University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Scaling Network Embeddings

Master's Thesis in Computer Science
by

Vladyslav Maksyk

Internal Supervisors

Vinay Setty

June 15, 2020

"Our greatest glory is not in never falling, but in rising every time we fall."

Confucius

Abstract

A Recommendation System is an intelligent machine learning system that seeks to predict a customer ranked set of personalized products from a dynamic pool of diverse choices. We can define the main objective of such systems as ranking edges in an undirected unweighted graph consisting of user and item nodes.

Deep Graph embeddings have recently attracted the interests of both academia and industry, mainly because of its simplicity and effectiveness in a variety of applications. This thesis's primary purpose is to perform research on the existing graph embeddings methods for recommendation algorithms. We aim to transform undirected unweighted graphs into vectors, also known as graph embeddings, to make a representation that would be suitable for different machine learning algorithms. At first, we introduce the reader to some existing and conventional approaches that allow us to create such embeddings. We then present several modifications and improvements to the existing methods. Finally, we use several evaluation metrics to showcase the performance evaluations of such modifications.

Acknowledgements

I want to express my sincere appreciation to my supervisor Vinay Setty for the professional guidance throughout the extend of this thesis. I would like to thank him for his dedicated involvement and assistance in every stage throughout the process. Vinay was always available to meet either in person or through video conferencing and consistently provided invaluable feedback towards my research.

I would also like to thank all the teachers and professors from the Faculty of Science and Technology Department of Computer Science for sharing their valuable knowledge, which definitely contributes to this thesis work.

I would like to thank my former group-mates who show up with smiles to lectures, labs, and project meetings during my memorable time here at the University of Stavanger.

Finally, I would like to thank my family: my parents, sisters, and brothers for their continuous support. Very special thanks to my friends for being my second family and for giving me lots of inspiration.

Contents

Abstract vi

Acknowledgements viii

Abbreviations xi

1 Introduction 1
- 1.1 Recommender Systems for the world . . . 1
- 1.2 Graph Perspective of Recommender Systems . . . 2
- 1.3 Capturing the Graph Structure . . . 3
- 1.4 Problem Definition . . . 3
- 1.5 Challenges . . . 4
- 1.6 Contributions . . . 4
- 1.7 Outline . . . 5

2 Background 7
- 2.1 Definitions and Preliminaries . . . 7
  - 2.1.1 Graph properties . . . 7
  - 2.1.2 Graph Representations . . . 8
  - 2.1.3 Proximities . . . 9
  - 2.1.4 Graph Embedding . . . 10
- 2.2 Machine Learning Fundamentals . . . 11
  - 2.2.1 Data Mining . . . 11
  - 2.2.2 Machine Learning . . . 12
  - 2.2.3 Artificial Neural Networks . . . 13
  - 2.2.4 Performance Evaluation . . . 16
- 2.3 Graph Embedding Methods . . . 17
  - 2.3.1 Factorisation based Approaches . . . 18
  - 2.3.2 Random Walk based Approaches . . . 19
  - 2.3.3 Deep Learning based Approaches . . . 28
  - 2.3.4 Other approaches . . . 32
  - 2.3.5 Conclusion . . . 32

3 Implementation 35

3.1 Method modifications . . . 35
3.1.1 APP . . . 35
3.1.2 DeepWalk . . . 37
3.1.3 Approximated DeepWalk . . . 39
3.1.4 Parallelized DeepWalk . . . 39

4 Evaluation and Experimental Setup 41
4.1 Data-sets . . . 41
4.2 Performance Evaluation . . . 42
4.2.1 Embedding generation . . . 42
4.2.2 Node recommendation with APP . . . 44
4.2.3 Multi-label node classification with Deepwalk . . . 44
4.2.4 Discussion . . . 45
4.3 Technical Documentation . . . 46
4.3.1 Programming Languages and Libraries . . . 47
4.3.2 Functional Interface . . . 47
4.3.3 Dependencies . . . 48
4.3.4 Installation . . . 48
4.3.5 Running the models . . . 49

5 Conclusion and Future Work 53
5.1 Conclusion . . . 53
5.2 Future Work . . . 54

List of Figures 54

List of Tables 57

Bibliography 59

Abbreviations

AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
APP	**A**symmetric **P**roximity **P**reserving
APR	**A**rtificial **P**recision **R**ate
ARR	**A**rtificial **R**ecall **R**ate
BFS	**B**readth**F**irst **S**earch
CBOW	**C**ontinuous **B**ag **O**f **W**ords
DFS	**D**epth **F**irst **S**earch
DL	**D**eep **L**earning
DW	**D**eep **W**alk
LLE	**L**ocally **L**inear **E**mbedding
ML	**M**achine **L**earning
MSE	**M**ean **S**quared **E**rror
NPL	**N**atural **L**anguage **P**rocessing
RS	**R**ecommender **S**ystem
RW	**R**andom **W**alk
SDNE	**S**tructural **D**eep **N**etwork **E**mbedding
SE	**S**earch **E**ngine
SGNS	**S**kip **G**ram **N**egative **S**ampling
SVM	**S**upport **V**ector **M**achine

Chapter 1

Introduction

1.1 Recommender Systems for the world

World's leading companies like Amazon, Netflix, Google, and many other such services offer a wide variety of products to their end-users. With the help of search engines, costumers can easily navigate through these products. However, despite the apparent advantages that a search engine provides, its ability is strictly limited. The search engine can only help users query products that they are already aware of. Recommender systems have erased this limitation allowing users to discover products that they may like but are unfamiliar with, based on previous records.

Generally, recommender systems are complex algorithms that are focused on suggesting relevant items to the end-user. These items can be books, movie products, and anything else depending on industries.

Recommender system's machine learning algorithms are generally divided into two main categories - collaborative filtering and content-based methods, although modern approaches use a combination of both methods. Collaborative filtering methods are based on the similarity from interactions, whereas content based on the similarity of item attributes. Collaborative methods use a rating matrix that stores explicit ratings given by users to items. The rating matrix is then used as input to the machine-leaning algorithm, which learns a function that predicts the utility of items to each other [1].

Products recommendation helps companies leverage huge incomes. Netflix estimates that its recommendation engine helps them save $1 billion a year [2]. Amazon's recommendations account for 35 percent of its sales [3]. Youtube's video recommender system accounts for 60 % of user views [4].

1.2 Graph Perspective of Recommender Systems

Throughout the years, graphs have evolved into a universal data structure that can be used in a variety of applications from pathfinding to social network analysis. Graphs are known for their excellent ability to express relationships between objects and are considered to be among the most convenient means of representing and storing structured knowledge.

Several techniques are used to represent such data structure, such as edge lists, adjacency lists, and adjacency matrices. The suitability of a certain method for a specific task is mostly defined by three main criteria. One is how much space we would require for each representation. Since modern days datasets have substantially grown in size in the latest years, this criteria remains to be crucial. One is the time duration that it takes to determine whether a particular edge is a member of a graph. The other is how long it takes to find the neighbors of a specific node.

Although these methods are sufficient for most of the graph-processing algorithms, they cannot be used as input to train machine learning models. This is because none of the methods described above are suitable for similarity detection within the graph structure. Since, an adjacency matrix is a representation of all nodes in a |V|-dimensional vector space, two linked vertices in the graph may have orthogonal representations. List of edges and Adjacency lists, due to their varying sizes, cannot be applied to any of the existing machine learning algorithms. Finally, conventional representations can significantly impact the time and memory consumption due to their overwhelmingly large size.

Throughout the years, several methods were introduced to address this issue. The most common solution was to express data structure within a continuous multi-dimensional vector space, where the nodes of a graph are represented with vectors of real numbers, also known as graph embeddings. This method achieves a significantly smaller dimension of the target space compared to the total number of vertices in the graph. Finally, these vectors are used to detect similarities between nodes.

Deep embeddings have proved to be successfully in several applications such as Computer Vision [5] [6] [7] , Natural Language Processing [8] [9], Speech Processing [10]. Deep Graph embeddings as an extension for graph structured data has also achieved significant successes in such applications as semantic segmentation [11], robot design [12], medical diagnosis [13], object annotation [14] and others.

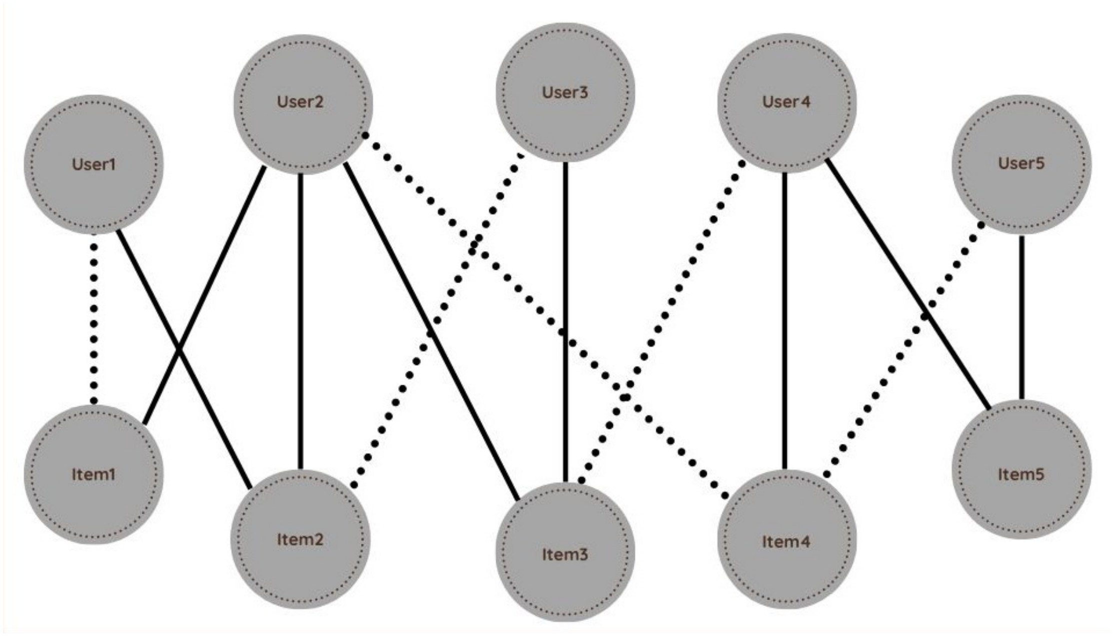

Figure 1.1: Rating matrix as a Bipartite Graph

1.3 Capturing the Graph Structure

Recommender systems can be expressed as a bipartite graph of users and items. The edge weights represent the similarity score between items and users. A bipartite graph is preferred since the systems are generally focused on the connections between item-user pairs.

In Figure 1.1, we represent a recommender system that contains five users and five items as a bipartite graph. The solid edges of the graph show a historical interconnection between the item and the user, such as a purchase, whereas the dashed edges are the probable connections that have to be predicted. The task is to rank these dashed edges so that they result in a higher score and then recommend the highest-scoring edges for each user node. This approach allows specialists to create graph-based algorithms for recommendation systems.

1.4 Problem Definition

For this study, we primarily consider undirected graphs. Let G be the network, and V be the set of all nodes within that network.

Given a social network $G = (V, E)$ with vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E where the vertices of the network represent the members and the edges represent the connections between them, we want to implement an approach for learning latent representations of vertices in that network. These latent representations represent vectors that encode the social relations of vertices in a continuous vector space, which is easily exploited by statistical models. The idea is to classify members of a social network into different categories. We propose an unsupervised method that learns features that capture the graph structure independent of the labels' distribution.

1.5 Challenges

Representing each node of a network as a vector can impose multiple challenges which have been driving research in this area:

(i) Choice of property: It is important for a vector representation model to retain the structure of the graph and the relations between the nodes. The first challenge is to determine the property of the graph that should be preserved in vector representation.

(ii) Scalability: Most networks are extensive, and for a method to process such networks, great scalability is required. This cannot be easy, given the importance of preserving the global properties of the graph.

(iii) Dimensionality of the embedding: It is challenging to obtain optimal dimensions of the representation. For instance, a high degree of dimensions can result in an increase in reconstruction precision while preserving a high space and time complexity. However, a lower degree of dimensions may boost the accuracy of link prediction if the chosen model only considers local relations between nodes.

1.6 Contributions

The main goal of this thesis is to analyze and mutually compare a selected set of methods for learning latent representations of vertices in a network, characterize their features, and point out their advantages and disadvantages. In this study, we will mainly focus on how to improve already existing techniques that learn latent representations of graphs such as Asymmetric Proximity Preserving (APP) and DeepWalk. DeepWalk uses local information obtained from truncated random walks to learn latent representations by treating walks as the equivalent of sentences. We plan to improve the performance of this technique by implementing a unique approach of random walks sampling in which we are achieving the effect of multiple random walks in a single one with the help of Breadth-First Search algorithm and simultaneously capture the context of the nodes.

This thesis focuses on Deep Graph Embeddings for recommendation systems as they offer promising new models that have not yet been well studied in the recommendation space.

1.7 Outline

This thesis consists of several chapters.

In Chapter 1, we describe the recommendation problem, cover the graph perspective of recommender systems, and introduce modern techniques of capturing the graph structure.

In Chapter 2, we provide a background for this thesis. We cover the foundation, definitions, and notation used within this thesis. We describe the basics of graphs, graph representation methods, related terms, and proceed towards machine learning and neural networks. We present the evaluating metrics used to measure the performance of our experiments. And then introduce modern techniques for creating graph embeddings and their appliance in the recommend systems. We additionally emphasize the underlying architecture of these methods and their fundamental basics.

In Chapter 3, we describe our contribution within this thesis work and provide a detailed interpretation of the implemented solution.

In Chapter 4, we analyze the properties of recommender system datasets, as well as cover the recommender datasets used within this thesis and point out the challenges associated with them. We provide experimental results of the model in the form of a table and discuss achieved improvements. We then describe the technical details and useful guidelines necessary for the installation process.

In Chapter 5, we present a short conclusion of this thesis and our accomplishments and mention improvements for future work.

