Eötvös Loránd University

Faculty of Science

Node embedding algorithms and their evaluation through link prediction

BSc Thesis

Katalin Nicole Petro-de Chalendar

Mathematics BSc.

Supervisor:

András Benczúr, Ph.D.

Department of Operational Research

Budapest, 2020.

STATEMENT

Name: Katalin Nicole Petro-de Chalendar

Neptun ID: NDV6LN

ELTE Faculty of Science: Mathematics BSc.

Specialization: Mathematical Analyst Specialization

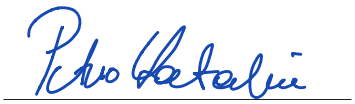
Title of diploma work: Node embedding algorithms and their evaluation through link prediction

As the author of the diploma work I declare, with disciplinary responsibility that my thesis is my own intellectual product and the result of my own work. Furthermore I declare that I have consistently applied the standard rules of references and citations.

I acknowledge that the following cases are considered plagiarism:
– using a literal quotation without quotation mark and adding citation;
– referencing content without citing the source;
– representing another person's published thoughts as my own thoughts.

Furthermore, I declare that the printed and electronical versions of the submitted diploma work are textually and contextually identical.

Budapest, 2020



Signature of Student

Acknowledgement

I wish to express my deepest gratitude to my supervisor, András Benczúr for his guidance throughout the last months, for introducing me to the field of node embeddings and making it possible for me to work on a state-of-the-art topic. His patience, kindness, help and motivation made this entire process very instructive and pleasant.

I am particularly grateful for the assistance given by Ferenc Béres during the implementation part of this dissertation, thanks to his kind help it was fascinating.

I would like to thank my professors at the university for helping me acquire the knowledge I gained in the last four years. I wish to express a special gratitude to István Fekete for introducing me to the world of graph algorithms.

I would like to thank for the invaluable assistance that Kitti Varga provided me during my studies. She inspired me throughout all these years with her knowledge, motivation and kindness.

I wish to pay special regards to Roland Molontay who often gave me guidance during my studies and also helped me choose the field of my thesis work.

I owe a special thank you to Anna Ország as well who helped me a lot at the beginning of this process with a lot of technical information.

My family has been by my side throughout all the years of my studies. They comforted and encouraged me whenever I needed it. I would like to express my great appreciation to my mother and father, the biggest supporters of my dreams and of everything I do, to my grandmother who spent a lot of time with me during this period and was a very big support, to my brother who is always one of my biggest help whatever it comes to and to Maria who gave me a very strong emotional support during good and bad times as well.

Finally, I am particularly grateful for all of my friends from the university and from MCC for making these years so much fun and for always being there for me.

Contents

Abstract

The purpose of this dissertation is to provide insights into the study of node embeddings by machine learning algorithms and to evaluate static node embedding models from Rozenberczki's general purpose community detection and network embedding library Python package `karateclub` through a link prediction task on the Dribbble dataset.

The aim of node embedding is to map each node of a graph to a low-dimensional vector to then use them as inputs for mining and learning algorithms such as link prediction. In this paper, we will present several static and online graph embedding methods such as Laplacian Eigenmaps, DeepWalk, Node2vec, StreamWalk and Online Second Order Similarity.

Keywords: node embedding, Word2Vec, link prediction, dynamic graphs, online node embedding, Dribbble.

1 Introduction

The use of graphs is widespread in many different scientific fields. Social networks are large graphs that represent people's acquaintances, biologists also use them to illustrate protein interactions, but any network that describes relationships or dependencies are representable with graphs. As a result, interest in machine learning on graphs has increased significantly.

Machine learning is a method for building models automatically by learning to identify patterns from data sets. The main idea is that the systems can then make decisions without human assistance. The primary challenge in this area is how to represent graphs, since mathematical and statistical operations on graphs and the direct application of machine learning methods are limited. The new approach consisting of learning how to place graphs in low-dimensional vector spaces has emerged over the past decade. Online machine learning of graphs, which is about evaluating different algorithms in a time-sensitive manner, has also become more widespread. In these models new data points are made available to the model in chronological order.

This dissertation will start with the elaboration of the standard link prediction task followed by the time-variant online link prediction problem. We will then focus on the vector space representation of words, the Word2Vec method that serves as basis for many node embedding methods. When describing node embeddings, we first enumerate methods that work on static graphs. These can be used for dynamic graph embedding as well by considering the sequence of snapshots in time, however it has been shown that time-preserving and online methods that update the node representations at the arrival of every new edge often perform better. These methods learn time-dependent embeddings on temporal networks that include edges with unique timestamps that mark when the given edge is active, see Figure 1.

Definition 1 (Temporal Network or Time-varying Network)**.** *A given temporal network* $G = (V, E, \tau)$ *consists of a set of vertices* V*, a set of edges* E*, and a function* τ *that assigns a singular timestamp* t *to each edge* $e_i = (u, v)$*.*

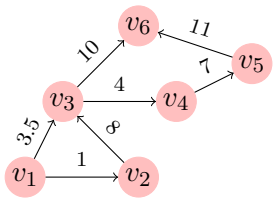


Figure 1: Time-varying Network

The last part of this dissertation will focus on the evaluation of static node embedding models from Rozenberczki's general purpose community detection and network embedding library Python package `karateclub` [28] to see how they perform on a link prediction task on the Dribbble dataset. Dribbble is a social network functioning as a self-promotion and networking platform for graphic and web designs, illustrations and photographs.

2 Link prediction problem

One of the main object of study in data analysis is the evolution of graphs and networks. A core problem is link prediction: considering an older or partial snapshot of a graph at a given time we aim to predict the edges that will appear in the graph until a given future time. Link prediction can be applied in a large variety of scientific fields. The capacity to predict interactions between data objects is fundamental for prediction of genetics and protein interactions, product recommendation on platforms like Amazon or Ebay, friendship recommendation on social networks and a wide range of data mining tasks.

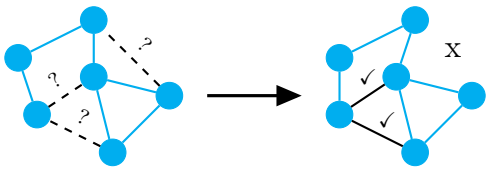


Figure 2: Link prediction problem

Before using node embedding methods to predict interactions between vertices, several approaches have been introduced in [1] that analyze network topology to answer the question of the link prediction problem.

2.1 Link prediction methods

In the following methods, the input consists of a graph $G = (V, E)$ and a timestamp function t over E where E is the multi-set of edges and V is the set of nodes. The edge $e = (u, v)$ represents an interaction between node u and v with the timestamp $t(e)$ when the interaction happened.

Definition 2 (Link prediction problem)**.** *Given four times $t_1 < t'_1 < t_2 < t'_2$ and the subgraph $G[t_1, t'_1]$ that contains all the edges with timestamps between t_1 and t'_1, we wish to create an algorithm that outputs a list of edges that do not appear in $G[t_1, t'_1]$ but are likely to be present in $G[t_2, t'_2]$. In this case $[t_1, t'_1]$ is considered the training interval and $[t_2, t'_2]$ the test interval* [1].

The link prediction methods described in this chapter all allocate a score to every node pair according to the input graph and then create a decreasing ranked list of the scores to determine the next linkage, see Figure 3.

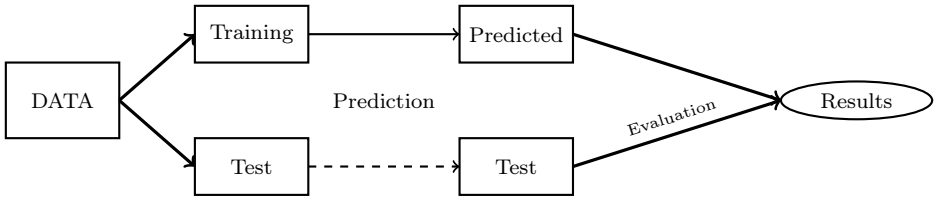


Figure 3: **Link prediction process:** First we separate the training and test data. We build predictions based on the training data, which we evaluate over the test network. Prediction consists of a score for each node pair turned into a ranked list of the potential linkages. We evaluate by using the test network as ground truth [4]

.

In the following sections, we will present several algorithms and functions that can be used for link prediction.

2.2 Local predictors

Local predictor approaches are based on the idea of neighbourhood similarity: the more the neighborhood of two nodes overlap, the more likely they will be linked in the future.

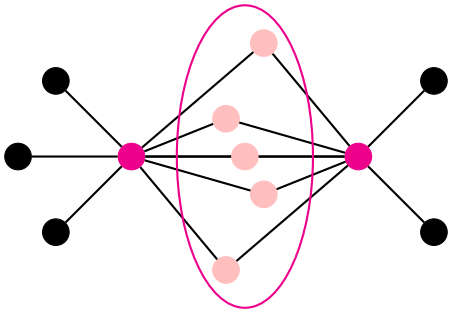


Figure 4: Neighbourhood overlap

Let $\Gamma(u)$ denote the set of neighbors of node u. The following functions are defined to implement link prediction based on neighbourhood similarity.

The **Common neighbours** method defines the scores with the number of neighbors that node pairs have in common.

$$\text{score}(u, v) = |\Gamma(u) \cap \Gamma(v)|$$

Neighbourhood overlap can also be defined with the **Jaccard coefficient**

$$\text{score}(u, v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|},$$

the **Adamic-Adar index**

$$\text{score}(u, v) = \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(w)|},$$

the **Resource allocation index**

$$\text{score}(u, v) = \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{|\Gamma(w)|},$$

and the **Preferential attachment**

$$\text{score}(u, v) = |\Gamma(u)| \cdot |\Gamma(v)|.$$

2.3 Global predictors

Global predictors are based on the idea that a linkage can appear in a graph between two nodes that do not share any common neighbours.

2.3.1 Shortest-path distance

The most intuitive approach to calculate a score for node pairs is to measure the length of the shortest path between them. In case there are multiple shortest path between node pairs that have an equal length, we randomly choose between them.

$$\text{score}(u, v) = \frac{1}{d(u, v)},$$

where $d(u, v)$ is the length of the shortest path between node u and node v. If we take the tie strength between nodes into account, the scores can be calculated as the following:

$$\text{score}(u, v) = \left(\sum_{i=1}^{d} \frac{1}{w_i} \right)^{-1},$$

where w_i is the strength of the ith connection of a shortest path of length d between nodes u and v. Node pairs that have a shortest-path distance of 1 are connected by an edge that belongs to the training edge set and thus are not taken into account.

