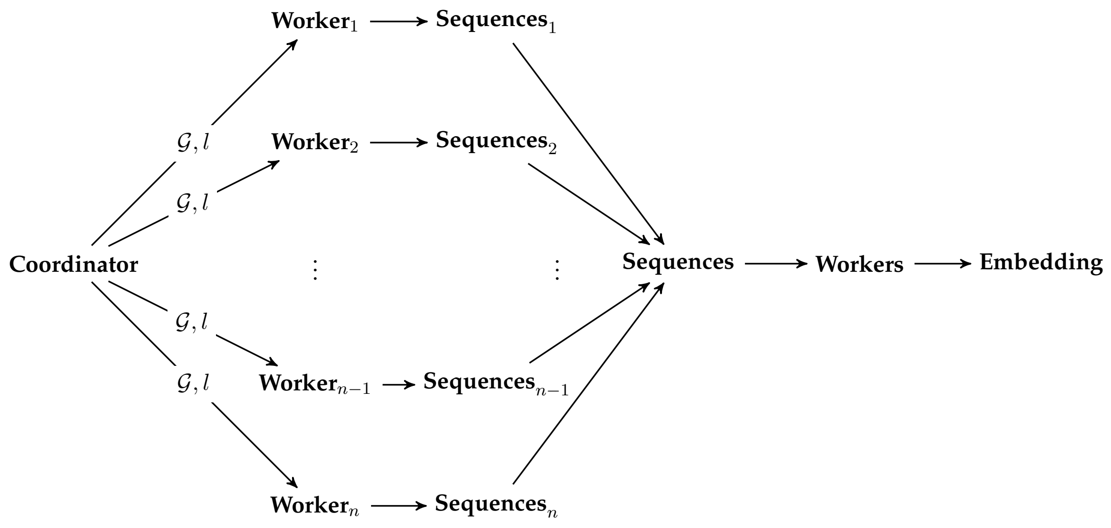


Figure 3.3: Diffusion to vector – outline of parallel processing. The sequences are generated by n workers who each create a copy of the graph $\mathcal{G}$ and they receive the sequence length controlling parameter l. Each of them generates diffusion tree based sequences for the vertices in V. Sequence generation is random seeded and the sequences are aggregated when each of the workers finish. The hitting frequency features described in Section 2.3 are extracted based on parameter $\widehat{w}$. Finally, the embedding is learned based on the hitting frequency features with parameters α, d, k using asynchronous gradient descent with the architecture described in Subsection 2.3.2 of Chapter 2 .

It has to be noted that the sample generation that happens between lines 6 and 17 inclusive in Algorithm 3 can be distributed among n workers. Each of the workers gets the sequence length controlling parameter and creates a graph object representing the network itself. After this the workers would generate samples for every vertex in V in a seeded random way. The seed could be the identifier of the worker itself. So we would have n sequences for each node in V – altogether this means $|V| \times n$ vertex sequences where the maximal length of the sequences is controlled by l. Each of the nodes would appear in at least n sequence out of the $|V| \times n$ vertex sequences. Using the sequences that we just created in a distributed fashion the workers could extract hitting frequency vectors based on the sliding window parameter $\widehat{w}$. Lastly, the hitting frequency features are utilized to generate the embedding with parameters α, d, k with asynchronous gradient descent and the architecture described in Subsection 2.3.2 of Chapter 2 . Essentially we have parallel sequence creation, feature generation and embedding learning. We have to note that the sample generation has a computational complexity of $\mathcal{O}(n \cdot l \cdot |V|)$. This complexity becomes $\mathcal{O}(l \cdot |V|)$ when parallelization with n workers is introduced.

Chapter 4

Computational performance

In this chapter we focus on the computational performance of the proposed node sequence generation algorithms. The motivation behind this is simple, most industry application would require that an embedding can be created in reasonable time even when the graph is large. Moreover, understanding how the possible evolution of a graph affects running time is a crucial aspect in real life applications. We will compare the time needed for performing node sequence generation and graph pre-processing to other similar node sequence based embedding methods. The reason for the separate investigation of sequence generation and pre-processing is the fact that the embedding methods have considerably different bottlenecks. Learning the embedding itself based on the node sequences is not in the focus of the chapter as all of the methods use the same learning procedure introduced by Mikolov et al. (2013a,b). Investigated graph embedding algorithms are the following:

(i) Diffusion2Vec with Eulerian sequence traceback – ED2V (our algorithm).

(ii) Diffusion2Vec with Endpoint sequence traceback – FD2V (our algorithm).

(iii) Node2Vec (Grover & Leskovec, 2016) – N2V.

(iv) DeepWalk (Perozzi et al., 2014) – DW.

The Python reference implementations of the D2V variants are enclosed with the thesis submission. In the benchmarks we only used a single core. Benchmarks for N2V were created with the Python reference implementation of Grover & Leskovec (2016). This is accessible at: *https://github.com/aditya-grover/node2vec*. Using this implementation one could also measure the performance of DW. However, it has to be pointed out that DW is a corner case of N2V, because of this running benchmarking experiments with the same reference implementation for both of them would be meaningless. At the same time, the original reference implementation of DW does not include the generation of transition probabilities and because of this it is considerably faster than N2V. It is true both for the pre-processing and sequence generation phases. This original Python implementation of DW is accessible at: *https://github.com/phanein/deepwalk*. Finally, I modified the DW and N2V implementations with lines that allow for performance measurement in the pre-processing and sequence generation phases.

First, in Section 4.1 we test the sensitivity of pre-processing and sequence generation times on synthetic graphs. We investigate in a series of experiments how performance changes with the increase in graph size and average degree. Second, in Section 4.2 we run experiments on a number of real world graphs that were used for benchmarking the representation quality of graph embedding methods in earlier works.

4.1 Peformance on synthetic graphs

Besides the comparison of the algorithms, we also want to point out some simple empirical regularities regarding the performance of these methods. The chosen synthetic graphs used for our experiments share two common properties. Namely, that the size of these graphs can be manipulated arbitrarily and in a similar manner the average degree is controllable. We did experiments with 3 synthetic graph generation models:

(i) **Watts-Strogatz graph** (Watts & Strogatz, 1998). This model creates a d-regular graph and later the edges of the created graph are perturbed. In our experiments we only consider the special case when edges are not randomized.

(ii) **Barabási-Albert graph** (Albert & Barabási, 2002). This model creates random graphs that have a power-law degree distribution, which is a property observed quite commonly in real world networks.

(iii) **Erdős-Rényi graph** (Erdős & Rényi, 1960). This model creates random graphs that have a Poisson degree distribution. This is not a realistic model of real world networks, but it is useful as a baseline benchmark.

Results of the graph pre-processing experiments are discussed in Subsection 4.1.1 while results of sequence generation are presented in Subsection 4.1.2.

4.1.1 Graph pre-processing experiments

The pre-processing step consists a series of distinct operations for every algorithm. First, in case of ED2V and FD2V it includes the loading of the edge list, generation of a graph object and the separation of the graph into connected components. Second, the pre-processing of the graph for N2V includes the read up of edges, the generation of the graph object and the calculation of transmission probabilities. Finally, in case of DW it includes the edge list read up and the graph object generation. Each of the graph-pre-processing experiments was repeated 100 times. When the graph-size – pre-processing time relationship was investigated the average degree was fixed to be 10. The number of nodes was 10^3, 10^4 and 10^5 respectively. While the average degree – pre-processing time connection was examined the graph size was fixed to be 10^4. The average degree was set as 8, 16 and 32.

Results obtained by pre-processing the Watts-Strogatz graph are plotted on Figure 4.1. Pre-processing time as a function of graph size is plotted on Subfigure 4.1a. The pre-processing time is visualized with a log-scale on the vertical axis. One can observe that DW and variants of D2V have a similar performance. It is also evident that the pre-processing is a linear function of the graph size when these methods are used. A 10 times larger graphs results in a 10 times longer pre-processing time. Compared to these methods N2V is considerably slower even when the graph is minuscule with a thousand nodes. The pre-processing time as a function of average degree is plotted on Subfigure 4.1b. Based on the average degree similar conclusions can be drawn: increasing the average degree increases the pre-processing time. In addition, DW and D2V variants outperform N2V again. Interestingly, doubling the average degree results in a pre-processing time increase that is disproportionate.

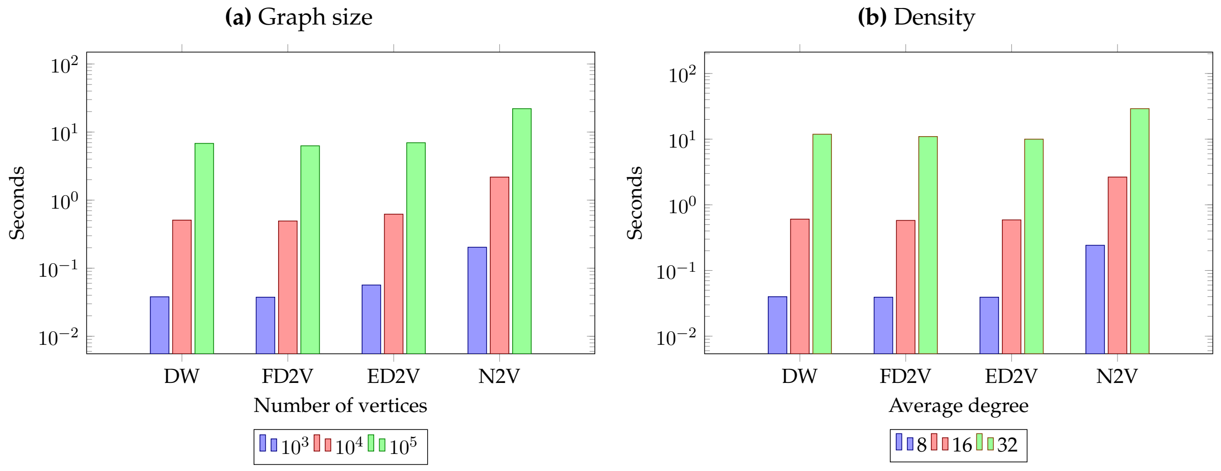


Figure 4.1: Watts-Strogatz graph – mean graph pre-processing time. Columns report mean graph pre-processing times based on 100 replications on a Watts-Strogatz graph. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed to be 10. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4.

Experimental data collected from pre-processing of the Barabási-Albert graph is plotted on Figure 4.2. Just as before the vertical axis measures mean pre-processing time on a log-scale in seconds. Based on Subfigure 4.2a one can conclude that increasing the graph size increases the pre-processing time for every model linearly. However, the performance of N2V is a magnitude worse compared to the other methods – just the pre-processing of the graph is about 10 times slower. It is a quite intriguing fact considering the fact that most of the real world networks have a power-law degree distribution like the Barabási-Albert graph. Exemplars that have this property include the physical internet (Faloutsos et al., 1999), world wide web (Clauset et al., 2009), citation graphs (Redner, 1998), protein-protein interaction networks (Jeong et al., 2001) and graphs of social network friendships such as Facebook, Myspace and Twitter (Backstrom et al., 2012; Thelwall, 2008; Java et al., 2007).

The sensitivity of the pre-processing time to average degree changes is plotted on Subfigure 4.2b when a Barabási-Albert graph is considered. We see that N2V performs poorly compared to other methods when the number of edges is increasing. Moreover, the pre-processing time itself is quite high already when a graph with an average degree of 8 is considered. The D2V variants and DW can process an approximately 4 times denser graph in a considerably lower time. This is also an important property to be considered in real life applications as there is supporting evidence that networks with power-law degree distribution show a densification of the network (Leskovec et al., 2007). Based on our findings the pre-processing performance of N2V deteriorates as the graph densification takes place.

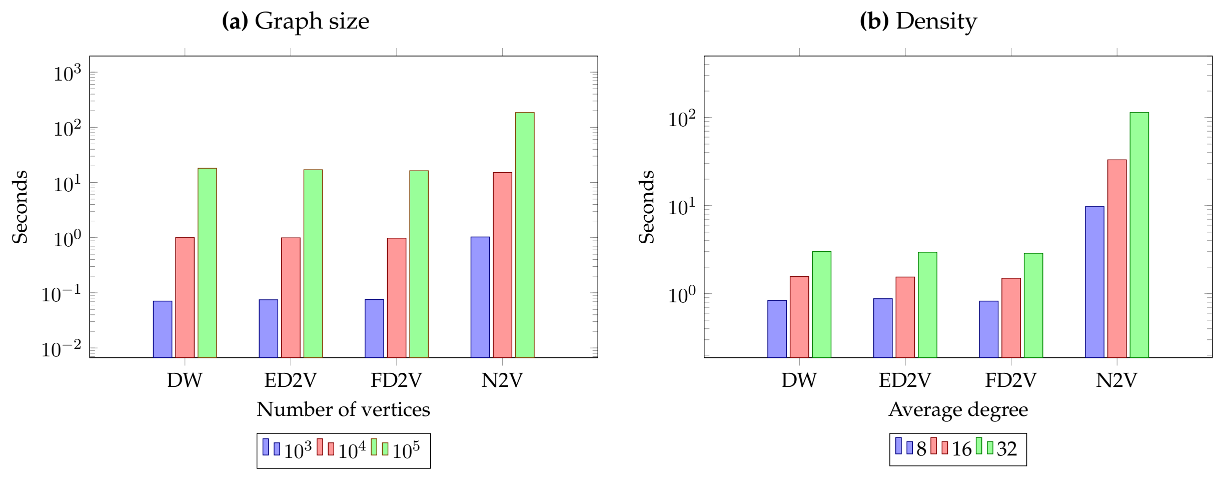


Figure 4.2: Barabási-Albert graph – mean graph pre-processing time. Columns report mean graph pre-processing times based on 100 replications on a Barabási-Albert graph. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed to be 10. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4. Time was measured in seconds.

4.1.2 Sequence generation experiments

After the preprocessing of the graph every algorithm starts to generate sequences of nodes for every vertex in the graph. As sequence generation is an essential element of embedding creation its sensitivity to the graph size has to be investigated. To have comparable results I have set the parameters controlling sequence length as follows:

(i) ED2V – Vertex set size of 40. This setting results in sequences that have an approximate sequence length of 80.

(ii) FD2V – Vertex set size of 25. Similarly, this setting results in sequences that have an approximate sequence length of 80.

(iii) N2V – Second order random walks with length 80.

(iv) DW – Random walks with length 80.

The properties of the graphs used for the sequence generation benchmarks themselves are the same as in Section 4.1.1. A single sequence generation run generates a sequence for each vertex of the pre-processed graph. Sequence generation is repeated 100 times and we calculate the mean generation time. Experimental results of generating node sequences on the Watts-Strogatz graph are plotted on Figure 4.3. Graph size increase related time measurements are on a log scale and density expansion related ones are on a linear scale. Sensitivity analysis of sequence creation time regarding the vertex set size is on Subfigure 4.3a. The first insight is that the diffusion based methods underperform compared to the procedures that generate sequences with a random walk. It is also evident that DW performs better on this graph than any other model. When the increase of the average degree is investigated on Subfigure 4.3b we see that performance of DW and N2V drops slightly. Mean sequence generation time increases with the density. While FD2V and ED2V generally performs poorly the average time needed for generating sequences with these methods decreases as the average degree of nodes increases.

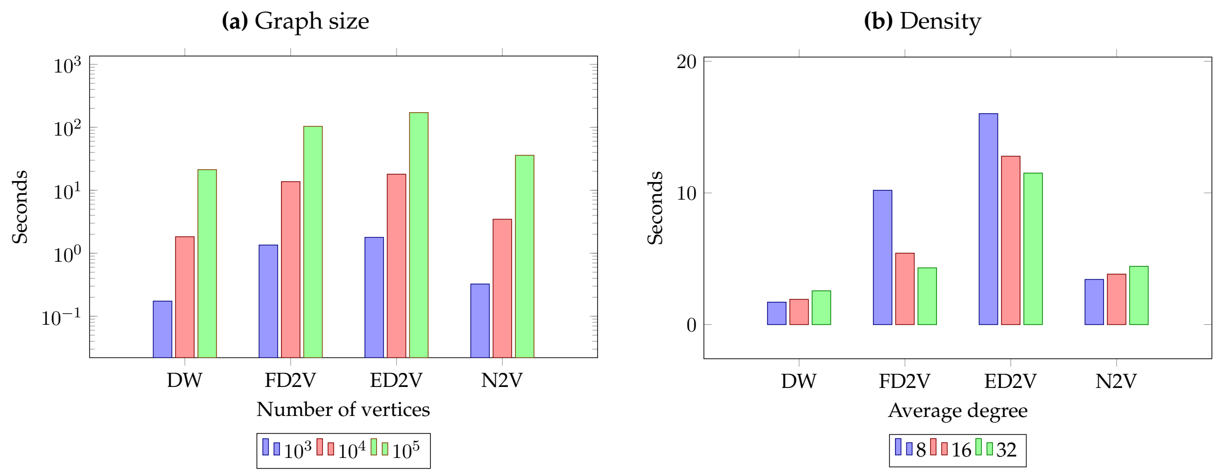


Figure 4.3: Watts-Strogatz graph – mean sequence generation time. Columns report mean sequence generation times based on 100 replications on a Watts-Strogatz graph. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed as 10. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4.

This result might seem counterintuitive, but there is a simple reason why this is the case. It is a known fact that a non-perturbed Watts-Strogatz graph has a high neighbourhood overlap – two neighbours have a large number of shared neighbours. Because of this in a large number of cases the diffusion tree sampled nodes are already in the diffuser set. This forces our method to draw a new sample in order to extend the tree. On the other hand, random walk based sampling allows for sequences that contain the same node multiple times.

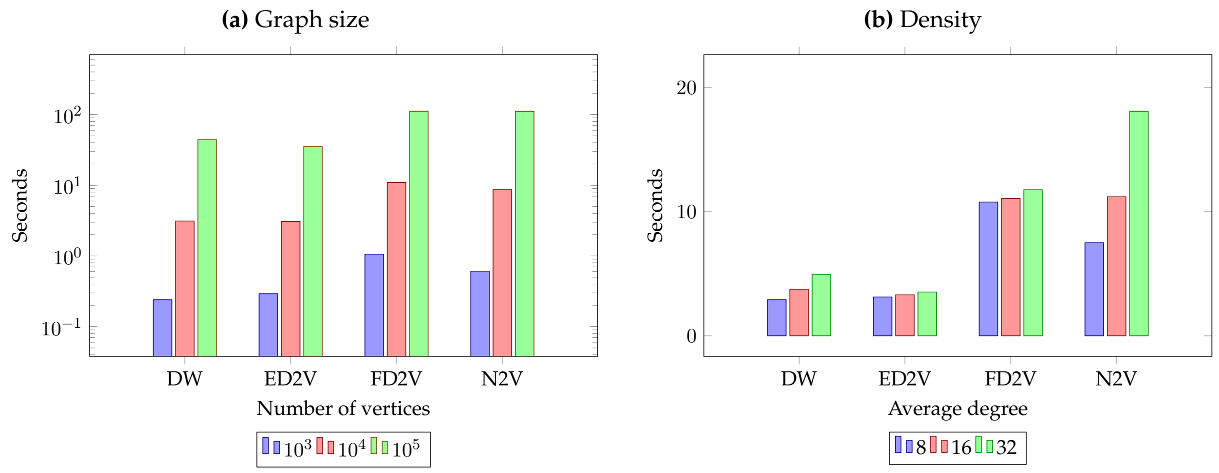


Figure 4.4: Barabási-Albert graph – mean sequence generation time. Columns report mean sequence generation times based on 100 replications on a Barabási-Albert graph. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed to be 10. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4.

Our benchmark results on the Barabási-Albert graph are plotted on subfigures of Figure 4.4. We have to emphasize again that these findings are the most relevant to possible applications of these methods as the graph's topology is a fair model of real networks. Findings with respect to manipulation of the graph size are on Subfigure 4.4a. Our core finding again is that linear increases in the graph size result in linear increases in the sequence generation time. Moreover, in these types of graphs N2V is way slower than other methods. Finally, sequence generation time as a function of average degree is on Subfigure 4.4b. Incrementation of the average degree only slightly increases the sequence generation time when D2V is used for the sequence generation. When N2V is considered the increase in density results in a quite considerable increase in the sequence generation time. A 4 times denser graph results in approximately two times longer sequence generation. The reason that we observe this phenomenon is quite fascinating. Our methods sample nodes in a local manner – when one samples a tree for a non central and low-degree node other nodes in the tree are likely to be low degree nodes. Taking samples from these list of neighbourhoods is computationally cheap. On the other hand, random walks escape the vicinity of the periphery nodes and end up in the core of the network. Sampling from the neighbourhood of high degree nodes and calculating the transmission probabilities for vertices with a high degree in the centre of the network is costly. Moreover, versions of D2V have to sample a lower number of nodes from the graph.

Bar charts comparing the performance of algorithms on Erdős-Rényi graphs are enclosed in Appendix A. The graph pre-processing results are plotted on Figure A.1, while the sequence generation results are on Figure A.2. In addition, tables summarizing the computational performance statistics on the synthetic graphs are also in Appendix A. Results regarding graph pre-processing are enclosed as Table A.1, while results of sequence generation are enclosed as Table A.2.

4.2 Performance on real graphs

In this section we provide evidence that our method outperforms N2V when one considers real networks. Our results were obtained by using the Python reference implementations of the algorithms. The graph pre-processing and sequence generation steps were both repeated 100 times. The parameters controlling the sequence length were taken from Subsection 4.1.2. As these are the settings used to benchmark these algorithms in the literature. From the pre-processing and sequence generation time measurements we calculated the means. For the sequence generation we also present minimal and maximal sequence creation times. The results are enclosed as Table 4.1. We present benchmarks on the following real life networks that we will later also use to investigate the representation quality of embeddings:

(i) **Blogcatalog:** A social network of bloggers – nodes represent users and links are social relationships (Agarwal et al., 2009).

(ii) **PPI:** A protein-protein interaction network (Chatr-Aryamontri et al., 2014). We use the subset of nodes and edges extracted by (Grover & Leskovec, 2016).

(iii) **Wikipedia:** A word co-occurrence graph derived from the Wikipedia corpus (Mahoney, 2011). This is the same dataset as the one created and used by (Grover & Leskovec, 2016).

First, let us consider the results on the BlogCatalog social network. Our methods are the fastest when it comes to the graph pre-processing phase. It is not surprising as the graph pre-processing steps of ED2V and FD2V are the same and indeed we observe that the time needed for this step is approximately the same. On this graph the performance of N2V in terms of time needed for the pre-processing is almost 250 times worth than the performance of D2V variants. The mean sequence generation time of ED2V is 3 times faster than N2V. We also see that DW is 1.25 times slower than ED2V on the BlogCatalog dataset. Finally, FD2V is the fastest of all, it outperform every other method even DW. Our earlier results had shown that N2V performs poorly compared to D2V when the average degree is high and the degree distribution of the graph is described by a power-law. Considering, that the BlogCatalog network has a power-law degree distribution and a high average degree (it has an approximate average degree of 64.775) the result that we have was expectable.

Second, let us review the finding on the PPI network. The graph pre-processing is again fastest when diffusion based methods are used and DW performs comparably. Pre-processing time of N2V is again relatively long – it is roughly 40 times slower than variants of D2V. The sequence generation is the fastest with DW and one can also observe that even N2V outperforms the Eulerian D2V method. These results are also somewhat expected as the PPI network has a low average degree (it has an approximate average degree of 19.732) and consists of small sized subgraphs that are disjoint. Sequence generation time of FD2V is slightly slower than that of DW.

Third, we focus on the results with respect to the Wikipedia word co-occurrence network. We see that the time gap between D2V and N2V is quite considerable when pre-processing performance is measured. This phase takes roughly 280 times longer for N2V. Also we can establish that the graph-pre processing performance of D2V versions is 1.5 faster than DW – this seems to be a constant proportion of pre-processing time differences between the two. The sequence generation results show that hat ED2V is faster than N2V but slower than DW. Moreover, the performance of FD2V is the best of all. If one looks at average degree these findings are in line with our earlier findings (the graph has an approximate average degree of 38.734), namely that a high average degree favours D2V in the sequence generation phase.

