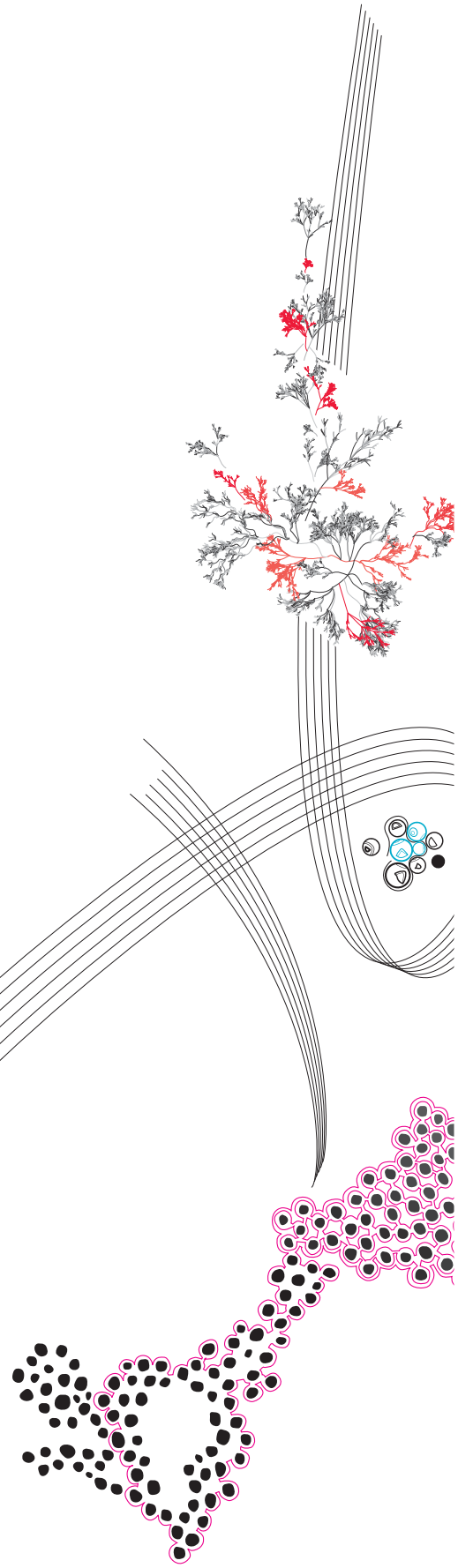

BSc Thesis Applied Mathematics

How well can personal network features be retrieved?

Bo Arends

Supervisor:
Clara Stegehuis
Riccardo Michielan

January, 2023

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

I wrote this paper for my Bachelor's Thesis Applied Mathematics at the University of Twente. I want to thank my supervisors Clara Stegehuis and Riccardo Michielan for helping me with the process.

How well can personal network features be retrieved?

B. Arends*

January, 2023

Abstract

This paper investigates how good certain algorithms can retrieve node features of nodes in networks when only observing the network connections on random uniformly distributed graphs up to 300 nodes. Algorithms like node2vec use random walks to retrieve information from a network, after which it makes an embedding of the graph. We are interested in the statistics of the original graph and the embedding to see if the algorithm can really capture the node features of the graph well. We investigated the degree of the nodes, the amount of triangles, the clustering and the number of edges that are in the graph but not in the embedding and vice versa.
We also looked if changing parameters to favor breadth first search or depth first search of the random walks changed the outcome. Changing certain parameters did not change the properties of the embedded graph, for example changing the parameters of node2vec that favor breadth first search or depth first search. One parameter that really changed the properties of the embedded graphs was changing the dimension of the the random geometric graph. We could conclude that changing the parameters that favor either depth first search or breadth first search did not change the statistics for the random graph and the embedding. Changing the dimension of the embedding also did not influence the statistics. However, changing the dimension of the random generated graph did significantly change the outcome.