Katz [6] introduces similarity by the sum of paths with the help of a parameter $0 < \beta < 1$ that controls the contribution of each path by giving larger weight to shorter paths.

$$\text{score}(u, v) = \sum_{l=1}^{\infty} \beta^l \cdot \left|\text{paths}_{u,v}^{(l)}\right|.$$

2.3.2 PageRank

Let us take a random walker that randomly chooses a neighbor of the vertex on which they are currently standing in every step, however there is always a chance that the walker will be transported to another random node in the network. Let $(0 < \alpha < 1)$ denote the probability that the walk will move to a random neighbor and $1 - \alpha$ the probability that it will be transported to another random node in the network. Rooted PageRank is a variant of this where the walker starting at node u can be transported back with $1 - \alpha$ probability only to the starting node u. The score of the node pair (u, v) is measured by the stationary probability of node v in the random walk with the transition probabilities described above.

$$\text{score}(u, v) = \pi_{u,v},$$

$$(\text{for symmetry: } \text{score}(u, v) = \pi_{u,v} + \pi_{v,u}),$$

where

$$\pi_{u,v} = (1 - \alpha) \sum_{w:u \to v} P_w \cdot \alpha^{d(w)}.$$

The summation happens over all walks w going from node u to node v. P_{w_*} describes the chance of traveling walk w_* in the graph. For a walk $w_* = (x_0, x_1, \ldots x_k)$ P_{w_*} is defined as $\prod_{i=0}^{k-1} \frac{1}{|\Gamma(x_i)|}$.

2.3.3 SimRank

The idea behind the SimRank approach is that two nodes are similar if their neighborhood is similar. We assume that all vertices are similar to themselves, i.e.,

similarity$(u, u) = 1$. The score(u, v) is defined as similarity(u, v) which is calculated with the following recursion:

$$\text{similarity}(u, v) = \frac{c}{|\Gamma(u) \cap \Gamma(v)|} \cdot \sum_{a \in \Gamma(u)} \sum_{b \in \Gamma(v)} \text{similarity}(a, b),$$

where $c \in [0, 1]$ is a factor that controls how fast similarities decrease.

2.4 Online link prediction

The link prediction methods presented so far were based on static graphs, or, in time, fixed snapshots of a dynamic graph. We now want to solve the time-aware variant of the link prediction problem and process dynamic graphs edge by edge. The aim is to be able to predict the next linkage based on all the edges that appeared before this specific time. The evaluation of online link prediction happens with the prequential evaluation framework. At the arrival of a new edge uv, this framework tries to predict this edge and reveals it to the model only afterwards. It then updates the model with the help of this new edge. This ensures that that the information of the most recent edges will also be included in the model.

Some of the static models presented earlier in this section can be extended to temporal models. For example, the static PageRank or Katz centrality can be modified by considering only temporal walks instead of all possible walks.

Definition 3 (Temporal walk)**.** *Given a temporal network $G = (V, E, \tau)$, a temporal walk from v_1 to v_j is a sequence of edges $(v_1v_2, v_2v_3, \ldots, v_{j-1}v_j)$, where $\tau(v_iv_{i+1}) \leq \tau(v_{i+1}v_{i+2})$ for all $1 \leq i \leq j - 1$.*

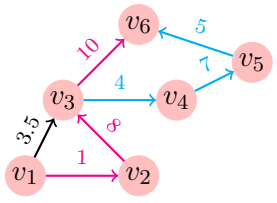


Figure 5: **Examples of Temporal Walks:** The pink walk $v_1v_2v_3v_6$ forms a valid temporal walk whereas the blue walk $v_3v_4v_5v_6$ is not valid since edge $e = (v_4, v_5)$ appears anterior to edge $e = (v_5, v_6)$ in the graph.

To create a temporal random walk, we first select an initial edge $e_i = (v_1, v_2)$; let t be its timestamp. The next node will be chosen from the neighborhood of v_2 but only from nodes that can be reached through an edge whose timestamp is greater than t. Let us call this multiset temporal neighborhood.

Definition 4 (Temporal Neighborhood)**.** *The temporal neighborhood of a node u at time t is*

$$\Gamma_t(u) = \{w \mid e = (u, w) \in E, \tau(e) > t\}.$$

Note that since several edges can appear between two vertices, temporal neighborhoods are multisets.

3 Vector space representation of words

Before discussing the embedding of nodes with the help of neural networks, we will present two models for word representation in a vector space which are the antecedents of node embeddings. When processing text using machine learning and neural network algorithms, we have the difficulty that these algorithms require numerical inputs to set up classification, regression or other models. Thus, it is necessary to represent words numerically. Vector space representations encode words according to their similarity in a vector space, thus helping learning algorithm achieve better results in natural language processing tasks. There are several model architectures for representing words as continuous vectors. All of them have the aim of maximizing accuracy while also minimizing computational complexity.

3.1 Word2vec

Word2vec models represent each word in a given vocabulary by a vector. The models learn the semantic similarities of words and place them in a vector space in a way that words with high similarity will be closer to each other. Two vectors are similar if their enclosed angle is small, more precisely:

$$\text{similarity}\left(\vec{\text{word}}_1, \vec{\text{word}}_2\right) = \frac{\vec{\text{word}}_1 \cdot \vec{\text{word}}_2}{\|\vec{\text{word}}_1\| \cdot \|\vec{\text{word}}_2\|}.$$

In the following chapter, we will present two Word2vec models, the Continuous Bag Of Words (CBOW) and the Continuous Skip-Gram architectures.

The choice of the word embedding models depend on the final prediction task. We can use CBOW if we predict a word based on its context, while Skip-Gram if we want to predict the words surrounding the current one. Figure 6 illustrates the structural difference between the two models.

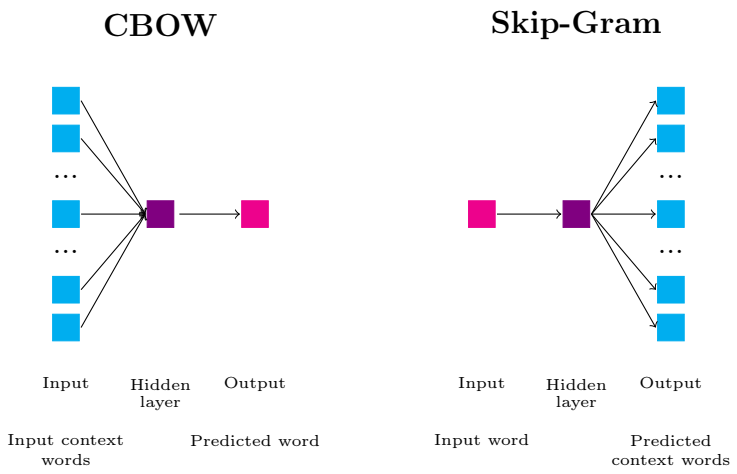


Figure 6: **CBOW and Skip-Gram model architectures:** The CBOW model uses a context to make a prediction for a specific word. (For example using the surrounding words in a sentence it predict the one in the middle). The Skip-Gram model on the other hand predicts the context using an input word. Both of these models use neural networks with a hidden layer.

3.2 CBOW model

3.2.1 One-word context

We introduce CBOW by describing the simplest version when the context consists of a single word. In this case, based on a single other word, we want to predict another (not necessary adjacent) word, which we call the **focus word**.

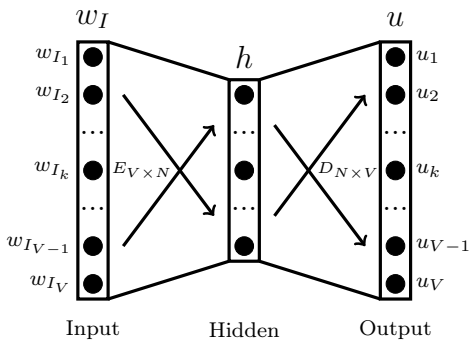


Figure 7: CBOW model for a one-word context

As Figure 7 shows, this model consists of three layers, an input, a hidden, and an output one. In this configuration, the size of our vocabulary $\mathcal{V}$ is V and the number of neurons in the hidden layer is N. The input layer is the 1-of-V-coding column vector w_I of the context word. The weights between the input and the hidden layers are represented by matrix E of size $D \times N$, while the weights between the hidden and output layers are represented by matrix D of size $N \times V$. Row i of the weight matrix E is an N dimensional vector representation v_w for a given input word w. The hidden layer h can be calculated from the input layer as:

$$h^\top := w_I^\top E.$$

We use vector h and weight matrix D to determine the output layer from the hidden layer, which assigns a score to each word in the dictionary

$$u := hD.$$

To obtain the 1-of-V-coding of the output word w_O, we first need to make a probability vector y from vector u. For this purpose we apply the softmax activation function. It is a function that normalizes an input vector to a probability distribution.

Definition 5. *The following function is called the softmax activation function:*

$$F(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

If u_j denotes the score of the j-th word in the dictionary and $p(w_j|w_I)$ approximates the probability that given a one-word context w_I the output word is w_j, than applying this activation function to vector u will result in a probability vector y.

$$y_j^T = p(w_j|w_I) = \frac{e^{u_j}}{\sum_i e^{u_i}}$$

The most likely element in the vector y indicates what the model predicts as the focus word from the dictionary (w_O).

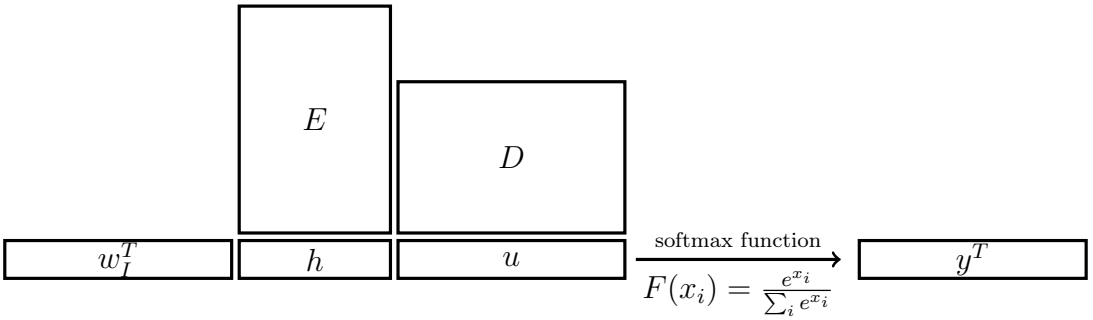


Figure 8: CBOW model operation for a one-word context

Notice that the column vectors v'_w of the weight matrix D also correspond to a vector representation of the word w. Hereinafter, v_w is called the first vector representation, v'_w is called the second vector representation of word w. Each word has two vector representations; v_w and v'_w. Using these notations:

$$y_j^T = p(w_j|w_I) = \frac{e^{v'^{\top}_{w_j} v_{w_I}}}{\sum_{k=1}^{V} e^{v'^{\top}_{w_k} v_{w_I}}}$$

By training the model, the aim is to create matrices E and D that give accurate predictions.

3.2.2 Multi-word context

In general, we do not make predictions based on a single-word, but rather on a multi-word context. Compared to the simplified model, a more accurate one applies the weight matrix E to all input words. The model takes the average of the input vectors:

$$h^T = \frac{1}{K}\left(w_1^{\top} + w_2^{\top} + ...w_K^{\top}\right) E = \frac{1}{K}\left(v_{w_1} + v_{w_2} + ... + v_{w_K}\right),$$

where K is the number of words in the context, $w_1, ...w_K$ are the 1-of-V-coding vectors of the context words and v_w is the first vector representation of a word w.