Chapter 2

Background

This section describes the background of this study. It provides the reader with all the necessary context to the information discussed throughout this study. We at least expect the reader to be familiar with basic mathematical concepts and terms like vectors and matrices. However, it is not mandatory to have any prior knowledge of Artificial Intelligence since we will cover that in detail throughout this chapter. Furthermore, we will provide useful references to certain conventions, notation, and definitions. We also recommend these books [15] [16] to the reader as a reliable and verified source of information related to this thesis work.

2.1 Definitions and Preliminaries

We will consider graphs and their representations as to the primary data structure within this thesis. Below we will provide the necessary preliminaries on this data structure its representations and critical notations.

2.1.1 Graph properties

A **Graph** $G = (V, E)$ is a set of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges $E = \{e_{ij}\}_{i,j=1}^{n}$, each connecting two different vertices. The edges and vertices of a graph can posses a wide variety of different properties, where some of them are more common than others. In particular, two edge properties stand out. These properties are edge *directionality* and *edge weight*.

- **Directed:** A graph is considered to be *directed* if all of its edges represent a one-way relationship. In directed graph an edge $uv \in E$ has start point u and endpoint v.
- **Undirected:** A graph is considered to be *undirected* if all edges are bidirectional. In a simple undirected graph $uv = vu$ and $uu \notin E$.

A *weight* property of an edge is defined by a numerical value associated with an individual edge [17].

- **Weighted:** A graph is considered as *weighted graph* when all of its edges $E = \{e_1, e_2, ..., e_n\} \in G$ are assigned a positive number $w(e)$, called the *weight* of e.
- **Unweighted:** An *unweighted graph* is a graph with no edge value. We can consider an unweighted graph to be weighted in which all edges e are assigned a weight $w(e) = 1$.

Adding weights to a graph allows us to represent complex objects like road networks and probabilistic models. In this thesis, we will only consider simple undirected, unweighted graphs.

2.1.2 Graph Representations

A graph can be represented in several ways. Each method has its advantages and disadvantages. Hence a method should be chosen based on the algorithm we want to run with graphs as input. Three main criteria define the performance of a method:

- Required memory for each representation
- Time for querying all neighbours of a node
- Time for querying a given edge

Here we will present three methods for representing graphs.

- **Edge lists** A graph is represented as a nested list or an array of $E = \{e_1, e_2, ..., e_n\}$ edges. An edge is represented with an array of two vector numbers, where each number refers to a vertex. The total space complexity of this method would be $\Theta(E)$ since each edge would contain a defined amount of numbers. However, if we want to search for a given edge in an unsorted edge list, it would take linear time to search through $|E|$ edges.

- **Adjacency matrix.** For a graph with $|V|$ vertices, an **adjacency matrix** is a $|V| \times |V|$ matrix consisting of $0s$ and $1s$, where the entry is 1 in i row and j column only if the edge (i, j) exists in the graph. For instance the adjacency matrix of graph Figure 2.1 can be represented with the following matrix (2.1). This method has an advantage in edge lookup. We can check whether an edge is present in a graph in constant time by looking at G[i][j], where G is the graph, and (i, j) represents an edge. However, this method has two disadvantages. First, it would take $\Theta(V^2)$ space. Second, search for neighbors of a given vertex i requires to look up the entire row of $|V|$ entries in row i.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{2.1}$$

- **Adjacency list.** The graph is represented by a vertex i and a list of vertexes that are adjacent to it. The adjacency list L_1 for a vertex v_1 is a composition of all vertexes that are target of all the edges, which start at v_1. Consequently, there will be $|V|$ amount of adjacency lists, one for each vertex. For instance, the adjacency list of the same graph in Figure 2.1 could be represented as sown in (2.2). The clear advantage of this method is finding the neighbors of a node, since it will take constant time $O(1)$ to get to each vertex's adjacency list [18]. A disadvantage is that to find out whether an edge (i, j) is present in the graph, we would have to extract i's adjacency list and then search for j in i's adjacency list. The time complexity of this operation would be $O(D)$, where D is the degree of vertex i.

$$\{L_A = [B], L_B = [A, C, D, E], L_C = [B, E], L_D = [B, E], L_E = [B, C, D]\} \tag{2.2}$$

2.1.3 Proximities

In a real-world network, the similarity between nodes is defined by *first order proximity*. For instance, friends in a social network usually share common interests. In order to capture these relationships, many existing graph embeddings methods tend to preserve first-order proximity. However, not always fundamentally similar nodes on the network have a direct connecting link, therefore to preserve the network structure, alternative methods are required.

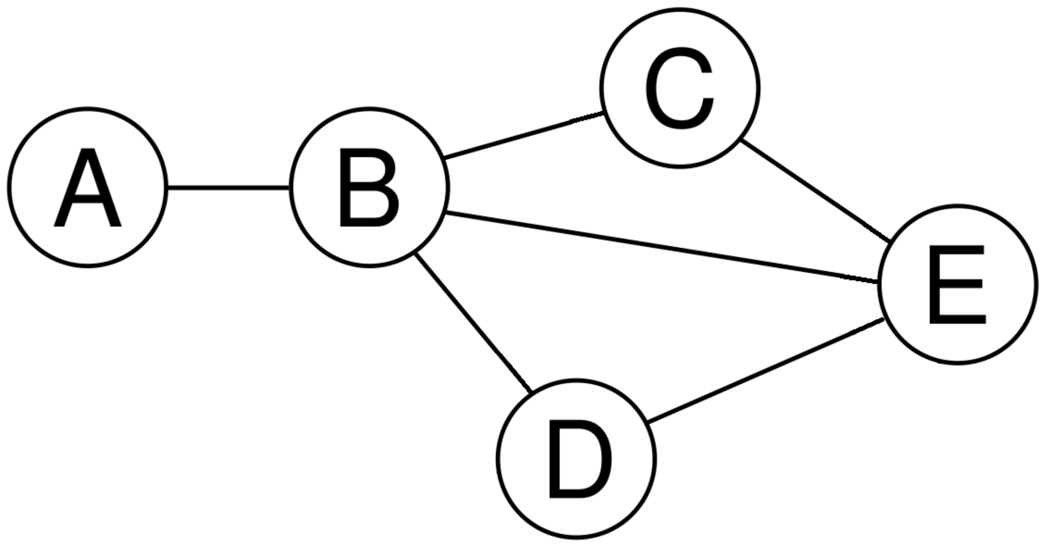

Figure 2.1: Unweighted undirected graph

It is generally accepted that similar nodes on the network tend to share common neighbors. For instance, words that always appear in the same context of words tend to have a similar meaning. Hence, we define the second-order proximity, which captures the similarity of the nodes' neighborhood structure [19].

Definition 2.1. First-Order Proximity [20] The first-order proximity in a network is the local pairwise proximity between two vertices. For each pair of vertices linked by an edge (u, v), the weight on that edge, w_{uv}, indicates the first-order proximity between u and v. If no edge is observed between u and v, their first-order proximity is equal to 0.

Definition 2.2. Second-Order Proximity [20] The second-order proximity between a pair of vertices (u, v) in a network is the similarity between their neighborhood network structures. Mathematically, let $p_u = (w_{u,1}, ..., w_{u,|V|})$ denote the first-order proximity of u with all the other vertices, then the second-order proximity between u and v is determined by the similarity between p_u and p_v. If no vertex is linked from/to both u and v, the second-order proximity between u and v is 0.

2.1.4 Graph Embedding

Given a graph $G = (V, E)$ a graph embedding is a mapping $f : v_i \rightarrow y_i \in \mathbb{R}^d \; \forall \in [n]$ such that $d \ll |V|$ and function f preserves proximity structure of graph G. Consequently, the goal of an embedding is to capture the network structure by learning a low-dimensional vector representation of each node. The majority of existing methods train node embeddings to differentiate the positive edges in E from some randomly sampled node pairs (negative edges). Essentially, the edges are used as training data. An embedding preserving the *first-order proximity* can be acquired by minimizing:

$$\sum_{i,j} S_{i,j} ||\boldsymbol{y_i} - \boldsymbol{y_j}||_2^2 \tag{2.3}$$

2.2 Machine Learning Fundamentals

This section provides a detailed elaboration on relevant machine learning terms and definitions closely associated with this thesis. We provide a description of Machine Learning in general, followed by a definition of Data Mining and relevant algorithms.

2.2.1 Data Mining

Data mining defines a process of applying analysis algorithms on a dataset to discover hidden patterns within the data as well as classify the data into distinct classes. Data Mining tasks are classified into *Supervised* and *Unsupervised* depending on the information the algorithm has about the available labels in the dataset.

- **Supervised Learning**[1] models are trained on data containing both input and output values. Input values are essentially the attribute values and metadata, whereas output values represent the labels of a class attribute. This model aims to predict the correct class of the new data based on previous records.
- **Unsupervised Learning**[2] models, on the contrary, have no access to output values. Therefore they attempt to discover patterns within the data by creating their own classes.
- **Semi-supervised Learning**[3] model represent a combination of both *Supervised* and *Unsupervised* learning models and usually work on partially-labeled data.

Moreover, data mining has two principal objectives, *verification* and *discovery*, where *verification* attempts to justify user's conjecture and *discovery* is further divided into *description* and *prediction*, where *description* is a method that discovers structures for

[1]Supervised Learning: https://en.wikipedia.org/wiki/Supervised_learning
[2]Unsupervised Learning: https://en.wikipedia.org/wiki/Unsupervised_learning
[3]Semi-supervised Learning: https://en.wikipedia.org/wiki/Semi-supervised_learning

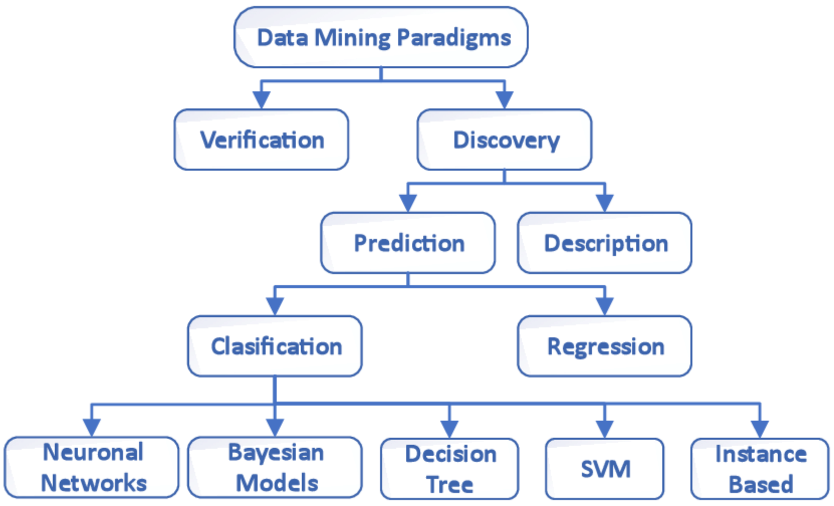

Figure 2.2: Data mining taxonomy

the purpose of representing the data in a clear layout and *prediction* attempts to predict the future labels of records based on known patterns.

Furthermore, data mining can be differentiated into two main objectives: verification and discovery. While verification tries to prove the user's hypothesis, discovery looks for yet unknown patterns within the data. The discovery step splits up into description, where the system finds patterns in order to present the data in an understandable format and prediction, where the system tries to predict the future outcomes of data from patterns.

The sub-task Prediction is consequently divided into classification and regression tasks, where classification tasks have categorical labels, whereas regression tasks have numerical, continuous labels.

This work focuses on algorithms that attempt to learn latent representations of vertices in a network. Therefore, the problem consists of multi-label network classification tasks and belongs to the discovery-prediction sector of data mining. In Figure 2.2, we represent the general overview of data mining taxonomy.

2.2.2 Machine Learning

A Machine Learning model can be defined as a computer program that is able to learn from provided data and consequently make predictions/decisions on unseen data. Conventional machine learning models learn a mapping taken a feature vector as input, which represents an object with the means of categorical and numerical characteristics. The main objective is to generate an output of the desired form, which can be categorized into class labels, regression score, an unsupervised cluster-id, or a latent vector (embedding).

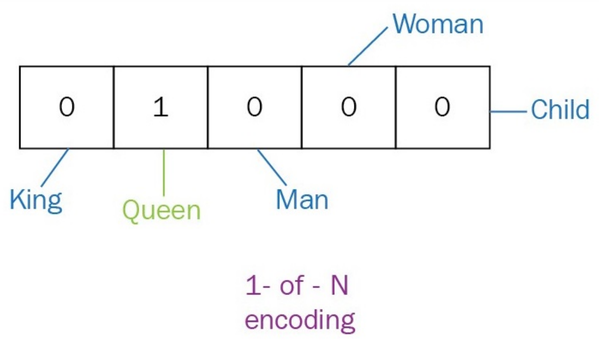

Figure 2.3: One-Hot representation. Image source: https://subscription.packtpub.com/book/web_development/9781786465825/3/ch03lvl1sec32/encoding-and-embedding

In this study, we will mainly focus on data in the form of a graph consisting of entities (nodes) and edges (relationships between nodes) with the purpose of clustering based on node connectivity.

Machine learning models can be classified into **parametric**[4] and **non-parametric**[5] depending on their inner structure. Parametric models employ a finite set of parameters θ to retain the required structure of the given data, whereas in non-parametric models, the numbers are not predefined.

Data Representation

Data-sets with categorical variables usually require conversion into numerical form in order to be fed as input to Machine learning models. One method to represent such categorical data is **One-Hot**[6] encoding. Such encoding method represents categorical attributes with binary vectors. First, it maps categorical attributes into integer values. Then, it represents every integer value in the form of a binary vector, where the index of the integer is equal to one, and the rest are zeros. In Figure 2.3, we represent one-hot encoding for word *Queen* given the vocabulary $V = \{King, Queen, Man, Woman, Child\}$.

2.2.3 Artificial Neural Networks

Since neural networks are related to this thesis work, we consider that it is necessary to familiarize the reader with important terms and notations from this area. For a broad overview of the neural network concept we recommend the book written by Deng et al. [21].