Keywords: Information networks, feature learning, random walks

*Email: b.arends@student.utwente.nl

Contents

1 Introduction

1.1 Problem description

In network data sets, often only the network edges are observed and not the information that comes with the network nodes. However these nodes often have certain features. Since similar features often make it more likely for two nodes to connect, these features can be used for making predictions on personal recommendations or people's behavior. Think of social media networks like Instagram or Facebook, nodes stand for social media profile of people that get recommendations to connect with certain profiles because of other shared connections. Several algorithms have been developed that try to retrieve these features using random walks and make an embedding of the original graphs, where the embedding is a vector representation of the graph capturing node characteristics. Nodes with similar neighbors will have similar vectors and will be points that are closer together in the embedding.
Algorithms that may be able to retrieve these node features may have a possible influence on the privacy on for example social media networks. If algorithms are able to retrieve node features from a network, how much privacy do we have?
I will investigate how good these embedding algorithms are at retrieving features that created the original graph. We will investigate the following research questions:

- How well can node2vec retrieve features from the original graph?
- What is the influence of knowing the correct node2vec and graph parameters on retrieving these features?
- Can we apply network-based methods to compute the accuracy of node2vec?

2 Feature learning algorithms

2.1 Prior work

There has been a lot of research in networks in recent years. Networks consist of nodes and edges and contain information. The nodes contain certain attributes and are linked to each other with edges. Think of a social media network where people are connected to each other. The edges between nodes tell us what people are connected to each other. The nodes, the social media profile of a person, contains certain attributes like gender, age and interests. Two nodes with similar attributes are often more likely to be linked to each other.

With networks like social media or websites like Wikipedia that are constantly expanding, there is a need for algorithms that can handle the data of large networks and analyze these networks.
In recent years there have been several approaches for analyzing these networks. Finding a good approach is challenging because they have to perform well on networks with millions of nodes and edges. Developing more efficient algorithms usually consists of a trade-off between accuracy and computational efficiency.

One method to analyze these networks is to use embedding methods. The embedding method transforms the network into n amount of vectors in a d-dimensional space, a vector for each node in the graph. These embedding methods place nodes that are similar in the graph closer together in the embedding.

The input of embedding algorithms are a graph with nodes and edges. In this paper, we look at algorithms that extract a set of random walks from the graph. There are a certain amount of random walks per node in the graph with a predefined random walk length. After the random walks we have information of the neighbourhoods of these nodes.

One of these algorithms is DeepWalk [2]. DeepWalk uses uniform random walks. The downside of this is that the user has no control over the explored neighborhood because there is no ability to favor depth first search or breadth first search. In depth first search, we start at a root node and proceed to travel through nodes to go as far as possible. In breadth first search, we start at a root node and go to all the direct or first level neighbours of that node. After that we travel to the first level neighbours of those nodes, etcetera.

Another algorithm is LINE [3], where a breadth-first strategy is used. The downside here is that nodes and clusters at further depth of the graph are not explored as much.

An algorithm that is flexible and controllable as opposed to the two examples mentioned above is node2vec [1].

However, there are only few theoretical results on these types of embedding algorithms. Suppose that we know the features that generated the network, and that the network is created based on a mathematical rule based on these features. How well can these algorithms, in particular node2vec, then retrieve the node features that generated the network?

2.2 Node2vec algorithm

Node2vec is one of the most recent algorithms used for multi-label classification on network nodes and link prediction of edges between network nodes.
Node2vec allows users to map nodes of a graph G to an embedding space, which usually is of lower dimension than the number of nodes in the graph. In this embedding, points that are close together in proximity are similar to each other. This way you can more easily see the structure of your input graph.

2.2.1 How the algorithm works

The input of the algorithm is a graph $G = (V, E)$, where E are the graph edges and V are the graph nodes. We start with the sampling process.

The algorithm simulates random walks starting at each node in the graph. The random walks have a length l, and we can choose how many random walks per node we want. For the random walk node2vec uses a formula to determine the transition probabilities between two nodes.

This is the transition probability for moving to node x given we were at node v:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Here Z is a normalizing constant and π_{vx} is the unnormalized transition probability between x and v.

The easiest way to bias these random walks is giving weights to the edges. With unweighted networks this does not work. Node2vec introduces two parameters p and q of which we can change the value. The parameter p controls the likelihood of immediately revisiting a node in the walk. If p is higher, the walk will most likely move away more from the starting node. If p is low, the walk would likely backtrack a step and keep it closer to the starting node. The parameter q differentiates between inward and outward nodes. If q is higher, the random walk is biased towards nodes closer to the source node, which favors breadth first search. If q is lower, the random walk is more inclined to visit nodes further away from the search node, favoring depth first search.
If we just travelled from node t to node v on edge (t, v), we need to transition probabilities to evaluate what node we travel to next. These transition probabilities on edges (v, x) are π_{vx}. The unnormalized transition probabilities are $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$, where w_{vx} are the static edge weights, which are 1 in case of unweighted graphs. Node2vec sets $\alpha_{pq}(t, x)$ to:

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

where d_{tx} is the shortest path distance between nodes t and x.