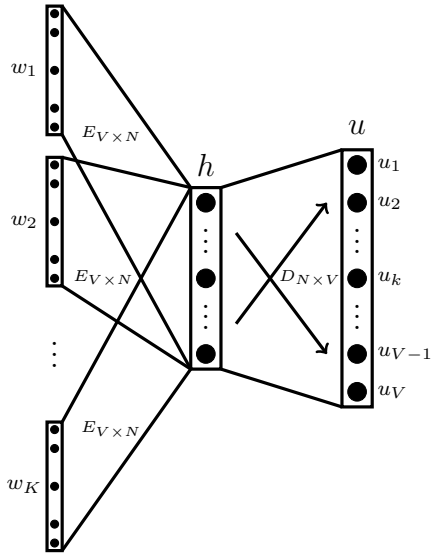


Figure 9: Multi-word CBOW model

During training, the model seeks to maximize the loss function

$$\frac{1}{K}\log\sum_{k=1}^{K}\mathbb{P}(w_k|\{w_{k-t}\dots w_{k+t}\}\setminus\{w_k\})$$

where we predict the center word w_k, given the context $\{w_{k-t}\dots w_{k+t}\}\setminus\{w_k\}$.

3.3 Skip-Gram model

Counter to the CBOW model, the continuous Skip-Gram model wishes to maximize the probability of accurately predicting a context of words from a given word, (see Figure 10). During the following computations we use the assumption that the words appear independently from each other in the text.

$$\mathbb{P}(\{w_{j-t}\dots w_{j+t}\}\setminus\{w_j\}|w_j)=\prod_{\substack{k=j-t,\\k\neq j}}^{j+t}\mathbb{P}(w_k|w_j).$$

Rather than maximizing the probability as a product, we maximize the log-likelihood

$$\frac{1}{K}\sum_{j=1}^{K}\sum_{\substack{k=j-t,\\k\neq j}}^{j+t}\log\mathbb{P}(w_k|w_j).$$

Optimization is computed by minimizing a loss function which in this case is

$$\Lambda_{SG} = -\frac{1}{K}\sum_{j=1}^{K}\sum_{\substack{k=j-t,\\ k\neq j}}^{j+t} \log \mathbb{P}(w_k|w_j).$$

Using

$$\mathbb{P}(w_k|w_j) \approx \frac{e^{v'^{\top}_{w_k} v_{w_j}}}{\sum_{i=1}^{V} e^{v'^{\top}_{w_i} v_{w_j}}}$$

we approximate the loss function Λ_{SG} with

$$\mathcal{L}_{SG} = -\frac{1}{K}\sum_{j=1}^{K}\sum_{\substack{k=j-t,\\ k\neq j}}^{j+t} \log \frac{e^{v'^{\top}_{w_k} v_{w_j}}}{\sum_{i=1}^{V} e^{v'^{\top}_{w_i} v_{w_j}}}.$$

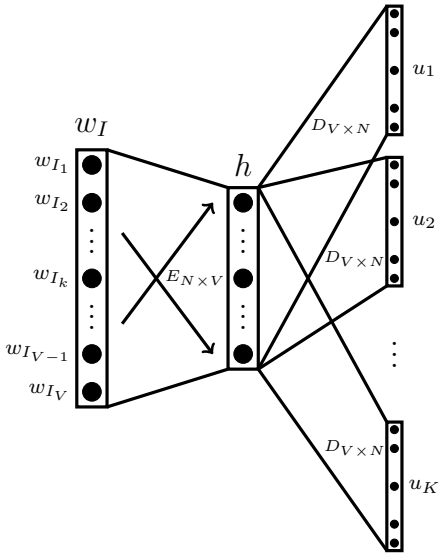


Figure 10: Skip-Gram model

3.3.1 Stochastic Gradient Descent (SGD)

When performing the Skip-Gram algorithm, we wish to minimize an empirical risk that occurs on the training data. In order to do this we often use the Gradient Descent (GD) method, which iteratively moves in the direction of the negative

gradient of the function at the current point to find a local minimum. In the Skip-Gram algorithm, the objective function takes the form of a sum:

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x),$$

where x is the parameter we look for in order to minimize $f(x)$. In these cases, we use the stochastic gradient descent (SGD), where we approximate $\nabla f(x)$ by $\nabla f_i(x)$.

If $f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x)$ is the function we wish to minimize, x is the initial parameter of the function and α is the learning rate, the SGD method will perform the following computation n times in a shuffled order of $\{1, \dots n\}$:

$$x \leftarrow x - \alpha \cdot \nabla f_i(x)$$

The Skip-Gram algorithm follows the steps of the the SGD algorithm described above on the function $\mathcal{L}_{SG}$ as the following:

Algorithm 1: Basic Skip-Gram$(t, R, \mathcal{D})$

Input: dictionary $\mathcal{D}$, window size t, current vector representations R

for $i \in \mathcal{D}$ **do**

  for $k = j - t \dots j + t, k \neq j$ **do**

    $\mathcal{L}_{i,k}(V) = -\log \frac{e^{v'^{\top}_{w_k} v_{w_j}}}{\sum_{i=1}^{R} e^{v'^{\top}_{w_i} v_{w_j}}}$

    $R \leftarrow R - \alpha \cdot \frac{\partial \mathcal{L}_{i,k}}{\partial R}$

  end

end

Note that in the above basic Skip-Gram algorithm, the gradient step involves the vectors corresponding all other words in the vocabulary, which makes the gradient step computationally very expensive. The basic algorithm can be enhanced by techniques to speed up the computation of the gradient that we will describe in the following section.

3.4 Heuristic gradient computation

In this section, we give two heuristics to speed up the gradient computation. We saw earlier in section 3.2.1 that each word in the dictionary has two vector space representation: the first vector v_w from the rows of the weight matrix E and the second vector v'_w from the column vectors of the weight matrix D. Learning the second vector representation v'_w is computationally complex since we have to iterate through every word of the vocabulary [11]. To speed up the output vector computation, Word2vec models use Hierarchical Softmax or Negative Sampling heuristics. Both reduce complexity by updating only the most influential output vectors.

3.4.1 Hierarchical Softmax

Hierarchical Softmax is a technique for approximating the softmax function to reduce the computational cost of training the network. It encodes all the words of the dictionary in a binary tree. The V words appear as leaf units and can be explicitly represented by the path from the root to that leaf. If the tree is balanced, any word can be defined by a sequence of $O(\log V)$ binary decisions. This model uses a Huffmann binary tree [13], which takes the frequency distribution of each word into account. Rare words are down at a deeper level while frequent words are at a shallower level. There are $V - 1$ internal nodes (including the root). For the inner nodes, we use the following annotation: $n(w, k)$ is the k-th unit on the path from the root to word w. Figure 11 shows an example where the output word is w_3.

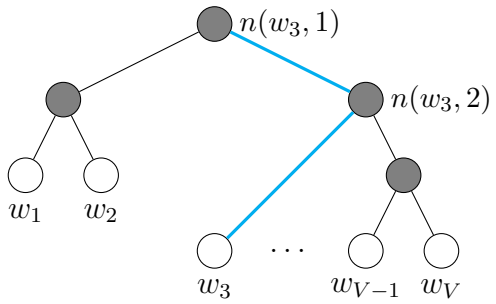


Figure 11: An example for hierarchical softmax model

In this model there is no output vector representation for the words. We are going to associate each row of our output matrix with one of the internal nodes. Let $v'_{n(w,k)}$ be the output vector for inner node $n(w,k)$. The hierarchical softmax model defines $p(w_j|w_I)$ as

$$p(w_j|w_I) = \prod_{l=1}^{L(w)-1} \sigma([\![n(w_j, l+1) = ch(n(w_j, l))]\!] \cdot v'^{\top}_{n(w_j,l)} v_{w_I}),$$

where $\sigma(x) = \frac{1}{1+e^x}$, $L(w)$ is the length of the path from the root to word w, $[\![x]\!]$ is 1 if x is true and -1 otherwise, and $ch(n)$ is the left child of unit n [10].

By using the hierarchical softmax model, the computational complexity per training instance per context word can be reduced from $O(V)$ to $O\log(V)$.

3.4.2 Negative Sampling

Negative Sampling is a more intuitive approach for improving the computational complexity. Instead of updating every output vector at every training instance, we update only a sample of them. During training, we keep the correct output word w_O in the sample and we determine the negative samples with a probabilistic distribution that we call noise distribution. We minimize the negative sampling loss function

$$E = -\log \sigma(v'^{\top}_{w_O} h) - \sum_{w_j \in W_{neg}} \log \sigma(-v'^{\top}_{w_j} h),$$

where $h = \frac{1}{K} \sum_{k=1}^{K} v_{w_k}$ in the CBOW model and $h = v_{w_I}$ in the Skip-Gram model, since when using negative sampling for the Skip-Gram model, we need to use this equation one at a time for each word that appears in the context [11].

4 Node embedding algorithms

Mining and machine learning algorithms on graphs start with the exploration of their structural features. In the last decade a new and computationally efficient representation learning technique emerged called node embedding. Given an undirected and weighted graph $G = (V, E, A)$, where E is the set of edges, V is the set of nodes and A is the adjacency matrix, node embedding algorithms place the vertices of the graph in a d-dimensional vector space by learning a $V \to \mathbb{R}^d$ function that maps each node to a vector that represents well the structural properties of the vertices [18]. These vectors can then be used as inputs for mining and learning algorithms such as node classification, link prediction and network similarity. In this chapter we will present some of the most widespread node embedding algorithms: Laplacian Eigenmaps, Large-Scale Information Network Embedding, Deepwalk and Node2vec.

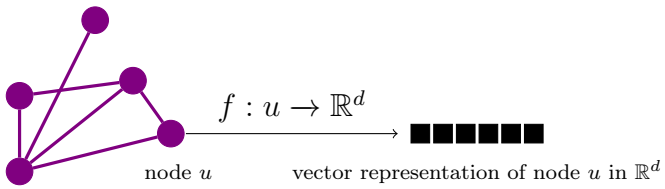


Figure 12: The aim of node embedding

4.1 The process of node embedding

Node embedding consists of four components: a pairwise node similarity function $S_G : V \times V \to \mathbb{R}^+$ that defines the similarity in the original graph, an embedding $\text{emb}(x)$ that maps the nodes to vectors, an embedding similarity function that defines the similarity in the vector space, and a loss function Λ along which we will optimize and train the node embedding model [16]. The difference between the various techniques is how we interpret these components. In this dissertation, similarity between two embedding vectors is defined as their dot product.

The input of the embedding is a one-hot encoded vector u and the output is the vector representation z_u of the given node: $\text{emb}(u) = z_u$.