[4]Parametric statistics: https://en.wikipedia.org/wiki/Parametric_statistics
[5]Non-parametric statistics: https://en.wikipedia.org/wiki/Nonparametric_statistics
[6]One-hot: https://en.wikipedia.org/wiki/One-hot

In comparison to conventional algorithms, neural networks attempt to address rather complex problems with a significantly easier approach in regard to algorithm complexity. Consequently, Artificial Neural Networks are mainly employed due to their straightforward structure and self-organizing nature, which allows them to deal with a wide variety of issues without additional intervention by the programmer.

An Artificial Neural Network contains nodes (neurons), weighted connections between these nodes, adjustable in the course of the learning procedure of the network, and an activation function that defines the output of every node given an input of a set of inputs. A Neural Network consists of multiple different layers. The input layer collects the data, such as attribute values of particular data entry, the output layer generates the output, and the hidden layers represent the connections between the input and output layers. Furthermore, Neural networks are classified into two main types.

- **Feedforward Networks** are networks that do not collect feedback from the network itself, which means that the data flows in one direction, from the input to the output layers, without further readjustments of the system.
- **Recurrent Networks**, in contrast, has a feedback option and is capable of reusing data from later stages for the earlier stages in the learning procedure.

The output value is generated wit the means of an activation function. The most frequently used function is the *Sigmoid function*[7], defined in the following manner:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

In Figure 2.4, we present a Feedforward Neural Network with four input nodes in the input layer, two hidden layers, and two output nodes.

A neural network is able to evaluate various functions by readjusting the parameters of the model. Further, we will briefly describe the main phases of the training process of a neural network:

- **Forwardpropagation**. Describes a process of passing the training data over the network, where every neuron applies modifications to the received data from the previous layer and sends it to the next layer. Hence, when the data has passed through all the layers and all the computations are made by its neurons, the final layer outputs the class predictions for the given input data.

[7] Sigmoid function: https://en.wikipedia.org/wiki/Sigmoid_function

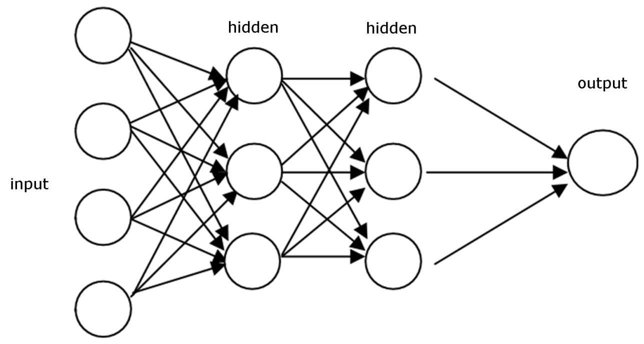

Figure 2.4: Example of an Artificial Neural Network

- **Loss Function**. The training process is generally formulated as an optimization problem. Hence it is subject to a minimization of the loss function L, which tends to estimate the prediction quality. More precisely, the loss function is required to estimate how close a specific neural network is to an optimal weight during the training procedure. All available types of loss functions are listed on the Keras manual page[8].

- **Backpropagation**[9]. Artificial Neural Networks are often trained with a backpropagation algorithm, where the weights of the neural network connections are adjusted based upon the local error rates. Consequently, this method calculates the gradients $\frac{\delta L}{\delta w_i}$ for all weights $w \in \Theta$ in a progressive manner.

- **Optimization algorithms** make use of the gradients obtained from the backpropagation algorithm to adjust the weights of the interconnections, aiming to lower the loss function. Most frequently used techniques are:

  – **Gradient Descent**[10] and its variants such as **Stochastic Gradient Descend**[11] and **SGD with momentum**[12].

  – Adaptive gradient descent algorithms such as **Adagrad**[13] or **Adam**[14].

In Figure 2.5 we visually summarize the training process described above.

[8]Keras loss functions: https://keras.io/api/losses/
[9]Backpropagation: https://en.wikipedia.org/wiki/Backpropagation
[10]Gradient Descent: https://en.wikipedia.org/wiki/Gradient_descent
[11]Stochastic Gradient Descend: https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[12]Momentum: https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Momentum
[13]Adagrad: https://en.wikipedia.org/wiki/Stochastic_gradient_descent#AdaGrad
[14]Adam: https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Adam

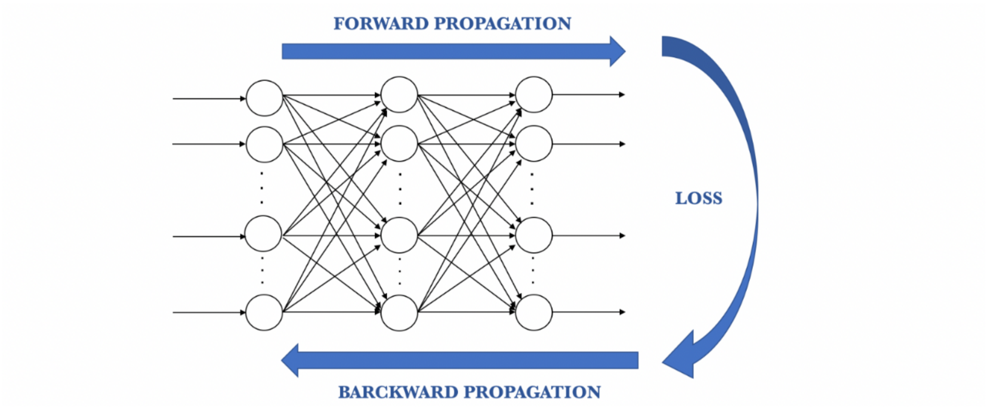

Figure 2.5: Training scheme of a Neural Network.

2.2.4 Performance Evaluation

Another crucial step in building a machine learning model is performance evaluation. The model should be able to estimate the target variable with a small error margin, which can be calculated with different metrics. The most important factors for performance evaluation of a machine learning model are the following:

- **Mean Squared Error**
- **Misclassification rate** estimates the relative amount of misclassified data in a given dataset. A formula for calculation misclassification rate is defined as follows:

$$misc_n = \frac{1}{n} * \sum_i (y_i \neq \hat{y_i}) \tag{2.5}$$

  where y_i is the real label and $\hat{y_i}$ defines the prediction for the data point i. However, misclassification heavily depends on class label distribution in the dataset.

- **F1-Measure**[15] denotes a Harmonic Mean[16] between precision and recall values and estimates the precision and robustness of a classifier. It can be mathematically expressed as follows:

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{2.6}$$

  - **Precision value**[17] is defined as the relative amount of instances classified as true among all instances classified as true.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{2.7}$$

[15]F1-Measure: https://en.wikipedia.org/wiki/F1_score
[16]Harmonic Mean: https://en.wikipedia.org/wiki/Harmonic_mean
[17]Precision value: https://en.wikipedia.org/wiki/Precision_and_recall#Precision

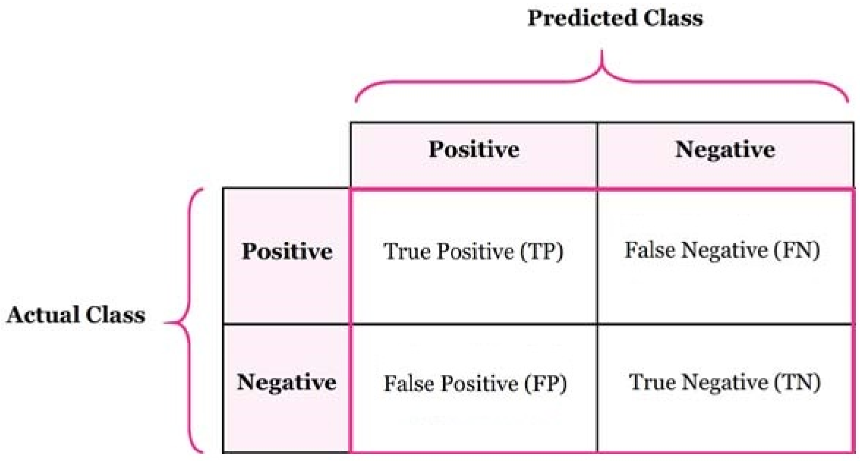

Figure 2.6: Confusion Matrix

- **Recall value**[18] is defined as the relative amount of instances classified as true among all true instances.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegative} \tag{2.8}$$

- **Confusion matrix** is considered to be one of the best methods to illustrate the performance of a machine learning model, which differentiates between true-negative, false-negative, true-positive, and false-positive predictions. A confusion matrix is depicted in Figure 2.6.

2.3 Graph Embedding Methods

The use of neural networks in different applications has increased substantially. It has successfully proved its efficiency in such applications as image segmentation, natural language processing, time-series forecasting, and embedding. The application we are most interested in throughout this thesis is embedding, where embedding is a method used to represent discrete variables as continuous vectors.

The term embedding is closely related to the Natural Language Processing domain, where Tomas Mikolov [22] introduced a method for learning high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships. However, these models were unsuitable for sparse graphs and alternative methods we required. Consequently, researchers have shifted their focus towards the development of scalable graph embedding algorithms that would address the issue of

[18]Recall value: `https://en.wikipedia.org/wiki/Precision_and_recall#Recall`

graph sparsity. In this section, we will briefly describe modern approaches used for generating graph embeddings.

A desirable embedding method should meet several requirements. One is that it must be scalable for very large graphs with millions of nodes and edges. Second, it should desirably preserve both *first-order proximity* and *second-order proximity.* Finally, it should be able to process any type of edges: directed/undirected and weighted/unweighted.

Due to the extreme sparsity of the real-world networks, such as Youtube and Facebook, many embedding approaches perform random walks on the original network to capture its connectivity. In other words, they connect nodes within the distance of an arbitrary walk length, known as network augmenting.

Furthermore, based on the samples from the augmented network, we train the node embeddings. Essentially, embeddings consist of two sets, vertex and context embedding matrices. A positive edge is then predicted with the dot product of vertex $[v]$ and context $[v]$ where (u, v) is an edge sample. This results in similar embeddings for neighboring nodes and disparate embeddings for remote nodes.

Overall, node embeddings are produced by two sequential methods: network augmentation and embedding training. Usually, both methods can be parallelized by multiple CPU threads to improve running time. We will use the taxonomy of approaches to graph embeddings proposed by Goyal and Ferrara [23] to cover the most popular, currently used methods for creating embeddings. We will describe three different graph embedding approaches with several representative algorithms for each approach. For each algorithm, we will provide performance evaluation, analyze preserved properties, and accuracy.

2.3.1 Factorisation based Approaches

Factorization based methods represent the relations between vertexes as a matrix and use factorization methods to create embeddings. The node connectivity can be represented with an adjacency matrix, Laplacian Matrix, Katz similarity matrix, etc. The choice of a factorization method depends on the matrix properties.

Locally Linear Embedding

One simple approach to reduce the adjacency matrix sparsity is to use Locally Linear Embedding method, which relies on the assumption that every node represents a linear composition of its direct neighbor nodes. It signifies that the embedding E_i can be formulated according to Equation (2.9) , where N_i is a collection of i neighbours and E_i is a vector with the following dimensions $d \ll n$.

$$E_i = \sum_{j \in N_i} W_{i,j} \times E_j \tag{2.9}$$

$$\phi(E) = \sum_i (E_i - \sum_{j \in N_i} W_{i,j} \times E_j)^2 \tag{2.10}$$

The embedding is produced by minimizing the equation (2.10). This approach allows for retaining the structure of the graph.

Laplacian Eigenmaps

Laplacian Eigenmaps [24] algorithm emerged in 2002 and was one of the first algorithms for creating graph embeddings. Laplacian Eigenmaps method is used to create a low-dimensional image of the graph while preserving the local geometric characteristics of the embedding manifold. Laplacian Eigenmaps approximates the geodesic distances between nodes with the help of the neighborhood graph.

The objective function to obtain the embeddings is subject to a minimalization problem of seeking vectors $\boldsymbol{y} \in \mathbb{R}^{|V| \times d}$ and utilize them for embedding creation:

$$\min_y = \sum_{(v_i,v_j) \in E} w_{i,j}(y_i - y_j)^2 \tag{2.11}$$

where w_i is the weight of edge (v_i, v_j) and $\boldsymbol{y} = (y_1, \ldots, y_{|V|})^T$.

2.3.2 Random Walk based Approaches

Random walks have a broad appliance in many different areas, such as community detection [25] and content recommendation [26]. A random walk is a stochastic process with random variables that can be considered as $W^1_{v_i}, W^2_{v_i}, \ldots, W^k_{v_i}$ such that $W^k_{v_i} + 1$ is a randomly chosen node from v_k neighbors, where W_{v_i} indicates the root node v_i of a random walk. A set of random walks with dedicated length allows us to capture the local structure of the graph. The main advantage of this method is its parallelizability. It is relatively easy to explore the graph with an arbitrary set of random walks running simultaneously in different threads. Moreover, it allows adapting small changes to the graph structure without the need for recalculation.

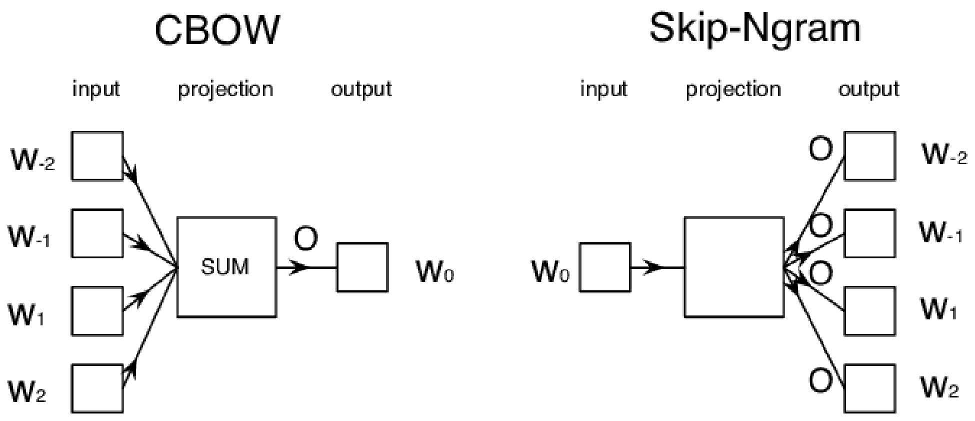

Figure 2.7: CBOW and Skip-Gram

Many different types of a random walk are of interest, which varies in several ways. Below we will describe the best currently used methods. We will first introduce the reader to the NLP *word2vec* models and then describe two graph-based models *DeepWalk* and *node2vec*.