The sequence generation phase allows for parallelization of sequence generation in the reference implementation of DW. Intriguingly, because of the use of the object that contains the transmission probabilities neither the reference Python N2V implementation nor its high performance C++ version is parallelized when the sequences are generated. This simply means that besides the fact that sequence generation is slow in N2V compared to other methods on power-law networks only the embedding learning phase benefits from the use of multiple cores. Contrarily, both variants of D2V are parallelized and DW also generates the vertex sequences in a parallel way.

With the results outlined in Table 4.1 and the fact regarding the parallelization of current D2V and N2V implementations additional performance comparisons are possible. Let us consider the following scenario. We want to embed the BlogCatalog network with N2V and ED2V. We choose the sequence length controlling parameters as before – a sequence of nodes with length approximately equal to 80 is quite generally used when these models are considered. Moreover,

	BLOGCATALOG $\lvert V\rvert$=10,312 $\lvert E\rvert$=333,982				PPI $\lvert V\rvert$=3,890 $\lvert E\rvert$=38,739				WIKIPEDIA $\lvert V\rvert$=4,777 $\lvert E\rvert$=92,517			
	DW	N2V	ED2V	FD2V	DW	N2V	ED2V	FD2V	DW	N2V	ED2V	FD2V
Mean sequence generation time	14.453	59.089	19.983	**8.896**	**1.403**	4.253	4.684	1.504	3.940	12.135	6.879	**2.727**
	(0.014)	(0.037)	(0.028)	(0.005)	(0.001)	(0.002)	(0.005)	(0.002)	(0.004)	(0.005)	(0.004)	(0.003)
Mean graph pre-processing time	4.756	784.899	**3.231**	**3.189**	0.480	12.797	**0.362**	**0.362**	0.927	185.287	**0.667**	**0.680**
	(0.019)	(2.440)	(0.006)	(0.003)	(0.003)	(0.129)	(0.001)	(0.001)	(0.004)	(0.557)	(0.001)	(0.001)
Maximal sequence generation time	16.212	62.247	26.04	9.903	1.792	4.675	5.724	2.094	4.568	12.887	7.875	3.591
Minimal sequence generation time	13.174	56.492	18.075	7.876	1.271	4.174	4.391	1.386	3.442	11.779	6.488	2.485

Table 4.1: Computational performance results on real life graphs. Headers contain the name of the dataset, the number of vertices and edges. The sub-headers denote the algorithms used. Columns report running time statistics extracted from 100 experiments on the datasets. Values in the parentheses are standard errors calculated for the means. Bold numbers denote the fastest sequence and graph generation methods. If the 95% confidence intervals overlap for the two means and one of them is for the fastest algorithm they are both highlighted, as there is no significant difference between the two measurements.

let we assume that we generate 10 sequences for each unique node in the graph. Doing this with the unparallelized N2V Python reference implementation would take about 1375 seconds – preprocessing takes 785 seconds and sequence generation takes 10 times 59 seconds (using the mean sequence generation time). To put it simply, the optimization starts after 23 minutes have passed. Now let us focus on ED2V. If our machine has 10 cores we can leverage on the parallel sequence generation framework introduced by our reference implementation. In this case each of the machines creates a copy of the graph and creates a sequence for each of the nodes. Because of this, the whole pre-processing and generation process takes 29 seconds – assuming average graph read times and maximal sequence generation time. It has to be noted that graph pre-processing is somewhat deterministic while sequence generation is more random. We took the maximal value of 26 seconds, because idle workers have to wait for the worker that finishes last. Considering all of these numbers it means that ED2V is 47 times faster than N2V. In a scenario where parallelization is not available the same graph pre-processing and sequence generation process with ED2V takes 203 seconds – using mean sequence generation times. This means that even without parallelization one could still obtain a 4 times speed up compared to N2V.

Chapter 5

Properties of samples and embeddings

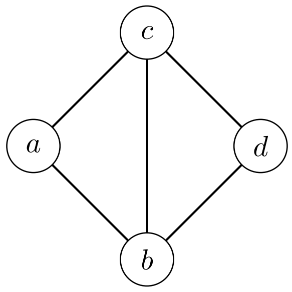
When we embed the graph our primary goal is to represent the graph in a latent space such way that pairwise distances and proximities between pairs of nodes on the graph are approximately maintained. In this chapter we will demonstrate that our method achieves this target quite well. Moreover, we argue that other topological properties (centrality measures and neighbourhood overlap) of the nodes and edges are correlated with certain distances in the latent space. Investigating these properties is essential to our later applications as solutions to downstream machine learning tasks such as node labeling and clustering have certain assumptions about the position of nodes in the latent space. We will discuss how these assumptions relate to properties of the embeddings.

First, in Section 5.1 we consider the differences between the topologies of subgraphs extracted with random walks and diffusion trees. Second, we will present basic empirical regularities regarding the position of nodes and node pairs in the latent space in Section 5.2. Third, in Section 5.3 we demonstrate that our proposed methods allow for visualizing graphs in two dimensional space. We also give examples of graph layouts with other sequence based graph embedding methods.

5.1 Why diffusion tree sampling is novel?

Earlier approaches to node sequence based graph embedding use random walks to generate linear sequences of nodes (Perozzi et al., 2014; Grover & Leskovec, 2016). The graph exploration of random walks is fundamentally different from the way diffusion processes traverse the graph. A truncated random walk which visits a fixed number of nodes tends to leave the neighbourhood of the starting node after a few hops. On the contrary, a diffusion process stays local and explores the neighbourhood of the starting node. Because of this the nodes sampled by a diffusion tree are from neighbourhoods that are remarkably different from the ones that are sampled by random walks. The induced subgraphs defined by the diffusion tree tend to be denser and more clustered than the induced subgraphs extracted with random walks. We will demonstrate this by two examples. First, we will consider a small synthetic toy example. Second, we will show that the sampled subgraphs are different when real graphs are considered.

Let us now consider a simple toy example to show that the sampling procedures have a quite distinctive behaviour even on small graphs. An undirected graph is depicted on Figure 5.1. It has a vertex and edge sets defined respectively by $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (c, d), (b, d)\}$. Now let us imagine that we want to sample 3 nodes with diffusion trees and random walks starting from node a. The exact probabilities of obtaining certain node sequences with the sampling methods are on the right hand side.



	Random Walk	Diffusion
$a - b - c$	$\frac{1}{6}$	$\frac{5}{14}$
$a - b - d$	$\frac{1}{6}$	$\frac{2}{14}$
$a - c - b$	$\frac{1}{6}$	$\frac{5}{14}$
$a - c - d$	$\frac{1}{6}$	$\frac{2}{14}$
$a - b - a$	$\frac{1}{6}$	
$a - c - a$	$\frac{1}{6}$	

Figure 5.1: Example undirected graph with sample sequences and exact sequence sampling probabilities. The graph $\mathcal{G}(V, E)$ is defined by vertex set $V = \{a, b, c, d\}$ and edge set $E = \{(a, b), (a, c), (c, b), (b, d), (c, d)\}$. The sequences have length 3 and all of them originates from vertex a. Sequence sampling probabilities show that diffusion trees are biased towards clustered neighbourhoods with short average path lengths.

We can conclude that random walks sample the possible sequences with equal probability. Moreover, we also see that random walks might sample nodes multiple times. We also observe that the diffusion process is biased towards sampling nodes in a manner that the induced subgraph has a lower diameter, shortest average path length and higher clustering. Importantly, this originates from the fact that a node ends up in the tree with higher probability if it has a higher number of neighbours in the seeder set. Because of this diffusion tree sampled node sets describe local neighbourhoods. Now we will demonstrate that this is true when we have a large graph that represents a real network.

To show that samples obtained by random walks and diffusion processes are quite different on a real network we will run a number of simple experiments. We will initiate 10^4 random walks and diffusion processes from a given vertex in the PPI network (Chatr-Aryamontri et al., 2014). With both sampling methods we will extract 40 nodes per sample graph and we will investigate the properties of the resulting induced subgraphs. We define the induced subgraphs by keeping the 40 nodes and the edges between them that appear in the original graph. Using the induced subgraphs we calculate three macro-level descriptive statistics to point out differences between the obtained sample graphs. These quantities are the graph density, transitivity (global clustering coefficient) and average shortest path length. These metrics are defined by Equations (5.1), (5.2) and (5.3) respectively. We use these measures to compare the subsampled graphs as they are quite informative regarding the sampled neighbourhoods.

$$\text{Density} = \frac{2 \cdot |E|}{|V| \cdot (|V| - 1)} \tag{5.1}$$

$$\text{Transitivity} = \frac{3 \times \text{Number of triangles}}{\text{Number of connected triangles of vertices}} \tag{5.2}$$

$$\text{Average shortest path length} = \frac{\sum\limits_{v \in V} \sum\limits_{w \in V} d(v,w)}{|V| \cdot (|V|-1)}, \quad v \neq w \tag{5.3}$$

Using the sampled subgraphs one can plot the distribution of the graph density, transitivity and average shortest path length and compare the distributions of these metrics obtained by the random walks and diffusion process. The distributions of induced subgraph statistics are plotted on the subfigures of Figure 5.2. Based on Subfigure 5.2a it is quite evident that nodes sampled with a diffusion tree are from more dense neighbourhoods on average than nodes sampled with random walks. The distributions on Subfigure 5.2b back up that diffusion tree sampled nodes are expected to be from more clustered subgraphs than nodes sampled with random walks. Finally, the results about average distance in the sample graphs are on Subfigure 5.2c. We can conclude that the induced subgraphs obtained with a diffusion tree have lower shortest path lengths on average. This was expected based on the fact that the induced subgraph density is higher on average when nodes are sampled with a diffusion tree. Essentially these observations imply that the regularities observed on the toy example hold for larger real networks. These findings also hint that diffusion tree based sampling will result in an embedding that is better at describing local network structure and worse in representing macro-level network structure. Furthermore, it also foreshadows that on the downstream machine learning task of community detection the diffusion tree generated embedding features outperform might the random walk generated ones.

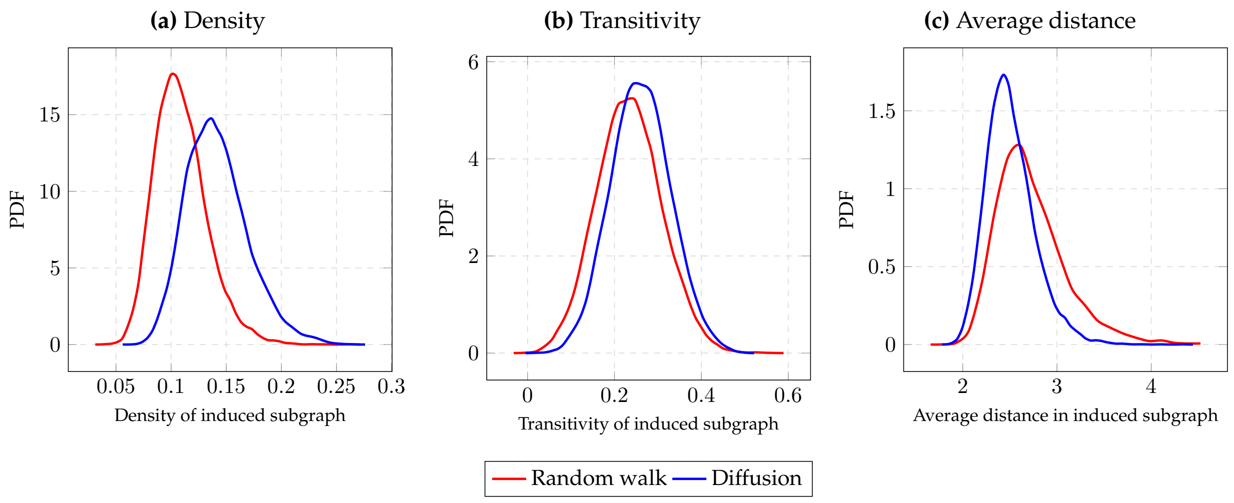


Figure 5.2: The distribution of random walk and diffusion tree sampled induced subgraph statistics on the PPI network. The sampling was initiated from the node with ID 100 and the size of the sampled subgraphs was set as 40. Altogether we sampled 10^4 graphs and for each of the induced subgraphs we calculated the graph density, transitivity and average shortest path length. The distribution of the statistics is plotted respectively on the left, centre and right subfigures. Based on the plots one can deduce that diffusion tree sampling takes nodes from more dense and clustered neighbourhoods than random walks.

5.2 Node position in the latent space and centrality

In this section we discuss how topological properties of nodes are related to properties of nodes in the latent space. When we use the expression *pairwise distances* of nodes on the graph we mean the length of the shortest path(s) between nodes. Now we formalize the meaning of approximate distance preservation. Let us consider that on a graph we have the randomly selected nodes e, f, g and h. The distance d is a distance on the graph and d' is a distance in the latent space. An embedding preserves pairwise distances between nodes if $d(e,f) < d(g,h)$ implies that $E[d'(e,f)] < E[d'(g,h)]$. To put it simply, if a randomly selected pair of nodes is closer than an other randomly selected pair on the graph we expect that the distance between pairs of nodes in the latent space has the same relation.

Now we create a simple experiment to show that embedding methods that we introduce have this property. We will use ED2V on the PPI network (Chatr-Aryamontri et al., 2014). The parameter settings of the algorithm were $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We randomly sample pairs of nodes from the graph that are at $1, 2, \ldots, 5$ distance from each other on the graph. For each of these graph distances we took 10^4 unique vertex pairs. After this we calculated the ℓ_1 distance in the latent space between these pairs of nodes. Finally, we plot the distribution of distances in the latent space conditional on the graph distance. These conditional distributions of pairwise distances are plotted on Figure 5.3.

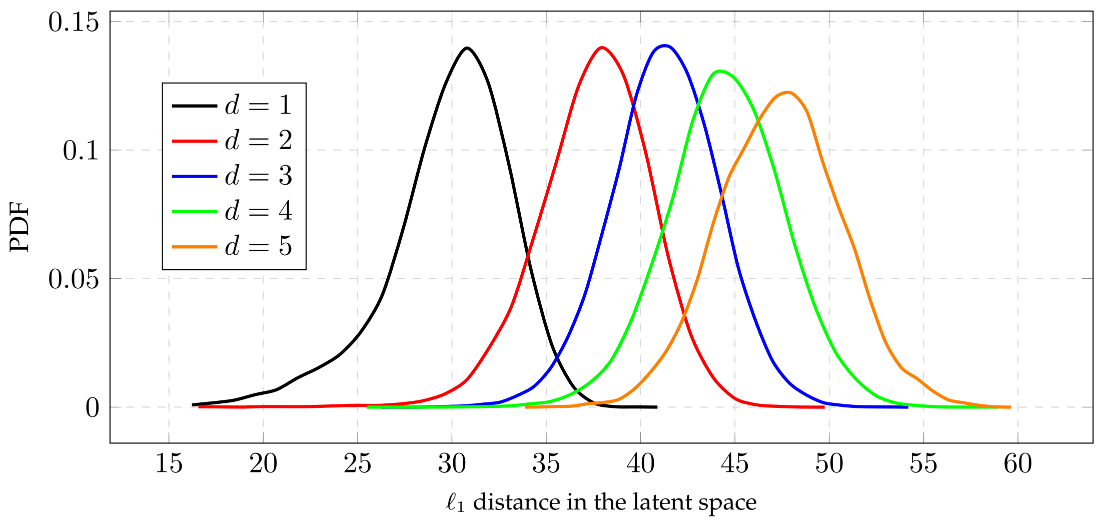


Figure 5.3: Distribution of distances in the latent embedding space of the PPI network conditional on the graph distance between randomly selected nodes. The embedding itself was generated by ED2V with settings such as $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. Each curve was estimated from 10^4 unique node pairs – we only sampled nodes from the largest connected component of the graph.

Based on Figure 5.3 it is evident that the relation of pairwise distances on graphs is only preserved between pairs of nodes approximately in the latent space. Moreover, it is also quite noticeable that the means of the conditional distributions of distances increase with the graph distance. The fact that relation of distances is approximately preserved in the latent space is quite helpful in the following cases:

(i) **Node classification:** Nodes that share the same label are expected to be close to each other when distances are measured with graph distances. As we mentioned earlier, this phenomenon is known as homophily. Because, pairwise distances are preserved in the latent space labels of a node can be predicted based solely on its position in the latent space.

(ii) **Edge prediction:** Nodes that have an edge between (nodes that are at distance 1 from each other) them have a distribution of distances that is separated from other distance distributions. This is particularly fascinating as later we will do edge prediction and having a well separated distribution will help with this task.

(iii) **Graph visualization**: Based on the embedding we can create low-dimension layouts where distances between nodes are approximately maintained.

Besides the pairwise distances other properties of nodes are also preserved approximately in the latent space. One can also investigate the association between node centrality on the graph and node distance from the origin in the latent space. We investigated two basic relationships:

(i) Distance of a node from the origin in the latent space and its closeness centrality. We define closeness centrality of node v as $CC_v = \sum_{u=1}^{|V|} \frac{1}{d(v,u)}$ using the standard assumption regarding disconnected nodes that $1/\infty = 0$.

(ii) Distance of a node from the origin in the latent space and its degree centrality. By degree centrality of node v we simply mean $\deg(v)$.

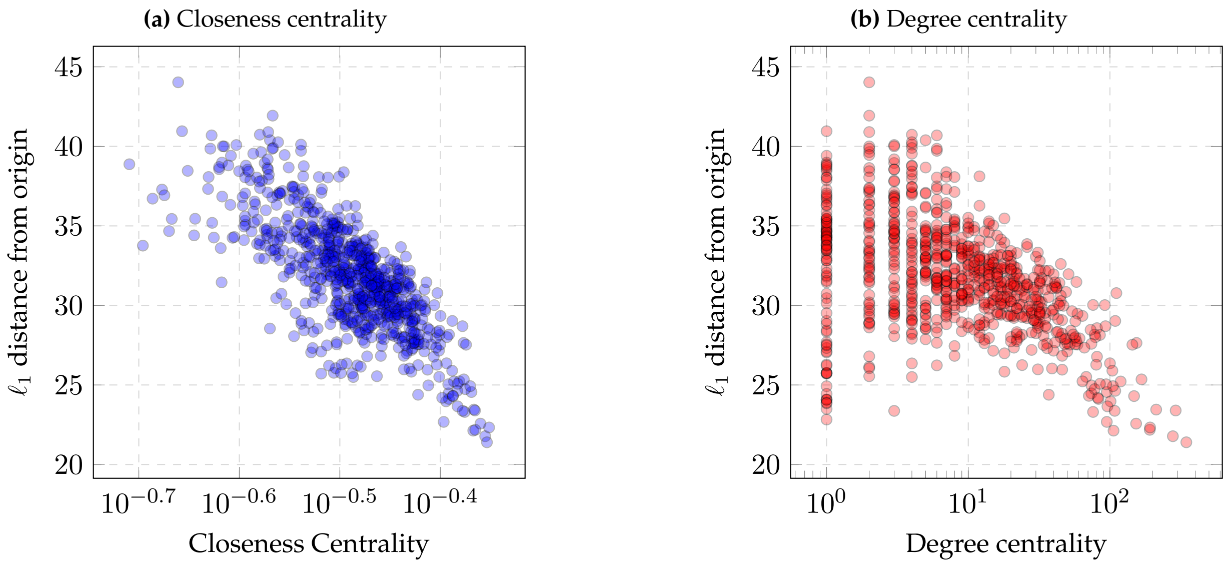


Figure 5.4: Association between the distance of vertices from the origin in latent space and node centrality measures. The dataset we used to create the plots is the PPI network. The embedding itself was generated by ED2V with settings $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. Each scatter plot was made with a subsample of 800 randomly selected nodes. The horizontal axes measure centrality on a log scale while the verticals measure $\ell_1$distance from the origin.

We use the ED2V embedding of the PPI network to investigate these relationships between node centrality and position in the latent space. The parameter settings of the algorithm were $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We took a sample of 800 nodes to create scatter plots that are informative. These are plotted on subfigures of Figure 5.4. Association between closeness centrality and ℓ_1 distance from the origin is plotted on Subfigure 5.4a. We observe that nodes that have a high closeness centrality are less distant from the origin – these vertices are in the centre of the latent space. Meaning that nodes that are central in the graph are also central in the latent space. This is a useful property to have when one does network visualization. The degree centrality and distance from origin relationship is plotted on Subfigure 5.4b. Importantly, there are two things to note. First, that a higher degree is associated with a position close to the origin in the latent space. Second, when the degree is low the variance of the distances from the origin is higher. This is due to small sized components of the graph that are not connected to the largest component and the fact that weights of the neural network are randomly initialized. Nodes from small components end up in random locations of the latent space. The FD2V embedding gives similar results and it is enclosed in Appendix B as Figure B.1. The distance and closeness centrality relationship is on Subfigure B.1a while the distance and degree scatter plot is on Subfigure B.1b.

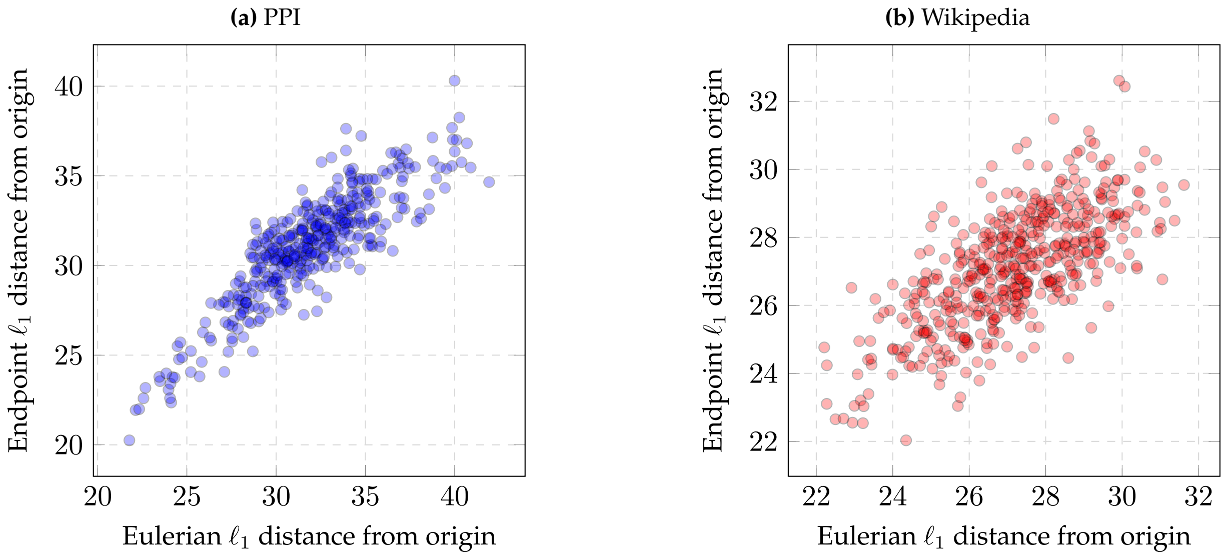


Figure 5.5: Association between the distance of vertices from the origin in latent spaces. We embedded the PPI and Wikipedia graphs with our proposed methods. Using the graph embeddings we calculated the ℓ_1 distances of vertices from the origin. The ED2V embeddings were generated with settings $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. Similarly, the FD2V embeddings were generated with settings $l = 25$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. Each scatter plot was created with a subsample of 400 randomly selected nodes.

As the results plotted on Figures 5.4 and B.1 show the behaviour of ED2V and FD2V is similar. Both of them conserves the centrality of nodes in the latent space approximately. This can be shown with a fairly simple simple experiment. First, we create embeddings of graphs with the two methods. Second, we calculate the distance of nodes from the origin in the two created latent spaces. Third, we create a scatter plot of the resulting distances based on a subsample of nodes. We created such scatter plots for the PPI and Wikipedia datasets – these are the subfigures of Figure 5.5. We see that the distances from the origin in the different embedding spaces are positively associated. In addition, we see that the association depicted on Subfigure 5.5a is stronger than the one on Subfigure 5.5b. Generally we deduce that nodes which are central in one of the embedding spaces will be central in the other one.

Besides the position of vertices in the latent space one can also investigate the distance of edge endpoints and how their distance relates to their centrality and neighbourhood overlap. In the following we will investigate these relationships. Once again we do analysis on the PPI network as it has a considerably small size and calculating centrality measures is less costly on smaller graphs. We generated embeddings with ED2V and the chosen parameter settings of the algorithm were $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. From the set of edges we draw subsamples of 400 edges to create the plots which are the subfigures of Figure 5.6. We calculated the ℓ_1 distance between the endpoint nodes of edges and plotted this distance as a function of the two quantities.