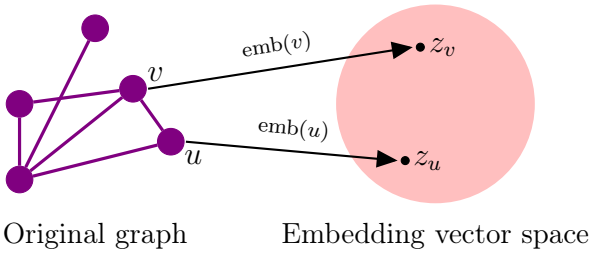


Figure 13: Illustration of node embedding

The aim is to represent the nodes as low dimensional vectors such as that similarity in the vector space reflects similarity in the original graph.

$$S_G(u, v) \approx z_u^\top z_v$$

The algorithms learn a matrix $Z \in \mathbb{R}^{d \times |V|}$ and each column vector z of the matrix represents the embedding of a specific node, where the vector space is d-dimensional, and V is the set of vertices in the graph.

4.2 Laplacian Eigenmaps for Embedding

The first node embedding method presented in this work defines the embedded vectors with the help of eigenvalues and eigenvectors of the Laplacian matrix of the graph [23]. We consider the following generalized eigenvector problem:

$$Lx = \lambda Dx,$$

where $D \in \mathbb{R}^{|V| \times |V|}$ is the diagonal weight matrix and $L \in \mathbb{R}^{|V| \times |V|}$ is the Laplacian matrix of the graph;

$$L_{i,j} = \begin{cases} \deg(u_i) & \text{if } i = j, \\ -1 & \text{if } i \neq j \text{ and } \exists \text{ edge}(u_i, u_j), \\ 0 & \text{otherwise.} \end{cases}$$

Let $x_0, x_1 \ldots x_{k-1}$ denote the solutions of the above equation where $\lambda_0 \leq \lambda_1 \leq \ldots \leq \lambda_{k-1}$. It can be proved that 0 is the smallest eigenvalue with eigenvector $(1, \ldots, 1)$, i.e., $\lambda_0 = 0$ and the vector is constant over the nodes. As seen in Figure 14, after

removing the constant and keeping the first few eigenvectors, the embedding of node u_i into the lower dimensional space $\mathbb{R}^m$ is defined by the vector $\text{emb}(u_i) = (x_1(i), x_2(i)...x_m(i))$.

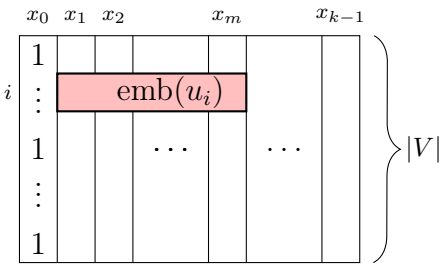


Figure 14: **Illustration of node embedding with Laplacian Eigenmaps:** The vector representation of node u_i is $(x_1(i), x_2(i)...x_m(i))$. We do not use the first column since it is (1,. . . ,1).

4.3 Neighborhood similarity

In this subsection, we discuss several different node similarity functions. We often base the measure of similarity between nodes on features of their neighborhoods. We present similarity algorithms through adjacency, k-hop adjacency, neighborhood overlap, and their combination.

4.3.1 Adjacency-based similarity

The adjacency-based similarity function is the edge weight between two nodes in the original graph $z_u^T z_v \approx \text{weight}(u, v)$. We want to learn a matrix $Z \in \mathbb{R}^{d \times V}$ in a way that minimizes the following loss function:

$$\Lambda = \sum_{(u,v) \in V \times V} \|z_u^\top z_v - A_{u,v}\|^2,$$

where A is the weighted adjacency matrix [17].

If we want to consider the k-hop neighbors, the loss function changes to the following:

$$\Lambda = \sum_{(u,v) \in V \times V} \|z_u^\top z_v - A_{u,v}^k\|^2,$$

4.3.2 Neighborhood overlap

In real-world graphs, adjacency is sometimes insufficient to present the global structure. In some cases, nodes that share neighbors are likely to be similar. If we look at social media for example, people whose connections are very similar tend to share interests and other properties. In the example of Figure 15, vertices v_1 and v_3 are connected, and so they should be placed closely in the vector space. Though there is no edge between vertices v_1 and v_2, they share all their neighbors, hence their embedding vectors should also appear close to each other.

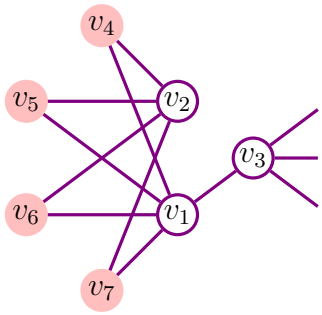


Figure 15: Shared neighbors in a graph

In this case the loss function we wish to minimize is the following:

$$\Lambda = \sum_{(u,v)\in V\times V} \|z_u^\top z_v - \mathrm{NO}_{u,v}\|^2,$$

where $\mathrm{NO}_{u,v}$ is the neighborhood overlap between nodes u and v that can be measured with the Jaccard coefficient, Adamic-Adar index and other methods mentioned in Section 2.3.1.

4.3.3 The LINE algorithm

There are methods that combine different neighborhood similarity measures. The LINE (Large-Scale Information Network Embedding) algorithm creates an embedding that preserves first- and second-order proximity of a graph as well by modeling them separately and then concatenating the embedded vectors for each node [21].

4.4 DeepWalk

Another option to interpret node similarity is if they co-occur on randomly generated short walks in the original graph. In this section, we introduce the DeepWalk algorithm that learns the node embedding with the help of random walks.

Definition 6. *A Random walk is a sequence of vertices in a graph starting with a given node and following a randomly selected neighbor at every vertex.*

The DeepWalk algorithm consists of two steps: it first generates a multiset $\mathcal{B}$ that contains node-context vertex pairs and then trains the Skip-Gram model on multiset $\mathcal{B}$. See Skip-Gram algorithm in Section 3.3.

Algorithm 2: DeepWalk(R, L, T) [19]

Input:
Graph $G(V, E)$
R: number of random walks
L: length of random walks
t: window size
for $i = 1, 2 \dots R$ **do**
  Select $w_{u_i}^1$ root note for random walk i with a probability distribution;
  Produce a node sequence of length L by a random walk: $\{w_{u_i}^1, \dots, w_{u_i}^L\}$;
  for $k = 1, 2 \dots R - t$ **do**
    for $r = 1, 2 \dots t$ **do**
      Add vertex-context pair $(w_{u_i}^k, w_{u_i}^{k+r})$ to multiset $\mathcal{B}$.
      Add vertex-context pair $(w_{u_i}^{k+r}, w_{u_i}^k)$ to multiset $\mathcal{B}$.
    end
  end
end
Run Skip-Gram model on multiset $\mathcal{B}$. (See Skip-Gram algorithm in Section 3.3)

Let W_{u_i} indicate the random walk rooted at the node u_i $(i = 1, 2, \dots R)$. The nodes appearing in random walk W_{u_i} are $\{w_{u_i}^1, w_{u_i}^2, \dots, w_{u_i}^L\}$ where $w_{u_i}^{j+1}$ is a vertex randomly chosen from the neighbors of $w_{u_i}^j$.

4.5 Node2vec

The Node2vec algorithm is different from DeepWalk in that it introduces two new parameters P and Q that bias the random walk.

There are two classic strategies to sample the neighbourhood of j nodes of a given vertex: breadth-first search (BFS) and depth-first search (DFS).

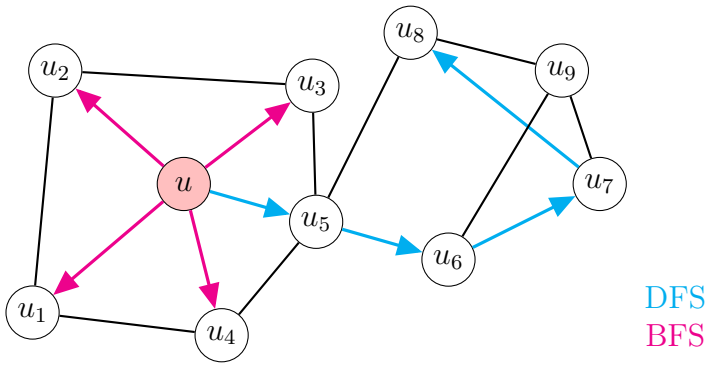


Figure 16: Difference between breadth-first search and depth-first search from node u if $j = 4$

BFS sampling explores a node's immediate neighbourhood, while DFS explores at an increasing distance from the node. For example, in Figure 16 the BFS sample is $\{u_1, u_2, u_3, u_4\}$ and the DFS sample is $\{u_5, u_6, u_7, u_8\}$ [20]. As we will describe, the Node2vec algorithm uses a sampling strategy that lies in between BFS and DFS.

Let us consider a random walk that steps from node u_s to node u_* (Figure 17). We now want to determine the next step. Let $\mu_{u_* x_i}$ denote the transition probabilities from node u_* to its neighbors $(x_1, x_2, \ldots x_{N_{u_*}})$ and set them to $\mu_{u_* x_i} = \alpha_{PQ}(u_s, x_i) \cdot w_{u_* x_i}$, where $w_{u_* x_i}$ is the weight of edge $(u_* x_i)$,

$$\alpha_{PQ}(u_s, x_i) = \begin{cases} 1/P & \text{if } d_{u_s x_i} = 0, \\ 1 & \text{if } d_{u_s x_i} = 1, \\ 1/Q & \text{if } d_{u_s x_i} = 2, \end{cases}$$

and $d_{u_s x_i}$ is the shortest path from node u_s to node x_i. Parameter P and Q determine the the walk, since $d_{u_s x_i}$ can only take the values $\{0, 1, 2\}$.

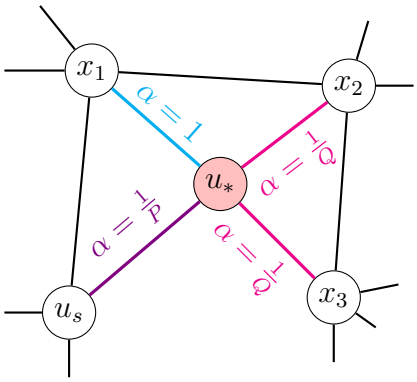


Figure 17: Illustration of Node2vec on an example

Parameter Q controls how far we can travel from node u_s, while parameter P gives the probability of revisiting the node u_s. The Node2vec model is close to a BFS sampling if both $P > 1$ and $Q > 1$. In this case, the highest weight α_{PQ} from node u_* is 1, which means that the walk tends to visit the neighbors of node u_s. If P takes a high (> 1) and Q a low (< 1) value, the model will be similar to DFS sampling, since the highest weight α_{PQ} from node u_* is $\frac{1}{Q}$.

5 Node embedding in dynamic graphs

In the previous sections, we focused on the embedding of static snapshots, ignoring the time-dependency and dynamics of real-world graphs that evolve over time with new edges and nodes appearing and disappearing. The embedding methods described earlier can all be adapted to dynamic networks by considering the sequence of snapshots in time where we define static snapshot graphs as $G_1, \dots G_T$, where $G_i = (V, E_t)$ and the edge set E_t are present in the time period $[t_{i-1}, t_i]$. The computation of these models are very time-consuming since they create a new representation for every snapshot.

In the following chapter we will discuss methods that learn time-preserving embeddings online. To simplify the model, from now on we will work with directed graphs but optionally we can add each new edge in both directions to make it undirected.