Word2Vec

In 2013, Tomas Mikolov [27] introduced an approach to capture the linguistic context of the words. Given a large corpus of words, it produces a multidimensional vector space model that typically consists of several hundred dimensions. Each vector becomes associated with a corresponding word from the given corpus. The vectors are arranged in a way that words with a similar context in the corpus are consequently placed in close range to each other in the vector space model.

Word2Vec is highly computationally efficient and comes in two algorithmically similar models: Skip-Gram and Continuous Bag-of-Words (CBOW). Given a vocabulary V, the concept of the two methods is to map the input sentence into a one-hot-encoded vector representation, that is a vector of size V, where each word is represented as a V-dimensional vector with $|V| - 1$ zeroes and one 1.

CBOW model shown in Figure 2.8 attempts to predict a word from the context of a word, given as input. The word embeddings are formed from the hidden layer of the network with a significantly lower dimension compared to one-hot encodings, however still able to hold all the necessary information for fitting word detection.

Skip-Gram model does the opposite as it tries to predict surrounding context words from the target words. According to Mikolov [27] both methods have their advantages as well as disadvantages. *Skip-Gram* tends to perform better with a smaller amount of data and

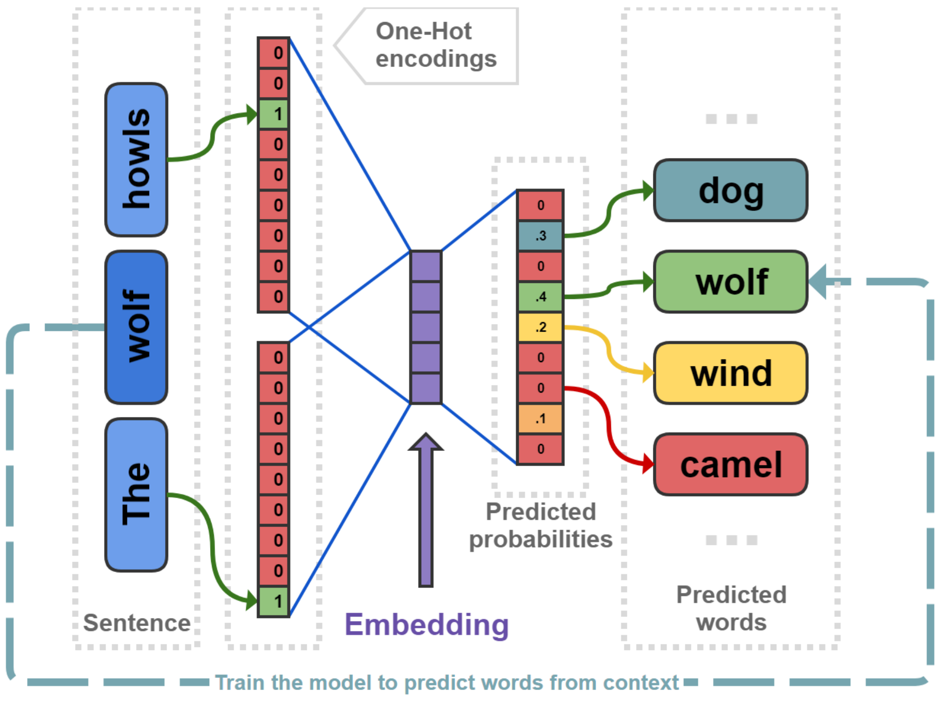

Figure 2.8: CBOW architecture proposed by Mikolov et al. [27]

can represent rare words well, whereas *CBOW* has a better performance and is better at representing more common words.

The architecture of *Skip-Gram* shown in Figure 2.9 is comparable to the auto-encoders one. Skip-Gram compresses the input vector to reduced density vector and outputs the probability distribution of target words. Skip-Gram uses a neural network for training, and the architecture can be represented in 3 layers:

- **Input layer**. The input layer contains a corresponding amount of neurons to the words in the vocabulary.
- **Hidden layer**. A standard fully connected layer with word embeddings as weights.
- **Output layer**. Contains the output probabilities for the target words from the vocabulary.

According to Mikolov [27], the Skip-Gram model defines the probability function as:

$$p(w_{c,j} = w_{O,c}|w_I) = \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(u_{j'})} \tag{2.12}$$

where $w_{c,j}$ denotes the j-th word predicted on the c-th panel of the output layer; w_O represents the actual c-th word present on the c-th context position; w_i denotes the input word;$u_{c,j}$ denotes the input of the j-th value on the c-th context position.

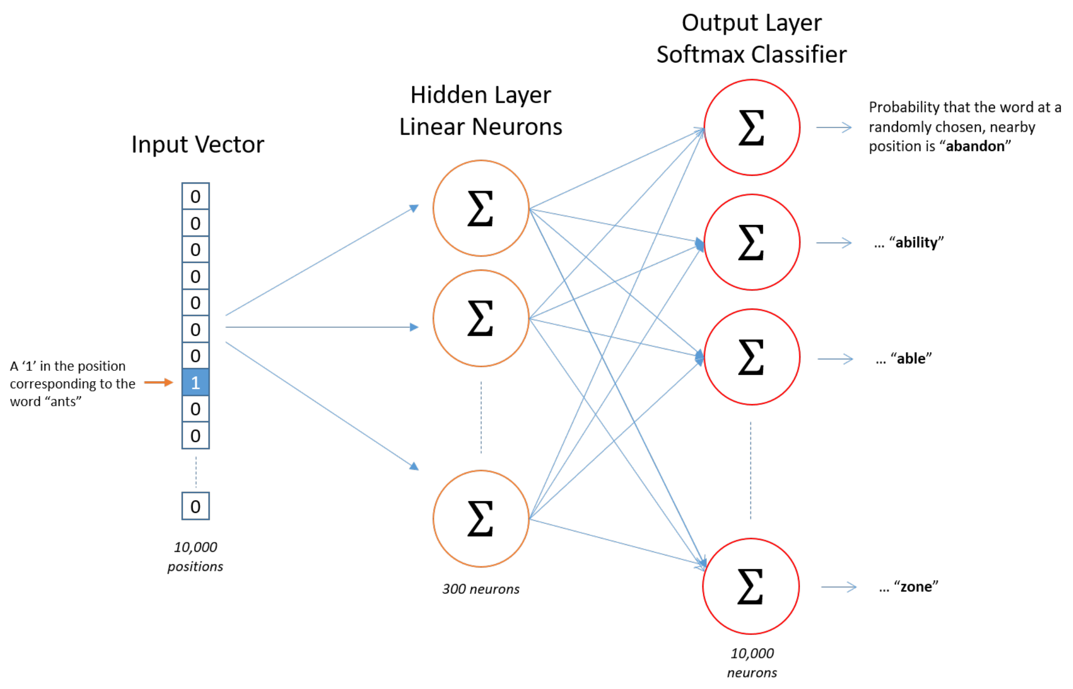

Figure 2.9: Skip-Gram architecture

The training algorithm of Word2Vec includes *Hierarchical Softmax* and/or *Negative Sampling*, however in most cases just *Negative Sampling* is used. We will briefly describe the main principles of these methods based on the detailed derivations and explanations proposed by Rong [28].

- **Hierarchical Softmax.** Hierarchical softmax is an improved method for computing the softmax function defined in Equation (2.12). This method represents the words of the vocabulary as a binary tree. See Figure 2.10 for an example tree.

  Hierarchical Softmax does not have vector representations of words, instead for every inner unit $n(w, j)$ there is an output vector representation $\boldsymbol{v}'_{n(w,j)}$. It is subject to the probability function of a word being the output word defined as:

$$p(w = w_O) = \prod_{j=1}^{L(w)-1} = \sigma\left([\![n(w, j+1) = ch(n(w, j))]\!] \cdot {\boldsymbol{v}'_{n(w,j)}}^T \boldsymbol{h} \right) \tag{2.13}$$

  where ch represents the left child of unit n; h is the output value of the hidden layer; $[\![x]\!]$ as a special function defined as:

$$[\![x]\!] = \begin{cases} 1 & \text{if } x \text{ is true;} \\ -1 & \text{otherwise.} \end{cases} \tag{2.14}$$

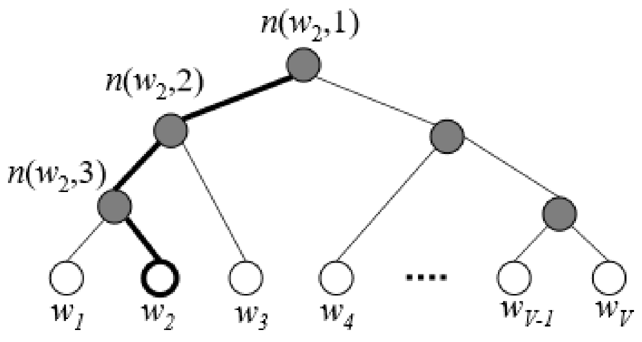

Figure 2.10: A binary tree for the hierarchical softmax. An example path from the root node to the word w_2 is highlighted, where the white dots on the path represent words, and the gray dots are the inner units.

- **Negative Sampling.** Negative sampling is used to differentiate between fake and real signal by means of logistic regression. The motivation behind negative sampling is to randomly sample k negative samples from a noise distribution P_N and compute the probabilities instead of summing over the entire vocabulary.

  Negative sampling objective for a single observation is calculated as follows :

$$\log p(w|w_I) = \log \sigma({v'_w} \cdot {}^T v) + \sum_{i=k}^{K} E_{w_i \sim P_n(w)} [\log \sigma(-{v'_w} \cdot {}^T v_{w_I})] \tag{2.15}$$

  In order to distinguish fake data from real, the noise distribution function is used. Noise distribution is defined as:

$$P(W_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n}(f(w_j)^{3/4})} \tag{2.16}$$

  where f_w denotes the frequency of the word appearing in the corpus; 3/4 is a value obtained from carrying out experiments.

DeepWalk

DeepWalk proposed by Perozzi et al. [29] appeared as the first algorithm to learn latent representations of the graph in an unsupervised manner. The idea behind DeepWalk is to use some of the advantages of language modeling to perform graph modeling. DeepWalk uses short random walks to learn a latent representation of the network, which encodes community structure and makes it applicable for basic classification models. After exploring attainable datasets, the author concludes that the distribution of a word in a corpus and node in a graph follows a power-law distribution[19] as shown in Figure 2.11.

[19]Power law distribution: https://en.wikipedia.org/wiki/Power_law

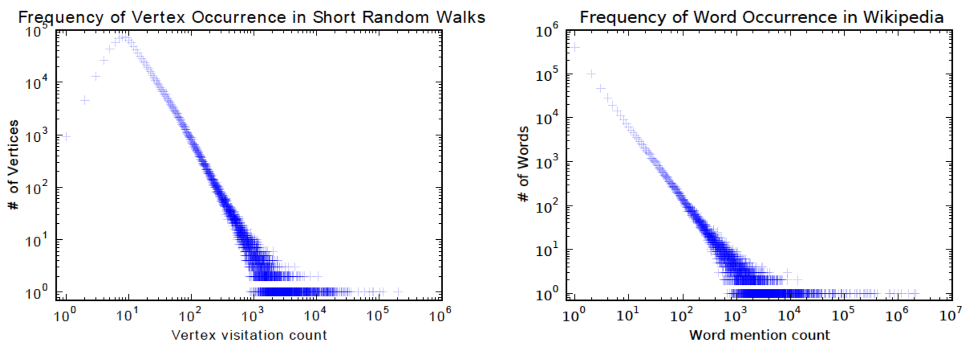

Figure 2.11: The power-law distribution of modes in truncated random walks compared to the distribution of words in natural languages proposed by Perozzi et al. [29]

DeepWalk is based on Skip-Gram architecture. Hence it tends to predict the neighbors of a given node. The algorithm consists of two main stages: a random walk generator and an update procedure. A random walk generator traverses the network to derive local structure based on the neighboring connections. An update procedure involves the Skip-Gram model that learns the generated embeddings. The algorithm is defined as follows:

- **Local structure discovery.** The random walk generator streams random truncated walks W_{v_i} of length l from a randomly chosen source node v_i to generate random embeddings. For every step in the sequence of a random walk, the next node is uniformly sampled from the neighbors of the previous node.
- **Skip-Gram.** Skip-gram attempts to maximize the similarity of embeddings of nodes that appear in the same walks.

  First, random vectors of dimension d are generated. In consequence, the node embeddings are updated by gradient descent throughout the collection of random walks in order to maximize the probability of the neighboring nodes. This is obtained with the softmax function defined in Equation (2.12). Finally, optimization can be applied with additional passes over the same walk path.

node2vec

Similar to the *DeepWalk* algorithm, *node2vec* [30] tends to maintain higher-order proximity between vertices in the graph, obtained through maximizing the probability of appearance of consequent nodes in a truncated random walk. The main difference between these two methods is that *node2vec* attempts to control the path sampling process, instead of sampling the paths randomly. The proposed algorithm employs the right balance

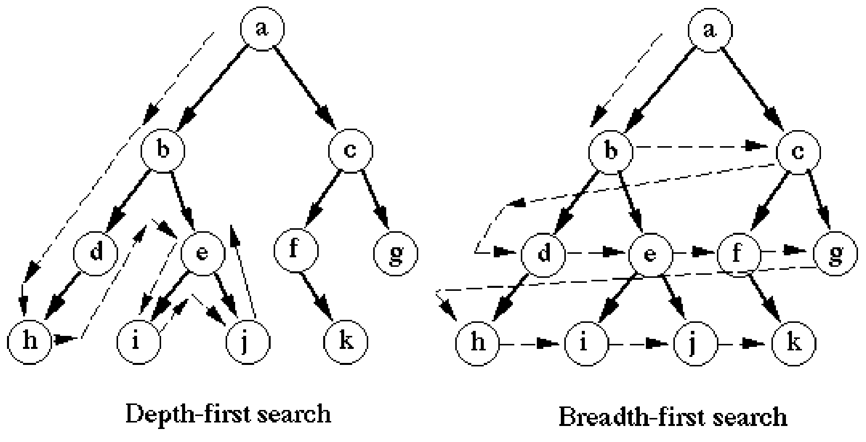

Figure 2.12: Depth-first and Breadth-first graph traversing algorithms.

between two graph search methods such as breadth-first search (BFS) and depth-first search (DFS), which results in a higher quality embeddings than produced by*DeepWalk*.