(i) **Neighbourhood overlap:** We define the neighbourhood overlap of nodes v and u as the fraction $\frac{|\mathcal{N}(v)\cap\mathcal{N}(u)|}{|\mathcal{N}(v)\cup\mathcal{N}(u)|}$, where $\mathcal{N}(u)$ and $\mathcal{N}(v)$ is the set of neighbours of nodes v and u respectively.

(ii) **Current flow betweenness:** Is the probability that a random walker on the graph goes through a given edge. For detailed discussion see Newman (2005) and Brandes & Fleischer (2005).

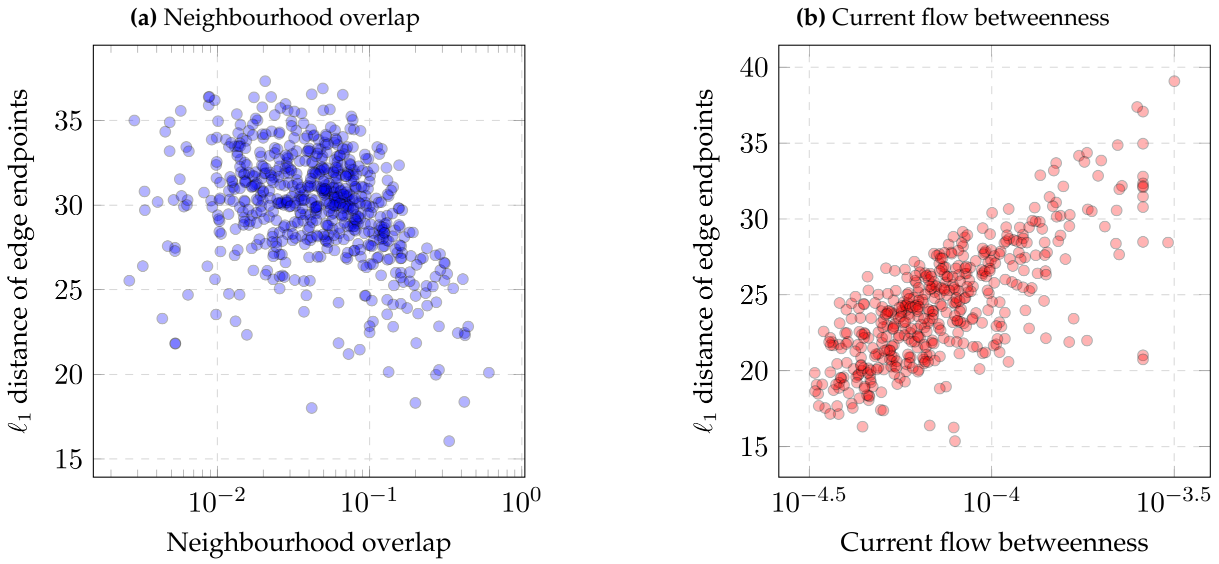


Figure 5.6: Association between neighbourhood overlap–edge endpoint distance and endpoint distance. The dataset we used to create the plots is the PPI network. The embedding itself was generated by ED2V with settings $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. Each scatter plot was made with a subsample of 400 randomly selected edges. The horizontal axes measure neighbourhood overlap and centrality on a log scale while the vertical axes measure ℓ_1 distance of endpoint nodes on a linear scale.

The distance of vertices at edge endpoints is plotted as a function of neighbourhood overlap on Subfigure 5.6a. We can observe that nodes that have an edge between them and have a high neighbourhood overlap are also located closer to each other in the latent space. This observed regularity is a fairly useful property when one does community detection. Namely because, clustering nodes in the latent space will also result in a partitioning of the nodes that is a dense subgraph. Our later results on the task of community detection tasks will underpin this. The current flow betweenness of edges and the distance of endpoints is plotted on Subfigure 5.6b.

The results on this plot are also fairly intuitive. If two nodes are distant in the latent space, but there is an edge between them it will have a high centrality. In a certain sense an edge like that is a bridge that fills a structural hole in the graph.

5.3 Visualizing graphs

In this section we will demonstrate that embeddings created by variants of D2V are useful tools when one creates network visualizations. This usefulness is not a unique property of D2V embeddings, because of this reference visualizations created with DW are included in the Appendix. In Section 5.2 we already demonstrated with our experiments that D2V approximately preserves the relation of graph distances between nodes in the latent space. Earlier we also provided considerable evidence that nodes with high centrality will end up in the centre of the latent space. These properties theoretically make D2V ideal for graph visualizations. We have visualized a number of synthetic graphs that have comparable sizes. Finally, we have to note, that we chose these graphs specifically because one might have prior assumptions about how a fair quality visualization of these graphs look like. The visualized synthetic graphs were the following:

- **Barabási-Albert graph:** We used a Barabási-Albert tree with 150 nodes. One expects that a high quality visualization positions nodes with high degree in the centre and branches of the tree are laid out in the embedding space uniformly (Albert & Barabási, 2002).
- **Watts-Strogatz graph:** Our experiment was done with a Watts-Strogatz graph with 150 nodes, 10 neighbours per node and a rewiring probability of 0.03 (Watts & Strogatz, 1998). A high quality visualization should lay out the nodes on a circle. In addition nodes with long range connections should gravitate towards the centre of the embedding space.
- **Kleinberg navigable small-world graph:** The embedding was created using a graph with 144 nodes, 1^{st} order neighbours, 1 random edge per node and connection probability exponent equal to -2 (Kleinberg, 2000). A good visualization should convey the message that rarely extant long edges shrink the graph. Nodes with these edges should be central. At the same time peripheral nodes should be laid out sparsely far from the centre.
- **Erdős-Rényi graph:** We chose an Erdős-Rényi graph with 150 nodes and 0.05 connection probability for creating the visualization (Erdős & Rényi, 1960). A fair layout of nodes would have a high number of nodes in the centre and the spatial concentration should decrease with the distance from this centre. In addition, a node's distance from the centre should be negatively associated with its degree.

The embedding algorithms parameter settings were set as $l = 40,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Visualizations were created based on the normalized 1^{st} and 3^{rd} vector of embeddings and plots are subfigures of Figure 5.7. Our plots support that the introduced embedding method is capable of generating informative visualizations for a range of synthetic graphs. However, the visualizations of the Barabási-Albert and Watts-Strogatz graphs have minor imperfections. First, the Barabási-Albert graph's layout on Subfigure 5.7a positions high centrality nodes in the centre and lays out the branches somewhat uniformly – these are favourable features. At the same time nodes with a single edge gravitate towards the centre of the embedding space, which is less favourable. Second, looking at Subfigure 5.7b we can conclude that the circular layout of the the Watts-Strogatz graph is generally quite informative. We also see that high-centrality nodes with rewired edges gravitate towards the centre. Furthermore, the shape of the circle is deformed by these nodes that gravitate towards the centre.

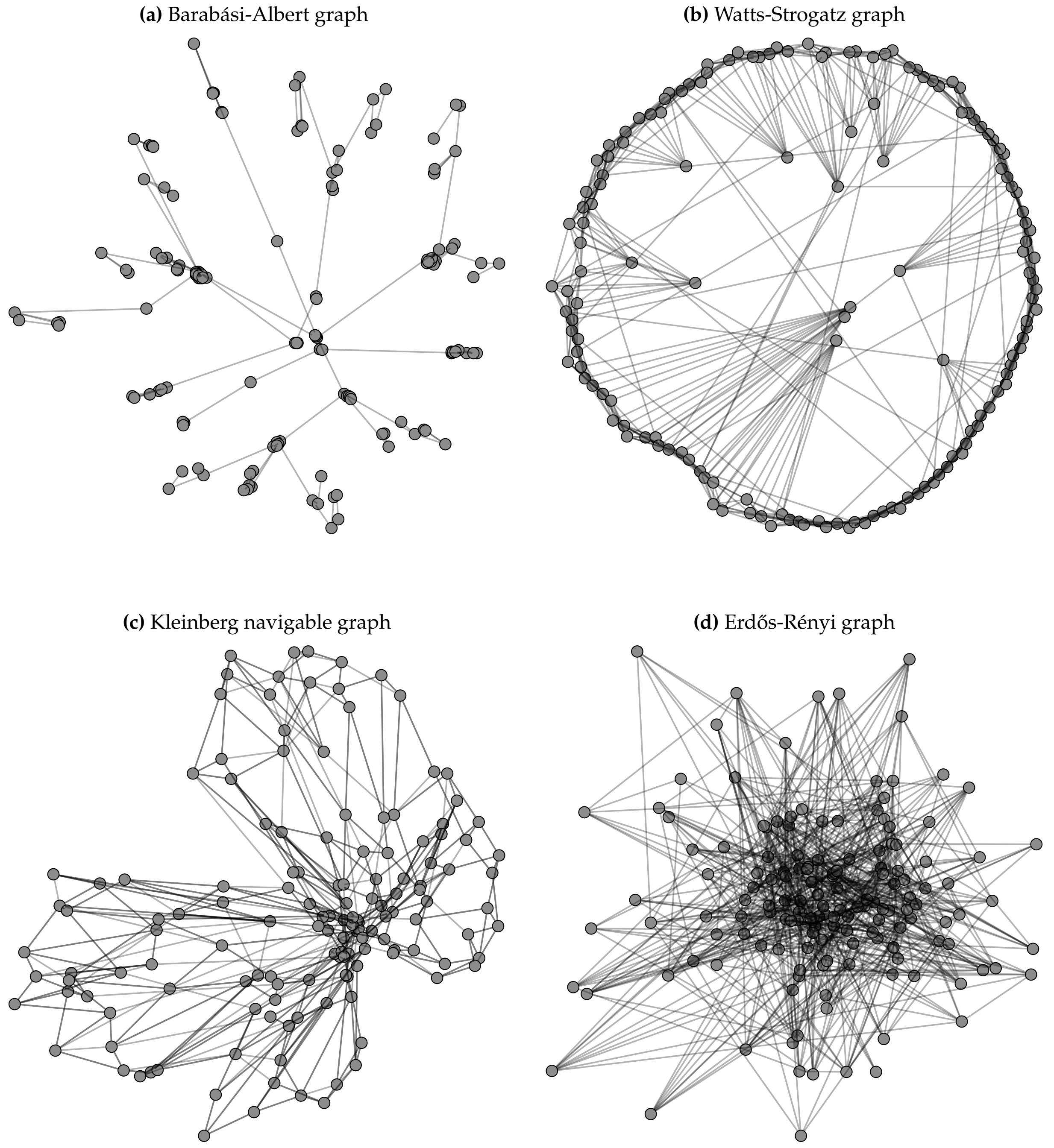


Figure 5.7: Visualizing graphs with ED2V embeddings. All of the embeddings were created with settings such that $l = 40$, $d = 3$, $n = 100$, $\widehat{w} = 10$, $\alpha = 0.025$. Feature vectors were normalized and the 1^{st} and 3^{rd} feature vector was used to create the visualizations. **(a)** Barabási-Albert tree with 150 nodes. **(b)** Watts-Strogatz graph with 150 nodes, 10 neighbours per node and rewiring probability equal to 0.03. **(c)** Kleinberg's Navigable Small-World graph with 144 nodes, 1^{st} order neighbours, 1 random edge per node and connection probability exponent equal to -2. **(d)** Erdős-Rényi graph with 150 nodes and random connection rate of 0.05.

Visualizations created with FD2V and DW are enclosed in Appendix B. Parameters of FD2V were $l = 25$, $d = 3$, $n = 100$, $\widehat{w} = 10$, $\alpha = 0.025$, while settings of DW were $l = 80$, $d = 3$, $n = 100$, $\widehat{w} = 10$, $\alpha = 0.025$. The Barabási-Albert and Watts-Strogatz graphs are on Figures B.2 and B.3. The Kleinberg navigable small-world graph is on Figure B.4 and the Erdős-Rényi graph is enclosed as Figure B.5. The quality of these layouts is similar except for the Watts-Strogatz graph, ED2V creates a considerably better layout of nodes than DW or FD2V.

Chapter 6

Applications of the embeddings

In this chapter we will use the node representations as features in downstream machine learning tasks. The chapter has two main goals. First, we want to demonstrate that the representation are useful for a number of supervised and unsupervised learning task. Second, we argue that the quality of the learnt representation is fairly competitive with other node sequence based graph embedding methods. Earlier in Chapter 5 we have implied that certain properties of the embeddings can help with semi-supervised node labeling, community detection and edge prediction. In the literature representation quality of sequence based embeddings is mostly measured by the performance on semi-supervised node labeling and edge prediction tasks. However, we believe that providing results for graph-clustering is an important contribution and also a highly relevant possible application of the created embeddings. We provide benchmarks on these downstream tasks with a wide variety of networks in terms of size and type.

Results on the multi-label node classification task are discussed in Section 6.1. Findings about the community detection are presented in Section 6.2. Edge prediction results are overviewed in Section 6.3 where we also provide algorithms to create synthetic datasets for edge prediction.

6.1 Multi-label node classification

This section solely focuses on how the learnt graph representations behave when they are used to classify nodes. In Subsection 6.1.1 we provide benchmarks on a number of networks that are used in the literature to describe the representation quality of sequence based graph embedding algorithms. We compare the results of sequence based methods to other baseline node labeling procedures in Subsection 6.1.2. Finally, the sensitivity of labeling performance to parameter changes is discussed in Subsection 6.1.3.

We utilize some widely used datasets of real networks to asses the representation quality obtained by using the node sequence based embedding methods. These are the following:

(i) **BlogCatalog:** Is a social network of bloggers, nodes are bloggers and links are social relationships – labels represent the interest of bloggers (Agarwal et al., 2009).

(ii) **PPI:** Is a protein-protein interaction network of humans – labels express biological states (Chatr-Aryamontri et al., 2014).

(iii) **Wikipedia:** Is a word co-occurrence network based on a chunk of the Wikipedia corpus – labels represent part of speech tags (Mahoney, 2011).

(iv) **Flickr:** A network of Flickr users – labels describe interests in types of photos (McAuley & Leskovec, 2012).

(v) **Youtube:** Is a friendship network of Youtube users – labels are common interests in video genres (Yang & Leskovec, 2015).

(vi) **Markercafe:** Is a social network mostly used in Israel, nodes are users and edges are social ties between them – labels speak for group memberships (Fire et al., 2011, 2013).

A node in these networks can have multiple labels – a Youtuber might have interest in multiple genres or a Markercafe user might be a member of multiple groups. The number of nodes, edges and unique labels are summarized by Table 6.1 for these networks. One can easily see that we present results on a wide variety of networks in terms of size, density and unique label number. This helps to ensure that our comparison is fair and we are able to show that the relative and absolute performance of our methods is consistent across datasets.

	BLOGCATALOG	**PPI**	**WIKIPEDIA**	**FLICKR**	**YOUTUBE**	**MARKERCAFE**
|**V**|	10,312	3,890	4,777	80,513	1,128,499	53,253
|**E**|	333,982	38,379	92,517	5,899,882	2,990,443	1,744,194
Unique labels	39	50	40	195	47	88

Table 6.1: Size and number of labels on the graphs used to asses classification performance. The number of vertices, edges and unique labels are respectively in the first, second and third row of the table. We included each of the graphs used for benchmarking the quality of representations for multi-label classification. These datasets later are also used for assessing representation quality for community detection.

6.1.1 Semi-supervised multi-label classification

In this subsection we will use logistic regression to predict the labels of nodes. For each label we will fit a separate logistic regression with ℓ_1 regularization and the shrinkage parameter is set as $\lambda = 1$. The features extracted with embeddings are all normalized and the use of ℓ_1 regularization allows for potentially shrinking the weights of the regression to zero. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). These settings for N2V and DW are the ones that were used by Grover & Leskovec (2016) and (Perozzi et al., 2014). For Flickr, Youtube and MarkerCafe the best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 10% of data as the training set. We chose the pair of *inout* and *return* parameters that resulted in the highest micro F-1 score. For the other datasets we took the optimal parameter settings of Grover & Leskovec (2016). Because of the computational performance issues of the N2V Python reference implementation, embeddings of the Flickr, Youtube and Markercafe graphs were created with the high performance C++ version of N2V. All of the D2V embeddings were generated by our Python reference implementation. On the subfigures of Figure 6.1 we plotted the average classification performance as a function of the training dataset size. The horizontal axis represents the fraction of training data while the vertical measures performance with micro averaged

F-1 score. Each point represents average performance on the test set based on 10 random training and test splits. Performance measurements on the same datasets using macro F-1 scores are enclosed in Appendix C as Figure C.1.

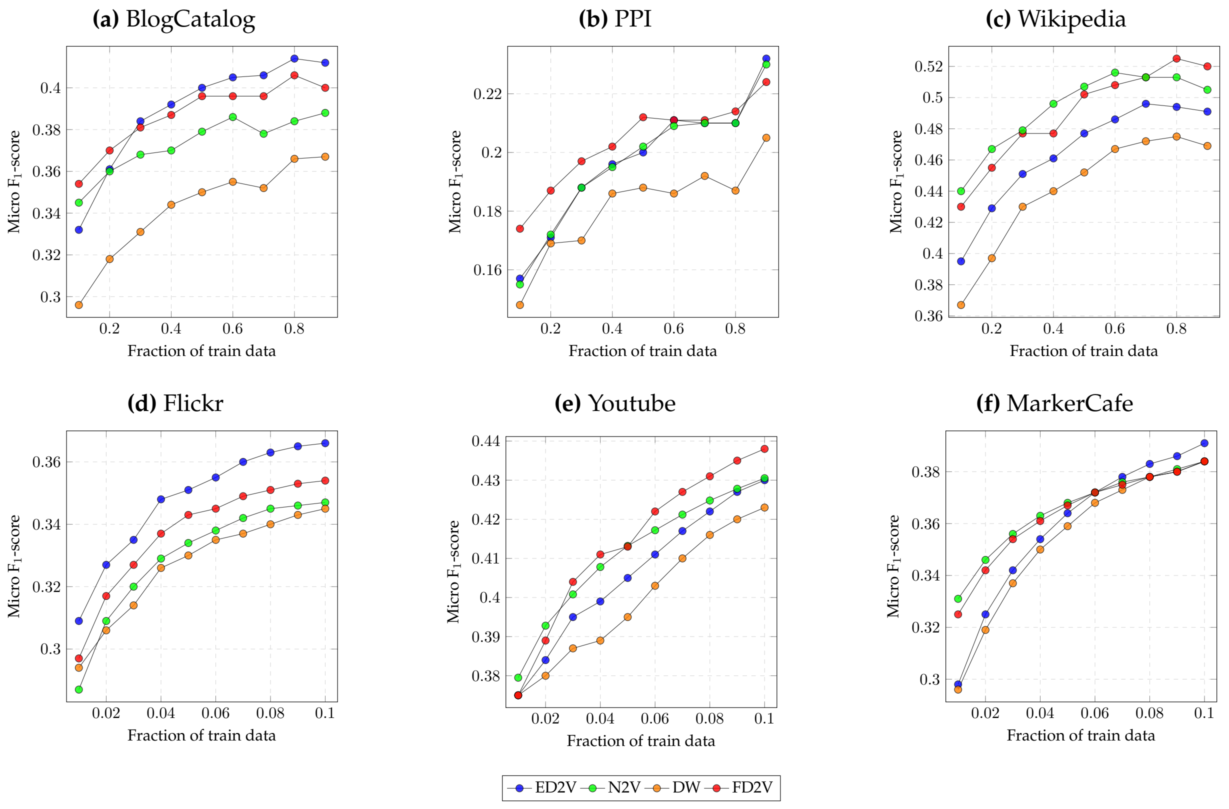


Figure 6.1: Classification performance of node sequence based embedding methods measured by micro F-1 score. Multi-label classification performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization the shrinkage parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 10% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. The fraction of training data is on the horizontal axis and the classification performance measured by micro F-1 score on the test set is on the vertical axis. Each point represents the average micro F-1 score based on 10 random train-test splits. The DW and N2V embeddings for the Flickr, Youtube and Markercafe datasets were generated with the high performance C++ implementation.

On the BlogCatalog dataset with sufficient amount of data ED2V clearly outperforms N2V and DW. The performance of FD2V is poor compared to other methods. Considering the PPI dataset it is evident that DW and ED2V underperform compared to N2V when the fraction of training data is small. Just as before FD2V performs poorly compared to other embedding methods. When one looks at the results regarding the Wikipedia dataset it is obvious thaty random walk based methods outperform the diffusion based ones and the performance difference is not just marginal. Also it seems like that the parameter settings of N2V given in Grover & Leskovec (2016) are suboptimal as theoretically DW should not be able to outperform it. Looking at the results on the Blogcatalog and Wikipedia networks we also observe that some of the models are slightly overfitted when a large fraction of training data is available. On the Flickr dataset ED2V outperforms every other method. Roughly with 5% of training data we are able to outperform random walk based methods even when those have 10% of labelled data available.

Performance on the Youtube network is in favour of N2V and DW. Contrarily, on the Marker-Cafe dataset random walk based methods show good performance when only a small fraction of nodes is labelled. Besides these dataset specific observations we can also infer that generally the increased dataset size results in decreasing marginal performance returns in most of the case. These findings imply that on this task ED2V is competitive with random walk based methods while FD2V is not. It has to be emphasized that N2V required a quite exhaustive grid-search. We created 5×5 embeddings for each of the networks and we had to evaluate the node classification performance for each of them to pick the embedding with the best representation quality. This is a serious drawback when the dataset is large – e.g. Flickr, Youtube and Markercafe datasets.

6.1.2 Comparison to other graph embedding methods

In this subsection we compare the classification performance of our methods to some baseline node labeling mechanisms. The baseline methods used for comparison are:

1. **EdgeCluster:** Clusters the adjacency matrix of the graph with k-means clustering and hot-one encoded cluster memberships are the features used for classification (Tang & Liu, 2009b). It is fairly scalable compared to methods that require the calculation of eigenvalues from the adjacency or modularity matrices.

2. **Modularity:** Extracts eigenvectors from the modularity matrix of the graph and uses the eigenvectors as features for classification (Tang & Liu, 2009a).

3. **WVRN:** A considerably simple baseline – a node has a label if majority of its labelled neighbours has the label. The abbreviation stands for weighted vote relational neighbour classifier (Macskassy & Provost, 2003). If a node has no labelled neighbours it receives the most common labels.

4. **Majority:** The simplest baseline. Each node receives the labels that are the most common among labelled nodes.

We use results from Perozzi et al. (2014), so we can compare our findings to these baselines on the BlogCatalog, Flickr and Youtube datasets. Our results used for the comparison of algorithms is directly taken from Subsection 6.1.1. The micro F-1 values obtained on the test data are tallied in Table 6.2 where rows denote algorithms, column headers are the datasets used for benchmarking and subheaders note the fraction of training data used. The macro F-1 summary results are enclosed in Appendix C as Table C.1. The first and most important notion is that sequence based embedding methods outperform the baselines on every given benchmark dataset. Even FD2V has a solid advantage over them. On the BlogCatalog and Flickr graphs ED2V has a clear advantage over other methods and the same is true for N2V on the Youtube graph. Besides these we can also see that the performance gap between the top performing sequence based embedding methods and top baseline classifier changes sometimes as more training data is available. The performance gap on BlogCatalog between ED2V and EdgeCluster is 0.0628 points initially in terms of micro F-1, just doubling the training data size reduces the gap to 0.0548. On the Flickr graph we cannot observe such tendencies as changes in the performance gap are negligible. Considering the Youtube dataset we see that the initial advantage of N2V is 0.0551 and this is shrank to 0.0440 when the size of the dataset is doubled. Conclusive results about this observation would need more investigation possibly with the inclusion of other datasets and parametric settings.