5.1 Time-preserving node embedding

In this section we describe methods that learn time-dependent embeddings on temporal networks (see Definition 1).

5.1.1 Embedding with temporal walks

Similarly to the DeepWalk model, we will interpret node similarity with the help of randomly generated short walks in the original graph, but this time with the use of temporal random walks where the edges are in a time respecting order (see Definition 3).

Since the edges arrive in a time-preserving order in the temporal walk, we do not expect a fixed length for the walks. We define a minimum length ω and a maximum length L. A temporal random walk R_{t_i} is valid if $\omega \leq R_{t_i} \leq L$. We now sample a set of temporal walks and run the Skip-Gram algorithm presented in Section 3.3 on the multiset, where the context window is the minimum length ω of the temporal walks.

5.2 Online node embedding

We talk about online embedding algorithms when the model continuously updates embeddings after the arrival of (almost) every arriving edge as we process the graph stream.

5.2.1 StreamWalk

The goal of StreamWalk is to update the embedding of the node v after the appearance of edge uv in order to be similar to the vector representations of the vertices that can reach u through recently appeared edges. This online algorithm is based on the idea of temporal walks described in Section 5.1.1. StreamWalk samples temporal walks and optimizes the embedding for the similarity of the new node u and the source node, see Figure 18.

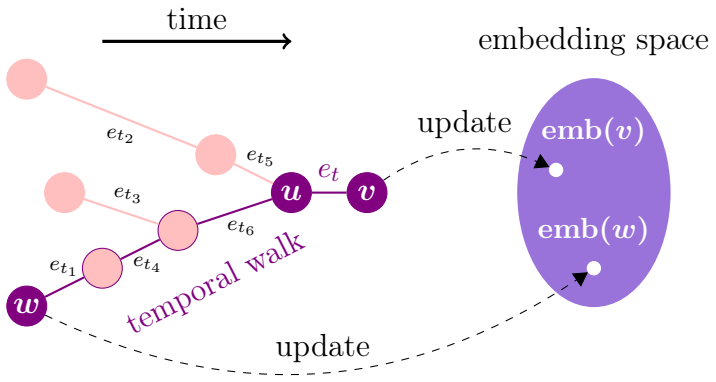


Figure 18: **Illustration of the main idea of StreamWalk:** The algorithm will update the embedding of the source node w of a sampled temporal walk and node v to be similar to each other if the latest arriving edge uv is the last edge of the temporal walk with $t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t$.

To ensure that we generate temporal walks, we start from node u and step backwards. We always select the next edge taking the waiting time into consideration. For example, in Figure 18, we first select edge e_{t_6}, then e_{t_4}.

To define the weight of a walk, we use the weighting function $\gamma(t) := e^{-c \cdot t}$, which takes the delay t of two adjacent edges as a parameter. This time-aware

function is exponential to give a greater weight to walks where the edges appeared more recently. We can now define the weight of a walk $\varpi = (e_1, e_2, \dots e_k)$ with timestamps $t_1 < t_2 < \dots < t_k$ at a time $t > t_k$ as

$$p(\varpi, t) = \alpha^{|\varpi|} \prod_{i=1}^{k} \gamma(t_{i+1} - t_i) = \alpha^{|\varpi|} \cdot e^{-c(t-t_k)} \cdot \dots \cdot e^{-c(t_2-t_1)} = \alpha^{|\varpi|} \cdot e^{-c(t-t_1)},$$

where $t_{k+1} = t$ and $0 < \alpha \leq 1$ is a parameter that controls the length of the walk.

StreamWalk uses two procedures during implementation: UpdateWalk(u, v) recursively updates the weights of all the temporal walks ending with edge uv, and SampleWalk(u, t) samples a temporal walk that ends in node u at time t.

Let $p(v, \tau(uv))$ denote the sum of weights of all walks that end in node v (but the last edge of the walk is not necessarily uv) at the arrival time $\tau(uv)$ of edge uv. UpdateWalk calculates this sum recursively by dividing the walks ending in v into three categories and summing their weights:

- the new single edge walk uv
- the walks which terminate with the edge uv
- the walks ending before the arrival of the edge uv.

The weight of the walk of the first category is obviously α. If t_u denotes the timestamp of the last edge entering node u before $\tau(uv)$, then the total weight of the walks of second type is

$$\alpha \cdot p(u, t_u) \cdot e^{-c(\tau(uv)-t_u)}.$$

The total weight of the third category walks is

$$p(v, t_v) \cdot e^{-c(\tau(uv)-t_v)},$$

where similarly as before t_v is the timestamp of the last edge entering node v before $\tau(uv)$. Thus,

$$p\big(v, \tau(uv)\big) = \alpha + \alpha \cdot p(u, t_u) \cdot e^{-c\left(\tau(uv)-t_u\right)} + p(v, t_v) \cdot e^{-c\left(\tau(uv)-t_v\right)},$$

see Figure 19. UpdateWalk brings to date the value of $p(v, \tau(uv))$ after the arrival of every edge.

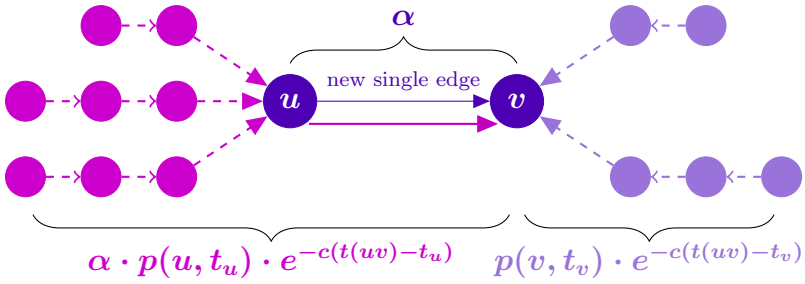


Figure 19: Illustration of the recursive method of UpdateWalk.

SampleWalk is the procedure that samples a temporal random walk in the graph after updating the weights of the temporal walks with algorithm UpdateWalk. It generates a random walk backwards starting from node u after the arrival of the last edge $\tau(uv)$. To select a backward edge u_1u we use the weight of the walks ending with this edge

$$p(u_1u) = \alpha \cdot \left(1 + p(u_1, t_{u_1}) \cdot e^{-c\left(t(u_1u) - t_{u_1}\right)}\right),$$

thus, the probability of selecting edge u_1u is

$$\frac{p(u_1u) \cdot e^{-c\left(\tau(uv) - \tau(u_1u)\right)}}{p\big(u, \tau(u,v)\big)}.$$

Let us terminate the sampled temporal walk some node u^* at some time t^* with probability

$$\frac{1}{1 + p(u^*, t_{u^*}) \cdot e^{-c(t^* - t_{u^*})}}.$$

We can also end the walk after reaching a predefined length. SampleWalk algorithm gives as output the source node w of the generated temporal random walk.

For each new arriving edge uv, the StreamWalk algorithm first updates the weights of the temporal walks with UpdateWalk, then samples a random walk with SampleWalk and optimizes the embedding for the similarity of node v and the source node w of the sampled walk with stochastic gradient descent as presented in Section 3.3.1. We train the model by sampling k walks per edge.

Algorithm 3: StreamWalk(u, v)

UpdateWalk(u, v)

Repeat k times:

w :=SampleWalks(u, now)

Optimize the embedding emb(v) and emb(w) of node u and w with SGD.

5.2.2 Online Second Order node similarity

The next online algorithm presented in this section optimizes the node embeddings to preserve second order similarity by using a time-aware variant of the Jaccard similarity described in Section 2.2. At the arrival of a new edge uv, we optimize the vector representations of the node pairs (u, v_*), where v_* are the in-neighbors of node v (see Figure 20).

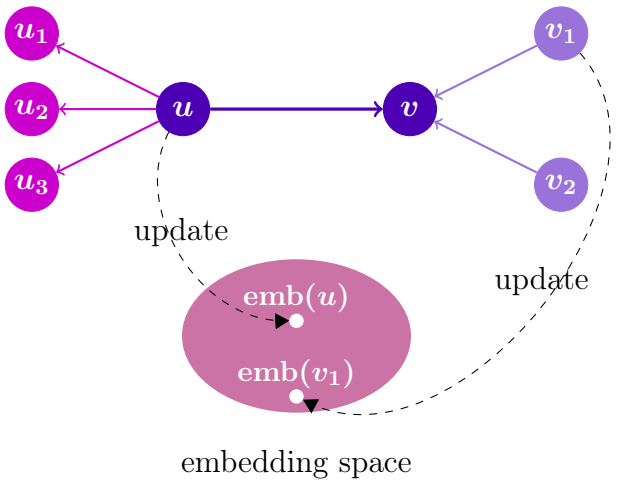


Figure 20: **The aim of online second order similarity.** Our goal is to separately optimize the vector representations of the node pair (u, v_1) and (u, v_2).

Let $N(u, t)$ denote the neighborhood of a node u at a time t and let

$$\varpi(uv) := e^{-c\left(t-\tau(uv)\right)}$$

be the weight of an edge uv, where $\tau(uv)$ is the timestamp of the edge uv. To accentuate the influence of newer edges, let us dispose of a node $u_1 \in N(u, t)$ with

probability $1 - \varpi(u_1)$. More precisely let $N^*(u,t)$ be the pruned neighborhood of node u after the arrival of edge uv.

We wish to optimize for the squared error

$$\big[q_u^\top q_{v_1} - \mathrm{sim}(u, v_1)\big]^2,$$

where we will define $\mathrm{sim}(u, v_1)$ later as the time-aware Jaccard similarity. In fact, sim will be a MinHash fingerprint approximation of the Jaccard similarity to avoid the costly exact computation.

Min-hashing is a procedure for the representation of large sets of data and the assessment of similarity between items. Let $\pi_1, \pi_2, \ldots, \pi_k$ be random and independent permutations of the node set. Let us define the fingerprints of a set A as:

$$h_i(A) := \min_{a \in A} \pi_i(a), \qquad i = 1 \ldots k.$$

It is easy to see that given two sets A and B:

$$\mathbb{P}\big(h_i(A) = h_i(B)\big) = \frac{|A \cap B|}{|A \cup B|}.$$

Let us use the MinHash fingerprinting for the pruned neighborhood $N^*(u,t)$ of a node u at time t:

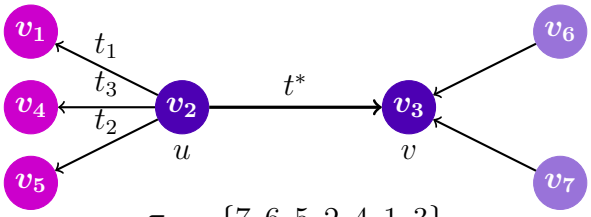
$$h_i(u,t) := h_i\big(N^*(u,t)\big).$$

To approximate the time-aware Jaccard similarity of a node pair (u,v) at time t, we use the following function [26]:

$$\mathrm{sim}(u,v,t) \approx \frac{1}{k} \sum_{i=1}^{k} \mathbb{I}[h_i(u,t) = h_i(v,t)].$$

After the arrival of a new egde uv, the online second order similarity algorithm first updates the fingerprints of node u with the procedure UpdateFingerprints(u, v).