BFS and DFS graph search methods shown in Figure 2.12 can be described as follows:

- **Breadth-first search** is an graph traversal algorithm that traverses the network layer-wise. Given a network G and a starting node v the algorithm first traverses nodes adjacent to v at the present depth level, before moving to the next depth level.
- **Depth-first search** is another graph traversal algorithm that traverses a tree depth-wise. Given a network G and a starting node v, the algorithm traverses to the maximum depth of the tree before backtracking.

Node2vec aims to solve one of the biggest problems in network analysis: create embeddings for edges and vertices in a task-independent way. In order to accomplish that goal, the algorithm can be defined with three main stages:

1. **Calculation of transition probabilities for every edge in the network.** The edge weights are calculated to produce a *biased random walk* of length l. Assuming that c_i represents the i^{th} vertex in the walk path, then vertices are chosen according to the following distribution:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\alpha_{pq} \cdot w_{vx}}{Z} & \text{if } (v, x) \in E; \\ 0 & \text{otherwise.} \end{cases} \tag{2.17}$$

The *search bias* is defined as α and relies upon parameters p and q.

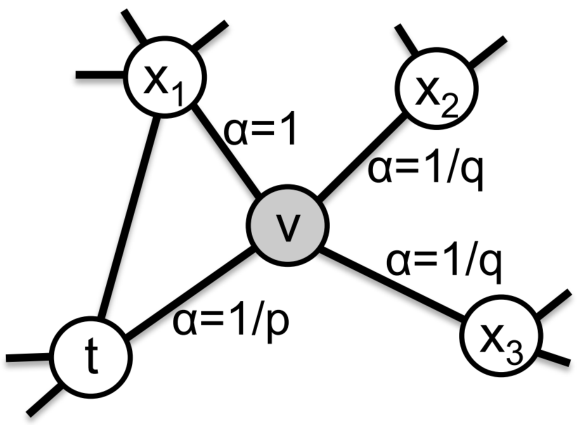

Figure 2.13: 2^{nd} order random walk with parameters p and q. Figure 2 of Grover and Leskovec [30] : *node2vec: Scalable Feature Learning for Networks.*

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0; \\ 1 & \text{if } d_{tx} = 1. \\ \frac{1}{q} & \text{if } d_{tx} = 2. \end{cases} \tag{2.18}$$

where d_{tx} is the shortest path between nodes t and x. Parameters, p, and q describe the speed of walk exploration. The variation of these parameters allows this method to differentiate between two opposite graph search strategies BFS and DFS. Specifically, the parameter p denotes a *return parameter* that controls the probability of revisiting a vertex during the walk, whereas q stands for the *in-out parameter* that enables interpolation between outward and inward vertexes. The parameter tuning of parameters p and q allows performing the sampling of the neighborhood in a very flexible way. The illustration of a random walk procedure is shown in Figure 2.13.

2. **Biased random walks execution.** For every vertex v, a set of context nodes is sampled from the nodes extracted during the walk in the previous step. Random walks need to be re-executed several times for every vertex, in order to balance the bias.

3. **Optimization problem for node embedding computation.** Vector representations are obtained by maximizing the *log* probability of observing a network context $N_s(v)$ for vertex v, that results in the following feature representation:

$$\max_f \sum_{v \in V} log(P(N_s(v)|f(v))) \tag{2.19}$$

Considering that the probability of observing a neighboring vertex is independent of observing another neighboring vertex given the embeddings of the source vertex, where the source vertex and the neighboring vertex have a proportional effect over each other in the feature space, the optimization problem can be formulated as follows:

$$\max_{f} \sum_{v \in V} log[-\log Z_v + \sum_{n_i \in N_S(v)} f(n_i) \cdot f(v)] \tag{2.20}$$

assuming a per node partitioning function defined as follows:

$$Z_v = \sum_{u \in V} exp(f(v) \cdot f(u)) \tag{2.21}$$

which is approximated using negative sampling technique. Finally, to define the features f, the equation is optimized with a Stochastic Gradient Ascent (SGA) method over the model's characteristics. The final created feature vector $f(v)$ for every node v in the network, can be utilized as input to any Machine Learning models.

Asymmetric Embedding Approach

Even though such graph embedding methods like Deepwalk [29] and Node2Vec [31] employ random walk sampling procedure, neither of these methods is able to capture the asymmetric proximities in both directed and undirected graphs that could be vital in several tasks such as link prediction in social networks and recommendation systems. Furthermore, we will briefly introduce an asymmetric proximity preserving (APP) graph embedding method proposed by Zhou et al. [32] that effectively resolves this issue. This method is based on random walks with restart, that allows stochastic gradient updates only along the forward direction of the walk.

In order to capture the asymmetric proximity each node should serve both as a source node $\vec{s_v}$ and a target node $\vec{t_v}$. A softmax function that consequently defines the probability of the source node u predicting the target node v can be formulated as follows:

$$p(v|u) = \frac{exp(\vec{s_u}, \vec{t_v})}{\sum_{n \in V} exp(\vec{s_u}, \vec{t_n})} \tag{2.22}$$

where V is the set of nodes present in the graph. In addition, the author highlights the need for adopting the Skip-Gram model with Negative Sampling(SGNS) [27] for performance improvements. The objective function is then defined as follows:

$$\log \sigma(\vec{s_u} \cdot \vec{t_v}) + k \cdot E_{t_n \sim P_D}[\log \sigma(-\vec{s_u} \cdot \vec{t_n})] \tag{2.23}$$

where k denotes randomly sampled negative pairs in accordance to a vertex distribution $P_D(N)$. Finally, the global objective function is shown as follows:

$$l = \sum_u \sum_v \#Sampled_u(v) \cdot (\log \sigma(\vec{s_u} \cdot \vec{t_v}) + k \cdot E_{t_n \sim P_D}[\log \sigma(-\vec{s_u} \cdot \vec{t_n})]) \tag{2.24}$$

where $\#Sampled$ is the number of sampled pair (u, v) and $\sigma(x) = \frac{1}{(1+exp(-x))}$ is the sigmoid function.

Algorithm 2.1 PPREMBEDDING (G, α)

```
Input: $G(V, E, W)$, Jumping Factor $\alpha$, Learning rate $\eta$
Output: Embedded Vector of $\vec{s_v}, \vec{t_v}$ for each $v \in V$
 1: Initialize: $\vec{s_v}, \vec{t_v}, \forall v \in V$
 2: for each $v \in V$ do
 3:   for $i = 0; i < \#Sample; i++$ do
 4:     $u = SampleEndPoint(v)$
 5:     $StochasticGradientDescent(v, u, 1)$
 6:     for $j = 0; i < k; j++$ do
 7:       $p = RandomUniform(V)$
 8:       $StochasticGradientDescent(v, p, 0)$
 9:     end for
10:   end for
11: end for
```

Algorithm 2.2 STOCHASTICGRADIENTDESCENT $(v, u, label)$

```
1: $\vec{s_v} = \vec{s_v} - \eta(\sigma(\vec{s_v}, \vec{t_u}) - label) \cdot \vec{t_u}$
2: $\vec{t_u} = \vec{t_u} - \eta(\sigma(\vec{s_v}, \vec{t_u}) - label) \cdot \vec{s_v}$
```

The author describes the sampling and the asymmetric learning strategy in Algorithm 2.1 *SampleEndPoint* samples a walk path p starting from node v and ending at node u assuming a stopping probability α. Proximity of a vertex pair (u, v) can be represented with the inner product of $\vec{s_u}, \vec{t_v}$ as soon as we get a source and target vector of each node.

2.3.3 Deep Learning based Approaches

Despite all the success of Neural networks in machine learning, the original models appeared to be computationally inefficient due to the complexity of training and severe computational requirements. However, it was until recently when Hinton et al. [33] proposed a back-propagation greedy learning algorithm, able to train deeper neural networks that are capable of capturing comprehensive data with highly non-linear structures. This led to a successful appliance of Deep Learning in a variety of tasks such as image recognition, language processing, and speech recognition.

Furthermore, we will describe a method called Structural Deep Network Embedding (SDNE), that tends to capture proximities within the source graph by introducing supervised methods coupled with unsupervised deep auto-encoders.

SDNE

In 2016 Wang et al. [34] introduced Structural Deep Network Embedding (SDNE), a semi-supervised deep model to perform node embeddings able to capture both local and global network structure as well as cope with sparse networks. Notably, the author proposes a model with a multi-layer architecture that is composed of several non-linear functions. The author highlights the non-linearity in the underlying structure of the real-world graph networks, which rules out the possibility to be captured by shallow models. However, the proposed model can preserve the high non-linearity in the graph structure, with a multiple layer composition of non-linear functions capable of mapping the data into a highly non-linear latent space.

The deep architecture of the proposed model consists of:

- **Unsupervised Component** preserves the second-order proximity with an extended version of a traditional *deep auto-encoder* by capturing the neighborhood structure of each vertex.
- **Supervised Component** exploits the first-order proximity as the supervised information to refine the representation of the latent space.

An **auto-encoder** is a neural network that attempts to train the representation of a given dataset in an unsupervised manner. At first, the input vector x is mapped to a hidden representation $z = f(x)$, and then function f is parametrized by the encoder with one or several layers of non-linearity. Hidden representation z is consequently mapped the output $\hat{x} = g(z)$ with the means of decoder network that parametrizes the decoder function g. Similar to the encoder network, a decoder network may be composed of several layers of non-linearity. The parameters are subject to a minimization function:

$$\mathcal{L} = (x, g(f(x))) = ||\hat{x} - x||_2^2 \tag{2.25}$$

Primarily a deep auto-encoder can be thought of as a multi-layer neural network. In Figure 2.14, we present the architecture of a deep auto-encoder with multiple layers of

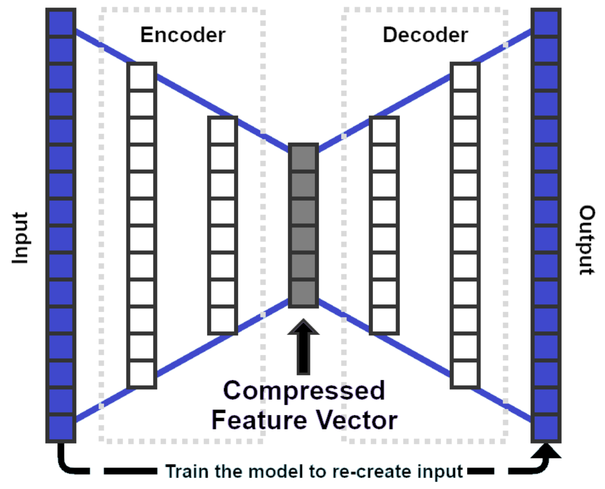

Figure 2.14: An example of an auto-encoder with multiple hidden layers.

non-linearity. The encoder attempts to compress the input data into a feature vector. In contrast, a decoder does the opposite and tries to reconstruct the original data from the given compressed representation with minimal loss.

Finally, the author claims that although minimizing the reconstruction loss does not explicitly preserve the similarity between samples, the reconstruction criterion can smoothly capture the data manifolds and thus preserve the similarity between samples.

Moreover, because real-world networks often contain a vast amount of unobserved edges, an auto-encoder may not create appropriate embeddings, unless modifications are applied. The goal is to archive minimal loss by prioritizing the existing links. Consequently, the author proposes to impose an additional penalty cost to the reconstruction error of the non-zero elements than that of zero elements. The resulting objective function is shown as follows:

$$\mathcal{L}_{2nd} = \sum_{i=1}^{N} ||(\hat{x}_i - x_i) \odot b_i||_2^2 = ||(\hat{X} - X) \odot B||_F^2 \tag{2.26}$$

where $\mathcal{L}$ defines the loss of the auto-encoder, which tends to predict $\hat{x}_i$ with edge weights from node i from the weight matrix W as x_i; $\odot$ stands for Hadamard product for matrices, and b is defined as follows:

$$b_i = \begin{cases} 1 & \text{if } w_{ij} = 0; \\ \beta & \text{if } w_{ij} > 0. \end{cases} \tag{2.27}$$

Considering the modifications made to the proposed **unsupervised component**, it is now able to capture the global network structure by reconstructing the second-order proximity between nodes. However, it is insufficient to capture the global network structure alone, so, the author proposes a **supervised component** to capture the local structure of the network additionally. The loss function of the **supervised component** is defined as follows:

$$\mathcal{L}_{1st} = \sum_{i,j=1}^{n} s_{i,j}||(y_i^{(K)} - (y_j^{(K)})||_2^2 = \sum_{i,j=1}^{n} s_{i,j}||(y_i - y_j)||_2^2 \tag{2.28}$$

To define the objective function in Equation (2.28), the author borrows an idea from Laplacian Eigenmaps [24], which applies a penalty to similar nodes that appear to be far off in the embedding space. Finally, the first-order proximity is preserved by incorporating the idea in the deep model.