After the sampling strategy we have a good overview of the neighborhoods of all the nodes in the graph. Node2vec aims to learn the mapping function

$$f : V \to \mathbb{R}^d \tag{3}$$

Here d is the number of dimensions of our feature representation. So the function f maps a node to vector of d dimensions to an embedding space. Suppose that we have two vectors, $f(v)$ and $f(u)$. These both get mapped to the embedding space. To measure similarity in the embedding space we use the dot product to measure the angle between the two. A smaller angle means that the vectors, so the nodes v and u, are similar to each other, so have similar neighborhoods in the original graph. If we have one node v, node2vec will normalize all the similarity scores of all the other nodes in the graph to see which nodes are the most similar to v.

We want to have a mapping for each source node u. Node2vec uses equation (4) below to apply to all the nodes. We optimize the sum (4) using stochastic gradient descent, optimizing the probability of observing a network neighborhood $N_S(u)$ of node u where it is conditioned on the feature representation that we defined with function f:

$$\max_f \sum_{u \in V} \log P(N_s(u)|f(u)) \tag{4}$$

For this optimization problem there are two standard assumptions. Conditional independence while observing difference neighborhood nodes. This means that the likelihood of observing a neighborhood node is independent of observing any other neighborhood node given the feature representation of the source. The second one is symmetry in the feature

space, which means a source node and neighborhood node have a symmetric effect on each other in the feature space. This reduces the above equation to:

$$\max_f \sum_{u \in V} \Big[-\log Z_u + \sum_{n_i \in N_s(u)} f(n_i) \cdot f(u) \Big] \tag{5}$$

where Z_u is equal to:

$$Z_u = \sum_{v \in V} \exp((f(u) \cdot f(v)) \tag{6}$$

and equals the per-node partition function.

The advantage of node2vec is that is scalable to certain network sizes and easily controllable by the parameters you can change. One of these advantages is that node2vec can easily favor either breadth first search and structural equivalence of depth first search and homophily, unlike algorithms like LINE and DeepWalk. If we take q=0.5 and p=1 as parameters, the random walk will favor breadth first search over depth first search. The values for p and q lie between 0 and 1.

3 Method

To investigate how good the node2vec algorithm is at finding certain node features, We use a Python package called NetworkX and the existing node2vec algorithm in Python. With NetworkX you can create random graphs. The simplest one is the random geometric graphs, which creates a random graph of N nodes within a unit cube.

We will start with a graph with a certain amount of nodes. We will run node2vec on that graph, which will create an embedding of the graph. By creating edges between the points in the embedding, we get another graph again, with different nodes and edges than in the original graph. After that we look at some statistics to see how good node2vec is at preserving the node features.

We started with a small graph where we chose the positions of the nodes ourselves. Below you see an example of a self-made graph of 10 nodes on the left and a random geometric graph with 10 nodes on the right, with a radius of $p = 0.4$

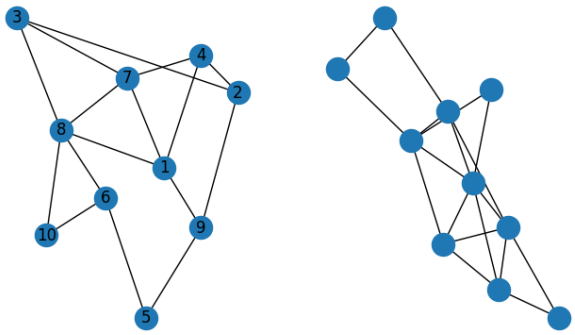

FIGURE 1: Left: Self made graph, Right: graph made by the random geometric graph generator

3.1 Random Geometric Graph

One of the most important benchmarks to test embeddings, is to use random geometric graphs that create random networks based on their feature vectors. One such random graph is the standard random geometric graph. This random graph creates a geometric graph in the unit cube based on a few parameters. The first one is the amount of nodes, these will be placed uniformly random within the unit cube. The second one is the radius, this determines what edges will be drawn based on what nodes are within the radius of each other. Another parameter is the dimension of the cube, the standard here is 2. The last one is the distance metric, where the standard is the Euclidean metric.