To avoid the recomputation of the fingerprints of node u at every time, in some cases we give $h_i(u,t)$ the value of its new neighbor v. See an illustration of how the fingerprints of node u are updated in Figure 21.



$\pi_1 = \{7, 6, 5, 2, 4, 1, 3\}$
$\pi_2 = \{6, 3, 2, 4, 7, 5, 1\}$
$\pi_3 = \{2, 1, 4, 3, 6, 7, 5\}$

	$t = t_3$	$t = t^*$
$N^*(u,t)$	$\{v_1, v_4, v_5\}$	$\{\cancel{v_1}, v_4, v_5, v_3\}$
I. $h_1(u,t)$	$\min\{7,2,4\} = 2$	$\pi_1(v_1) = 7$, $\pi_1(v_3) = 5$ not the minimum $\rightarrow 2$
II. $h_2(u,t)$	$\min\{6,4,7\} = 4$	$\pi_i(v) < \pi_i\big(h_i(u,t_u)\big)$ $\rightarrow 3$
III. $h_3(u,t)$	$\min\{2,3,6\} = 2$	the minimum is too old and removed $\rightarrow 3$

Figure 21: **Illustration of UpdateWalk procedure at the arrival of a new edge uv:** The neighbors of node u arrive in time $t_1 < t_2 < t_3 < t_*$ and we have three random independent permutations π_1, π_2, π_3 over the node set. In the second column we can see the MinHash fingerprints of node u before the arrival of the new edge. Let us say that in this example the oldest node v_1 gets removed from the pruned neighborhood $N^*(u, t_3)$. We check if the removed edge or the new edge has the minimum hash value. In the first case neither so $h_1(u,t)$ maintains its former value 2. In the second case (II.) the hash value of node v is smaller than $h_1(u,t_3)$ so we set $h_i(u,t^*)$ with the new value of v. If the removed old edge was the minimum like in the third case (III.), we also give $h_i(u,t^*)$ the value of v.

Algorithm 4: UpdateFingerprints(u,v) [26]

for $i = 1, 2 \ldots k$ **do**
  if $h_i\big(u, \tau(uv)\big)$ is old **or** $\pi_i(v) < \pi_i\big(h_i(u,t_u)\big)$ **then**
  $h_i\big(u, \tau(uv)\big) := v$
end

After updating the fingerprint of u, for all in-neighbors v_* of node v we compute the number of fingerprints l that are equal to the fingerprints of u and take the value of v. We then optimize l times the vector representations of nodes v_* and u by using the squared error loss function $\left[q_u^\top q_{v_*} - \mathrm{sim}(u, v_1)\right]^2$, see Algorithm 5.

Algorithm 5: OnlineSecondOrderSimilarity(u, v) [26]

UpdateFingerprints(u, v), see Algorithm 4.
for *all in neighbors* v_* *of node* v *that are not too old* **do**
  $l := 0$
  for $i = 1, 2 \dots k$ **do**
    if $h_i\big(u, \tau(uv)\big) = h_i\big(v_*, \tau(uv)\big) = v$ **then**
    $l := l + 1$
    Optimize q_u and q_{v_*} l times by using $\left[q_u^\top q_{v_*} - \mathrm{sim}(u, v_*)\right]^2$.
  end
end

The purpose of the dissertation was to give an overview of different node embedding, there are many other static and online node embedding models besides the one presented in the previous chapters.

6 Evaluation of static models through link prediction

In the following chapter we evaluate the static node embedding methods of the Python framework `karateclub` [28] by examining how they perform on a link prediction task.

6.1 Rozenberczki's network embedding framework

We use the Python framework `karateclub` that contains 30 graph mining algorithms for community detection, node embedding, and whole graph embedding and summarization. The node embedding models of this package include factorization-based techniques that use matrix factorization on adjacency or proximity matrices along with graph-sampling based techniques that sample pairs of nodes from input graphs via random walks and then use the Skip-Gram model to output the embeddings. The models evaluated in this dissertation are: DeepWalk, Role2Vec, NodeSketch, GraphWave, NMFADMM, GraRep, BoostNE, NetMF and Walklets. GraphWave, GraRep, NMFADMM and BoostNE are all factorization-based models while similarly to the DeepWalk algorithm Walklets and Role2Vec both use sampled random walks to create the node embeddings. Walklets generates paths with fixed lengths by skipping steps in the random walks and Role2Vec uses attributed random walks that differentiate edges according to vertex attributes and structural features. There are also models that combine the two techniques: NetMF for example is a model that unifies DeepWalk, Node2Vec and LINE through a matrix factorization framework. NodeSketch tries to improve the two technique's computational complexity. It is a node embedding method that preserves k-order proximity by first creating low-order embeddings from the Self-Loop-Augmented adjacency matrix with a random hash function and recursively generates the higher order embeddings. These algorithms are presented in more detail in the papers collected in [31].

6.1.1 Node embedding algorithms with `karateclub`

The node embedding models of the `karateclub` framework place each node of the input graph in a 128-dimensional vector space. To illustrate how it works we will first use a small social network, Zachary's karate club which represents connections between 34 members of a university karate club outside of the club, see Figure 22.

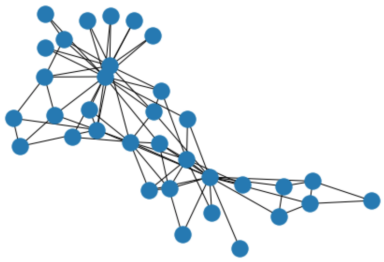


Figure 22: Illustration of Zachary's karate club network using the Python `NetworkX package`.

By using for example the DeepWalk algorithm of the `karateclub` package as the following:

```
from karateclub import DeepWalk
G = nx.karate_club_graph()
model = DeepWalk()
model.fit(G)
embedding = model.get_embedding()
```

as a result we get an array that contains the 128-dimensional embedded vector representation of each of the 34 nodes:

```
array([[-0.10468774, -0.42117506,  0.16994181, ...,  0.13140915,
         0.05461558, -0.01364159],
       [-0.02065941, -0.16449502,  0.04885819, ...,  0.10764173,
         0.10261779, -0.00961214],
       [ 0.08251422, -0.18667659,  0.00494064, ...,  0.09822474,
         0.22129805,  0.02225673],
       ...,
       [ 0.1367532 , -0.13823353, -0.03607513, ...,  0.05142015,
         0.26117924,  0.05435172],
       [ 0.19620116, -0.09208958, -0.08562755, ...,  0.12960388,
         0.3632858 ,  0.04683062],
       [ 0.20123258, -0.10593056, -0.0849262 , ...,  0.12033035,
         0.35455897,  0.0407094 ]], dtype=float32)
```

6.2 Dribbble

In this experiments we used the Dribbble dataset [27]. Dribbble is a social network for graphic and web designs, illustrations and photographs. We use the following dataframe as input:

shot_id	timestamp	user_id	user_id_creator	date
1	1248333459	frogandcode	simplebits	2009-07-23 07:17:39
1	1248339803	owltastic	simplebits	2009-07-23 09:03:23

Figure 23: Each row of this dataframe represents a like interaction between two users: at *timestamp* *user_id* liked *shot_id* created by *user_id_creator*. You can see here the first two rows of the dataset that contains likes between 2009 and 2017.

During the link prediction task we are going to predict likes that originally happened between users and shots but to avoid working with bipartite user-shot graphs we work with user-user graphs. By using the `NetworkX` package we create graphs where the nodes are users, and each edge represents an interactions between the users (in this case likes).

6.2.1 Preprocessing

The first part of the implementation process is to prepare the dataset. For this evaluation we will only consider interactions that happened in 2010 and 2011. We drop rows with missing values, loops and duplicates, thus edges will now only appear once in the dataset.

6.3 Link prediction task

Based on the weekly interactions we wish to predict the likes that will occur during the following week. We group the likes and create graphs for each week with `NetworkX`. Figure 24 shows the number of weekly interactions in 2010 and 2011.

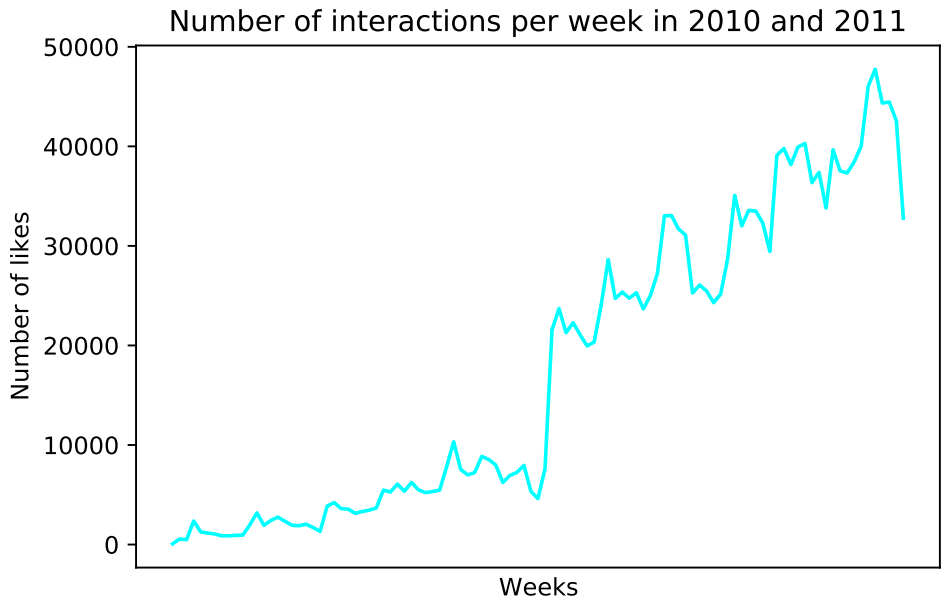


Figure 24: Number of interactions per week in 2010 and 2011

Since `karateclub` models only work on undirected, connected graphs where the indexes of the nodes are integers we convert the names of the nodes to integers for each week and if the graphs are not connected we consider the largest connected subgraphs.

During the link prediction each weekly graph serves as training dataset and the graph created from the following week's data as test dataset. We use a node embedding algorithm from the `karateclub` package and place the nodes of the training graph in a 128-dimensional vector space.

For each node of a training graph we create a ranked list of all the other nodes. We rank the nodes in the list in descending order based on the dot product of the embeddings of the current vertex and the other vertices in the graph. For example the first element of the ranked list of a specific node is the user that is closest to the examined node according to their dot product. For each edge of the test dataset where the nodes appear in the training graph we look up the index of the node $user\_id\_creator$ in the ranked list of the node $user\_id$. Figure 25 shows the first five rows of the dataframe we get as a result.

timestamp	user_id_converted	user_id_creator_converted	user_id	user_id_creator	rank_of_creator
1270426272	314	604	ackernaut	Vonster	273
1270427914	310	86	micahbrich	cameronmoll	390
1270428164	310	551	micahbrich	bastianallgeier	350
1270428497	287	8	kiryn	clayglobal	92
1270428503	522	239	mg	jenniferdaniel	502

Figure 25: The dataframe contains the IDs and the converted IDs of the users along with the rank of *user_id_creator* in the ranked list of *user_id* for every edge of the test dataset.