The **semi-supervised** deep model architecture shown in Figure 2.15 eventually represents a combination of supervised and unsupervised models and is defined by the following loss function:

$$\begin{aligned}\mathcal{L}_{mix} &= \mathcal{L}_{2nd} + \alpha\mathcal{L}_{1st} + v\mathcal{L}_{reg} \\ &= ||(\hat{X} - X) \odot B||_F^2 + \alpha \sum_{i,j=1}^{n} s_{i,j}||(y_i - y_j)||_2^2 + v\mathcal{L}_{reg}\end{aligned} \tag{2.29}$$

where $\mathcal{L}_{reg}$ represents an $\mathcal{L}2 - norm$ regularizer that prevent over fitting which is defined below:

$$\mathcal{L}_{reg} = \frac{1}{2}\sum_{k=1}^{K}(||W^{(k)}||_F^2 + ||\hat{W}^{(k)}||_F^2) \tag{2.30}$$

where K is the amount of network layers of the encoder or decoder and $W^{(k)}$,$\hat{W}^{(k)}$ are the parameters of the network in the k^{th} layer. Recall, that $||A||_F$ is the Frobenius norm[20].

Furthermore, the author claims that the model suffers from local optima in the parameter space due to the high non-linearity of the model. And in order to address this issue, the author resorts to utilize Deep Belief Network proposed by Hinton et al. [33] to initially pretrain the parameters. Finally, the algorithm of the semi-supervised model can be described as follows:

[20]Frobenius Norm: https://www.sciencedirect.com/topics/engineering/frobenius-norm

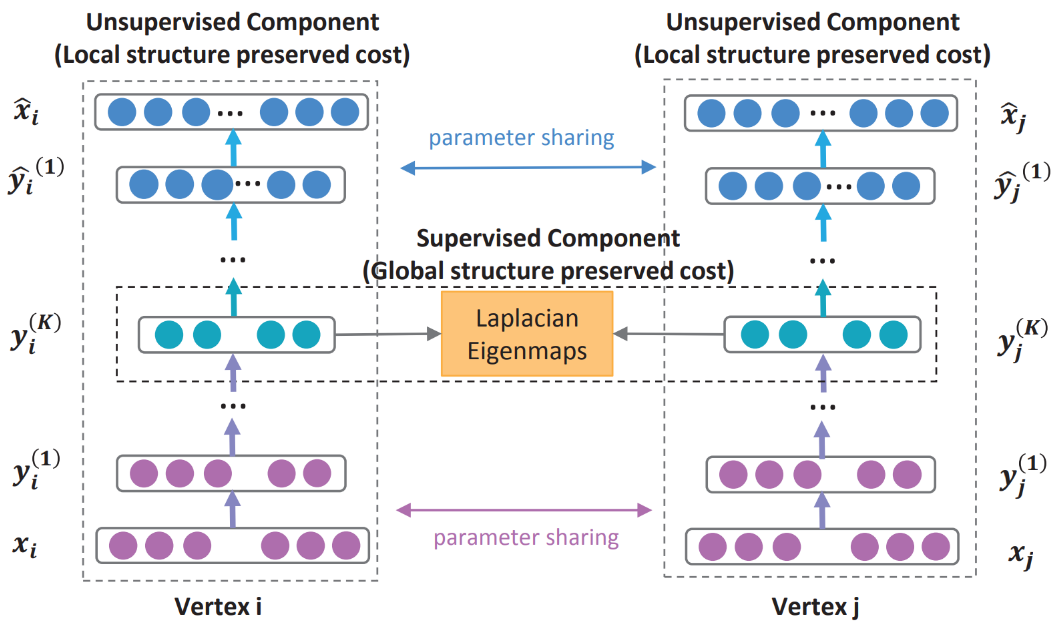

Figure 2.15: The architecture of the semi-supervised model of SDNE – Figure 2 of Hinton et al. [33]

Algorithm 2.3 Training Algorithm for the semi-supervised deep model of SDNE

```
Input: the network $G = (V, E)$ with adjacency matrix $S$,
    the parameters $\alpha$ and $v$
Output: Network representations $Y$ and updated Parameters: $\theta$
1: Pretrain the model through deep belief network to obtain
   the initialized parameters $\theta = \{\theta^{(1)}, ..., \theta^{(K)}\}$
2: $X = S$.
3: repeat
4:     Based on parameters $X$ and $\theta$, obtain $\hat{X}$ and $Y = Y^K$.
5:     Compute the loss function: $\mathcal{L}_{mix} = \mathcal{L}_{2nd} + \alpha\mathcal{L}_{1st} + v\mathcal{L}_{reg}$.
6:     Back-propagate through the entire network to get updated parameters $\theta$.
7: until Convergence
8: Obtain the network representations $Y = Y^{(K)}$
```

2.3.4 Other approaches

Furthermore, we will highlight an additional approach that doesn't fall into any class of the taxonomy but is still suitable for our purposes.

2.3.5 Conclusion

Throughout this chapter, we have described common graph embedding approaches with several representative algorithms for each approach. We can conclude that each model is unique and attempts to improve the process of creating graph embeddings in different aspects.

First, we presented two factorization-based approaches, namely Locally Linear Embedding and Laplacian Eigenmaps, where both methods attempt to employ numerical methods of adjacency matrix decomposition. Secondly, we described random walk based methods, namely node2vec, APP and DeepWalk, which heavily rely on the NLP Skip-Gram model. Finally, we introduced a deep learning based approach SDNE, which incorporates a deep auto-encoder in its architecture.

Chapter 3

Implementation

In this chapter, we introduce unsupervised random walk methods based on the unifying framework described in Khosla et al. [35]. We will analyze currently available approaches and propose improvements in accordance with recent studies in the field of deep learning. Finally, we will describe our approach of creating latent representations for unweighted graphs and dive deeper into the implementation part describing the algorithms. In an attempt to clearly present the implementation details for both APP and DeepWalk models, we will describe each of them in a separate section of this chapter.

3.1 Method modifications

In this section, we will describe how we intend to reduce the time complexity of such graph embedding methods as APP and DeepWalk. We will present our suggested modifications for these methods as well as reason about our hypothesis.

3.1.1 APP

The simulation of a random walk, or more generally a Markov chain, is a fundamental algorithmic paradigm in generating training data for many proposed graph representation learning approaches Tang et al. [20], Perozzi et al. [29], Tsitsulin et al. [36], Zhou et al. [37], Grover and Leskovec [31]. Random walks sample vertices from a given vertex neighborhood, local and global, to collect training data for downstream training of node embeddings. A standard procedure is to initiate multiple random walks from each neighborhood to explore their neighborhoods. Consequently, vertices are sampled from each random walk as *context* vertices. Most of the unsupervised approaches then use the source-context vertex pairs as training data for learning source vertex representation or embedding, including APP.

Given a graph $G = (V, E)$, we are interested in learning low dimensional representations of each node $v \in V$ such that similar nodes in V are embedded closer. We call C as the *context graph* and for each edge $(u, v) \in E'$, v is called the *context* of node u. Let C denote the corresponding adjacency matrix of $\mathcal{C}$ with $c_{i,j}$ denoting $(i, j)th$ element in C. For an edge $(u, v) \in E'$, we call u as the *source* node and v its *context.*

As each node can be a source or a context of some other node, we denote the *source* and *context* representations of nodes in $\mathcal{C}$ by $\Phi \in \mathbb{R}^{|v|} \times \mathbb{R}^d$ and $\theta \in \mathbb{R}^{|v|} \times \mathbb{R}^d$ respectively. For any node v_i, Φ_i and θ_i represent respectively its d-dimensional source and context vectors (representations). We are then interested in learning Φ and θ while minimizing the following loss function:

$$\mathcal{J} = -\sum_{i,j} c_{i,j} \cdot f(\Phi_i, \theta_j) \tag{3.1}$$

where, f is monotonically increasing in Φ_i, θ_j.

We recall that by our construction, for any two dissimilar nodes, $i, j, c_{i,j} = -1$. This imposes an additional constraint on the embedding vectors such that embedding vectors' corresponding dot product is minimized for dissimilar nodes. Note that by minimizing the loss function in Equation 3.1, a vertex (in its source representation) will be embedded closer to its context (in its context representation); and, therefore, two vertices sharing the same context will be embedded closer (in their source representations) by transitivity.

Generally, we implement a novel approach to random walk sampling. We are achieving the effect of multiple random walks in a single one with the help of the Breadth-First Search algorithm, that simultaneously generates context pairs while traversing the graph.

Furthermore, we will present the algorithm in the form of a pseudo-code to clearly convey our hypothesis.

Given a network $G(V, E)$, budget γ and termination factor ε as input to the modified APP Algorithm 3.1 we initially perform a uniform permutation of the vertices V and initiate a queue Q with the a single vertex v_1, where $v_1 \in \mathcal{O}$. Consequently, the algorithm performs a *RandomWalk* sampling for every $v_s \in V$, where v_s denotes the source node of a random walk procedure.

In Algorithm 3.2 a random walk sampling procedure starts with retrieving the first element v_0 of the queue Q and proceeds to obtain its neighbors N_{v_i}. In line 4 the

Algorithm 3.1 Modified-APP(G, ε, γ)

```
Input: Network Graph: $G(V, E)$
    Budget: $\gamma$
    Termination Factor: $\varepsilon$
Output: Context Graph $\to C$
1: $\mathcal{O} = Shuffle(V)$
2: $Q = Queue(\mathcal{O}_v)$
3: for $v_s \in \mathcal{O}$ do
4:     $C_{v_s} = RandomWalk(G, \varepsilon, \gamma)$
5: end for
```

Algorithm 3.2 RandomWalk(G, ε, γ)

```
1: while $Q \neq \emptyset$ do
2:     $v_0 = Pop(Q)$
3:     $N_{v_i} = GetNeighbors(v_0)$
4:     $\gamma_n = \left\lfloor \frac{\gamma}{N_{v_0}} \right\rfloor$
5:     for $n_i \in N_{v_0}$ do
6:         $\gamma_{t \to n_i} = \gamma_n + Select(\frac{\gamma - \gamma_n}{|N-(t)|})$
7:         $P_{n_i} = \lfloor \varepsilon \cdot \gamma_{t \to n_i} \rfloor$
8:         $Pairs(n_i) \leftarrow (v_s, n_i, P_{n_i})$
9:         if $rand < \varepsilon$ then
10:            $Q = Queue(n_i)$
11:        end if
12:    end for
13: end while
```

algorithm computes the budget γ_n that is to be shared between neighboring nodes of node v_0. In line 7 the algorithm estimates the probability P_{n_i} that the random walk would be terminated at node n_i. Consequently, for every neighbor n_i in the set of neighbors N_{v_0} we compute *context pairs* $Pairs(n_i)$ given a start node v_s and a target node n_i. Finally, node v_i is appended to the queue Q if the termination factor ε is grater than a uniformly sampled value $rand$. The procedure then repeats itself while $Q \neq \emptyset$.

3.1.2 DeepWalk

Deep Walk [29] uses local information obtained from truncated random walks to learn latent representations by treating walks as the equivalent of sentences. Deep Walk consists of three sequential stages that are performed to build latent representations of a graph.

- **Random walk generation.** Given a graph G, the random walk generator samples uniformly a random vertex v_i as the start node of the random walk W_{v_i} and then uniformly traverses the graph through neighboring nodes until a predefined length is reached.

- **Skip-Gram** Slides a window of length $2w + 1$ over the random walk W_{v_i} mapping the central vertex v_1 to its representation $\Phi(v_1)$.
- **Hierarchical Softmax.** Applies Hierarchical Softmax as the output activation function.

In this thesis work, we focused on devising an algorithm that would synchronize random walk generation and skip-gram operation into one unit, potentially improving the algorithm's time complexity.

Furthermore, we will present the main components of our approach in the form of pseudo-code. We will additionally show several variants of our approach and describe their advantages.

Algorithm 3.3 Modified-DeepWalk(G, w, γ, t)

```
Input: Network Graph: $G(V, E)$
    Budget: $\gamma$
    Walk Length: $t$
    Window Size: $w$
Output: Context Graph $\rightarrow C$
1: $\mathcal{O} = Shuffle(V)$
2: $Q = Queue(v_1)$
3: for $v_s \in \mathcal{O}$ do
4:     $C_{v_s} = RandomWalk(G, \gamma, t, w)$
5: end for
```

Algorithm 3.4 RandomWalk(G, γ, t, w)

```
1: while $Q \neq \emptyset$ do
2:     $v_0 = Pop(Q)$
3:     $N_{v_i} = GetNeighbors(v_0)$
4:     $\gamma_n = \left\lfloor \frac{\gamma}{N_{v_0} - (t)} \right\rfloor$
5:     for $n_i \in N_v$ do
6:         $W_{n_i} = Window(v_s, n_i, w, \gamma_n)$
7:         $C_{n_i} = UpdateContextGraph(W_{n_i})$
8:         $Q = Queue(n_i)$
9:     end for
10: end while
```

In Algorithm 3.3 we represent a process of building a context graph with the means of short truncated random walks, namely called *random walk generator*. In lines 1-5, we show the main part of the algorithm. First, we create a randomly permuted set of nodes $\mathcal{O}$ and a queue Q needed for the BFS graph traversal technique. The loop runs a $RandomWalk$ for each node in the set given the parameters γ, t, w as input.

In Algorithm 3.4, we showcase the core part of the random walk procedure. While the queue Q is not empty, we pop the first node, query its neighbors and compute the budget b_n that is to be shared between them. Furthermore, for every neighbor n_i we construct a window W_{n_i} with the given parameters of window size w, neighbor itself n_i and the budget for this neighbor b_n. The method *UpdateContextGraph* generates the context pairs in a similar way to Skip-Gram 3.1.2 given the source node v_s, a target node n_i, window size w and the budget γ_n. Finally, according to BFS, the neighbor node n_i is appended to the end of the queue Q, assuming that the walk length t is not reached and the random walk traversal repeats itself while $Q \neq \emptyset$.

3.1.3 Approximated DeepWalk

In Algorithm 3.5, we describe our approach to approximate the algorithm for the purpose of performance improvements. The algorithm for the approximated version remains to be as we described in Algorithms 3.3 3.4 except for the added sampling functionality.