3.2 Node2Vec Python

We use the existing node2vec python code in combination with the functions of NetworkX. We give the node2vec code a graph and decide on the parameters, and node2vec will make an embedding of that graph. A few parameters are the embedding dimensions, the length of the random walks, the number of random walks per node, and search bias parameters p and q.

Giving a graph and certain parameters will result in an embedding of the graph, where similar nodes are put close in the embedding. Below is an example with 3 small graphs consisting of 15 nodes. Node2vec makes an embedding, which is a scatter plot, after which we make a graph of the embedding by connecting edges that are close together. We choose a threshold of the distance between edges for which there are approximately the same amount of edges in the original graph and the graph after the embedding. If there are not approximately the same amount of edges in both the graph and the embedding, the results will be skewed. For example if the embedding has significantly less edges than the original graph. There will be less triangles within the embedded graph, less clustering, and a lot of edges from the original graph will disappear in the embedded graph, and there is no use in comparing these statistics with the original graph.

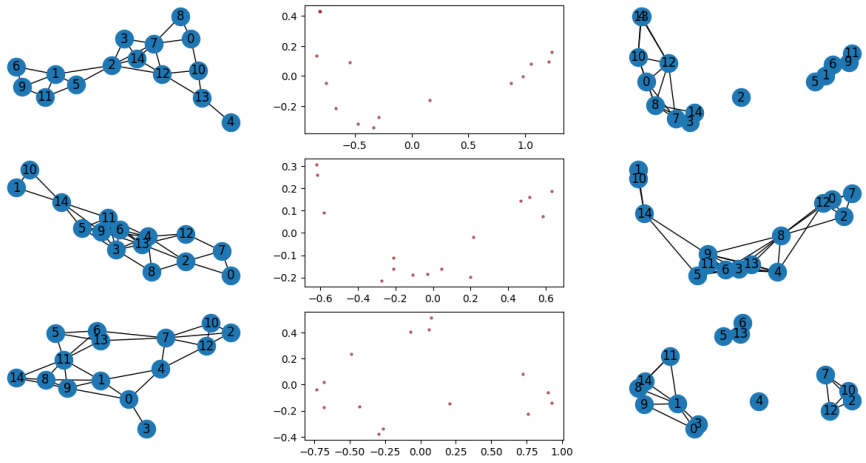

FIGURE 2: Left: random graph, middle: embedding, right: graph of embedding

3.3 Method

To analyse if the embeddings node2vec makes preserve all the node features, we look at a few different graph statistics.

First of all, we go through all the parameters we use for creating the random graphs and the node2vec algorithm. First of all, the radius for the random geometric graphs, which is set to 0.4. This radius is the distance for which the random geometric graph creates an edge if two nodes are within that distance of each other. The dimension of the embedding is equal to 128. The length of the random walks is 80, and there are 10 random walks per node in the graph.

After we set the parameters we will look at some statistics. The first one is the degree of the nodes. We want to know if we have a similar distribution of the degree of the nodes in the graph and the embedding of the graph. We do this by plotting a histogram with the degree on the x-axis and the frequency of that degree on the y-axis.

The second one is the clustering coefficient C_i, where i is the node. The clustering coefficient is:

$$C_i = \frac{\text{number of triangles connected to i}}{\text{number of triples centred around i}} \tag{7}$$

where a triple centred around node i is a set of two edges connected to i. The clustering coefficient for the whole graph is the average of all the clustering coefficients of all the nodes in the graph:

$$C_i = \frac{1}{n}\sum_{i=1}^{n} C_i \tag{8}$$

where n is the total number of nodes in the graph.

Another statistic to analyse is the number of triangles in a graph compared to the number of triangles in the embedding.

The last statistic is the number of edges that are in the original graph but not in the embedding, and the number of edges that are in the embedding but not in the original graph. We take this number of edges divided by the total number of edges in the graphs to get a percentage.