During the preprocessing we dropped all duplicates from the dataset so that edges only appear once, thus we only make predictions for edges not seen before. When making recommendations for the source node of the current edge in the test data we will only recommend users with which the current source node previously did not form an edge in the test data. We also do not recommend users with whom it already formed an edge in the previous weeks.

As some of the these models have a random factor (for example DeepWalk is based on random walks) to get a comprehensive result we performed each evaluation ten times for every model.

6.4 Evaluation of the models with DCG and running time

To evaluate and compare the different models, we use the discounted cumulative gain (DCG) metric, which is capable of evaluating online machine learning algorithms [30]. For each interaction we calculate the DCG as

$$\mathrm{DCG} = \frac{1}{\log_2(\mathrm{rank} + 1)}$$

timestamp	user_id_converted	user_id_creator_converted	user_id	user_id_creator	rank_of_creator	dcg
1270426272	314	604	ackernaut	Vonster	273	0.123487

Figure 26: An example of the calculated DCG

The overall evaluation of each model is the average of the DCG values over all interactions and samples. If we look at Figure 27 we can see that the performances of the models are mostly similar with GraRep outperforming the other models.

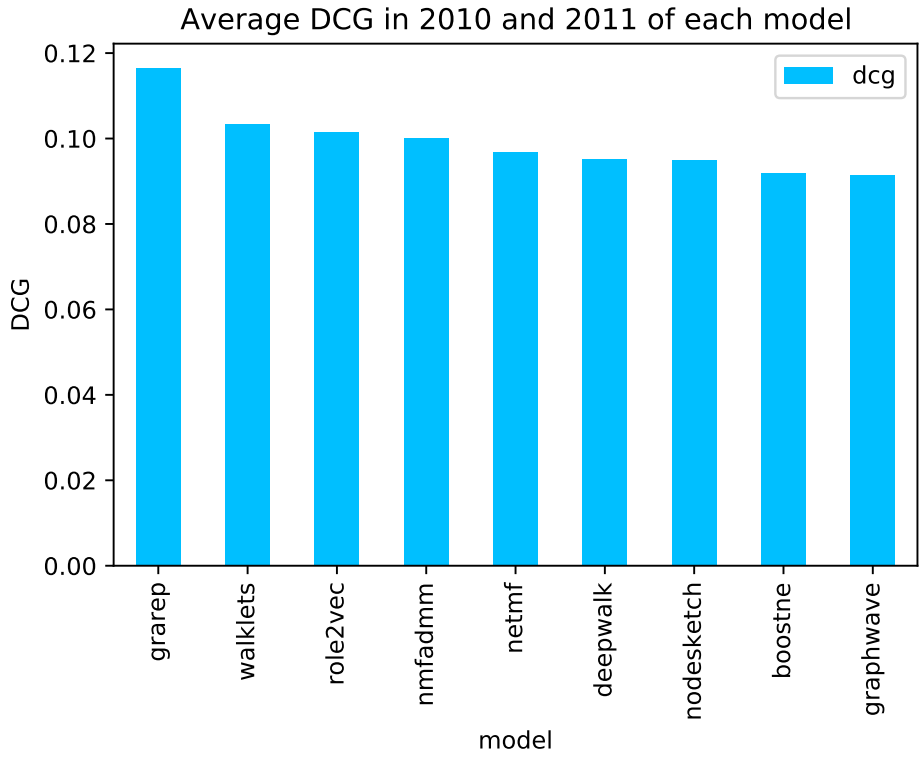


Figure 27: Average DCG in 2010 and 2011 of each model over all interactions and samples.

GraRep which appears to be the best model to solve this link prediction task is a factorization-based method that generates the node embeddings by the factorization of the k-step transition probability matrix A^k, where

$$A = D^{-1}S$$

if S is the adjacency matrix and D is the degree matrix of the input graph.

$$D = \begin{cases} \sum_l S_{il} & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

We can also examine performance tendencies by computing the weekly averages of the DCG. Figure 28 illustrates the weekly average DCG of each model.

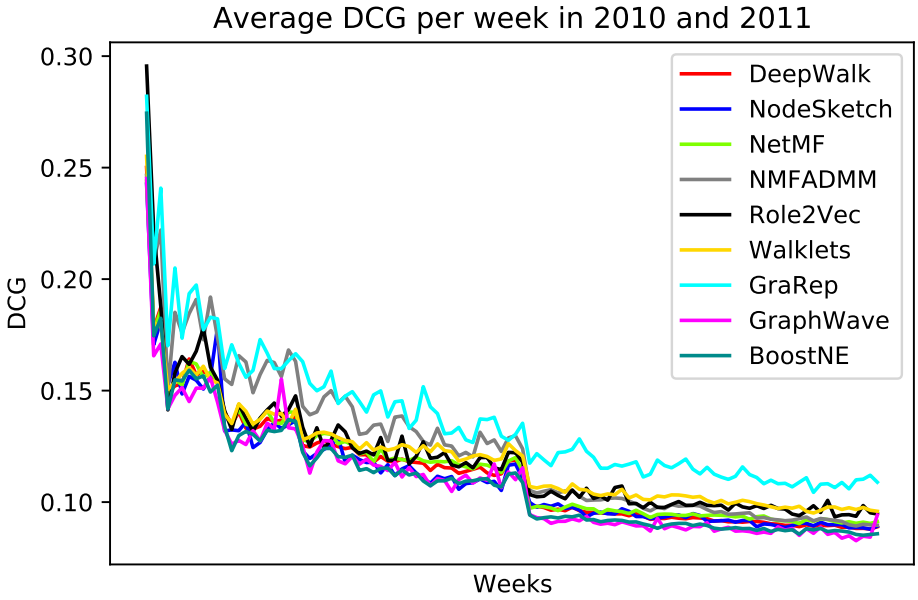
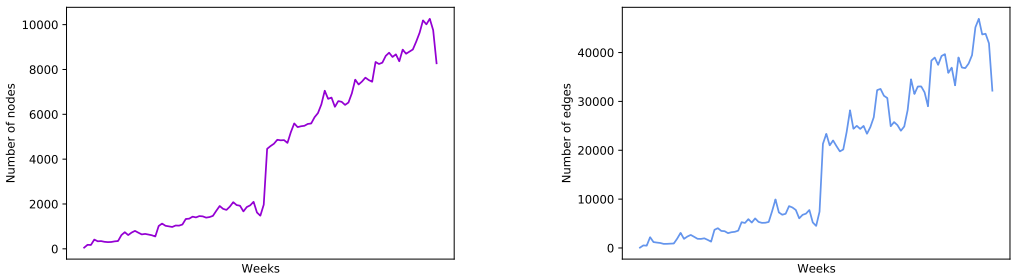


Figure 28: Weekly averages in 2010 and 2011 for each model

If we look at Figure 28 and 29 it is easy to see that as the graphs get bigger the performance of the models decrease on the link prediction task.

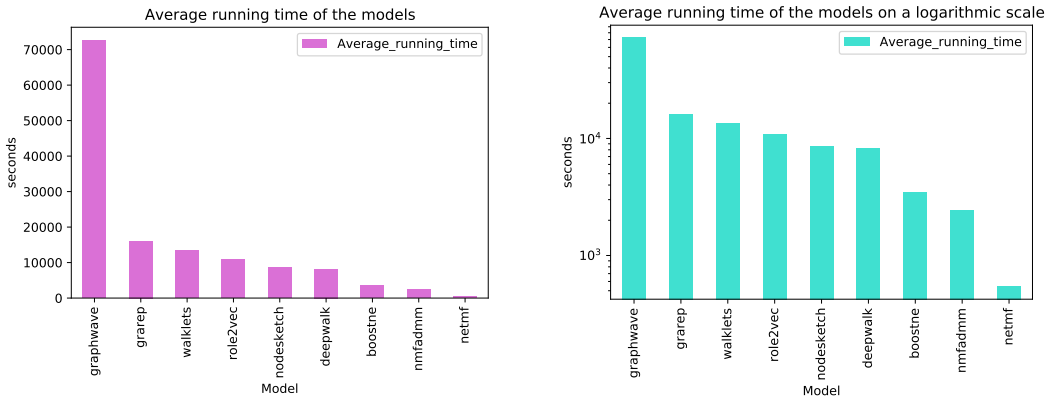


(a) Number of nodes

(b) Number of edges

Figure 29: Measurements of the largest connected components per week in 2010 and 2011

When evaluating different models another important aspect is the time necessary for its execution. Figure 30 shows that there is a much bigger difference between the models when it comes to running time.



(a) Linear scale

(b) Logarithmic scale

Figure 30: Average running time of the models on a linear and a logarithmic scale

7 Conclusion

In this dissertation we introduced the predecessor of node embedding methods; the vector space representation of words and several static node embedding methods along with online algorithms as well. The final part of the dissertation included the evaluation of static methods from the Python package of [28] based on their performance on a link prediction task using the Dribbble dataset. We measured these models by using weekly interactions as training and test dataset.

The experiment showed that the performance of these models decreases as the size of the training dataset increases. When examining the models DCG averages we could see that the models perform similarly but with very different running times.

7.1 Future work

Bigger differences between the models and better results could be achieved if the link prediction task would use for example several weeks data to make the prediction for the following week's interactions. However, the running time of these models would also considerably grow. Future work would include finding the best performing training and test dataset size and comparing the methods of [28] to online algorithms as well.

References

[1] David Liben-Nowell and Jon Kleinberg, *The Link Prediction Problem for Social Networks,* https://www.cs.cornell.edu/home/kleinber/link-pred.pdf (2004).

[2] Zan Huang and Dennis K.J. Lin, *The Time Series Link Prediction Problem with Applications in Communication Surveillance,* https://pdfs.semanticscholar.org/f8a2/6c822984681075491ee6601a66e6675d0e63.pdf .

[3] Alexandru Mara, Jefrey Lijffijt, and Tijl De Bie, *EvalNE: A Framework for Evaluating Network Embeddings on Link Prediction,* http://ceur-ws.org/Vol-2436/article_2.pdf (2004).

[4] Raf Guns, *Link prediction,* https://www.researchgate.net/publication/312813875_Link_Prediction/link/5bd1840f92851cabf2661399/download (2014).

[5] Muhan Zhang and Yixin Chen, *Link prediction,* LinkPredictionBasedonGraphNeuralNetworks (2018).

[6] Leo Katz, *A new status index derived from sociometric analysis,* http://people.cs.vt.edu/~badityap/classes/cs6604-Fall15/readings/katz-1953.pdf (1953).

[7] Liyan Dong, Yongli Li, Han Yin, Huang Le, and Mao Rui, *The Algorithm of Link Prediction on Social Network,* http://downloads.hindawi.com/journals/mpe/2013/125123.pdf (2013).