	BlogCatalog		Flickr		YouTube	
% Labeled nodes	**40%**	**80%**	**4%**	**8%**	**4%**	**8%**
ED2V	**0.3927**	**0.4147**	**0.3489**	**0.3638**	0.3992	0.4221
FD2V	0.3448	0.3665	0.3267	0.3400	0.3895	0.4162
N2V	0.3872	0.4064	0.3372	0.3596	**0.4227**	**0.4386**
DW	0.3701	0.3842	0.3290	0.3453	0.4078	0.4248
EdgeCluster	0.3299	0.3599	0.3031	0.3176	0.3676	0.3946
Modularity	0.3297	0.3723	0.2760	0.2889	–	–
WVRN	0.2882	0.3333	0.2097	0.2125	0.3288	0.3775
Majority	0.1670	0.1649	0.1646	0.1662	0.2523	0.2534

Table 6.2: Classification performance compared to other feature generation methods measured by micro F-1. Numbers in the columns represent micro F-1 test scores on the training dataset. Column headers are the dataset names and subheaders are the training dataset sizes. The results of node sequence based embedding methods are the same as the ones in Figure 6.1. Baseline results were taken from the work of Perozzi et al. (2014). Bold numbers denote the best performing method.

6.1.3 Sensitivity to parameters

As part of our investigation in this subsection we also present results regarding the sensitivity of the performance. We will show how the manipulation of model parameters effects the performance on the downstream machine learning task. We will use the Flickr dataset to demonstrate how the performance changes with model parameter changes. We assume that the baseline embeddings were created with graph embedding settings $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$, $k = 1$. We set the sequence length controlling parameters of algorithms to be $l = 40$ (ED2V) and $l = 25$ (FD2V). In each experiment we choose one of the parameters to be changed. We trained ℓ_1 regularized logistic regression on 10% of the data and evaluated the micro F-1 score on the remaining data. We repeated this process 10 times with a random test-train split and obtained the average test performance. Obtained results for the sensitivity to feature vector dimension, window size, vertex set size, diffusion number, epoch number and learning rate are subplots of Figure 6.2. On each plot the manipulated parameter is on the horizontal axis and the performance is on the vertical axis. Performance evaluation in terms of macro F-1 score with the same experimental setup is enclosed in Appendix C as Figure C.2.

Now let us look at the experimental findings. First, looking at Subfigure 6.2a we see that the size of the feature vectors is optimal when the dimension of the vector is 64. This is true for both variants of D2V. Increasing or decreasing the feature vector size deteriorates the classification performance. This implies that in a very low dimensional space the nodes are not well separated and also that a very sparse representation does not keep nodes with similar labels close to each other in the embedding space. Findings regarding the manipulation of the window size are on Subfigure 6.2b. Sliding window size varies between 2 and 20 with a step size of 2. We see a similar pattern – there is an optimal sliding window size in the chosen interval. For ED2V a window size of 6 is optimal, for FD2V a smaller window size is better. This also implies that including hitting frequencies of nodes that are further away from a source node introduces noise. The effect of increasing the number of vertices included in the diffusion tree is plotted on Subfigure 6.2c. It is not surprising that increasing the sample size increases classification performance. With a larger sample we get better estimates of hitting frequencies and that increases the representation quality. We also see that marginal micro F-1 gains are decreasing with the size of the vertex set size.

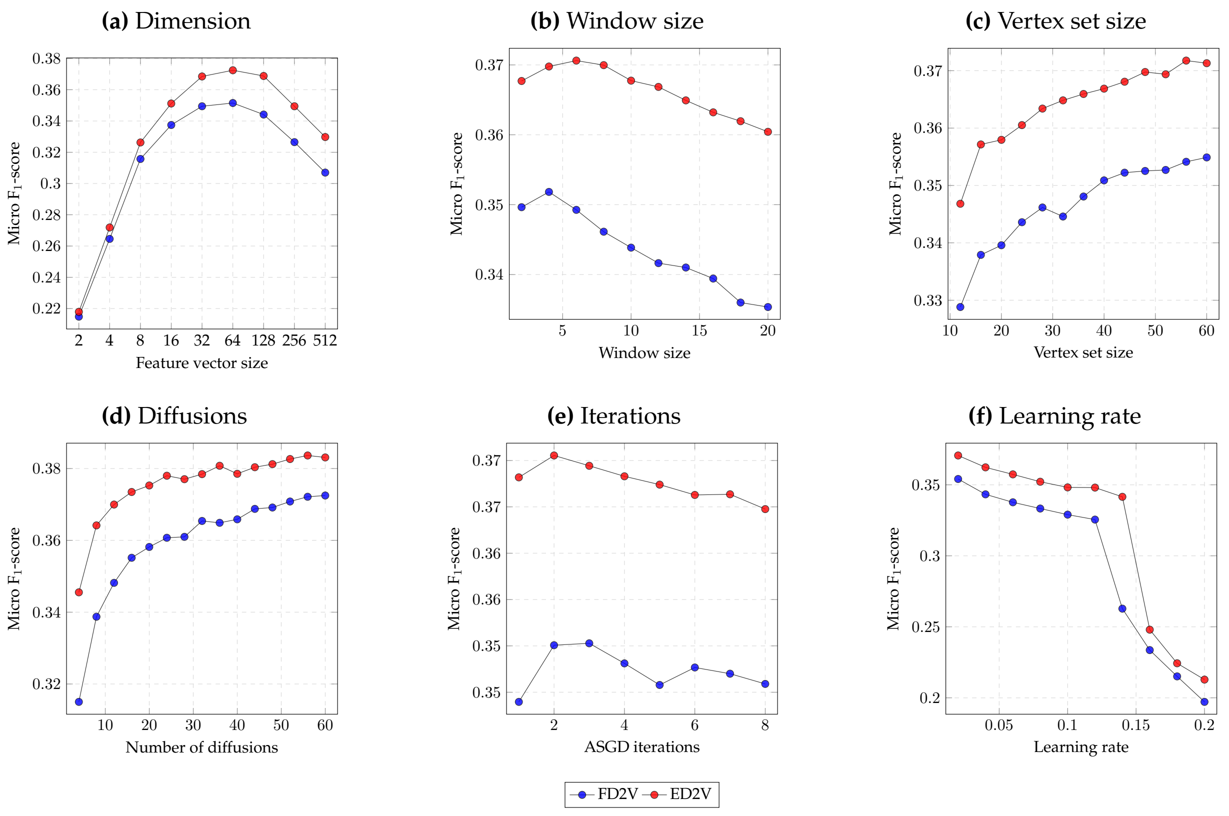


Figure 6.2: Sensitivity of multi-label classification performance of logistic regression using features extracted with the D2V variants to change of parameters. The classifier was fitted with ℓ_1 regularization and the regularization parameter was set as $\lambda = 1$. Embeddings were created with the baseline parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$, $k = 1$. We set the sequence length controlling parameters of algorithms to be $l = 40$ (ED2V) and $l = 25$ (FD2V). Embedding algorithm parameters were tuned to show the sensitivity of the results. The manipulated parameters are on the horizontal axis and the classification performance measured by micro F-1 score on the test set is on the vertical axis. Each point represents the average micro F-1 score based on 10 random train-test splits. In each split we used 10% of the dataset for training and the remainder for testing.

The classification performance depending on the number of diffusions per source node is plotted on Subfigure 6.2d. Once again one would expect that having more samples of node sequences would improve the quality of the learnt representation and this is what we see. In addition, the marginal performance improvements are decreasing again just as they did on Subfigure 6.2c when the vertex set size was increased. Subfigure 6.2e illustrates what happens to the classification performance when the number of asynchronous gradient descent iterations increases. We can see that increasing the number of epochs initially increases than later reduces the performance. This means that even when the representation learning is non supervised overfitting happens. The effect of learning rate variation on performance is plotted Subfigure 6.2f. Just as in case of the number of epochs we observe that a too high learning rate results in overfitting. When the learning rate reaches a certain starting value we observe a quite stark drop in labeling performance.

These findings have interesting implications. First, those parameter settings that are widely used to benchmark the node sequence based graph embedding methods are suboptimal for ED2V and FD2V in this dataset. Second, the performance gap between the D2V variants is quite consistent. The only exception is dimension it seems that in higher dimensions the performance gap is greater between the two. Third, it is indisputable that increasing the vertex set size and diffusion number per source with additional parameter changes (for example having 2 epochs and decreasing the window and dimension sizes) could lead to a significant performance increase on this downstream task.

6.2 Community detection

This section presents how the embeddings can be used to do community detection. Community detection is one of the most basic unsupervised machine learning tasks on graphs. The goal of it is to assign the nodes into clusters in a way that they form more dense subgraphs than the original graph. With the obtained clusters of nodes one might identify interest groups in social networks, research communities in citation graphs or groups of companies that perform financial transaction intensively. In certain regards this section is quite unique as the works that discuss node sequence based graph embedding methods (Perozzi et al., 2014; Tang et al., 2015; Grover & Leskovec, 2016; Pimentel et al., 2017) do not consider solving this downstream machine learning task. Moreover, it turns out that results obtained by clustering nodes in the embedding space are competitive with results attained by a number of widely used community detection algorithms.

The remainder of the section is structured as follows. In Subsection 6.2.1 we present graph clustering results on the networks that we used to present our node classification results. We have experimental results where we demonstrate that k-means and mini batch k-means clustering gives high quality results even on large graphs. Later we also support evidence that hierarchical clustering of nodes in the latent space gives fair grade clusters on smaller graphs. We compare the performance of our methods to other widely used graph clustering methods in Subsection 6.2.2. Lastly, we carry out a complete sensitivity analysis of clustering quality with respect to model parameters in Subsection 6.2.3.

6.2.1 Community detection with sequence based embeddings

In this subsection we discuss the idea of how the learned representations can be used to perform graph clustering. The graph embedding itself maps each of the nodes into a latent space. As our experiments in Chapter 5 have demonstrated this latent space has two properties that are fundamental to clustering the nodes. First, nodes that share an edge are expected to be closer to each other in the latent space than nodes that are at large distances from each other. Second, connected nodes that have a high neighbourhood overlap are closer to each other than those that have a lower neighbourhood overlap. This means that the procedure described below should be able to extract communities from a graph using the embeddings. Our proposed community detection method allows for parameter tuning in the graph embedding and clustering phases when nodes are already embedded in the latent space.

(i) Learn an embedding of the graph with a sequence based embedding method.

(ii) Cluster the data points of the embedding with a chosen clustering method.

(iii) Extract cluster memberships for each data point.

(iv) Assign nodes to clusters based on cluster memberships in the latent space.

(v) Calculate graph clustering quality metric.

The choice of the clustering quality metric also gives some freedom. We evaluate the cluster quality by using the unweighted modularity (Newman, 2006). Let as assume that the components of vector **c** describe the cluster membership of nodes. On a graph $\mathcal{G}(V, E)$ the unweighted modularity of the clustering vector **c** is defined by Equation (6.1). The matrix **A** is the adjacency matrix if v and w are neighbours it takes a value equal to 1 otherwise it is 0. Lastly, δ is the Kronecker delta function, if two nodes are in the same cluster it is 1 otherwise it is 0. Modularity is always in the $[-1/2, 1)$ interval and higher values suggest a better clustering of a graph.

$$\mathcal{Q}(\mathcal{G}, \mathbf{c}) = \frac{1}{2 \cdot |E|} \cdot \sum_{v \in V} \sum_{w \in V} \left[A_{v,w} - \frac{\deg(v) \cdot \deg(w)}{2 \cdot |E|} \right] \cdot \delta(\mathbf{c}_v, \mathbf{c}_w) \tag{6.1}$$

In the following we will demonstrate that k-means and hierarchical clustering gives good results on benchmark networks when we use modularity to evaluate the clustering quality.

K-means clustering

We use the standardized features extracted with sequence based graph embedding methods to do perform k-means clustering of nodes. We used the squared error as the loss function. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$ and $k = 1$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The N2V *inout* and *return* parameters were taken from the solution to the classification task. Performance as a function of cluster number is plotted on Figure 6.3 for the Blogcatalog, PPI, Wikipedia, Flickr, Youtube and Markercafe networks. On the horizontal axis we have the number of clusters and modularity is on the vertical axis. We varied the number of clusters between 2 and 50. Each point represents the maximal modularity score based on 10 random cluster centre initializations. The DW and N2V embeddings for the Flickr, Youtube and Markercafe datasets were generated with the high performance C++

implementation. Because of the size of the dataset, results on the Youtube dataset were created by mini-batch k-means clustering with a batch size equal to 100.

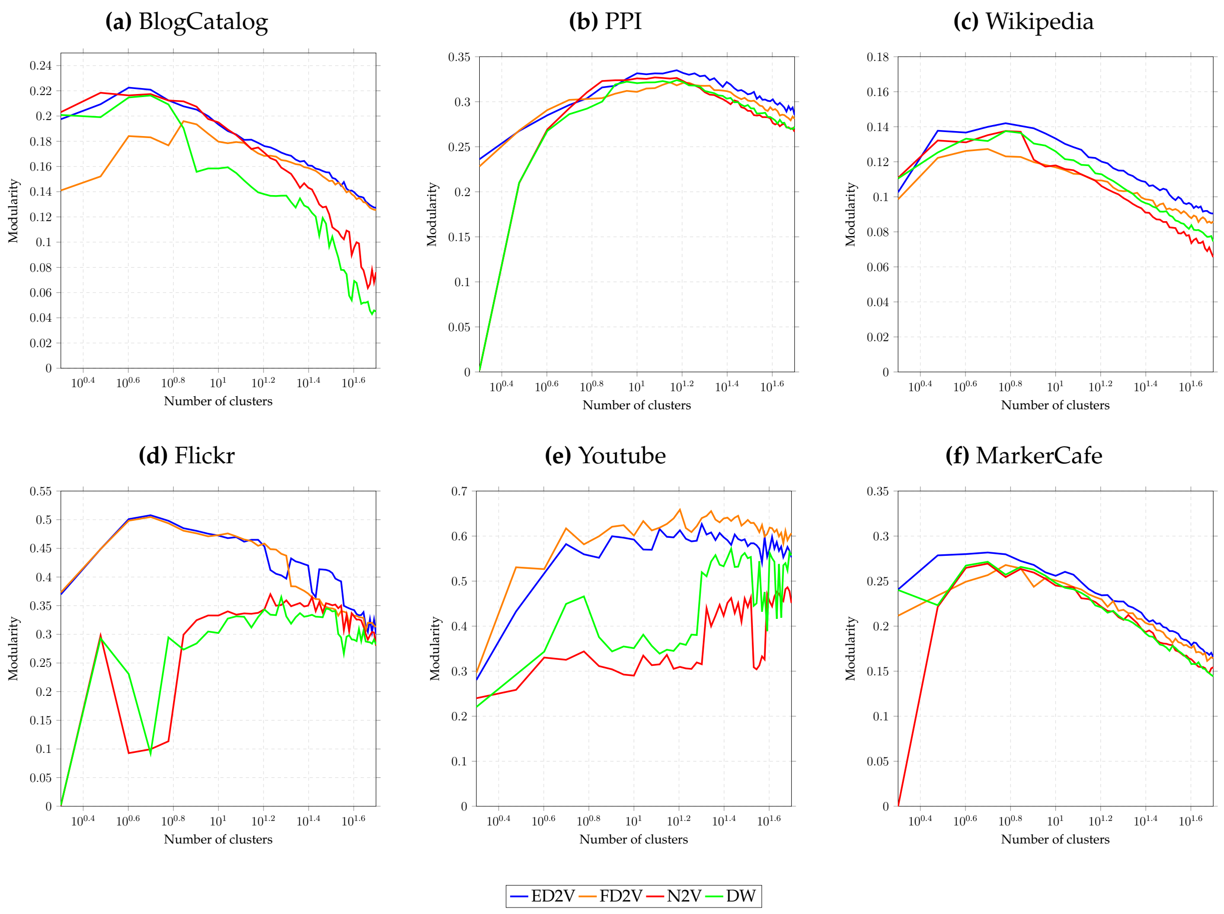


Figure 6.3: Community detection performance of k-means clustering using the standardized features extracted with the sequence based graph embedding methods. We used the squared error as the loss function. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The N2V *inout* and *return* parameters were taken from the solution to the classification task. On the horizontal axis we have the number of clusters and modularity is on the vertical axis. Each point represents the maximal modularity score based on 10 random cluster centre initializations. The DW and N2V embeddings for the Flickr, Youtube and Markercafe datasets were generated with the high performance C++ implementation.

Our results plotted on Subfigure 6.3a on the BlogCatalog network show that ED2V slightly outperforms other sequence based methods. In the high cluster number region the diffusion based algorithms do better than the random walk based ones. We also see that with these parameter settings N2V outperforms DW. Most of the methods performs well when the number of clusters is 4. Now let us review results on the PPI and Wikipedia networks plotted on Subfigures 6.3b and 6.3c. Once again we see that ED2V outperforms other methods by a margin in terms of modularity on both networks. Moreover, the diffusion based methods outrun the other two when we disassemble these graphs to smaller subgraphs. While the performance gaps between methods seem to be imperceptibly widening as the number of clusters increases we have to keep in mind that regions with a large number of subgraphs are not necessarily relevant.

On the larger networks results with DW and N2V are quite different from earlier findings. Results about the Flickr dataset are plotted on Subfigure 6.3d. We see that even with multiple initializations DW and N2V results in clusterings that have a particularly low modularity compared to D2V variants. We assume that this regularity and the noticeable modularity drop in the low cluster number region is caused by the same phenomenon. Namely, that 10 initialization of the cluster centres for k-means clustering is not sufficient for finding good centres or the initialization method itself is not suitable for choosing good initial centres. The Youtube network embedding was clustered by mini-batch k-means – modularity scores are plotted on Subfigure 6.3e. Our methods outperform the random walk based algorithms, but at the same time we observe again signs that the other methods ended up in local minima in most of the cases when the number of clusters was low. We assume this because sudden drops and jumps of modularity should not be observed when the cluster number is increased – a good clustering method should remove a weakly connected component from the graph when the cluster number is increased slightly. Surprisingly out of the two diffusion based methods FD2V is the better and it has a considerable performance lead over ED2V. Finally, results on the MarkerCafe graph are plotted on Subfigure 6.3f. On this graph the behaviour of the sequence based methods is similar to the one that we observed on Subfigures 6.3a, 6.3b and 6.3c. We see that ED2V outperforms every other method and also that most probably neither of the methods ended up in a local minimum.

Hierarchical clustering

Using the already obtained graph embeddings we can create other clusterings of the nodes. In the coming, we will demonstrate that hierarchical clustering is also an option when the graph is small. Community detection experiments with hierarchical clustering included the BlogCatalog, PPI and Wikipedia networks as these graphs all have somewhat limited vertex set sizes. We used ℓ_2 distance of nodes in the latent space and Ward's method to create the linkage matrix. The criterion used to form the clusters themselves is the maximal cluster distance. Because of the deterministic nature of the method we only did one experimental run while we varied the number of clusters between 2 and 50 inclusive. We plotted on subfigures of Figure 6.4 the modularity values as a function of cluster number. Once more we decided to use a log scale on the horizontal axis of figures.

First, the hierarchical clustering results on the BlogCatalog network are plotted on Subfigure 6.4a. Interestingly N2V performs poorly even when it is compared to DW. The Eulerian version of D2V is slightly better than DW and the endpoint traceback based D2V performs well when the number of clusters is large. Compared to the k-means clustering results hierarchical clustering performs quite poorly with every embedding feature set. Modularity values as function of cluster number on the PPI dataset are plotted on Subfigure 6.4b. Random walk based methods are clearly outperforming the diffusion based algorithms. However, we have to note again that k-means outperformed hierarchical clustering on this dataset. Interestingly k-means and hierarchical clustering both suggest that the optimal number of communities is between 10 and 16. Finally, performance measurements on the BlogCatalog dataset are plotted on Subfigure 6.4c. We observe again, that the random walk based graph embedding procedures outperform the diffusion based ones when we cluster nodes with hierarchical clustering. This is true for a wide range of potential cluster numbers. Altogether, we conclude that k-means clustering results in higher quality clusters. Besides, this performance gap, we also have to consider the fact that hierarchical clustering does not scale to the larger datasets e.g. Flickr with approximately 80,000 nodes.

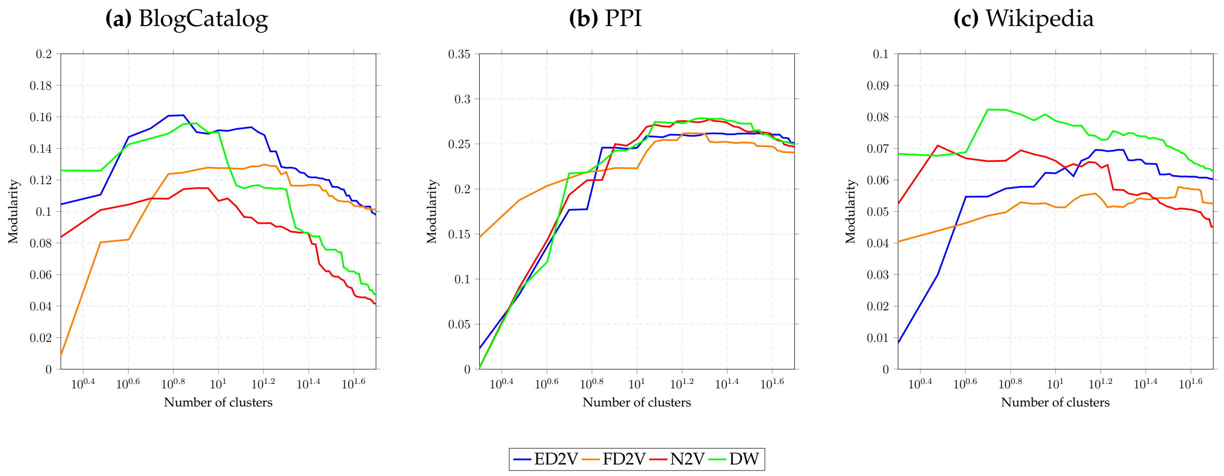


Figure 6.4: Community detection performance of hierarchical clustering using the standardized features extracted with the sequence based graph embedding methods. We used ℓ_2 distances and Ward's method to create the linkage matrix. Finally, the criterion used to form the flat clusters is the maximal cluster distance. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The N2V *inout* and *return* parameters were taken from the solution to the classification tasks. On the horizontal axis we have the number of clusters on a log scale and modularity is on the vertical axis.

6.2.2 Comparison to other community detection methods

In this section we will compare the performance of our proposed clustering method to community detection procedures. We will demonstrate that our algorithms outperform most of the widely used graph clustering methods that are implemented in the newest version of IGraph (Csardi & Nepusz, 2006). The community detection methods that we will use to benchmark our procedures are the following:

(i) **Fast-Greedy**: Is a hierarchical agglomeration algorithm which merges dense subgraphs to form communities (Clauset et al., 2004). It has a computational complexity of $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$ which makes it unfavourable on large graphs.

(ii) **Leading Eigenvector**: Uses the eigenvectors of the Laplacian matrix to extract higher than average density subgraphs from the network (Girvan & Newman, 2002). It is also a fairly complex algorithm considering that it has a $\mathcal{O}(|V|^2 + |E|)$ runtime.

(iii) **Louvain**: This procedure optimizes an objective function similar to modularity in a greedy way (Blondel et al., 2008). Moreover, it has a fairly low computational complexity of $\mathcal{O}(|E|)$ when the graph is sparse. Sparsity is a property observed in most of the real networks.

(iv) **WalkTrap**: Uses random walks on the graph to extract dense subgraphs where the random walker on the graph is stuck (Pascal & Latapy, 2005). It has a relatively high complexity of $\mathcal{O}(|E| \cdot |V|^2)$. We cannot use WalkTrap as a baseline on the Youtube graph because of this.

In our experiments we clustered the BlogCatalog, PPI, Wikipedia, Flickr and MarkerCafe networks with all of these baseline community detection methods. The Youtube graph was only clustered with the Louvain method as it has a fairly large size both in terms of vertices and edges. We used the default parameter settings of the algorithms to extract the communities and measured the clustering performance of the methods by modularity. The results obtained with the sequence based embeddings were taken from the experiments that were presented in Subsection 6.2.1. We took the highest modularity values for all of the embedding methods. Hierarchical clustering results are only presented for the small graphs – the Blogcatalog, PPI and Wikipedia networks. Our results are summarized by Table 6.3, where dashes denote missing measurements and the use of bold font stands for the best performing method on a given dataset.