For the histogram we use 10 graphs with 300 nodes per graph. For the other statistics we take graphs of 50, 100, 150, 200, 250 and 300 nodes to be able to see how it changes when the size of the graph increases. We take 10 graphs each time and take the average in order to give an accurate result. We also change the values of p and q and the graph dimension, to see if that changes the results. We have five different cases. First, we change the values of p and q in the node2vec algorithm to see if it makes a difference. In the first case, p and q are both 1. In the second case, p is 0.5 and q is 1, favoring breadth first search. In the third case, p is 1 and q is 0.5, favoring depth first search. In the fourth case we set p and q both to one, but we set the dimensions of the random geometric graph generator to 4 instead of 2, while keeping the dimension of the embedding the same. In the last case, we keep the dimension of the random geometric graph on 2, while changing the dimension of the embedding from 128 to 64.

4 Results

We start with 4 figures with 4 subplots each. In the subplots we can see the number of edges, the clustering of the graph, the number of triangles in the graph and the relative amount of edges that are in the graph but not the embedding, and vice versa.

In figure 3, both p and q are equal to 1. In figure 4 and 5 we changed the values of p and q. In figure 6, we changed the dimension of the random geometric graph from 2 to 4. We used 6 graphs, starting at 50 nodes and with the largest one having 300 nodes.

The x-axis in the figures below is the amount of nodes of the graphs we use. The number of edges and number of triangles are integers, and the clustering and relative edge differences are numbers between 0 and 1.