[8] Polina Rozenshtein and Aristides Gionis, *Temporal PageRank,* https://research.aalto.fi/files/26626013/temporal_pagerank.pdf (2016).

Rozenshtein, Polina; Gionis, Aristides

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, *Efficient estimation of word representations in vector space,* https://arxiv.org/abs/1301.3781 (2013).

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Grag S Corrado, and Jeff Dean, *Distributed representations of words and phrases and their compositionality,* https://arxiv.org/abs/1310.4546 , 2013.

[11] Xin Rong, *word2vec Parameter Learning Explained,* https://arxiv.org/pdf/1411.2738.pdf (2016).

[12] Janod Killian, Mohamed Morchid, Richard Dufour, and Georges Linares, *A Log-Linear Weighting Approach in the Word2vec Space for Spoken Language Understanding,* http://mohamedmorchid.free.fr/articles/slt2016_janod.pdf (2016).

[13] Hao Peng, Jianxin Li, Yangqiu Song, and Yaopeng Liu, *Incrementally Learning the Hierarchical Softmax Function for Neural Language Models,* https://pdfs.semanticscholar.org/28c7/430273a7f65b547f7a7e9c55eb5fde75dc3a.pdf (2016).

[14] Andriy Mnih and Geoffrey Hinton, *A Scalable Hierarchical Distributed Language Model,* https://www.cs.toronto.edu/~amnih/papers/hlbl_final.pdf (2008).

[15] Rasmus Hallén, *A Study of Gradient-Based Algorithms,* http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8904399&fileOId=8904400 (2017).

[16] William L Hamilton, Rex Ying, and Jure Leskovec, *Representation learning on graphs: Methods and applications,* https://arxiv.org/pdf/1709.05584.pdf (2018).

[17] Jure Leskovec, *Representation Learning on Networks,* http://snap.stanford.edu/proj/embeddings-www/ (2018).

[18] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena, *DeepWalk: Online Learning of Social Representations,* https://arxiv.org/pdf/1403.6652.pdf (2014).

[19] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang, *Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec,* https://arxiv.org/pdf/1710.02971.pdf (2018).

[20] Aditya Grover and Jure Leskovec, *node2vec: Scalable Feature Learning for Networks,* https://arxiv.org/pdf/1607.00653.pdf (2016).

[21] Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang, *Network Representation Learning: A Survey,* https://arxiv.org/pdf/1801.05852.pdf (2018).

[22] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei, *LINE: Large-scale Information Network Embedding,* https://arxiv.org/pdf/1503.03578.pdf (2015).

[23] Mikhail Belkin and Partha Niyogi, *Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering,* http://web.cse.ohio-state.edu/~belkin.8/papers/LEM_NIPS_01.pdf (2001).

[24] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim, *Continuous-Time Dynamic Network Embeddings,* https://dl.acm.org/doi/abs/10.1145/3184558.3191526 (2018).

[25] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu, *Embedding Temporal Network via Neighborhood Formation,* https://dl.acm.org/doi/10.1145/3219819.3220054 (2018).

[26] András A. Benczúr, Róbert Pálovics, Domokos M. Kelen, and Ferenc Béres, *Node embeddings in dynamic graphs,* https://link.springer.com/article/10.1007/s41109-019-0169-5 (2019).

[27] Johannes Wachs, Anikó Hannák, András Vörös, and Bálint Daróczy, *Why do men get more attention? Exploring factors behind success in an online design community,* https://arxiv.org/abs/1705.02972 (2017).

[28] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar, *An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs*, https://arxiv.org/abs/2003.04819 (2020).

[29] Kalervo Jarvelin and Jaana Kekalainen, *Cumulated Gain-Based Evaluation of IR Techniques*, https://www.cc.gatech.edu/~zha/CS8803WST/dcg.pdf.

[30] Róbert Pálovics, András Benczúr, Levente Kocsis, András Benczúr, Tamás Kiss, and Erzsébet Frigó, *Exploiting temporal influence in online recommendation*, https://dms.sztaki.hu/sites/dms.sztaki.hu/files/file/2014/recsys.pdf (2014).

[31] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar, *Karate Club Python package documentation*, https://karateclub.readthedocs.io/en/latest/notes/resources.html#neighbourhood-based-node-embedding (2020).

8 Appendix

8.1 Python code of the evaluation

```
import json, time, math
import scipy as sp
import pandas as pd
import numpy as np
import networkx as nx
from datetime import datetime

def convert_ids(G):
    user_map = dict(zip(list(G.nodes()), range(G.number_of_nodes()
    )))
    G_new = nx.Graph()
    new_edges = [(user_map[src], user_map[trg]) for src, trg in G.
    edges()]
    G_new.add_edges_from(new_edges)
    return G_new, user_map

def dot_ranking(user, others, row, M):
    index = list(range(M.shape[1]))
    # remove observed nodes from the toplist
    index.remove(user)
    for node in others:
```

```
        index.remove(node)
    # dot product is negated for ascending sort
    dists = -M[row,index]
    distances = list(zip(index,dists))
    distances.sort(key=lambda x: x[1])
    ranked_nodes = [a_tuple[0] for a_tuple in distances]
    return ranked_nodes

def dcg(arr):
    return 1.0 / np.log2(arr+1)

def fit_model(karate_obj, G):
    start_time = time.time()
    karate_obj.fit(G)
    embedding_arr = karate_obj.get_embedding()
    end_time = time.time()
    return embedding_arr, end_time - start_time

def preprocess_test(test, train_nodes, user_map):
    new_test = test[test['user_id'].isin(train_nodes) & test['user_id_creator'].isin(train_nodes)]
    new_test = new_test.reset_index()
    new_test['user_id_converted'] = new_test['user_id'].apply(lambda x: user_map[x])
    new_test['user_id_creator_converted'] = new_test['user_id_creator'].apply(lambda x: user_map[x])
    return new_test[['timestamp','user_id_converted','user_id_creator_converted',"user_id","user_id_creator"]].copy()

def eval_model(history_dict, training, test, model):
    # preprocessing
    G_train = nx.from_pandas_edgelist(training, 'user_id', 'user_id_creator', create_using=nx.Graph())
    if nx.is_connected(G_train) == False:
        GCC = sorted(nx.connected_components(G_train), key=len, reverse=True)
        G_train=G_train.subgraph(GCC[0])
    H, user2idx = convert_ids(G_train)
    # training model
```

```
    embedding, elapsed_training_time = fit_model(model, H)
    # link prediction
    train_nodes = nx.nodes(G_train)
    ranking_df = preprocess_test(test, train_nodes, user2idx)
    users_unique=list(ranking_df.user_id_converted.unique())
    M = np.dot(embedding[users_unique,:],embedding.T)
    ranking_dict={}
    ranks = []
    edges_list = list(zip(ranking_df['user_id_converted'],
ranking_df['user_id_creator_converted'], ranking_df['user_id'],
    ranking_df['user_id_creator']))
    for user, creator, user_global_id, creator_global_id in
edges_list:
        # prepare to observe unseen user
        if not user_global_id in history_dict:
            history_dict[user_global_id] = set()
        # extract recommendation for unseen user
        if not user in ranking_dict:
            others = []
            for u_id in history_dict[user_global_id]:
                if u_id in user2idx:
                    others.append(user2idx[u_id])
            ranking_dict[user] = dot_ranking(user, others,
users_unique.index(user), M)
        # query rank
        ranks.append(ranking_dict[user].index(creator))
        # update toplist
        ranking_dict[user].remove(creator)
        # store interaction for the upcoming time intervals
        history_dict[user_global_id].add(creator_global_id)
    # evaluation metrics
    ranking_df['rank_of_creator'] = np.array(ranks) + 1
    ranking_df["dcg"] = ranking_df["rank_of_creator"].apply(dcg)
    mean = ranking_df["dcg"].mean()
    return ranking_df, mean, elapsed_training_time
```

Listing 1: Python code of the evaluation of the models

```
import pandas as pd
from link_prediction import *
from karateclub import Diff2Vec, Role2Vec, DeepWalk, Walklets,
    NetMF, BoostNE,  GraRep, LaplacianEigenmaps, HOPE, NodeSketch,
    NMFADMM, GraphWave
from tqdm import tqdm
import pandas as pd
import numpy as np
import sys

def run(algo, time_column, sample_id="0"):
    # choose node embedding
    if algo == "diff2vec":
        model = Diff2Vec()
    elif algo == "role2vec":
        model = Role2Vec()
    elif algo == "deepwalk":
        model = DeepWalk()
    elif algo == "walklets":
        model = Walklets()
    elif algo == "netmf":
        model = NetMF()
    elif algo == "boostne":
        model = BoostNE()
    elif algo == "grarep":
        model = GraRep()
    elif algo == "laplacian":
        model = LaplacianEigenmaps()
    elif algo == "hope":
        model = HOPE()
    elif algo == "nodesketch":
        model = NodeSketch()
    elif algo == "nmfadmm":
        model = NMFADMM()
    elif algo == "graphwave":
        model = GraphWave()
    else:
        raise ValueError("Invalid algorithm!")
    # load data
```

```
38      likes =  pd.read_csv('/mnt/idms/home/petrok/szakdolgozat/
    likes_2010_11.csv')
39      # split data over time
40      parts = []
41      for group in likes.groupby(time_column):
42          #print(group[0])
43          parts.append(group[1])
44      print("Number of intervals:", len(parts))
45      # run experiment
46      history_dict = {}
47      ranks_arr, dcg_arr, time_arr = [], [], []
48      if algo == "laplacian" and time_column == "week":
49          # eigenvalue error for the first interval
50          from_index = 2
51      else:
52          from_index = 1
53      for i in tqdm(range(from_index,len(parts))):
54          tmp_ranks, tmp_dcg, tmp_time = eval_model(history_dict,
    parts[i-1], parts[i], model)
55          # new columns are needed for easier result aggregation
    later
56          tmp_ranks["model"] = algo
57          tmp_ranks["setting"] = time_column
58          tmp_ranks["sample_id"] = sample_id
59          ranks_arr.append(tmp_ranks)
60          dcg_arr.append(tmp_dcg)
61          time_arr.append(tmp_time)
62          tmp_ranks.to_csv('%s_%s_%i_%i_sample%s.csv' % (algo,
    time_column, i-1, i, sample_id))
63      return np.mean(dcg_arr), np.sum(time_arr)
64
65  if __name__ == "__main__":
66      if len(sys.argv) == 4:
67          algo = sys.argv[1]
68          time_column = sys.argv[2]
69          # in order to differentiate outputs of the same algorithm
70          sample_id = sys.argv[3]
71          mean_dcg, total_time = run(algo, time_column, sample_id)
72          print("Arguments:", algo, time_column, sample_id)
```

```
73          print("Mean DCG: %.4f" % mean_dcg)
74          print("Total training time: %.2f" % total_time)
75          with open("runtime_%s_%s_sample%s.txt" % (algo,
    time_column, sample_id), 'w') as f:
76              f.write("%.2f\n" % total_time)
77      else:
78          print("Usage: <algo_type> <time_column> <sample_id>")
```

Listing 2: Running the evaluation algorithm from the console