The main idea of is that every edge in the network maintains a local network of its neighborhood, obtained either from a truncated random walk or being sampled from other nodes. We intend to bypass a random walk assuming that there is a possibility to sample the context pairs for node n_i from local storage of another node S_{v_i} encountered while traversing the network. While this approach does not guarantee a better classification accuracy, it significantly improves the running time of the algorithm.

Algorithm 3.5 Sample(S_{v_i}, v_i)

```
Input: Local Storage: $S_{v_i}$, Vertex $v_i$
Output: Context Pairs: $C_{v_i}$
1: if $v_i \in S$ then
2:     $C_{v_i} \leftarrow$ Uniformly sample context pairs from: $S_{v_i}$
3:     $S_{v_i} \leftarrow UpdateLocalStorage(C_{v_i})$
4: end if
```

3.1.4 Parallelized DeepWalk

In an attempt to explore potential performance improvements, we incorporate parallelization into our algorithm. The algorithm for the parallelized version remains to be as we described in Algorithms 3.3 3.4 except for the added parallelization functionality.

Since every node $v \in G$ requires a separate random walk to be performed, the graph exploration part can be relatively easily implemented with a synchronous version using multi-threading. In Algorithm 3.6 we initially specify the amount of threads we intend to use. Then given a the network G to the *starmap* function of the *multiprocessing* library the algorithm executes the $DeepWalk$ function for each node $v \in G$ in a iterable. The item is sent to the function as a parameter. Finally, the *starmap* function maps the generated output C_v of each thread and combines it together into a *Context Graph C* .

Algorithm 3.6 Parallelize(T, G, w, γ, t)

```
Input: Number of threads: $T$
    Network Graph: $G(V, E)$
    Budget: $\gamma$
    Walk Length: $t$
    Window Size: $w$
Output: Context Graph $\rightarrow C$
 1: $p = Pool(T)$
 2: $C = p.starmap(DeepWalk, G)$
```

Chapter 4

Evaluation and Experimental Setup

In the following chapter we will experimentally evaluate the performance of the proposed adjusted models to prove the effectiveness of our modifications. We will first present the running times of our algorithms and then evaluate their performance for the Node recommendation and Multi-label classification tasks on several open-source datasets that contain millions of nodes. Moreover, we will provide the potential users of our algorithms with a detailed technical documentation that contains the necessary installation and setup manuals for different supported environments.

4.1 Data-sets

In this section, we will give a brief overview of the datasets used within our project. Since we mainly focus on representing unweighted graphs, all of the evaluated data sets are undirected, unweighted graphs. We will demonstrate the performance of our algorithms on several multi-label network classification tasks of such social networks as Flickr, Youtube, and BlogCatalog.

Name	BlogCatalog	Flickr	YouTube
$\|V\|$	10,312	80,513	1,138,499
$\|E\|$	333,983	5,899,882	2,990,443
$\|y\|$	39	195	47
Labels	Interests	Groups	Groups

Table 4.1: Graphs used within our experiments

An overview of the graphs we consider in our experiments is given in Table 4.1.

- **Arxiv** [38] represents a collaboration network generated from the w-print arXiv, where the nodes are the authors of the papers and the edges represent cooperations between different authors.
- **Amazon** [38] dataset represents products and co-purchasing relation between products. The original Amazon graph to an undirected graph to represent the co-purchasing relation in a symmetric manner.
- **BlogCatalog** [38] dataset represents a social relationships network presented by blogger authors. The labels represent the topic categories given by the authors.
- **Flickr** [39] dataset represents a network of contacts between users of the photo-sharing platform. Labels are represented as different interest groups.
- **Youtube** [40] dataset represents a social network users of the popular online video-sharing platform. The labels represent groups of users that are interested in similar video genres (e.g., soccer, video games).

4.2 Performance Evaluation

We run our experiments on a distributed system cluster[1]. A distributed systems cluster is a group of machines that are virtually or geographically separated, and that work together to provide the same service or application to clients. The hardware is managed by the Linux operating system and consists of multiple processors(80) with a 2.5 GHz frequency that can operate independently on shared memory.

4.2.1 Embedding generation

In order to estimate the influence of our modifications on the APP and DeepWalk algorithms, we will attempt to reconstruct the graphs with both algorithms, original and modified. Since we perform multiple simultaneous random walks that continuously generates context pairs, the modified versions should have a running time advantage.

In order to certify our hypothesis, we have conducted the multiple experiments with equivalent learning parameters, such as budget, walk length, and alpha parameter, to produce latent representations of three different social network datasets.

In Tables (4.2) (4.3) (4.4) (4.5) we present the modified and original algorithms' running time measurements in the following format HH:MM:SS.SSS, with different dimensions to

[1]Distributed system cluster: https://en.wikipedia.org/wiki/Computer_cluster

highlight the performance improvements. We have repeated the experiments ten times with various random number generator seeds and averaged the running time for each method variation. The presented result validates the effectiveness and efficiency of the proposed modifications to the APP algorithm. We can state that the modified APP indeed has a better time complexity.

Dataset	Dimensions	Algorithm	Random Walk	Training	Total Duration
BlogCatalog	64	Original	00:00:19.22	00:00:00.36	00:00:19.58
		Modified	00:00:00.88	00:00:00.33	00:00:01.21
	128	Original	00:00:20.22	00:00:00.44	00:00:20.66
		Modified	00:00:01.11	00:00:00.43	00:00:01.54
Flickr	64	Original	00:02:23	00:00:02	00:02:26
		Modified	00:00:22	00:00:01.64	00:00:24.61
	128	Original	00:03:09	00:00:04	00:03:13
		Modified	00:00:24	00:00:03	00:00:28
Youtube	64	Original	00:48:37	00:02:04	00:50:41
		Modified	00:25:53	00:02:18	00:28:11
	128	Original	00:53:38	00:03:18	00:56:56
		Modified	00:28:35	00:03:16	00:31:51

Table 4.2: Running time measurements of the APP algorithm in embedding generation task, Undirected graphs

Dataset	Dimensions	Algorithm	Total Duration
BlogCatalog	64	Original	00:25:31
		Modified	00:28:41
Flickr	64	Original	05:52:22
		Modified	06:33:51
Youtube	64	Original	51:03:14
		Modified	49:13:44

Table 4.3: Running time measurements of the DeepWalk algorithm in embedding generation task, Undirected graphs

Dataset	Dimensions	Algorithm	Total Duration
BlogCatalog	64	Original	00:25:31
		Modified	00:20:21
Flickr	64	Original	05:52:22
		Modified	03:14:51
Youtube	64	Original	51:03:14
		Modified	34:13:44

Table 4.4: Running time measurements of the approximated DeepWalk algorithm in embedding generation task, Undirected graphs

Dataset	Dimensions	Algorithm	Total Duration
BlogCatalog	64	Original	00:25:31
		Modified	00:18:41
Flickr	64	Original	05:52:22
		Modified	03:31:55
Youtube	64	Original	51:03:14
		Modified	29:51:01

Table 4.5: Running time measurements of the parallelized DeepWalk algorithm in embedding generation task

4.2.2 Node recommendation with APP

APP tends to outperform traditional methods such as Deepwalk, Line and Node2Vec in node recommendation task, which demonstrated the advantage of preserving asymmetric and higher order proximity. In order to create personalized service like product recommendation it is necessary to calculate the *top-k* feasible candidates for every individual user. Hence, we evaluate the performance of the APP in node recommendation task and present the results in Table (4.6). For experimentation purposes we separate a small fraction of the dataset to create a test set, and use the rest as a training corpus. We evaluate the methods for recommendation tasks with *Precision@k* and *Recall@k* metrics.

Dataset	Algorithm	P@10	R@10	P@20	R@20	P@50	R@50
Amazon	Original	0.091	0.655	0.057	0.82	0.026	0.922
	Modified	0.098	0.649	0.061	0.809	0.028	0.91
Arxiv	Original	0.124	0.701	0.069	0.831	0.034	0.969
	Modified	0.128	0.69	0.07	0.829	0.036	0.955

Table 4.6: Precision and Recall for top-k Node Recommendation with APP, Undirected Graphs

4.2.3 Multi-label node classification with Deepwalk

Multi-label node classification is considered to be an essential task to estimate the effectiveness of a graph embedding. For the evaluation purpose we randomly select 10% to 90% of the nodes as training data and use the remaining part for tests. To verify that we did not significantly decrease the models accuracy we run the evaluation for ten times and provide the results in Tables (4.7) (4.8) (4.9). We also perform the evaluations on the embeddings created with the original algorithm and combine them with our results to facilitate comparison. The measured F_1 scores are indeed close to the original algorithm which proves that our modifications retain initial classification accuracy and improves the time complexity.

Dataset	Metric	Algorithm	10%	30%	50%	70%	90%
BlogCatalog	Micro-F1 (%)	Original	36.00	39.60	41.00	41.50	42.00
		Modified	35.91	39.64	41.21	41.43	41.89
	Macro-F1 (%)	Original	21.30	25.30	27.30	27.90	28.90
		Modified	21.29	25.33	27.41	27.99	28.91
Flickr	Micro-F1 (%)	Original	32.4	35.9	37.2	38.1	38.5
		Modified	32.44	35.88	37.33	38.08	38.6
	Macro-F1 (%)	Original	14.0	19.6	22.1	23.6	24.6
		Modified	14.7	19.5	21.91	23.55	25.01
Youtube	Micro-F1 (%)	Original	37.95	40.08	41.32	42.12	42.78
		Modified	37.88	40.11	41.3	42.21	42.88
	Macro-F1 (%)	Original	29.22	33.06	34.35	34.96	35.42
		Modified	29.31	33.11	34.3	34.91	35.58

Table 4.7: Evaluation of the learned embeddings with DeepWalk on a multi-label node classification task.

Dataset	Metric	Algorithm	10%	30%	50%	70%	90%
BlogCatalog	Micro-F1 (%)	Original	36.00	39.60	41.00	41.50	42.00
		Modified	35.11	37.88	39.21	39.41	40.07
	Macro-F1 (%)	Original	21.30	25.30	27.30	27.90	28.90
		Modified	19.82	23.91	26.1	25.72	26.53
Flickr	Micro-F1 (%)	Original	32.4	35.9	37.2	38.1	38.5
		Modified	30.11	33.8	36.09	36.51	36.98
	Macro-F1 (%)	Original	14.0	19.6	22.1	23.6	24.6
		Modified	12.4	17.81	20.94	22.05	22.87
Youtube	Micro-F1 (%)	Original	37.95	40.08	41.32	42.12	42.78
		Modified	35.65	37.84	38.6	39.41	40.1
	Macro-F1 (%)	Original	29.22	33.06	34.35	34.96	35.42
		Modified	28.12	31.8	32.44	32.89	33.6

Table 4.8: Evaluation of the learned embeddings with approximated Deepwalk on a multi-label node classification task

4.2.4 Discussion

In this section, we will discuss the presented evaluation results and reason about the effectiveness of our modifications.

First, we can state that our modified APP algorithm indeed has a significantly lower running time in embedding generation task compared to the original algorithm (4.2). We

Dataset	Metric	Algorithm	10%	30%	50%	70%	90%
BlogCatalog	Micro-F1 (%)	Original	36.00	39.60	41.00	41.50	42.00
		Modified	35.78	39.41	41.1	41.53	42.31
	Macro-F1 (%)	Original	21.3	25.30	27.3	27.9	28.9
		Modified	21.4	25.41	27.22	27.97	29.02
Flickr	Micro-F1 (%)	Original	32.4	35.9	37.2	38.1	38.5
		Modified	32.46	36.07	37.2	38.14	38.39
	Macro-F1 (%)	Original	14.0	19.6	22.1	23.6	24.6
		Modified	14.1	19.22	21.87	23.45	24.57
Youtube	Micro-F1 (%)	Original	37.95	40.08	41.32	42.12	42.78
		Modified	38.4	40.46	41.5	42.6	42.97
	Macro-F1 (%)	Original	29.22	33.06	34.35	34.96	35.42
		Modified	30.07	33.1	34.87	35.21	35.72

Table 4.9: Evaluation of the learned embeddings with paralellized Deepwalk on a multi-label node classification task. Four threads

conclude that these modifications did not considerably affect the precision of the node recommendation task (4.6).

Furthermore, the evaluations of the modified DeepWalk (4.3) algorithm indicate that our modifications slightly increased the running time for BlogCatalog and Flickr datasets. However, we notice that for larger graphs like Youtube the modifications tend to have a positive effect. While there may be several possible explanations for this result, one possible reason is that *numpy* library widely used within our script generally performs better on large data quantities than the built-in Python's list constructor. Despite the running time differences, we conclude that the multi-label classification accuracy (4.7) remains generally close to each other.

Additionally, our attempt to approximate the random walk procedure of DeepWalk's algorithm shows a decent improvement in running time (4.4), however, this comes with a price of a slight decrease in precision values *Micro-F1* and *Macro-F1* (4.8) for multi-label node classification task.

Finally, we state that the parallelization of our approach to random walk sampling procedure resulted in an improved overall running time of the algorithm (4.5) and maintains the initial DeepWalk's precision values (4.9).

4.3 Technical Documentation

In order to dive deeper into the implementation details, we first describe the libraries and frameworks used within this project. Secondly, we describe the modules of the application and its functionality. Finally, we present our implementation.

4.3.1 Programming Languages and Libraries

The primary programming language used throughout this project is **Python** (v3.6.7). However, we additionally attempt to implement our application in a low-level language such as **C++** for performance purposes.

- **Gensim**[2] Gensim is a NLP package for topic modelling, document indexing and similarity retrieval.
- **NumPy**[3] In order to work with homogeneous arrays we have incorporated a widely used python library Numpy 2 (v 1.16.3). Numpy is written in C, which is a low-level language that makes it very fast for mathematical operations.
- **Networkx**[4] is a package for performing a variety of operations on complex networks, including loading the networks, storing edge weights, studying their structure, dynamics, and functions.
- **Matplotlib**[5] an extensive library for creating statically animated and interactive visualizations in Python, used to create all charts within this project.
- **Multiprocessing**[6] a library that provides both local and remote concurrency, avoiding the Global Interpreter Lock with the means of sub-processes. It is used within this thesis to run the application in multiple threads, consequently improving the running time of the application.