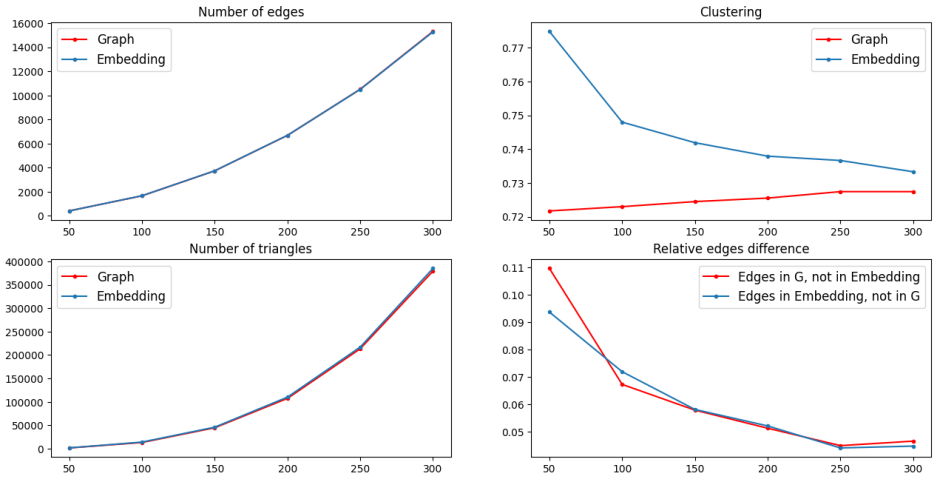

FIGURE 3: Statistics for p=1 and q=1

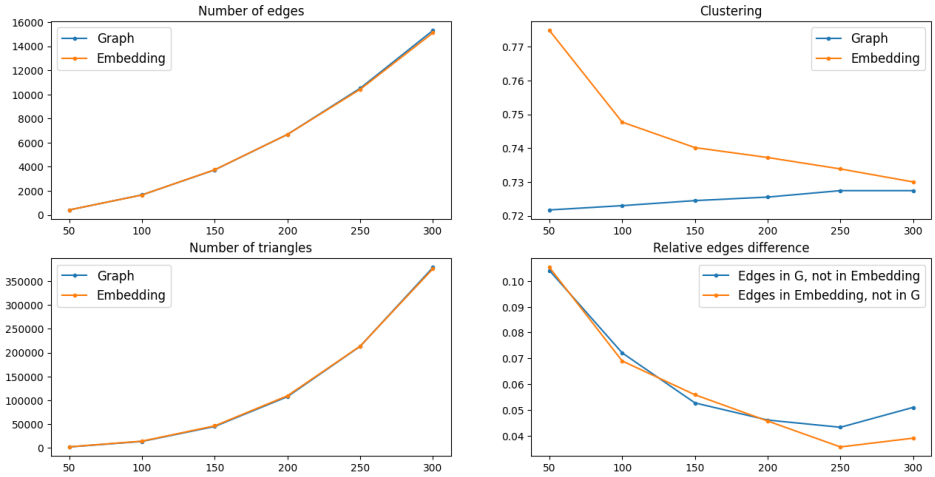

FIGURE 4: Statistics for p=1 and q=0.5

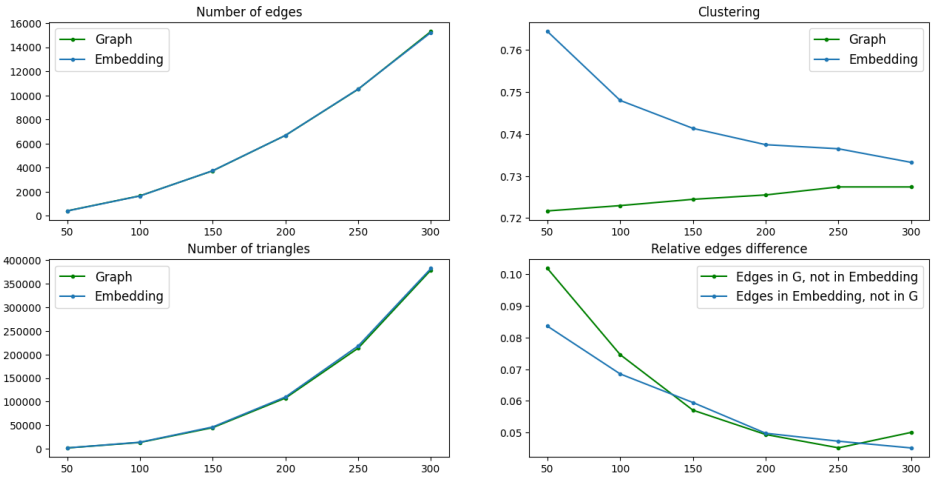

FIGURE 5: Statistics for p=0.5 and q=1

As we can see above for all 3 figures, the number of edges and triangles grow exponentially when the number of nodes increase. This holds for both the graph and the embedding, where the results overlap. For all figures, the clustering converges to around 73% when the amount of nodes increases. Since the clustering has to be between 0 and 1, a score of 0.73 means there is a decent amount of clustering in the graph, so a lot of node neighbours are connected to each other. Also the relative edge differences converges to around 5%.

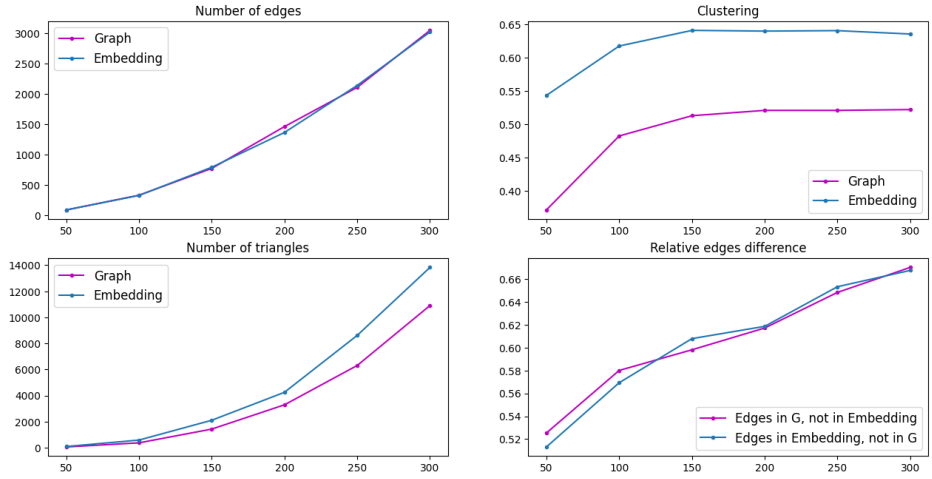

FIGURE 6: Statistics for p=1, q=1 but graph dimension=4

In the figure above, we changed the dimension of the random geometric generator to 4 instead of 2. We can see here that the results differ from the figures above. The clustering is lower for both the graphs, that converges to 50%, and the embedding, that converges to 65%. The relative edge difference seems to keep increasing when we increase the number of nodes, going above 66%.