ALGORITHM	BLOGCATALOG	PPI	WIKIPEDIA	FLICKR	YOUTUBE	MARKERCAFE
Fast Greedy	0.2069	0.3029	0.1456	0.4517	–	0.2597
Walktrap	0.1766	0.2571	0.0553	0.4873	–	0.2026
Leading Eigenvector	0.2035	0.2262	0.0915	0.4810	–	0.2455
Louvain	**0.2362**	0.3323	**0.1472**	**0.5196**	**0.7107**	**0.3004**
K-means DW	0.2163	0.3240	0.1375	0.3651	0.5718	0.2774
K-means FD2V	0.1959	0.3216	0.1273	0.5047	0.6553	0.2678
K-means ED2V	0.2225	**0.3349**	0.1420	0.5078	0.6265	0.2818
K-means N2V	0.2184	0.3270	0.1376	0.3647	0.4862	0.2630
Hierarchical DW	0.1559	0.2785	0.0823	–	–	–
Hierarchical FD2V	0.1298	0.2620	0.0577	–	–	–
Hierarchical ED2V	0.1610	0.2618	0.0696	–	–	–
Hierarchical N2V	0.1149	0.2774	0.0709	–	–	–

Table 6.3: Clustering performance compared to other methods measured by modularity. Numbers in the columns represent modularity scores of the clusterings. Column headers are the dataset names and row names are the algorithm names. Results of k-mean clustering applied to node sequence based embeddings are the best ones taken from Figure 6.3. Similarly, performance metrics of hierarchical clustering applied to node sequence based embeddings are the best ones from Figure 6.1. Baseline results were created with the listed community detection algorithms that were implemented in *IGraph* (Csardi & Nepusz, 2006). Bold numbers note the highest modularity value obtained on the dataset. Dashes denote missing modularity values when obtaining a clustering is not feasible due to complexity of the algorithm.

Earlier we established that results obtained with k-means outperform hierarchical clustering on every small dataset. This is evident if one looks at the first three columns of Table 6.3. On the BlogCatalog dataset ED2V with k-means is 38.1% better than the best sequence based embedding method combined with hierarchical clustering. On PPI the performance gap is 20.3% while on Wikipedia it is 72.5%. When we compare the results obtained with k-means clustering and sequence based embedding generation to other algorithms we see that ED2V outperforms most of the baselines except for Louvain on every dataset. The sequence based embeddings clustered with k-means outperform the Walktrap and Leading Eigenvector procedures. We also have to note that ED2V has the best performance on the PPI network but it only has a marginal advantage. In addition, in Subsection 6.2.3 we will support evidence that the performance of ED2V can be improved with careful parameter tuning of the embedding generation itself. The advantage of the Louvain method over ED2V varies between 2.3% and 13.4% on the different datasets.

These results show that the clustering of nodes in the latent space is fairly competitive with other graph clustering methods. It is also quite important that our methods allow for direct control of the cluster number. Other methods create a large number of small clusters. If our goal is the extraction of a fixed number of clusters that are somewhat interpretable, methods which result in thousands of small fragmented clusters will not be helpful.

6.2.3 Sensitivity of clustering quality to hyperparameters

In this section we investigate how the change of certain sequence generation and embedding learning parameters affects the downstream clustering performance. We only discuss the performance of our own diffusion based algorithms – specifically ED2V and FD2V. As earlier in case of multi-label classification we will use the Flickr graph to carry out the sensitivity analysis. We will consider a comparison embedding that was created with settings such that $d = 128,\ n = 10,$ $\widehat{w} = 10,\ \alpha = 0.025,\ k = 1$. We set the sequence length controlling parameters of algorithms to be $l = 40$ (ED2V) and $l = 25$ (FD2V). In each experiment we applied k-means clustering 10 times to the obtained embedding and took the clustering with the best modularity on the graph. The number of clusters was always fixed as 5. Obtained results for the sensitivity to feature vector dimension, window size, vertex set size, diffusion number, epoch number and learning rate are subplots of Figure 6.5. On each plot the manipulated parameters is on the horizontal axis and the modularity is on the vertical axis.

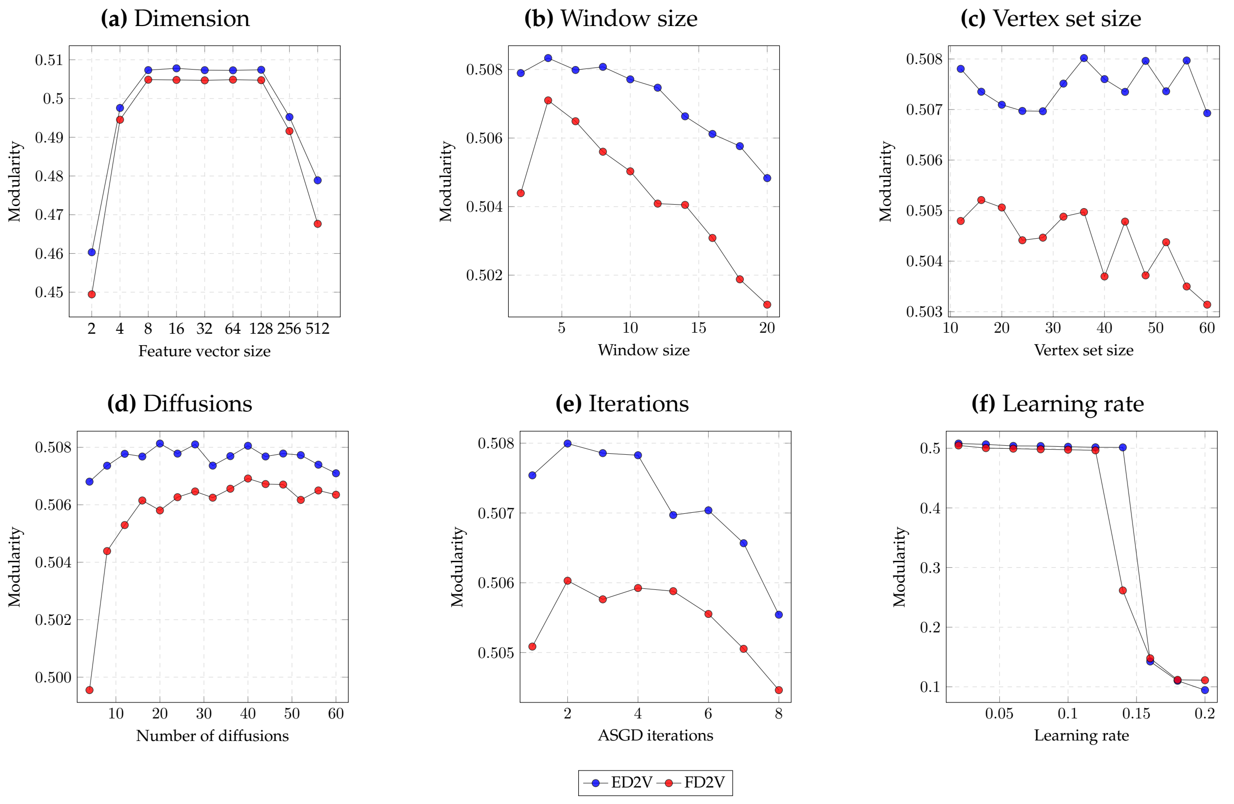


Figure 6.5: Sensitivity of k-means clustering performance to change of embedding method parameters. Embeddings were created with baseline parameter settings such that $d = 128,\ n = 10,\ \widehat{w} = 10,\ \alpha = 0.025,\ k = 1$. We set the sequence length controlling parameters of algorithms to be $l = 40$ (ED2V) and $l = 25$ (FD2V). Embedding algorithm parameters were tuned to show the sensitivity of community detection results. Manipulated parameters are on the horizontal axis and the clustering performance measured by modularity is on the vertical axis.

Performance as a function of feature vector dimension is on Subfigure 6.5a. It is quite surprising that our methods are quite robust to the change of the feature vector size – vectors with size between 8 and 128 have similar representation qualities when used for graph clustering. Similar to node classification having a high dimensional latent space weakens the performance. The effect of changing the sliding window's size is on Subfigure 6.5b. Once more, we see that a window size below 10 is beneficial – meaning that parameters used to evaluate sequence based embedding models are not optimal for clustering (at least on this dataset). Results obtained by changing the vertex set size are on Subfigure 6.5c. Our earlier results about the sensitivity of performance in multi-label classification and clustering had shown that behaviours of ED2V and FD2V are correlated. Meaning that the quality of the created representations behaves similarly when certain parameters are changed. Unusually, the quality of ED2V and FD2V representations reacts differently when we increase the number of sampled vertices. The modularity is unchanged by the vertex set size changes when we use ED2V and it drops when FD2V is applied. Changing the number of diffusions increases the performance initially as Subfigure 6.5d suggests. After a certain number of vertices is reached performances of both algorithms stagnate. About the number of asynchronous gradient descent iterations we can conclude based on Subfigure 6.5e that the optimal number of iterations is 2. The optimal number of epochs is the same as it was in case of the classification task. Finally, Subfigure 6.5f displays that increasing the learning rate results in a representation that has low value when one does clustering.

Altogether our findings about the sensitivity of performance have three important implications. First, one can obtain minor clustering performance gains by changing the parameters of the embedding methods. Reducing the window size and increasing the number of diffusions per node boosts the performance. Second, our findings suggest that having a low dimensional latent space representation and sampling less nodes than usual does not affect the clustering performance adversely. These changes can increase the computational performance, decrease memory usage and disk space consumption when we store the representation. Third, sensitivities of clustering and node classification performances are quite similar.

6.3 Edge prediction

We discuss edge prediction with sequence based embedding features in this section. Edge prediction is one of the most trivial downstream supervised learning tasks that one can do with network data. The edge predictions task is as follows: given two nodes in the graph predict whether an edge is formed between the two in the near future or not. Edge prediction has a wide range of possible applications. Trivial applications include the recommendation of new acquaintance on social networks (Backstrom & Leskovec, 2011; Dong et al., 2012), proposition of potential scientific collaborations (Liben-Nowell & Kleinberg, 2007; Clauset et al., 2017) and suggestion of pages that a webpage should link to (Chen et al., 2005). Besides these evident use cases there are other opportunities to use edge prediction. One is the improvement of recipes with ingredients that result in harmonic flavours (Ahn et al., 2011). Another one, is the detection of anomalous financial transactions among bank customers (Akoglu et al., 2015). Besides the useful possible applications it is self-evident that edge prediction requires either the collection of very special time series network data or the generation of synthetic data used for training and evaluation of the edge prediction model. Henceforth, we have an emphasis on explaining the methods that we use to generate synthetic data for edge prediction.

The synthetic data generation is discussed in Subsection 6.3.1 where we introduce a general framework for synthetic data creation for the edge prediction task. We also propose specific algorithms to help with the generation of more realistic synthetic data for this task. In Subsection 6.3.2 we show that our methods perform comparably to other node sequence based embedding methods on a range of datasets when the synthetic data is generated in a naive way. Lastly, in Section 6.3.3 we show that current accuracy benchmarks of edge prediction methods are biased upwards by the fact that synthetic data is created in an unsophisticated way. We also support evidence that our proposed algorithms outperform other methods in more realistic settings.

6.3.1 Synthetic sample generation for edge prediction

As we stated earlier, in an ideal edge prediction setting we have time series data about the network of interest. However, the collection of such data requires gradual data collection and can be a cumbersome task. This means that majority of edge prediction methods are evaluated on synthetic data. A generalized structure of the most widely used synthetic data creation procedures is summarized below.

(i) Remove β fraction of edges from the graph and create an attenuated graph.

(ii) Learn an embedding using the attenuated graph.

(iii) With a custom procedure sample $|E| \cdot \beta$ edges that are non existent in the original graph.

(iv) Create edge features based on the new set of edges and the embedding.

(v) Predict whether an edge existed in the original graph and evaluate performance.

First of all, the above described data generation procedure results in an edge set that has approximately $|E| \cdot (1+\beta)$ pairs of nodes. However, only $|E| \cdot (1-\beta)$ edges are used to create the embedding as a fraction was removed with a random edge removal strategy. Roughly $|E| \cdot 2 \cdot \beta$ edges are only used in the evaluation phase. Half of these does exist in the original network and the other half was drawn with an arbitrary sampling strategy. This stylized description of the synthetic data generation allows for arbitrarily chosen edge deletion, embedding creation and edge addition procedures. In most of the cases the edges are removed randomly ensuring that each of the nodes is connected to at least another node in the graph. Similarly edges are added in a completely random manner. The graph representation learning methods described by Perozzi et al. (2014); Grover & Leskovec (2016); Kipf & Welling (2016b); Pimentel et al. (2017) all use random edge deletion and addition to asses the performance of their models.

About the random deletions and addition we have to note that real complex networks are quite robust to non-targeted edge removals. This means that random edge removal does not influence the macro and micro level topological properties of the network significantly. These properties include the diameter, the distribution of centrality measures and the average length of shortest paths among others (Klau & Weiskircher, 2005). This also implies that the relative position of nodes in the latent space will be somewhat maintained when a graph is embedded after the random edge removal. Furthermore, when one samples random edges in the addition phase most of the node pairs will be between nodes that are distant and have a low neighbourhood overlap compared to real edges. To put it simply, these edges describe unrealistic relationships. Predicting that there is no link between such nodes is a considerably easy and somewhat irrelevant task. Therefore, we propose an edge sampling procedure which allows for controlling the neighbourhood overlap. Our model is formulated in a general way. Basically, the totally random edge addition is a corner case of our model. In the remainder we will describe the basic graph attenuation mechanism, our proposed edge sampling strategy and our edge feature generation procedure.

Graph attenuation

The randomized graph attenuation and embedding method is described by Algorithm 4. Besides the graph edge list, the parameter needed for creating the embedding one also needs β the rate of attenuation. The goal of the algorithm is to generate an embedding of the randomly attenuated graph. Before the edge deletion starts we need to create a graph object and based on the number of edges and the attenuation rate we have to set the number of edges to be deleted. We also set the counter of already deleted edges to be 0.

```
Data: Source – Path to the edge list used.
      $\beta$ – Rate of deletion.
      $k$ – Number of ASGD iterations.
      $l$ – Length of the vertex sequence/Vertex set size.
      $d$ – Feature vector size.
      $n$ – Number of diffusions per node.
      $\widehat{w}$ – Window size.
      $\alpha$ – Learning rate.
Result: X – Embedding of the attenuated graph where the rate of deletion is $\beta$.
1  $\mathcal{G} \leftarrow$ Read Graph(Source)
2  Deletion Cardinality $\leftarrow$ Ceil($\beta \cdot |E|$)
3  Deleted Counter $\leftarrow 0$
4  while Deletion Cardinality> Deleted Counter do
5      $(v, w) \leftarrow$ Random Sample$(E)$
6      $\widetilde{\mathcal{G}} \leftarrow \mathcal{G}(V, E \setminus (v, w))$
7      if $deg(v_{\widetilde{\mathcal{G}}}) > 0$ and $deg(w_{\widetilde{\mathcal{G}}}) > 0$ then
8          Deleted Counter $\leftarrow$ Deleted Counter $+ 1$
9          $\mathcal{G} \leftarrow \widetilde{\mathcal{G}}$
10     end
11 end
12 X $\leftarrow$ Create Vertex Seqiences and Learn Embedding$(\mathcal{G}, k, l, d, n, \widehat{w}, \alpha)$
```

Algorithm 4: Graph attenuation and embedding algorithm

After these initialization steps we start an iterative process that is halted by reaching the target number of deleted edges. We sample a random pair of nodes from the edge set. We create a graph where this edge is removed. If the degree of the edge endpoint nodes is above zero in the residual graph we increase the deleted counter. In addition, the temporarily created graph replaces the graph object if the condition on the degrees is satisfied. Based on the attenuated graph and the embedding parameters one can learn an embedding of the new graph with an arbitrary embedding method. Later the features of nodes can be used the generate edge features for the edge prediction task.

Edge sampling with potential neighbourhood overlap

Our proposed edge sampling strategy with neighbourhood overlap is described with pseudo-code by Algorithm 5. The algorithm needs a path to the graph edge list. Besides this it needs three parameters. One parameter is β the rate of addition which controls the number of edges being added to the edge set. The second one is γ which sets the minimal neighbourhood overlap that nodes at the end of an edge must have. Finally, the third parameter is η which controls the minimal degree that nodes at the end of the potential edge must have. Before creating the augmented edge set we read the graph. Using the addition rate β and the number of edges in the edge set we decide the number of edges to be added. Finally, we set the the newly added edge number to be 0 and the augmented edge set $\widetilde{E}$ to be the edge set itself.

```
Data: Source of graph – Path to the edge list used.
      $\beta$ – Rate of addition.
      $\gamma$ – Minimal neighbourhood overlap.
      $\eta$ – Minimal degree.
Result: $\widetilde{E}$ – Edges of the augmented graph where the rate of addition is $\beta$.
1  $\mathcal{G} \leftarrow$ Read Graph(Source of graph)
2  Addition Cardinality $\leftarrow$ Ceil($\beta \cdot |E|$)
3  Addition Counter $\leftarrow 0$
4  $\widetilde{E} \leftarrow E$
5  while Addition Cardinality > Addition Counter do
6      $v \leftarrow$ Random Sample($V$)
7      $w \leftarrow$ Random Sample($V$)
8      if $v \neq w$ and $(v, w) \notin \widetilde{E}$ then
9          Neighbourhood Overlap $\leftarrow \frac{N_G(v) \cap N_G(w)}{N_G(v) \cup N_G(w)}$
10         if Neighbourhood Overlap $\geq \gamma$ and $\deg(v) \geq \eta$ and $\deg(w) \geq \eta$ then
11             Addition Counter $\leftarrow$ Addition Counter $+ 1$
12             $\widetilde{E} \leftarrow \widetilde{E} \cup (v, w)$
13         end
14     end
15 end
16 Dump Edges($\widetilde{E}$)
```

Algorithm 5: Edge sampling algorithm with neighbourhood overlap constraint

Following these initialization steps we start an iterative process in which we add the new edges to the augmented edge set. This process is stopped when the required number of edges is reached. During the iteration first we sample the nodes v and w. If the two nodes are different and they are not in the augmented edge set we calculate the neighbourhood overlap. Using the neighbourhood overlap of v and w and their respective degrees we decide whether they should be added to $\widetilde{E}$. If their neighbourhood overlap is greater than γ and their degrees exceed η we add them to the augmented edge set. We also increment the counter of newly added edges. After the iteration halted we save the augmented edge list. It is worth noting that having γ and η set as 0 means that we sample edges randomly.

Edge feature generation

Using the augmented graph edge list and the embedding of the attenuated networks we can generate the edge features themselves. The edge feature and outcome variable generation mechanism that we use is described by Algorithm 6. First we read the original and the augmented edge lists – these are the sets E and $\widetilde{E}$ respectively. We also obtain the embedding created with the graph embedding procedure and extract d the number of columns it has. We create the matrix $\widetilde{\mathbf{X}}$ for the edge features and the vector $\widetilde{\mathbf{y}}$ describing the outcome variable. The edge feature matrix has $|\widetilde{E}|$ rows and d columns while the binary outcome vector has $|\widetilde{E}|$ rows. Before creating the edge features we set the augmented edge counter e to be 0.

```
Data: Source of graph – Path to the origimal edge list.
      Source of augmented graph – Path to the augmented edge list.
      Source of attenuated embedding – Path to the attenuated embedding.
Result: $E$ – Edges of the augmented graph where the rate of addition is $\beta$.
1  $E \leftarrow$ Read Edge List(Source of graph)
2  $\widetilde{E} \leftarrow$ Read Edge List(Source of augmented graph)
3  $\mathbf{X} \leftarrow$ Read Embedding(Source of attenuated embedding)
4  $d \leftarrow$ Get Number of Columns($\mathbf{X}$)
5  $\widetilde{\mathbf{X}} \leftarrow M_{|\widetilde{E}|\times d}$
6  $\widetilde{\mathbf{y}} \leftarrow M_{|\widetilde{E}|\times 1}$
7  $e \leftarrow 0$
8  for $(v, w)$ in $\widetilde{E}$ do
9      e $\leftarrow$ e $+ 1$
10     $\widetilde{\mathbf{X}}_e \leftarrow$ Calculate Operator($\mathbf{X}_v, \mathbf{X}_w$)
11     if $(v, w) \in E$ then
12         $\widetilde{\mathbf{y}}_e \leftarrow 1$
13     end
14     else
15         $\widetilde{\mathbf{y}}_e \leftarrow 0$
16     end
17 end
```

Algorithm 6: Edge prediction dataset generator algorithm

We iterate through the node pairs in the augmented edge set and increment the augmented edge counter. Using the edge endpoint nodes we take the corresponding rows of from the embedding and using an arbitrary operator we calculate an edge feature vector. This vector is a row vector with d columns and it can be created by applying an elementwise operator on the end node specific vectors. In our experiments we used the ℓ_1,ℓ_2, average and Hadamard operators. The resulting feature vector is the e^{th} row of the edge feature matrix. If the pair of nodes was in E we set the e^{th} element of the outcome variable to be 1 otherwise it is going to be 0. When the procedure is finished we have edge features and an outcome variable which we can use to evaluate the representation quality of the embedding with regards to edge prediction. We are also able to test the sensitivity of edge prediction to increasing the neighbourhood overlap. Increasing the neighbourhood overlap will result in adversarial examples of non-existent edges that are fairly hard to classify.

6.3.2 Edge prediction without overlap constraint

First, we perform edge prediction in a setup where there is no constraint on the neighbourhood overlap. This means that we will set γ and η to be 0. The datasets used for evaluating the edge prediction performance are datasets widely used for benchmarking node sequence embedding algorithms. These networks are the following:

(i) **Arxiv:** Is a collaboration network of scientists who do work related to astrophysics. Nodes represent researchers and edges are papers written by two researchers (Leskovec et al., 2007).

(ii) **Facebook:** Is a quite dense subsample of the Facebook network. The nodes are users of the site and edges are friendships (Leskovec & Mcauley, 2012).

(iii) **Citeseer:** Is a citation network created in a way that the whole graph is connected. Nodes are papers while the edges are citations (Lu & Getoor, 2003; Sen et al., 2008).

(iv) **Cora:** Is a citation network similar to Citeseer (Lu & Getoor, 2003; Sen et al., 2008).

(v) **Pubmed:** Is also a citation network, but it describes fairly niche scientific works. It only contains papers that investigate diabetes (Namata et al., 2012).

	ARXIV	FACEBOOK	CITESEER	CORA	PUBMED
$\|\mathbf{V}\|$	18,772	4,039	3,327	2,708	19,717
$\|\mathbf{E}\|$	198,110	88,234	9,464	5,429	88,676

Table 6.4: Size of the graphs used to asses link prediction. The number of vertices and edges are respectively in the first and second row of the table. We included each of the graphs used for benchmarking the quality of representations for link prediction.

The number of vertices and edges for the benchmark networks are listed in Table 6.4. Generally the graph that we use to evaluate the edge prediction performance are smaller than the ones used to asses the clustering and classification. We also want to emphasize that the Citeseer, Cora and Pubmed graphs are less dense than the Facebook and Arxiv networks. The embeddings were generated after 50% of edges was removed from the original set of edges. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as the training set. Where it was possible (Facebook and Arxiv datasets) we took the optimal parameters from Grover & Leskovec (2016). The augmented edge set contains an additional 50% of randomly sampled edges. We used the ℓ_1, ℓ_2, average and Hadamard operators to generate edge features. Finally, we trained our edge prediction models on half of the synthetic data and evaluated it on the remainder. Our results represent the average area under the curve value based on 10 random seeded 50%-50% train-test set splits.

	Algorithm	ArXiv	Facebook	Citeseer	Cora	Pubmed
Average	ED2V	0.9901	0.9887	0.9747	0.9779	0.9831
	DW	0.9881	0.9880	0.9613	0.9733	0.9934
	FD2V	**0.9913**	0.9899	0.9513	0.9761	0.9851
	N2V	0.9911	0.9895	0.9751	0.9794	0.9945
Hadamard	ED2V	0.9898	**0.9917**	0.9855	0.9855	**0.9952**
	DW	0.9834	0.9901	0.9872	0.9869	0.9931
	FD2V	0.9875	0.9904	0.9712	0.9856	0.9922
	N2V	0.9896	0.9910	**0.9898**	**0.9883**	0.9932
L_1	ED2V	0.9737	0.9876	0.9707	0.9855	0.9931
	DW	0.8951	0.9839	0.9779	0.9865	0.9870
	FD2V	0.9328	0.9845	0.9042	0.9776	0.9922
	N2V	0.9787	0.9880	0.9823	0.9877	0.9892
L_2	ED2V	0.9737	0.9876	0.9707	0.9855	0.9931
	DW	0.8989	0.9838	0.9779	0.9865	0.9870
	FD2V	0.9310	0.9845	0.9044	0.9776	0.9922
	N2V	0.9782	0.9880	0.9823	0.9877	0.9892

Table 6.5: Edge prediction performance of gradient boosted classification trees using features extracted with the sequence based graph embedding methods. The classifier was fitted with tree depth equal to 3, a learning rate of 0.1 and the number of trees was chosen based on early stopping and 5-fold cross-validation within the training set. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 50% of edges was attenuated before embedding. Later 50% of randomly chosen edges was added to the edge set without a neighbourhood overlap limit. Edge features were generated by applying the ℓ_1, ℓ_2, Hadamard and average operators elementwise on the vectors describing the edge endpoint nodes. The table reports performance measured by AUC on the test set. Each value is the average AUC based on 10 random seeded 50%-50% train-test splits. Bold numbers denote the best performing model-operator combination on a given dataset.