4.3.2 Functional Interface

In this section, we will present the functional interface that our models have in common. Hence we will briefly describe the individual functions of the models and their functionality.

- **Edge List Parser.** This function loads the edge list dataset represented as source node id, target node id, and transforms it into a suitable format for embedding generation.

[2]Gensim: `https://radimrehurek.com/gensim/`
[3]NumPy: `https://numpy.org/`
[4]Networkx: `https://networkx.github.io/`
[5]Matplotlib: `https://matplotlib.org/`
[6]Multiprocessing: `https://docs.python.org/2/library/multiprocessing.html`

- **Random Walk.** This function provides the random walk functionality. Given a start node s as input along with restart parameter ϵ and the walk length l, the algorithm proceeds to traverse the graph in a BFS fashion simultaneously producing the context pairs.
- **Approximation.** This function provides the approximation ability to the algorithm by enabling the sampling procedure.
- **Miscellaneous.** Contains all sorts of small helper functions required for the algorithm execution.
  - **Store context pair.** Provides the functionality to store the generated context pairs into a dictionary format.
  - **Display measurements.** Provides measurements necessary to estimate the performance of a specific execution.
  - **Write to file.** This function writes the generated context pairs into an *txt* file format suitable for machine learning algorithms that create graph embeddings.

4.3.3 Dependencies

The required dependencies for this project are handled with **pip** (v20.1.1) *dependencies.txt* file, which is located at the root of the project. We will provide a full description on how to handle the dependencies and run the models later in this chapter.

4.3.4 Installation

All provided commands should be launched from **Linux BASH**, **GIT BASH** or **Windows Subsystem for Linux**. All scripts should be executed from the project root. The scripts can be found on the project's github page[7].

1. Install requirements:
   - python v3.6 with pip
   - pipenv
   - g++
2. Install dependencies:

```
$ pipenv install --dependencies
```

[7]Scaling Network Embeddings: https://github.com/vladmaksyk/Scaling-Network-Embeddings

4.3.5 Running the models

Modified APP

In order to launch the APP model we first need to generate the executable files. This is done with the following command:

```
$ cd APP-Modified
$ make
```

Command line interface

The parameters of the script can be displayed by directly calling the execution file. Notice that not specifying some of the parameters will result in default values for the corresponding parameter.

```
$ ./cli/app
```

Some of the most important arguments of the script are shown below:

- **-train**<string> Train the Network data;
- **-save**<string> Save the Embedding;
- **-dimensions**<int> Dimension of vertex representation; default equals to 64;
- **-undirected**<int> Specify whether the graph is directed;
- **-window__size**<int> Size of skip-gram window;
- **-walk__times**<int> Amount of walks from the start vertex;
- **-walk__steps**<int> Train the Network data;
- **-threads** <int> Number of training threads;
- **-alpha**<int> Learning rate;

An example of the script usage:

```
$ cd APP-Modified
$ ./cli/app -train example.txt -save embeddings.txt
-undirected 1 -dimensions 64 -walk_times 10 -walk_steps 40
-window_size 10 -alpha 0.025 -threads 1
```

Modified DeepWalk

For simplicity, we have combined all three variations of our modified DeepWalk algorithms into one script.

- Exact
- Approximate
- Parallelized

The user can switch between the version with the use of command line arguments described below.

Command line interface

Some of the most important arguments of the script are shown below:

- **-type**<string> Variation of DeepWalk
  - exact
  - approximate
  - parallelized
- **-format**<string> Format of the network dataset;
  - mat
  - edgelist
- **-input**<string> Input dataset;
- **-number_walks**<int> Amount of walks from the start vertex;
- **-representation-size**<int> Dimension of vertex representation;
- **-walk-length**<int> Length of a walk in nodes;
- **-window_size**<int> Size of skip-gram window;
- **-workers** <int> Number of training threads;
- **-output**<string> Save generated embeddings;

An example of the script usage:

```
$ cd DeepWalk-Modified
$ ./src/deepwalk --type approximate --format edgelist
--input example.txt --number-walks 80
--representation-size 64 --walk-length 40 --window-size 10
--workers 1 --output example.embeddings
```

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Throughout this thesis work, we have performed research on the currently available approaches to creating latent representations of undirected unweighted graphs. In accordance with the recent progress in the field of deep learning, we successfully attempted to improve the time complexity of such methods as APP and Deepwalk while retaining their initial accuracy for advanced analytical tasks such as node recommendation and multi-label node classification.

Firstly, we introduced the reader to the conventional methods of graph representation, such as adjacency matrices, adjacency lists, and lists of edges. Consequently, we described the main principle behind the concept of creating graph embeddings, that intend to obtain the graph structure by creating a graph vector $\Phi(v) \in \mathcal{R}^d$ for every node v in the graph G, such that $d << |V|$, and thus generate appropriate input that can be used for further processing by different machine learning models.

Later, we concentrated on the real-world networks that tend to have missing links. This problem prompted our focus on capturing both local and global roles of a specific node in the graph, with the means of first and second-order proximities.

Furthermore, we provided a detailed description of currently available methods for graph embeddings generation. First, we introduced the reader to the factorization based approach Laplacian Eigenmaps. Secondly, we described random walk based methods DeepWalk, Node2Vec, and APP, and, finally, we presented a deep-learning-based approach SDNE. We analyzed the underlying structure for each of the described approached and presented a detailed description of the graph embeddings learning process.

Subsequently, we described our contribution within this thesis work. Particularly, we describe the improvements made to the APP and DeepWalk algorithms. By introducing a novel approach of random walk sampling we archived as decent enhancement in the running times of the algorithms.

We then perform an extensive evaluation of both the original and modified algorithms on various social network datasets. First, we estimated the embedding generation running times of each algorithm an presented the results in a convenient for comparison format. Secondly, we evaluated the performance of the modified APP algorithm on the node recommendation task with appropriate metrics such as *Precision@k* and *Recall@k*. Thirdly, we conducted multiple experiments with various DeepWalk modification on the multi-label node classification task and measured the accuracy with F_1 score. Finally, we provided the potential users of our software with useful technical documentation that contains detailed setup manual.

5.2 Future Work

Nevertheless, since most of the conventional methods focus on creating embeddings with undirected graphs, we thus have not attempted to create latent representations for directed graphs. Consequently, we consider adding this capability to our methods as potential future work. Another possible modification is to add distributed message-passing functionality to our algorithms. This functionality enables parallelization between different machines as a sequence improving the scalability of the algorithms.

List of Figures

1.1 Rating matrix as a Bipartite Graph 3

2.1 Unweighted undirected graph 10
2.2 Data mining taxonomy 12
2.3 One-Hot representation. Image source: `https://subscription.packtpub.com/book/web_development/9781786465825/3/ch03lvl1sec32/encoding-and-embedding`
2.4 Example of an Artificial Neural Network 15
2.5 Training scheme of a Neural Network. 16
2.6 Confusion Matrix 17
2.7 CBOW and Skip-Gram 20
2.8 CBOW architecture proposed by Mikolov et al. [27] 21
2.9 Skip-Gram architecture 22
2.10 A binary tree for the hierarchical softmax. An example path from the root node to the word w_2 is highlighted, where the white dots on the path represent words, and the gray dots are the inner units. 23
2.11 The power-law distribution of modes in truncated random walks compared to the distribution of words in natural languages proposed by Perozzi et al. [29] 24
2.12 Depth-first and Breadth-first graph traversing algorithms. 25
2.13 2^{nd} order random walk with parameters p and q. Figure 2 of Grover and Leskovec [30] : *node2vec: Scalable Feature Learning for Networks*. 26
2.14 An example of an auto-encoder with multiple hidden layers. 30
2.15 The architecture of the semi-supervised model of SDNE – Figure 2 of Hinton et al. [33] 32

List of Tables

4.1 Graphs used within our experiments 41
4.2 Running time measurements of the APP algorithm in embedding generation task, Undirected graphs 43
4.3 Running time measurements of the DeepWalk algorithm in embedding generation task, Undirected graphs 43
4.4 Running time measurements of the approximated DeepWalk algorithm in embedding generation task, Undirected graphs 43
4.5 Running time measurements of the parallelized DeepWalk algorithm in embedding generation task 44
4.6 Precision and Recall for top-k Node Recommendation with APP, Undirected Graphs 44
4.7 Evaluation of the learned embeddings with DeepWalk on a multi-label node classification task. 45
4.8 Evaluation of the learned embeddings with approximated Deepwalk on a multi-label node classification task 45
4.9 Evaluation of the learned embeddings with paralellized Deepwalk on a multi-label node classification task. Four threads 46

Bibliography

[1] Pavel Kordík. Machine learning for recommender systems. 2(1), 2018. doi: https://medium.com/recombee-blog/machine-learning-for-recommender-systems-part-1-algorithms-evaluation-and-cold-start-6f696683d0

[2] Vishnu Subramanian. How netflix's ai recommendation engine helps it save $1 billion a year. 1(1), 2019. doi: https://artelliq.com/blog/how-netflix-s-ai-recommendation-engine-helps-it-save-1-billion-a-year/.

[3] Ian MacKenzie, Chris Meyer, and Steve Noble. How retailers can keep up with consumers. 1(1), 2019. doi: https://www.mckinsey.com/industries/retail/our-insights/how-retailers-can-keep-up-with-consumers.

[4] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston. The youtube video recommendation system. *In Proceedings of the fourth ACM conference on Recommender systems*, 6(1):293–296, 2010. doi: https://www.mckinsey.com/industries/retail/our-insights/how-retailers-can-keep-up-with-consumers.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classifcation with deep convolutional neural networks. *In Advances in neural information processing systems*, 4(1), 2010.

[6] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. *In Proceedings of the IEEE international conference on computer vision*, 2:2980–2988, 2017.

[7] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2, 2015.

[8] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *In Advances in neural information processing systems*, pages 3111–3119, 2013.

[9] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[10] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[11] Li Yi, Hao Su, Xingwen Guo, and Leonidas J Guibas. Syncspeccnn: Synchronized spectral cnn for 3d shape segmentation. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2282–2290, 2017.

[12] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. 2018.

[13] Charles C Onu, Jonathan Lebensold, William L Hamilton, and Doina Precup. Neural transfer learning for cry-based diagnosis of perinatal asphyxia. *arXiv preprint arXiv:1906.10199*, 2019.

[14] Huan Ling, Jun Gao, Amlan Kar, Wenzheng Chen, and Sanja Fidler. Fast interactive object annotation with curve-gcn. *In CVPR*, 2019.

[15] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated, first edition, 2014. ISBN 1461471370, 9781461471370.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. 2016. doi: http://www.deeplearningbook.org.

[17] Tyler Elliot Bettilyon. Types of graphs. 1(1), 2019. doi: https://medium.com/tebs-lab/types-of-graphs-7f3891303ea8.

[18] André Platzer, Frank Pfenning, Rob Simmons, Penny Anderson, Iliano Cervesato. Representing graphs. 2(1):1–12, 2018. ISSN 2150-8097. doi: http://www.andrew.cmu.edu/user/hgommers/lec/23-graphs.pdf.

[19] Primož Godec. Graph embeddings. 2(1), 2018. doi: https://towardsdatascience.com/graph-embeddings-the-summary-cc6075aba007.

[20] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, Qiaozhu Mei. Line: Large-scale information network embedding. 2(1):2–3, 2015. doi: https://arxiv.org/pdf/1503.03578.pdf.

[21] Li Deng and Dong Yu. *Deep Learning Methods and Applications.* Foundations and Trends in Signal Processing, Santa Clara, CA, USA, first edition, 2014. ISBN 1932-8346.

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. 2(1), 2013. doi: https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.

[23] Palash Goyal, Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, pages 2282–2290, 2018. doi: http://www.sciencedirect.com/science/article/pii/S0950705118301540.

[24] Mikhail Belkin, Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. 2003. doi: https://www2.imm.dtu.dk/projects/manifold/Papers/Laplacian.pdf.

[25] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. *In Foundations of Computer Science,2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 475–486, 2006.

[26] F. Fouss, A. Pirotte, J.-M. Renders, and M. Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *Knowledge and Data Engineering, IEEE Transactions on*, 19(3):355–369, 2007.

[27] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[28] Xin Rong. word2vec, parameter learning. *arXiv:1411.2738v4*, pages 6–7, 2016.

[29] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[30] Aditya Grover, Jure Leskovec. node2vec: Scalable feature learning for networks. *arXiv:1607.00653v1 [cs.SI]*, pages 2–8, 2016.

[31] Grover A. and Leskovec J. node2vec: Scalable feature learning for networks. *In SIGKDD*, pages 855–864, 2016.

[32] Chang Zhou, Yuqiong Liu, Xiaofei Liu, Zhongyi Liu, Jun Gao. Scalable graph embedding for asymmetric proximity. *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, 2017.

[33] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527—-1554, 2006.

[34] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. *In Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, pages 1225–1234, 2016.

[35] Khosla M., Setty V., and Anand A. A comparative study for unsupervised network representation learning. *IEEE Transactions on Knowledge and Data Engineering.*, 2019.

[36] Tsitsulin A., Mottin D., Karras P., and Müller E. Verse: Versatile graph embeddings from similarity measures. *In The Web Conference*, pages 539–548, 2018.

[37] Zhou C., Liu Y., Liu X., Liu Z., and Gao J. Scalable graph embedding for asymmetric proximity. *In AAAI*, pages 2942–2948, 2017.

[38] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL http://networkrepository.com.

[39] J. McAuley and J. Leskovec. Image labeling on a network: Using social-network metadata for image classification. In *ECCV*, 2012. URL https://snap.stanford.edu/data/web-flickr.html.

[40] L. Tang and H. Liu. Scalable learning of collective behavior based on sparse social dimensions. In *ACM*, 2005.