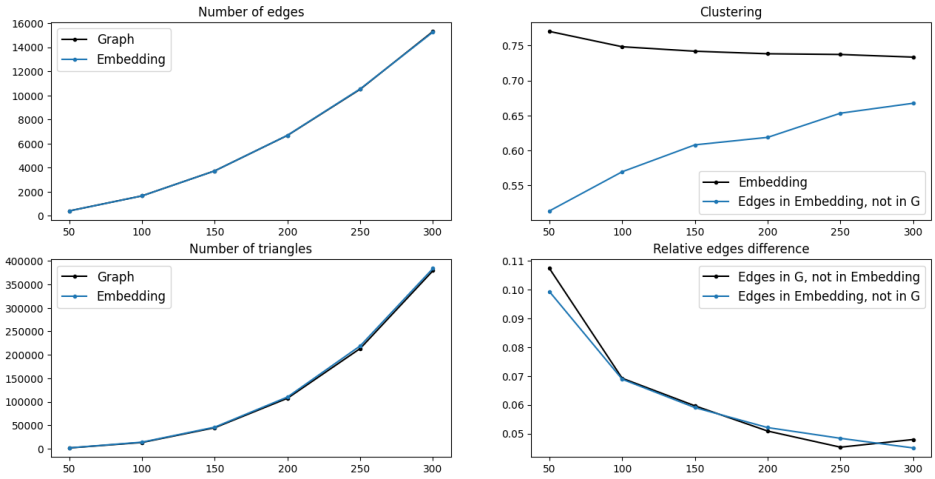

FIGURE 7: Statistics for p=1, q=1 but embedding dimension=64

In the figure above we kept the dimension of the random geometric graph at 2, but change the dimension of the embedding, from 128 to 64. We can see that the results are similar to the results above where we changed p and q. The clustering converges to around the same number, around 0.73. It does take graph with more nodes for the graph and embedding clustering be approximately the same number. The relative edge difference here also converges to around 5%.

In the figures below we have 3 histograms, where we have the frequency of the degree of the nodes. We took 10 graphs of 300 nodes each, and just added all the values together for the histogram. For one graph, these values could be divided by 10.

Below that is a table where we have the average and standard deviation for all 3 histograms.

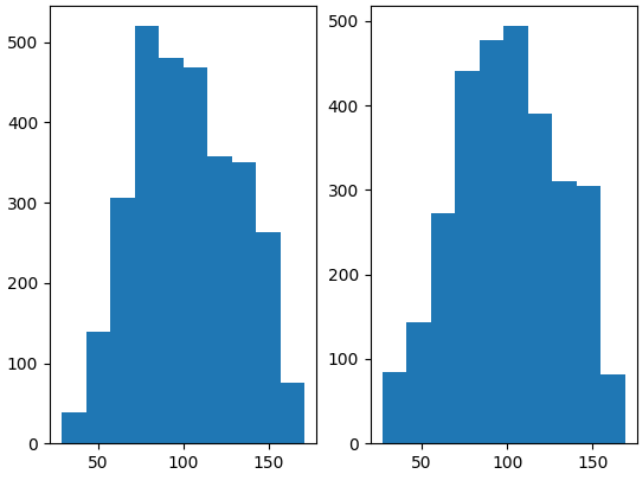

FIGURE 8: Histogram for the degree of the nodes, for 10 graphs. Parameters p=1 and q=1. Left original graph, right embedding.

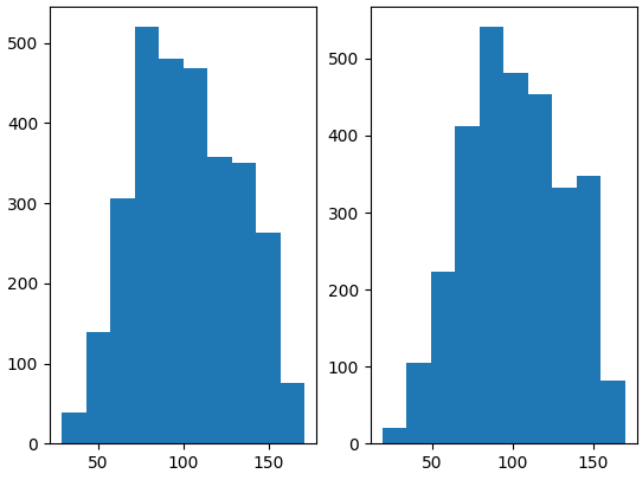

FIGURE 9: Histogram for the degree of the nodes, for 10 graphs. Parameters p=0.5 and q=1. Left original graph, right embedding.