We used two different classifiers to demonstrate that choosing a non-linear model can improve the edge classification performance. The chosen models and their settings are below.

- **Logistic regression:** We used ℓ_1 regularization and the shrinkage parameter was set as $\lambda = 1$. This setting is the same as the one used by Perozzi et al. (2014) and Grover & Leskovec (2016).
- **Gradient boosted classification trees:** This classifier was fitted with tree depth equal to 3, a learning rate of 0.1 and the number of trees was chosen based on early stopping and 5-fold cross-validation within the training set. We used gradient boosted tree implementation described by Chen & Guestrin (2016).

We enclosed results obtained with gradient boosted classification trees in the main body of the paper as Figure 6.5. Performance evaluation of logistic regression is in Section D.2 of Appendix D as Figure D.1. The edge prediction performance obtained with boosted trees is superior to the logistic regression performance. However, it should be highlighted that when logistic regression is used the random walk based embedding methods outperform the diffusion based ones on most of the datasets. Looking at the average and Hadamard operators it is fairly easy to see that performance of different embeddings is quite similar and performance differences are marginal. On the three most dense datasets ED2V outperforms other methods while on the two less dense ones N2V has the best performance. We also see that the Hadamard operator applied to the extracted edges has a fairly good performance across datasets. In case of logistic regression the performance gap between operators is quite considerable, but using boosted trees the difference is less sharp. About these results that seem to favour our model on the standard benchmarks we have to note that they were obtained on a highly unrealistic task. Most of the synthetically sampled edges are between nodes that are far from each other on the graph.

6.3.3 Edge prediction with overlap constraint

Next we are going to test how the presence of a neighbourhood overlap constraint effects the edge prediction performance. We will manipulate the required minimal neighbourhood overlap and evaluate predictive performance on the Citeseer, Cora and Pubmed datasets. A required overlap ensures that unconnected nodes are good potential candidates for being connected in the future and our classification task is not extremely primitive. We compare our models to random walk based algorithms and show that FD2V has good results in this setting.

Embeddings were created with the standard parameter settings of the algorithms used earlier. When we generated the synthetic data 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were respectively generated by applying the ℓ_1, ℓ_2, Hadamard and average operators elementwise on the vectors describing the edge endpoint nodes. We used logistic regression and gradient boosted classification trees to predict the existence of edges. The hyper parameter settings of the classifiers were taken from the previous section. Results regarding the ℓ_1 and average operators obtained with gradient boosted trees are on the subfigures of Figure 6.6. The neighbourhood overlap is on the horizontal axis and the classification performance measured by AUC on the test set is on the vertical axis. Each point represents the average AUC based on 10 random 50%-50% train-test set splits.

Looking at results on Subfigures 6.6a, 6.6b and 6.6c obtained with the ℓ_1 operator we see that having a neighbourhood overlap constraint reduces the performance. Moreover, as the required minimal neighbourhood overlap value is increased the classification accuracy drops. For most of the neighbourhood overlap values FD2V has a performance that is superior or equal to the other methods on the Citeseer and Pubmed datasets. This is quite intriguing considering the fact that it had a poor performance on the node classification and clustering tasks. Our measurements with edge features extracted with the average operator are on Subfigures 6.6d, 6.6e and

6.6f of Figure 6.6. Surprisingly the prediction accuracy is not always deteriorated by an increasing neighbourhood overlap when one uses the average operator. On the Citeseer and Pubmed datasets the performance increases with a higher neighbourhood overlap. Furthermore, FD2V is again able to outperform on these datasets the other embedding methods. On the Cora dataset FD2V also has a good performance, but at the same time the accuracy of all of the sequence based embedding methods decreases if the constraint is more stringent. Altogether we observe that performance is not neutral to setting a neighbourhood overlap limit.

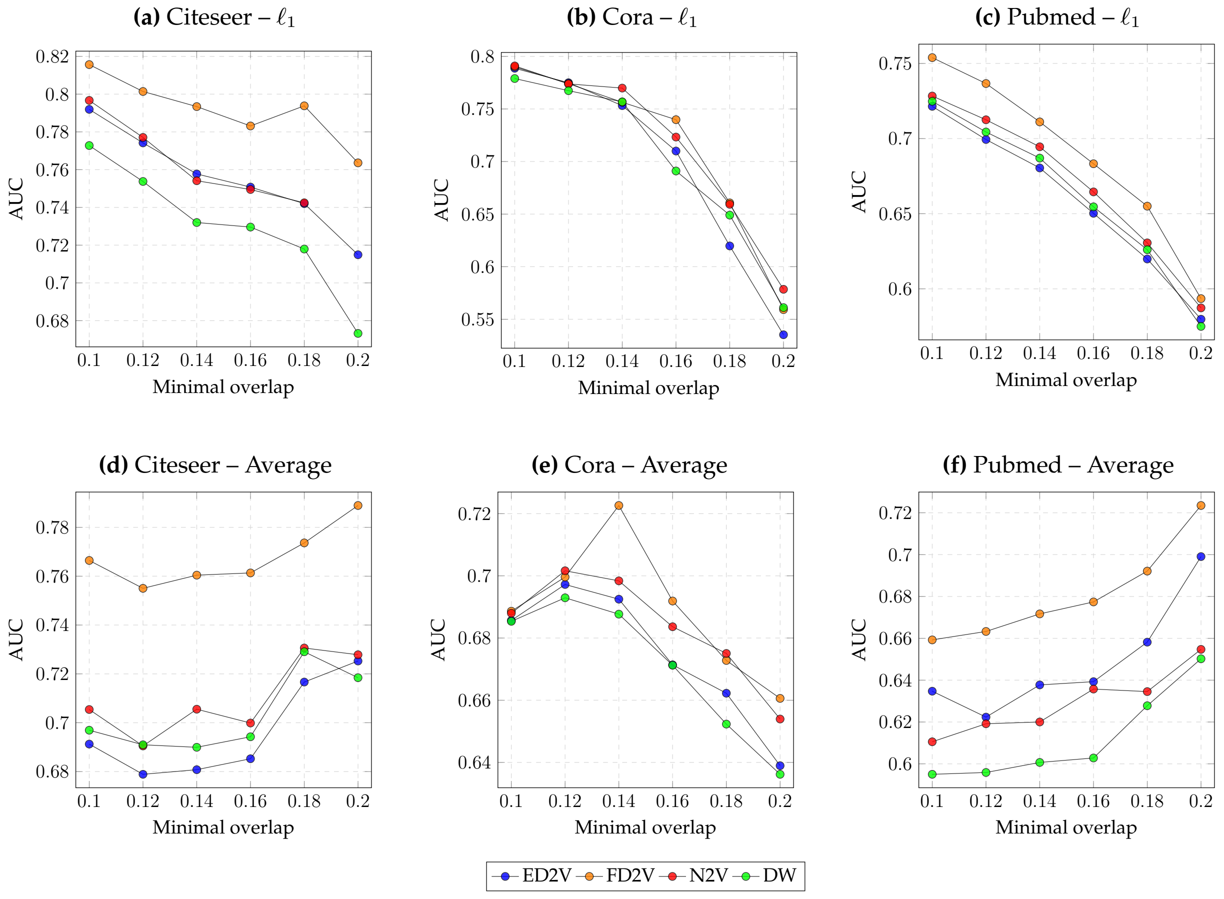


Figure 6.6: Edge prediction performance of gradient boosted classification trees using features extracted with the sequence based graph embedding methods. The classifier was fitted with tree depth equal to 3, a learning rate of 0.1 and the number of trees was chosen based on early stopping and 5-fold cross-validation within the training set. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were respectively generated by applying the ℓ_1 and average operators elementwise on the vectors describing the edge endpoint nodes. The neighbourhood overlap is on the horizontal axis and classification performance measured by AUC on the test set is on the vertical. Each point represents the average AUC based on 10 random train-test splits. The minimal degree parameter η was set as 2.

Additional supporting results obtained on the task of edge prediction with overlap constraint applying gradient boosted classification trees are enclosed in Section D.1 of Appendix D as Figure D.1. Results obtained with logistic regression are also in Appendix D. In Section D.2 we enclosed the experimental findings as Figures D.2, D.3, D.4 and D.5. The footnotes of the figures contain detailed description of the experimental settings.

Chapter 7

Conclusions

In this concluding chapter we summarize the main achievements and highlights of our work in Section 7.1. We discuss the limitations of our work and point out possible broadening of our investigation in Section 7.2.

7.1 Main findings

In this work we proposed ED2V and FD2V two node sequence based graph embedding models that use diffusion processes on graphs to create vertex sequences. We implemented these methods in Python and demonstrated that the design of these algorithms results in fast sequence creation in realistic settings and allows parallel vertex sequence generation which leads to additional speed up. We supported evidence that the computational performance of our method is robust to graph densification and growth.We highlighted that embedding vectors are useful for graph visualization. We established that vertex centralities and graph distances among nodes are conserved in the embedding space that we create. We confirmed that node features created with the ED2V and FD2V are useful explanatory variables for downstream machine learning tasks. We gave a detailed evaluation of the representation quality of embeddings on the machine learning tasks of multi-label node classification, community detection and edge prediction. Our findings reinforced that besides the favourable computational performance the representation quality itself is competitive with other methods. We conclude that our work is an important contribution towards solving large scale network analysis problems.

7.2 Limitations and possible future work

We discuss the limitations and shortcomings of our work in Subsection 7.2.1. We overview possible future extensions of the project in Subsection 7.2.2 where we list potential theoretical and empirical contributions.

7.2.1 Limitations

In the following we will discuss certain considerable limitations of the work that we carried out. Some of them is fairly theoretical, others are about our experiments and the data we used. We also mention some technical limitations of our creation. Specifically these are:

- **Theoretical limitations:** Currently the theory of node sequence based graph embedding algorithms is more ore less absent. Earlier claims about why these methods are so effective are not well grounded in terms of probability and graph theory. We only extended the theory with procedures that are more suitable for vertex sequence generation in our opinion.
- **Exclusion of vertex features:** The node sequence based graph embedding methods that we introduced exclude node features. Vertex specific data potentially helps to improve the learnt representation quality as it is known that a number of networks show strong autocorrelation of generic vertex features. We have to note that using only the network might be still beneficial when we have missing values. For details on this specific topic see the results of Yang et al. (2016); Defferrard et al. (2016) and Kipf & Welling (2016b).
- **Graphs with edge weights:** Random walk based graph embedding methods allow for edge weighted graphs. However, our methods are limited to graphs that have no edge weights. If weights represent link strength we exclude important information and our learnt representation is biased by weighting links equally.
- **Lack of high performance implementation:** We only created a reference implementation of our sequence generation methods. While these reference versions of the diffusion based methods are competitive with other reference implementations it is evident that in their current form they have limited value for possible industrial applications.
- **Real vertex sequence based benchmarks**: Our methods generate synthetic sequences and create graph representations based on these sequences. Sequence based embeddings created with real node sequences would be important reference baselines in downstream machine learning tasks.
- **Limitations of our experiments:**
  - When we considered the multi-label node classification we only used logistic regression to evaluate the representation quality as it is used in other related works (Perozzi et al., 2014; Tang et al., 2015; Grover & Leskovec, 2016; Pimentel et al., 2017). However, using non-linear models might show that the representation quality is competitive with more advanced node classification methods. Our early experiments with k-nearest neighbours, neural networks and tree based ensembles in this regard were negative.
  - On certain graphs poor initializations of k-means clustering resulted in inferior cluster quality. Finding a proper initialization method would be beneficial for fair comparison of sequence based embedding procedures.
  - We only used synthetic data for assessing the representation quality regarding the edge prediction. Using temporal network data would help with benchmarking the edge prediction performance in a realistic setting.

  - We did not perform regression – evaluating the value of representations for performing this task would be essential. However, it would require a number of networks with various sizes and types that have continuous node features.

7.2.2 Possible future work

Based on the above listed limitations of our work and findings presented earlier we see that there is a wide range of possible extensions. We enumerate a number of potentially interesting areas where additional research can be done. Some of these require addressing the above listed theoretical issues others require additional experiments or some programming.

- **General theory of sequence based graph embedding methods:** There is need for a better understanding why the sequence based graph embedding methods work. Our experiments about the basic properties highlighted that a number of distances and centralities are maintained in the embedding space. It is fair to assume that there are theoretical guarantees in this regard.
- **Integration of vertex features**: As we emphasized using generic vertex features could improve the representation quality. Concatenating these features to the hitting frequency vector would allow for the inclusion of generic vertex features. For random walk based procedures Yang et al. (2016) already made possible the use of generic vertex features.
- **Generalizations:** Currently there are certain generalizations of these methods. For example there is an algorithm that specifically works with signed graphs (Yuan et al., 2017; Goyal & Ferrara, 2017). However, there is no algorithm that can deal with multiplex graphs where nodes can have multiple types of edges.
- **High performance implementation:** Creating a C++ version of our methods with the SNAP network analysis package would foster possible industry application of our methods (Leskovec & Sosič, 2016). Using an implementation language different from Python would also allow for better parallelization of vertex sequence generation.
- **Additional time benchmarks:** Our experiments about computational performance only considered graphs with thousands of nodes. Scaling up the investigation to larger graphs would give additional evidence that our method has a considerable advantage. Also one could investigate how change of topological properties affect performance gains.
- **Downstream machine learning tasks:** The evaluation of the representation quality can be extended by applying regression, other classifiers or other clustering algorithms. Our assessment was limited to demonstrate that our embedding method is competitive with other sequence based methods.
- **Edge prediction in a realistic setting:** With proper temporal network data the assessment of the representation quality for edge prediction would be an interesting series of experiments. Collaboration networks or data extracted from publicly available API's of social networks[1] could support data for these investigations.

[1]For example the Deezer API practically allows for unlimited social network data extraction.

- **Diffusion sampling for graph convolutions:** Graph convolutional neural networks have problems when the size of the graph is large. Patch sampling small subgraphs with diffusion trees might help to scale up these models.

Appendix A

Synthetic graph embedding benchmarks

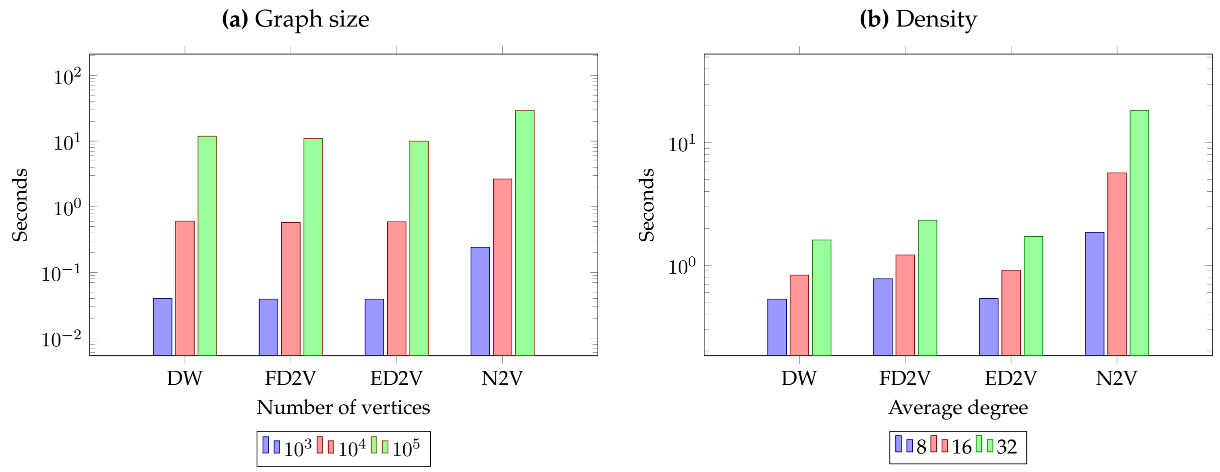


Figure A.1: Erdős-Rényi graph – mean graph pre-processing time. Columns report mean sequence generation times based on 100 replications on a Erdős-Rényi graph. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed as 10. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4.

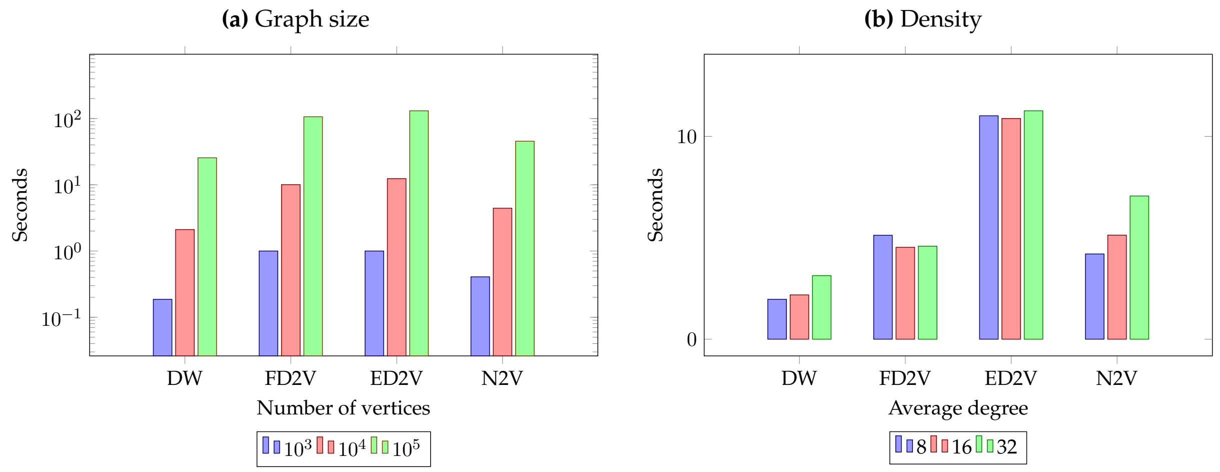


Figure A.2: Erdős-Rényi graph – mean sequence generation time. Columns report mean sequence generation times based on 100 replications on a Erdős-Rényi graph. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed as 10. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4.

	$\overline{\textbf{Degree}}$	**DW**	**N2V**	**ED2V**	**FD2V**
	8	0.8391	9.734	0.8237	0.8765
Barabási	16	1.563	33.1155	1.4983	1.5457
	32	3.0059	113.9357	2.8785	2.9591
	8	0.5294	1.8592	0.5354	0.7751
Erdős	16	0.8316	5.6721	0.9123	1.2132
	32	1.6084	18.2822	1.7174	2.3341
	8	0.4392	1.576	0.474	0.5072
Watts	16	0.6677	4.6116	0.6849	0.7907
	32	1.2132	14.3283	1.3359	1.3901

	|V|	**DW**	**N2V**	**ED2V**	**FD2V**
	10^3	0.0707	1.0283	0.0756	0.0747
Barabási	10^4	0.9994	15.1209	0.9797	0.9909
	10^5	18.1231	184.9882	16.3046	17.0141
	10^3	0.0399	0.2419	0.0392	0.0392
Erdős	10^4	0.6058	2.65	0.5891	0.5784
	10^5	11.8811	29.1177	9.9965	10.9557
	10^3	0.038	0.2032	0.0566	0.0375
Watts	10^4	0.5086	2.1823	0.6219	0.4937
	10^5	6.8146	22.0627	6.9634	1.2785

Table A.1: Mean graph prep-processing time. Columns report mean graph pre-processing times based on 100 experimental replications on Barabási-Albert, Erdős-Rényi and Watts-Strogatz graphs. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed as 10.

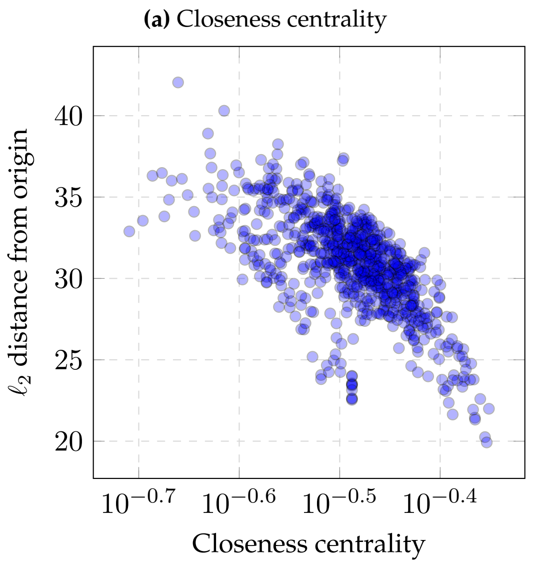
	$\overline{\text{Degree}}$	DW	N2V	ED2V	FD2V
Barabási	8	2.9059	7.4983	10.7822	3.128
	16	3.7542	11.2028	11.0561	3.2991
	32	4.9607	18.0961	11.7711	3.5287
Erdős	8	1.9647	4.2017	11.0151	5.1236
	16	2.183	5.1308	10.8804	4.5305
	32	3.1349	7.0603	11.2604	4.586
Watts	8	1.7035	3.4277	16.0224	10.1974
	16	1.9159	3.833	12.7878	5.4239
	32	2.5659	4.424	11.5099	4.3079

	$\lvert V \rvert$	DW	N2V	ED2V	FD2V
Barabási	10^3	0.24	0.6099	1.0602	0.2917
	10^4	3.1206	8.6544	10.9499	3.0832
	10^5	44.1338	111.0924	111.3172	35.1909
Erdős	10^3	0.1864	0.4076	0.9999	0.9999
	10^4	2.107	4.4284	12.3701	10.0363
	10^5	25.5295	45.4164	130.7681	106.2965
Watts	10^3	0.1738	0.3245	1.7858	1.3506
	10^4	1.8293	3.465	17.9917	13.7024
	10^5	21.2494	35.8815	171.218	130.4653

Table A.2: Mean sequence generation time. Columns report mean sequence generation times based on 100 experimental replications on Barabási-Albert, Erdős-Rényi and Watts-Strogatz graphs. In case of the average degree experiments the degree was set as 8, 16 and 32 while number of vertices was 10^4. For the graph size benchmarks the number of vertices was set at 10^3, 10^4 and 10^5 while the average degree was fixed as 10.

Appendix B

Basic properties of the embeddings



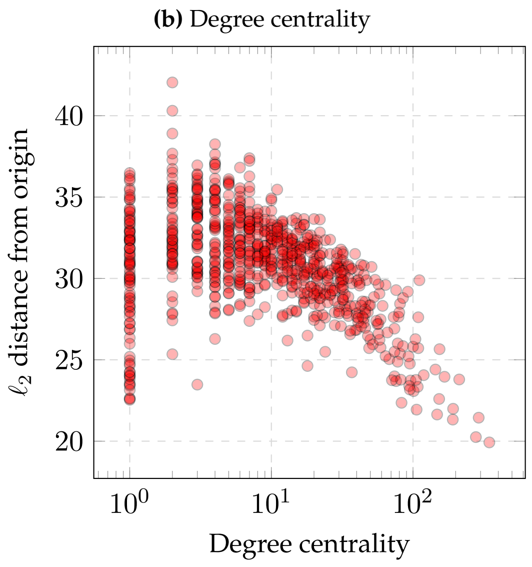


Figure B.1: Association between the distance of vertices from the origin in latent space and node centrality measures. The dataset we used to create the plots is the PPI network. The embedding itself was generated by FD2V with settings $l = 40$, $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. Each scatter plot was made with a subsample of 400 randomly selected nodes. The horizontal axes measure centrality on a log scale while the verticals measure $\ell_2$distance from the origin.

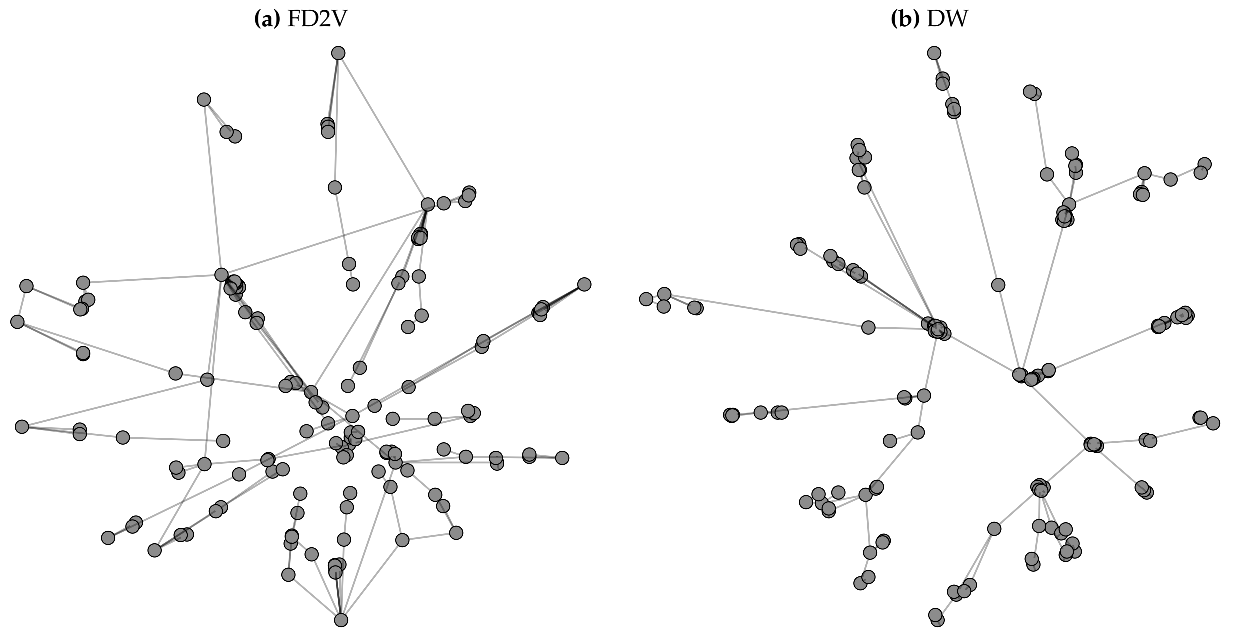


Figure B.2: Visualizing a Barabási-Albert graph based on FD2V and DW embeddings. The FD2V embedding was created with settings such that $l = 25,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Similarly, the DW embedding was created with parameter settings of $l = 80,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Feature vectors were normalized and the 1^{st} and 3^{rd} feature vector was used to create visualizations. The graph itself is a Barabási-Albert tree with 150 nodes.

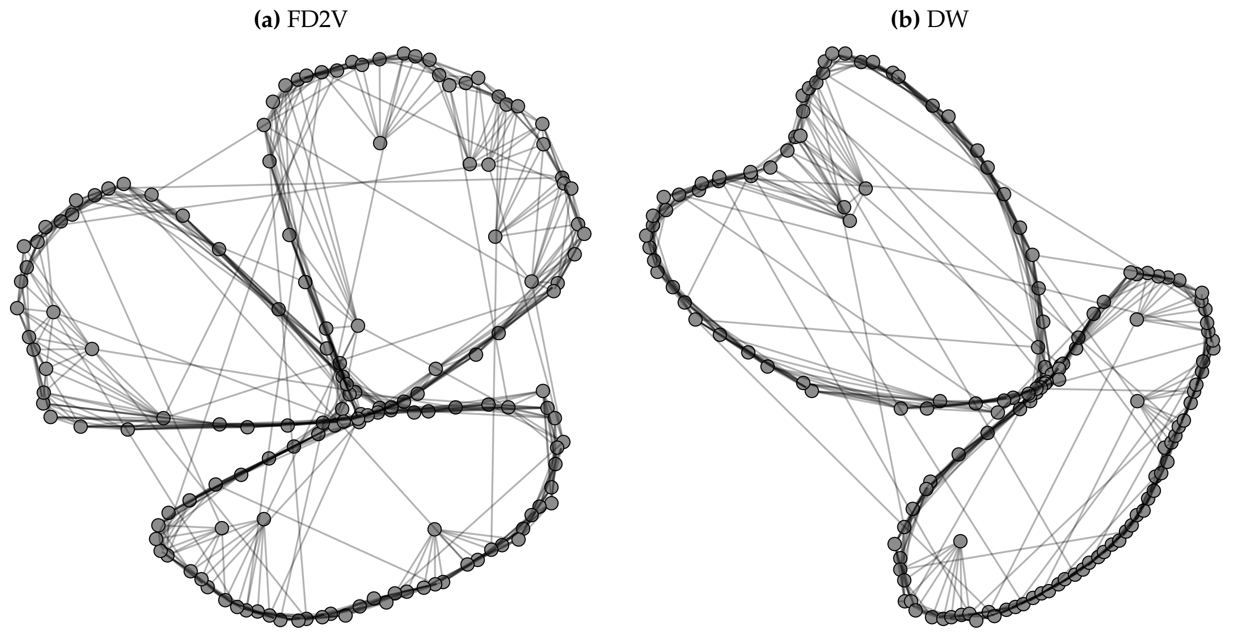


Figure B.3: Visualizing a Watts-Strogatz graph based on FD2V and DW embeddings. The FD2V embedding was created with settings such that $l = 25,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Similarly, the DW embedding was created with parameter settings of $l = 80,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Feature vectors were normalized and the 1^{st} and 3^{rd} feature vector was used to create visualizations. The graph is a Watts-Strogatz graph with 150 nodes, 10 neighbours per node and rewiring probability of 0.03.

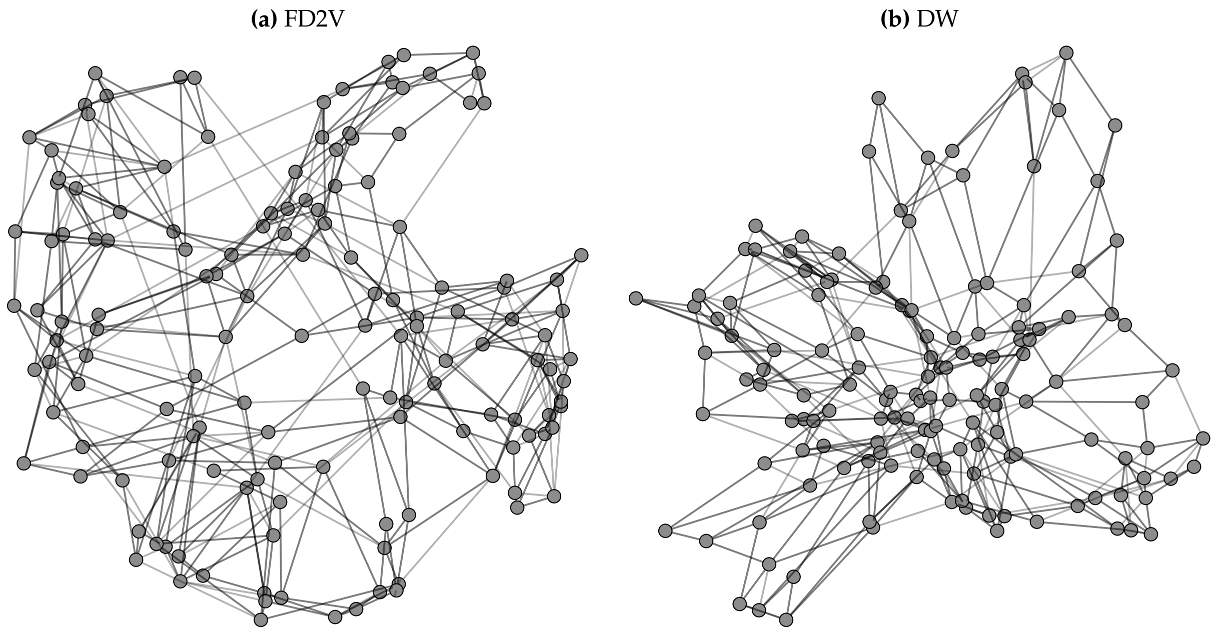


Figure B.4: Visualizing a Kleinberg Navigable Small-World graph based on FD2V and DW embeddings. The FD2V embedding was created with settings such that $l = 25,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Similarly, the DW embedding was created with parameter settings of $l = 80,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Feature vectors were normalized and the 1^{st} and 3^{rd} feature vector was used to create the visualizations. The graph is a Kleinberg navigable small-world graph with 144 nodes, 1^{st} order neighbours, 1 random edge per node and distance exponent of 2.

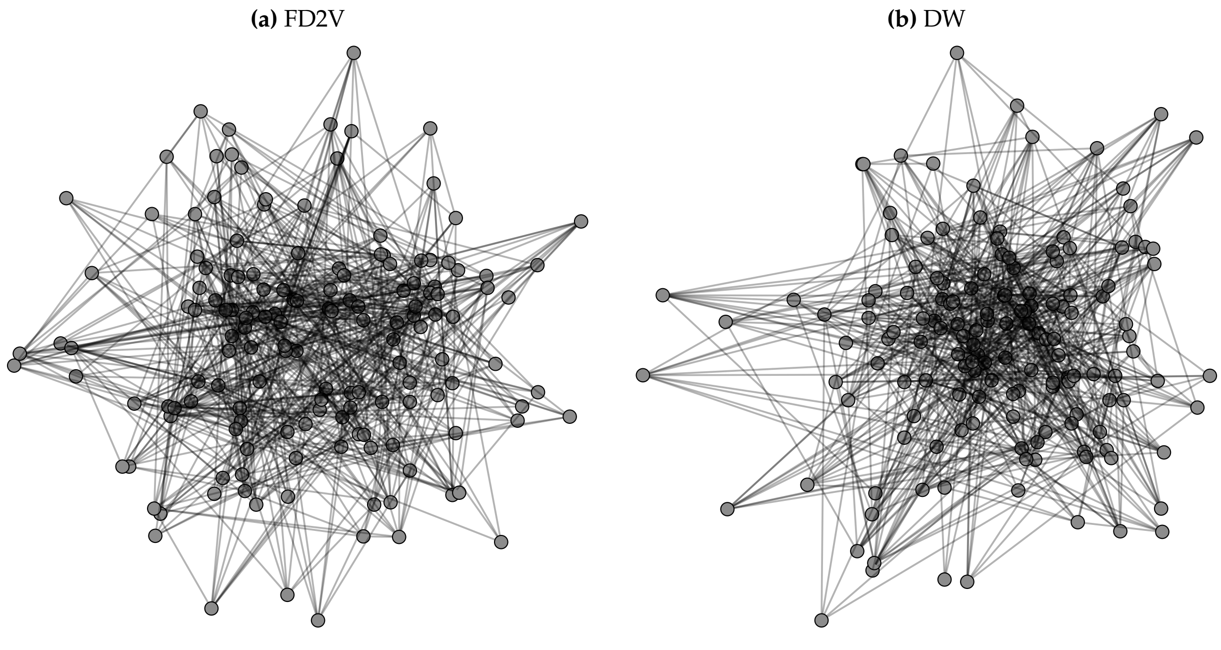


Figure B.5: Visualizing an Erdős-Rényi graph based on FD2V and DW embedding. The FD2V embedding was created with settings such that $l = 25,\ d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Similarly, the DW embedding was created with parameter settings of $l = 80$, $d = 3,\ n = 100,\ \widehat{w} = 10,\ \alpha = 0.025$. Feature vectors were normalized and the 1^{st} and 3^{rd} feature vector was used to create the visualizations. The graph is an Erdős-Rényi graph with 150 nodes and random connection rate of 0.05.

Appendix C

Multi-label classification

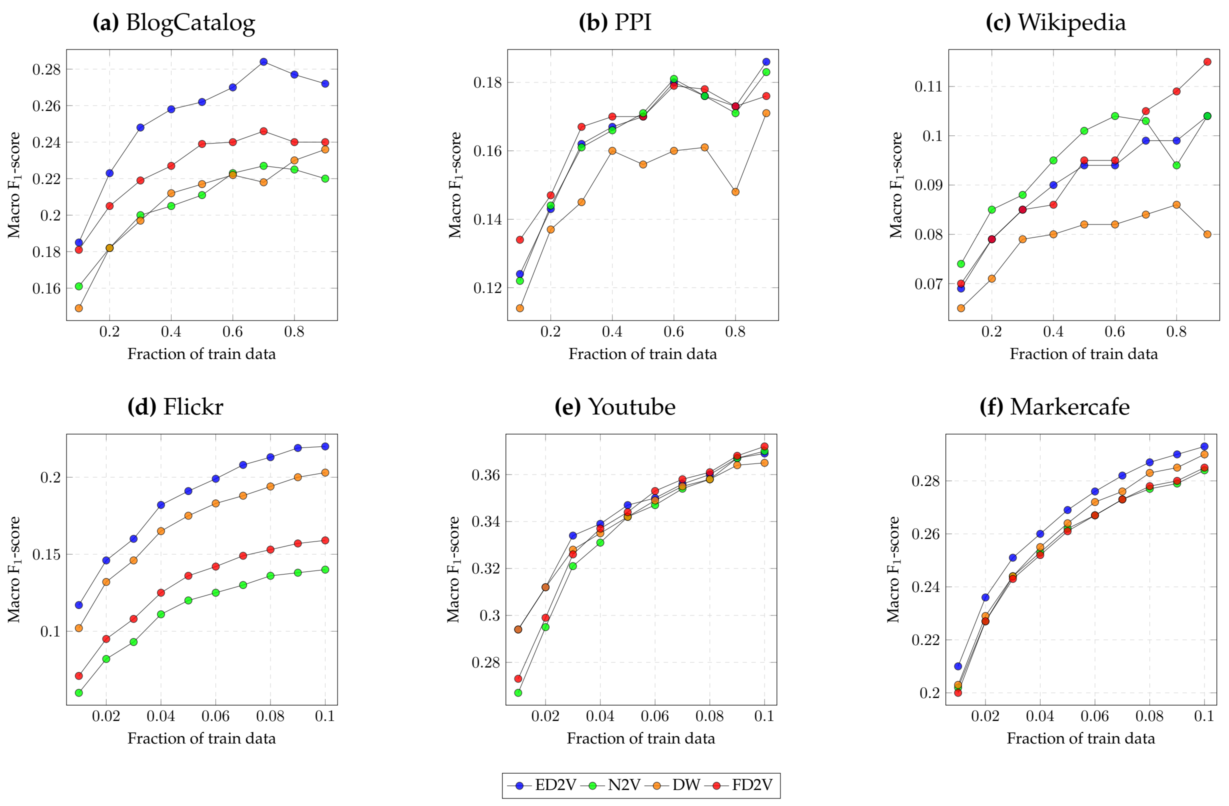


Figure C.1: Multi-label classification performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization and the regularization parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 10% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. The fraction of training data is on the horizontal axis and the classification performance measured by macro F-1 score on the test set is on the vertical axis. Each point represents the average macro F-1 score based on 10 random train-test splits. The DW and N2V embeddings for the Flickr, Youtube and Markercafe datasets were generated with the high performance C++ implementation.

	BlogCatalog		Flickr		YouTube	
% Labeled nodes	**40%**	**80%**	**4%**	**8%**	**4%**	**8%**
ED2V	**0.2583**	**0.2771**	**0.1824**	**0.2137**	0.3396	0.3605
FD2V	0.2123	0.2318	0.1759	0.1942	0.3355	0.3580
N2V	0.2274	0.2401	0.1257	0.1538	**0.3371**	**0.3618**
DW	0.2056	0.2256	0.1118	0.1362	0.3312	0.3582
EdgeCluster	0.2200	0.2461	0.1672	0.2018	0.2917	0.3123
Modularity	0.2185	0.2420	0.1511	0.1710	–	–
wvRN	0.1424	0.1886	0.0347	0.0659	0.2090	0.2648
Majority	0.0258	0.0248	0.0046	0.0047	0.0610	0.0616

Table C.1: Classification performance compared to other feature generation methods measured by macro F-1. Numbers in the columns represent micro F-1 test scores on the training dataset. Column headers are the dataset names and subheaders are the training dataset sizes. The results of node sequence based embedding methods are the same as the ones in Figure C.1. Baseline results were taken from the work of Perozzi et al. (2014). Bold numbers denote the best performing method.

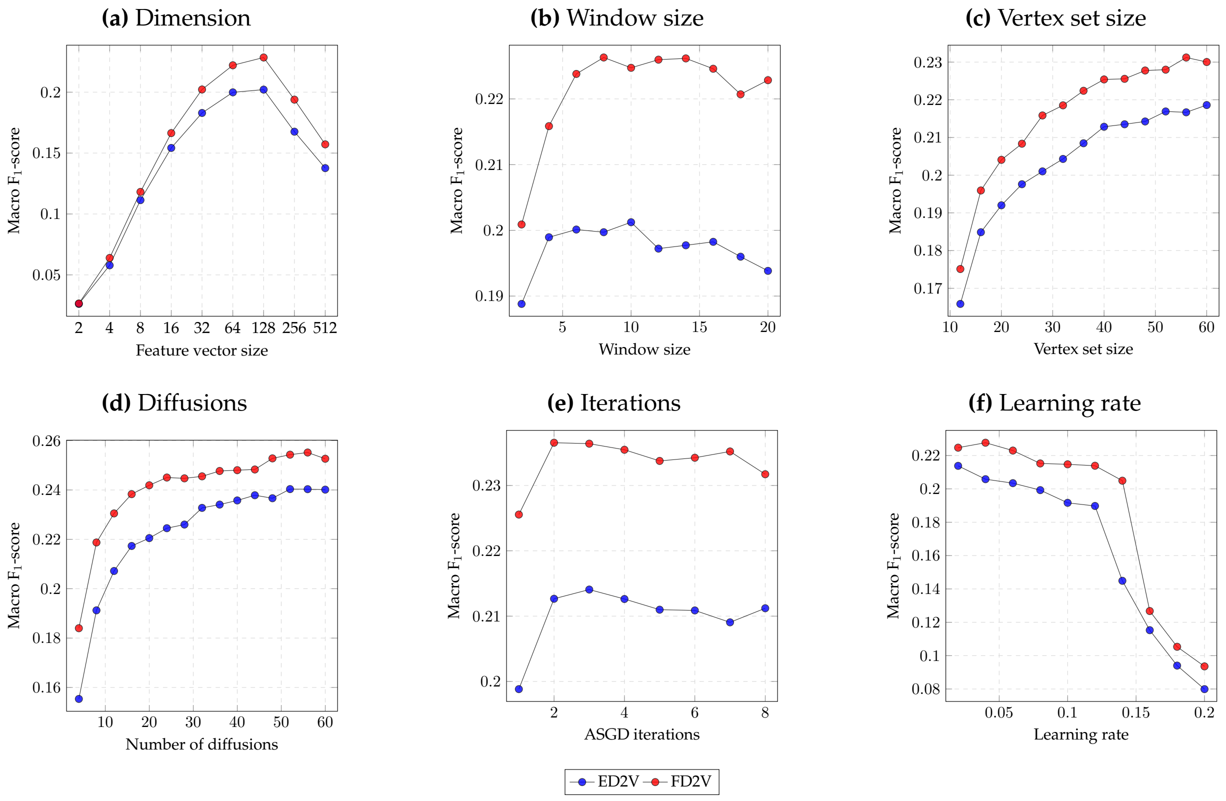


Figure C.2: Sensitivity of multi-label classification performance of logistic regression using features extracted with the D2V variants to change of parameters. The classifier was fitted with ℓ_1 regularization and the regularization parameter was set as $\lambda = 1$. Embeddings were created with the baseline parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$, $k = 1$. We set the sequence length controlling parameters of algorithms to be $l = 40$ (ED2V) and $l = 25$ (FD2V). Embedding algorithm parameters were tuned to show the sensitivity of the results. The manipulated parameters are on the horizontal axis and the classification performance measured by macro F-1 score on the test set is on the vertical axis. Each point represents the average macro F-1 score based on 10 random train-test splits. In each split we used 10% of the dataset for training and the remainder for testing.

Appendix D

Edge prediction

D.1 Gradient boosted trees

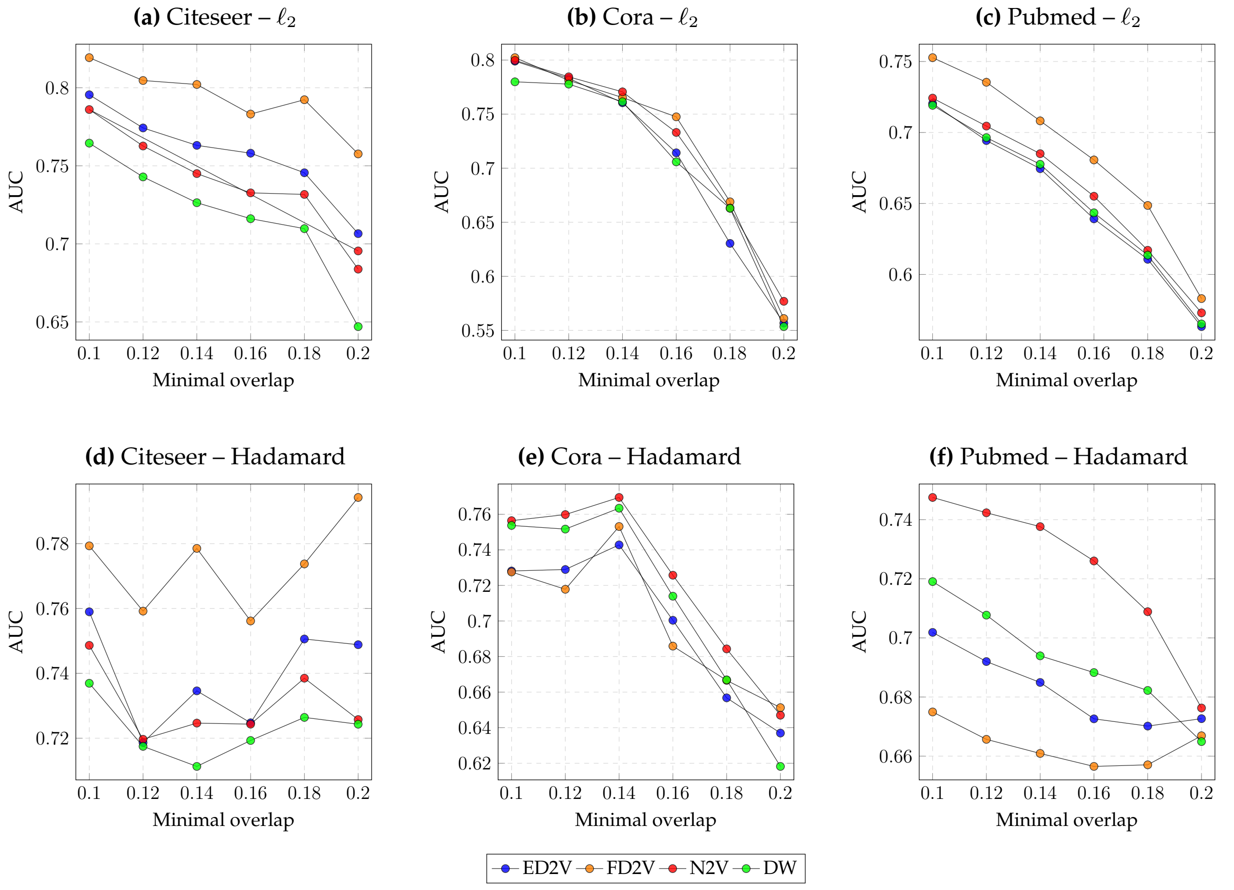


Figure D.1: Edge prediction performance of gradient boosted classification trees using features extracted with the sequence based graph embedding methods. The classifier was fitted with tree depth equal to 3, a learning rate of 0.1 and the number of trees was chosen based on early stopping and 5-fold cross-validation within the training set. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were respectively generated by applying the ℓ_2 and Hadamard operators elementwise on the vectors describing the edge endpoint nodes. The neighbourhood overlap is on the horizontal axis and the classification performance measured by AUC on the test set is on the vertical axis. Each point represents the average AUC based on 10 random train-test splits. The minimal degree parameter η was set as 2.