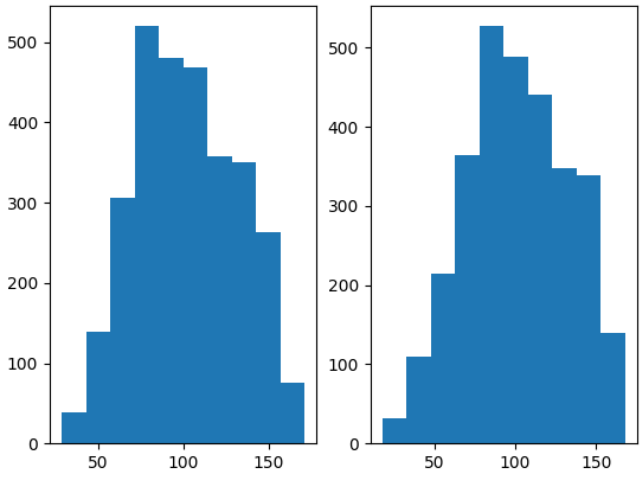

FIGURE 10: Histogram for the degree of the nodes, for 10 graphs. Parameters p=1 and q=0.5. Left original graph, right embedding.

p & q	Graph avg	Embed avg	Graph std	Embed std
p=1, q=1	102.0593	100.6907	29.8111	30.8690
p=0.5, q=1	102.0593	101.6553	29.8111	30.4012
q=1, p=0.5	102.0593	101.0793	29.8111	31.6194

TABLE 1: Averages and standard deviations of the histograms

We can see in the table above that the average values and the standard deviation for the graph and embedding barely differ.

5 Conclusions

We can see that when the graph dimension is equal to two and the embedding dimension is equal to 128, the measurements on the original graph and embedding grow close together when the amount of nodes on the graphs increases. For all three cases of p and q, the amount of triangles stay the same for the original graph and the embedding. For the clustering, we see that for all three cases when the node count increases, the clustering of the graphs converges to the same value. For the graph, the clustering converges from below and for the embedding it converges from above.
If we look at the relative edge difference, we see that when the amount of nodes increases, the amount of edges that are in the graph but not in the embedding and vice versa go down to a maximum of only 5 percent of the total amount of edges in the graph. From this we can conclude that node2vec really captures the node features well, since a very high percentage of edges in the original graph are found in the embedding.
We can also conclude that changing the values of p and q do not significantly change the outcome of the results.
Lowering the dimension of the embedding did not seem to affect the results, because the values for the amount of edges, triangles, clustering and the relative edge difference are all very similar to the graphs where the dimension was higher, as there is no noticeable difference.
After this we look at the case where the dimension of the original graph is 4. The results here are very different than when we choose the dimension to be 2. The clustering when the dimension is 4 is way lower, as well as the amount of triangles. The most remarkable here is that the relative edge difference does not stagnate but keeps increasing even at 300 nodes. These results are reasonable, because we generate a graph with a higher dimension while keeping the dimension of the embedding the same. So we have a random graph with more information than first, trying to capture it with the same amount of dimensions as first.

6 Discussion

All the statistics that we investigated are on relatively small graphs with 300 nodes on less. Node2vec is developed to work on networks that are way larger, with more than thousands of nodes. The results for these large graphs might differ from the smaller graphs we used, but since it takes approximately 45 minutes for python to run node2vec on a graph of 500 nodes, it was not feasible to do tests with these kinds of graphs.
This might change the results even more for the case where the dimension of the original graph is 4, where we use the same amount of nodes for when the dimension is equal to 2.

In this paper we used python to generate uniformly distributed graphs. Since in the real world networks are usually not uniformly distributed, this may influence the results.

References

[1] Leskovec-J. Grover, A. Node2vec: Scalable feature learning for networks. pages 855–864, August 2016.

[2] Al-Rfou R. Skiena S. Perozzi, B. Deepwalk: Online learning of social representations. 96:701–710, 2014.

[3] Qu-M. Wang M. Zhang M. Yan J. Mei Q. Tang, J. Line: Large-scale information network embedding. pages 1067–1077, May 2015.