D.2 Logistic regression

	Algorithm	ArXiv	Facebook	Citeseer	Cora	Pubmed
Average	ED2V	0.7549	0.7705	0.6395	0.5810	0.7159
	DW	0.7758	0.7855	0.6502	0.6215	0.6351
	FD2V	0.7291	0.7668	0.5522	0.5563	0.6355
	N2V	0.7968	0.8009	0.6542	0.6398	0.7594
Hadamard	ED2V	0.9793	0.9887	0.9819	0.9838	0.9911
	DW	0.9801	0.9892	0.9802	0.9841	0.9931
	FD2V	0.9737	0.9879	0.9496	0.9831	0.9898
	N2V	**0.9841**	**0.9895**	0.9830	0.9866	0.9935
L_1	ED2V	0.9197	0.9843	0.9721	0.9869	0.9938
	DW	0.9728	0.9876	0.9809	0.9872	0.9877
	FD2V	0.8679	0.9836	0.9008	0.9781	0.9929
	N2V	0.9766	0.9878	**0.9832**	0.9885	0.9896
L_2	ED2V	0.9210	0.9843	0.9717	0.9870	**0.9941**
	DW	0.9736	0.9878	0.9801	0.9874	0.9878
	FD2V	0.8704	0.9839	0.8971	0.9778	0.9930
	N2V	0.9773	0.9879	0.9816	**0.9889**	0.9898

Table D.1: Edge prediction performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization and the regularization parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 50% of edges was attenuated before embedding. Later 50% of randomly chosen edges was added to the edge set without a neighbourhood overlap limit. Edge features were generated by applying the ℓ_1, ℓ_2, Hadamard and average operators elementwise on the vectors describing the edge endpoint nodes. The table reports performance measured by AUC on the test set. Each value is the average AUC based on 10 seeded random train-test splits. Bold numbers denote the best performing model-operator combination on a given dataset.

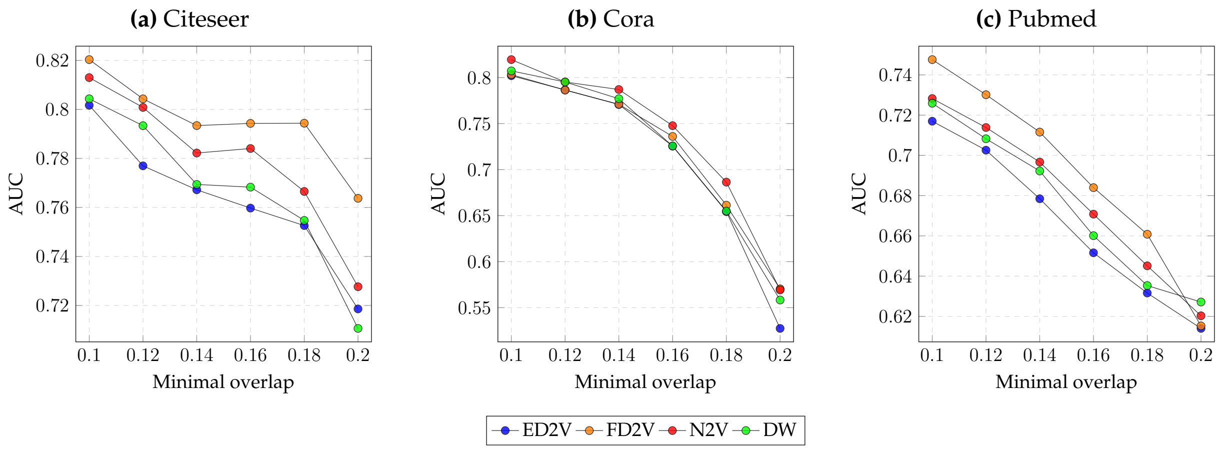


Figure D.2: Edge prediction performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization the regularization parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were generated by applying the ℓ_1 operator elementwise on the vectors describing the edge endpoint nodes. The neighbourhood overlap is on the horizontal axis and the classification performance measured by AUC on the test set is on the vertical axis. Each point represents the average AUC based on 10 random train-test splits. The minimal degree parameter η was set as 2.

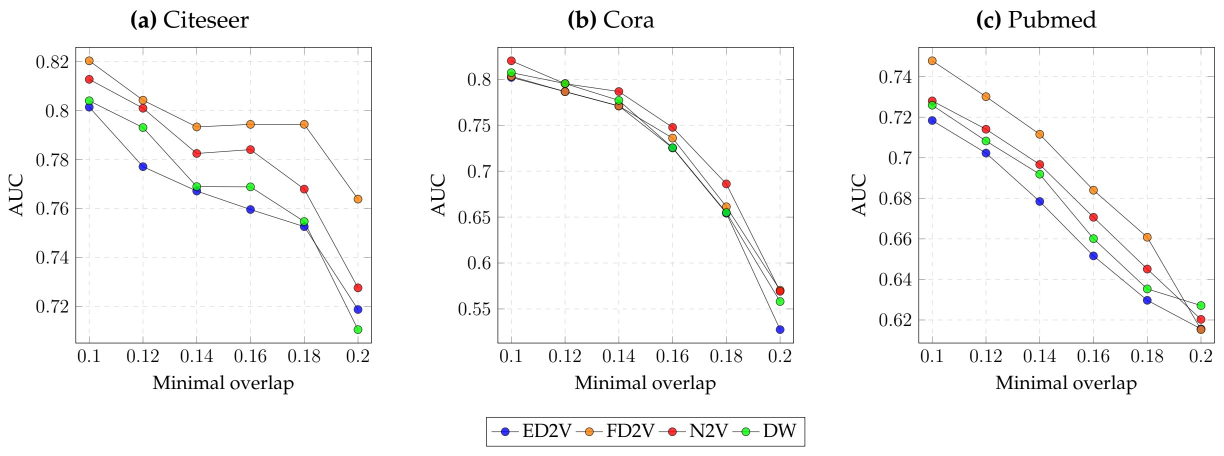


Figure D.3: Edge prediction performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization the regularization parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128$, $n = 10$, $\widehat{w} = 10$, $\alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were generated by applying the ℓ_2 operator elementwise on the vectors describing the edge endpoint nodes. The neighbourhood overlap is on the horizontal axis and the classification performance measured by AUC on the test set is on the vertical axis. Each point represents the average AUC based on 10 random train-test splits. The minimal degree parameter η was set as 2.

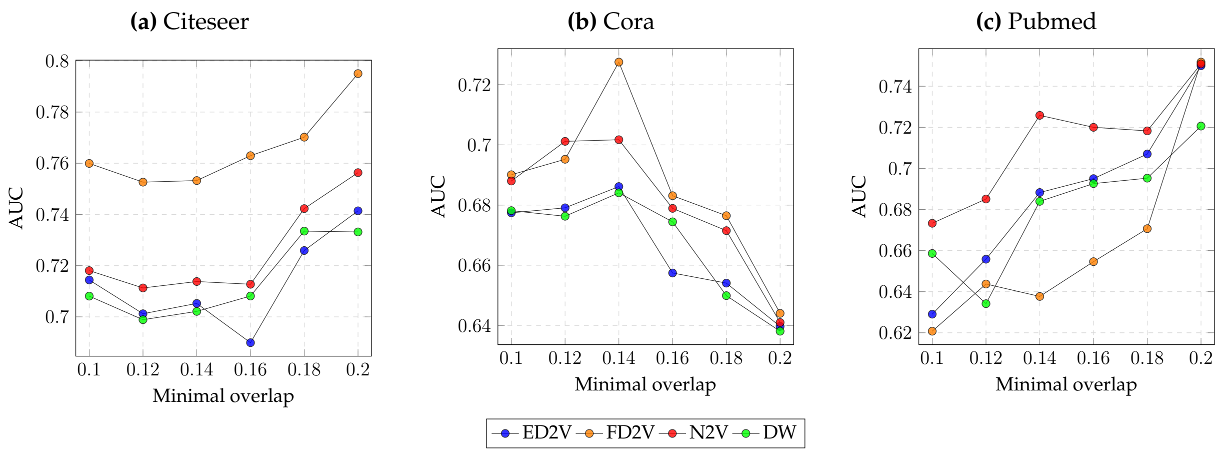


Figure D.4: Edge prediction performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization the regularization parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128,\ n = 10,\ \widehat{w} = 10,\ \alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were generated by applying the average operator elementwise on the vectors describing the edge endpoint nodes. The neighbourhood overlap is on the horizontal axis and the classification performance measured by AUC on the test set is on the vertical axis. Each point represents the average AUC based on 10 random train-test splits. The minimal degree parameter η was set as 2.

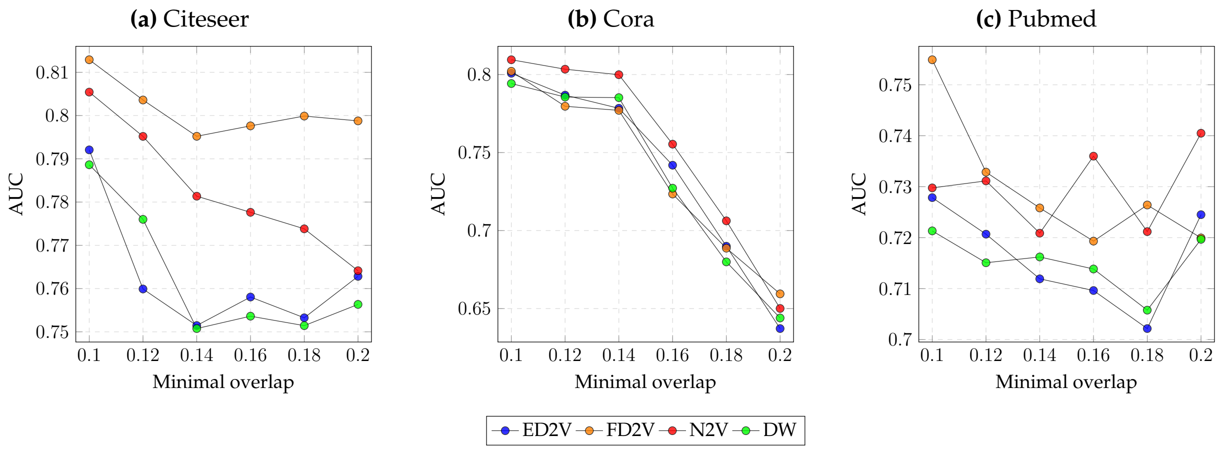


Figure D.5: Edge prediction performance of logistic regression using features extracted with the sequence based graph embedding methods. The classifier was fitted with ℓ_1 regularization the regularization parameter was set as $\lambda = 1$. Embeddings were created with parameter settings such that $d = 128,\ n = 10,\ \widehat{w} = 10,\ \alpha = 0.025$. We set the sequence length controlling parameters of algorithms to be $l = 80$ (N2V and DW), $l = 40$ (ED2V) and $l = 25$ (FD2V). The best performing N2V *inout* and *return* parameters were chosen with grid search over $\{0.25, 0.5, 1, 2, 4\}$ using 50% of data as training set. These parameter settings and practices ensure that we present a fair comparison of the embedding methods. In the dataset generation process 20% of edges was attenuated before embedding. Later 20% of randomly chosen edges was added later to the edge set with a neighbourhood overlap limit. Edge features were generated by applying the Hadamard operator elementwise on the vectors describing the edge endpoint nodes. The neighbourhood overlap is on the horizontal axis and the classification performance measured by AUC on the test set is on the vertical axis. Each point represents the average AUC based on 10 random train-test splits. The minimal degree parameter η was set as 2.

Appendix E

List of abbreviations

ED2V	Diffusion to vector with Eulerian traceback
FD2V	Diffusion to vector with endpoint traceback
DW	Deep walk
N2V	Node to vector
CC	Closeness centrality
PDF	Probability density function
AUC	Area under the curve
WVRN	Weighted vote relational neighbour classifier
PPI	Protein-protein interaction

Bibliography

N. Agarwal, et al. (2009). 'A Social Identity Approach to Identify Familiar Strangers in a Social Network.'. In *ICWSM*.

A. Ahmed, et al. (2013). 'Distributed Large-Scale Natural Graph Factorization'. In *Proceedings of the 22nd international conference on World Wide Web*, pp. 37–48. ACM.

Y.-Y. Ahn, et al. (2011). 'Flavor Network and the Principles of Food Pairing'. *Scientific reports* **1**.

L. Akoglu, et al. (2015). 'Graph Based Anomaly Detection and Description: a Survey'. *Data Mining and Knowledge Discovery* **29**(3):626–688.

F. Al Zamal, et al. (2012). 'Homophily and Latent Attribute Inference: Inferring Latent Attributes of Twitter Users from Neighbors.'. *ICWSM* **270**.

R. Albert & A.-L. Barabási (2002). 'Statistical Mechanics of Complex Networks'. *Reviews of modern physics* **74**(1):47.

L. Backstrom, et al. (2012). 'Four Degrees of Separation'. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pp. 33–42. ACM.

L. Backstrom & J. Leskovec (2011). 'Supervised Random Walks: Predicting and Recommending Links in Social Networks'. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pp. 635–644. ACM.

M. Belkin & P. Niyogi (2003). 'Laplacian Eigenmaps for Dimensionality Reduction and Data Representation'. *Neural computation* **15**(6):1373–1396.

N. Biggs, et al. (1976). *Graph Theory, 1736-1936*. Oxford University Press.

V. D. Blondel, et al. (2008). 'Fast Unfolding of Communities in Large Networks'. *Journal of statistical mechanics: theory and experiment* **10**.

S. Boccaletti, et al. (2006). 'Complex Networks: Structure and Dynamics'. *Physics reports* **424**(4):175–308.

L. Bottou (1991). 'Stochastic Gradient Learning in Neural Networks'. *Proceedings of Neuro-Nımes* **91**(8).

U. Brandes & D. Fleischer (2005). 'Centrality Measures Based on Current Flow.'. In *STACS*, vol. 3404, pp. 533–544. Springer.

S. Cao, et al. (2015). 'Grarep: Learning Graph Representations with Global Structural Information'. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 891–900. ACM.

S. Cao, et al. (2016). 'Deep Neural Networks for Learning Graph Representations'. In *AAAI*, pp. 1145–1152.

A. Chatr-Aryamontri, et al. (2014). 'The BioGRID Interaction Database: 2015 Update'. *Nucleic acids research* **43**(D1):D470–D478.

H. Chen, et al. (2005). 'Link Prediction Approach to Collaborative Filtering'. In *Digital Libraries, 2005. JCDL'05. Proceedings of the 5th ACM/IEEE-CS Joint Conference on*, pp. 141–142. IEEE.

T. Chen & C. Guestrin (2016). 'Xgboost: A Scalable Tree Boosting System'. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 785–794. ACM.

A. Clauset, et al. (2017). 'Data-Driven Predictions in the Science of Science'. *Science* **355**(6324):477–480.

A. Clauset, et al. (2004). 'Finding Community Structure in Very Large Networks'. *Physical Review* **70**(6).

A. Clauset, et al. (2009). 'Power-Law Distributions in Empirical Data'. *SIAM review* **51**(4):661–703.

G. Csardi & T. Nepusz (2006). 'The IGraph Software Package for Complex Network Research'. *InterJournal, Complex Systems* **1695**(5):1–9.

M. Defferrard, et al. (2016). 'Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering'. In *Advances in Neural Information Processing Systems*, pp. 3844–3852.

Y. Dong, et al. (2012). 'Link Prediction and Recommendation Across Heterogeneous Social Networks'. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pp. 181–190. IEEE.

J. Edmonds & E. L. Johnson (1973). 'Matching, Euler Tours and the Chinese Postman'. *Mathematical programming* **5**(1):88–124.

P. Erdős & A. Rényi (1960). 'On the Evolution of Random Graphs'. *Publ. Math. Inst. Hung. Acad. Sci* **5**(1):17–60.

M. Faloutsos, et al. (1999). 'On Power-Law Relationships of the Internet Topology'. In *ACM SIGCOMM computer communication review*, vol. 29, pp. 251–262. ACM.

M. Fire, et al. (2011). 'Link Prediction in Social Networks Using Computationally Efficient Topological Features'. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pp. 73–80. IEEE.

M. Fire, et al. (2013). 'Computationally Efficient Link Prediction in a Variety of Social Networks'. *ACM Transactions on Intelligent Systems and Technology (TIST)* **5**(1):10.

F. Fouss, et al. (2007). 'Random-Walk Computation of Similarities Between Nodes of a Graph with Application to Collaborative Recommendation'. *IEEE Transactions on knowledge and data engineering* **19**(3):355–369.

M. Girvan & M. E. Newman (2002). 'Community Structure in Social and Biological Networks'. *Proceedings of the National Academy of Sciences* **99**(12):7821–7826.

P. Goyal & E. Ferrara (2017). 'Graph Embedding Techniques, Applications, and Performance: A Survey'. *arXiv preprint arXiv:1705.02801* .

S. Gregory (2010). 'Finding Overlapping Communities in Networks by Label Propagation'. *New Journal of Physics* **12**(10):103018.

A. Grover & J. Leskovec (2016). 'node2vec: Scalable Feature Learning for Networks'. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

M. Gutmann & A. Hyvärinen (2010). 'Noise-Contrastive Estimation: A New Estimation Principle for Unnormalized Statistical Models'. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304.

M. Huisman (2009). 'Imputation of Missing Network Data: Some Simple Procedures'. *Journal of Social Structure* **10**(1):1–29.

A. Java, et al. (2007). 'Why We Twitter: Understanding Microblogging Usage and Communities'. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pp. 56–65. ACM.

H. Jeong, et al. (2001). 'Lethality and Centrality in Protein Networks'. *Nature* **411**(6833):41–42.

D. Kempe, et al. (2003). 'Maximizing the Spread of Influence Through a Social Network'. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 137–146. ACM.

T. N. Kipf & M. Welling (2016a). 'Semi-Supervised Classification with Graph Convolutional Networks'. *arXiv preprint arXiv:1609.02907* .

T. N. Kipf & M. Welling (2016b). 'Variational Graph Auto-Encoders'. *arXiv preprint arXiv:1611.07308* .

G. W. Klau & R. Weiskircher (2005). 'Robustness and Resilience'. In *Network analysis*, pp. 417–437. Springer.

J. M. Kleinberg (2000). 'Navigation in a Small World'. *Nature* **406**(6798):845.

G. Kossinets (2006). 'Effects of Missing Data in Social Networks'. *Social networks* **28**(3):247–268.

J. Leskovec, et al. (2007). 'Graph Evolution: Densification and Shrinking Diameters'. *ACM Transactions on Knowledge Discovery from Data (TKDD)* **1**(1):2.

J. Leskovec & J. J. Mcauley (2012). 'Learning to Discover Social Circles in Ego Networks'. In *Advances in neural information processing systems*, pp. 539–547.

J. Leskovec & R. Sosič (2016). 'SNAP: A General-Purpose Network Analysis and Graph-Mining Library'. *ACM Transactions on Intelligent Systems and Technology (TIST)* **8**(1):1.

D. Liben-Nowell & J. Kleinberg (2007). 'The Link-Prediction Problem for Social Networks'. *Journal of the Association for Information Science and Technology* **58**(7):1019–1031.

Q. Lu & L. Getoor (2003). 'Link-Based Classification'. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 496–503.

S. A. Macskassy & F. Provost (2003). 'A Simple Relational Classifier'. In *Proceedings of the Second Workshop on Multi-Relational Data Mining (MRDM-2003) at KDD-2003*. Citeseer.

M. Mahoney (2011). 'Large Text Compression Benchmark'.

J. McAuley & J. Leskovec (2012). 'Image Labeling on a Network: Using Social-Network Metadata for Image Classification'. In *Computer Vision-ECCV*, pp. 828–841.

M. McPherson, et al. (2001). 'Birds of a Feather: Homophily in Social Networks'. *Annual review of sociology* **27**(1):415–444.

T. Mikolov, et al. (2013a). 'Efficient Estimation of Word Representations in Vector Space'.

T. Mikolov, et al. (2013b). 'Distributed Representations of Words and Phrases and Their Compositionality'. In *Advances in neural information processing systems*, pp. 3111–3119.

G. Namata, et al. (2012). 'Query-Driven Active Surveying For Collective Classification'. In *10th International Workshop on Mining and Learning with Graphs*.

A. Narayanan, et al. (2016). 'subgraph2vec: Learning Distributed Representations of Rooted Sub-Graphs from Large Graphs'. *arXiv preprint arXiv:1606.08928* .

M. E. Newman (2003). 'The Structure and Function of Complex Networks'. *SIAM review* **45**(2):167–256.

M. E. Newman (2005). 'A Measure of Betweenness Centrality Based on Random Walks'. *Social networks* **27**(1):39–54.

M. E. Newman (2006). 'Modularity and Community Structure in Networks'. In *Proceedings of the National Academy of Sciences of the United States of America*, vol. 103 of *23*, p. 8577–8696.

M. Ou, et al. (2016). 'Asymmetric Transitivity Preserving Graph Embedding.'. In *KDD*, pp. 1105–1114.

P. Pascal & M. Latapy (2005). *In International Symposium on Computer and Information Sciences*, chap. Computing Communities in Large Networks Using Random Walks., pp. 284–293. Springer Berlin Heidelberg.

C. Peersman, et al. (2011). 'Predicting Age and Gender in Online Social Networks'. In *Proceedings of the 3rd international workshop on Search and mining user-generated contents*, pp. 37–44. ACM.

B. Perozzi, et al. (2014). 'Deepwalk: Online Learning of Social Representations.'. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*.

B. Perozzi & S. Skiena (2015). 'Exact Age Prediction in Social Networks'. In *Proceedings of the 24th International Conference on World Wide Web*, pp. 91–92. ACM.

T. Pimentel, et al. (2017). 'Unsupervised and Scalable Algorithm for Learning Node Representations' .

M. Rahman & M. Al Hasan (2016). 'Link Prediction in Dynamic Networks Using Graphlet'. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 394–409. Springer.

B. Recht, et al. (2011). 'Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent'. In *Advances in neural information processing systems*, pp. 693–701.

S. Redner (1998). 'How Popular is Your Paper? An Empirical Study of the Citation Distribution'. *The European Physical Journal B-Condensed Matter and Complex Systems* **4**(2):131–134.

E. Sarigöl, et al. (2014). 'Predicting Scientific Success Based on Coauthorship Networks'. *EPJ Data Science* **3**(1):9.

P. Sen, et al. (2008). 'Collective Classification in Network Data'. *AI magazine* **29**(3):93.

D. I. Shuman, et al. (2013). 'The Emerging Field of Signal Processing on Graphs: Extending High-Dimensional Data Analysis to Networks and Other Irregular Domains'. *IEEE Signal Processing Magazine* **30**(3):83–98.

S. H. Strogatz (2001). 'Exploring Complex Networks'. *Nature* **410**(6825):268.

J. Tang, et al. (2015). 'LINE: Large-Scale Information Network Embedding'. In *Proceedings of the 24th International Conference on World Wide Web*, pp. 1067–1077. International World Wide Web Conferences Steering Committee.

L. Tang & H. Liu (2009a). 'Relational Learning via Latent Social Dimensions'. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 817–826. ACM.

L. Tang & H. Liu (2009b). 'Scalable Learning of Collective Behavior Based on Sparse Social Dimensions'. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pp. 1107–1116. ACM.

M. Thelwall (2008). 'Social Networks, Gender, and Friending: An Analysis of MySpace Member Profiles'. *Journal of the Association for Information Science and Technology* **59**(8):1321–1330.

D. Wang, et al. (2016). 'Structural Deep Network Embedding'. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1225–1234. ACM.

D. J. Watts & S. H. Strogatz (1998). 'Collective Dynamics of 'Small-World' Networks'. *Nature* **393**(6684):440.

J. Yang & J. Leskovec (2015). 'Defining and Evaluating Network Communities Based on Ground-Truth'. *Knowledge and Information Systems* **42**(1):181–213.

Z. Yang, et al. (2016). 'Revisiting Semi-Supervised Learning with Graph Embeddings'. In *International Conference on Machine Learning*, pp. 40–48.

S. Yuan, et al. (2017). 'SNE: Signed Network Embedding'. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 183–195. Springer.