Estimating the Betweenness Centrality of Nodes with Network Embedding

Final project, Luca Schinnerl

June 2021

1 Introduction

In graph theory, a key area is to estimate the importance of specific nodes in a network. This is usually done with a centrality measure. There are many different types of these centrality measures and they can be classified under the following categories (among others):

1. Degree Centrality;
2. Eigenvector Centrality;
3. Closeness Centrality;
4. Shortest path Centrality;
5. Group Centrality.

However, a widely accepted method to quantify the importance of nodes is the betweenness centrality. This method is based on a shortest paths approach. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of the weights of the edges (for weighted graphs) is minimised. The betweenness centrality for each vertex is proportional to the number of these shortest paths that pass through the vertex. This shortest path approach has many real world use cases:

1. Applicable to a diverse set of problems in network theory;
2. Applicability in biology [1];
3. Applicability to many types of social networks [2];
4. Applicability to traffic flows [3].

The actual number of use cases far exceeds this short list. It can be concluded that the betweenness centrality is one of the most important centrality measures used within the context of graph theory.

1.1 Definition of betweenness centrality

Mathematically the betweenness centrality can be expressed through the following equation:

$$g(v) = \sum_{j \neq v \neq k} \frac{\sigma_{jk}(v)}{\sigma_{jk}}, \tag{1}$$

where v represents a node, σ_{jk} the total number of shortest paths from node j to node k and σ_{jk} the total amount of these shortest paths running through node v. It is important to note that the betweenness centrality scales with the number of pairs of nodes. This means that for larger networks, with a high number of nodes, the expected betweenness centrality for each node is huge. To counter this problem, equation 1 can be re-scaled to (for undirected graphs):

$$g(v) = \frac{1}{(N-1)(N-2)} \sum_{j=1;j\neq v}^{N} \sum_{k=1;j\neq v,j}^{N} \frac{\sigma_{jk}(v)}{\sigma_{jk}} \tag{2}$$

while N is the number of nodes in the network. The most efficient way of calculating the betweenness centrality for an undirected network is at least of order $O(|V||E|)$ using Brandes' algorithm [4].

1.2 Problem definition

This paper discusses a new algorithm which can be used to approximate the betweenness centrality for undirected and unweighted graphs. The discussion includes how the algorithm is able to approximate this centrality for very large networks, where a direct calculation with Brandes' algorithm is highly complex and takes a very long time. The goal of this research is to test whether or not it is possible to estimate exactly this centrality through network embedding. More formally, consider the problem of classifying nodes on a network with respect to the betweenness centrality of each node. We have a graph $G(V, E)$, where V represents the nodes of a specific graph G, while $E \subseteq (V \times V)$ are the edges. This graph can be extended to include labels $X \in \mathbb{R}^V$ which represents the betweenness centralities of each node in the graph. In this machine learning task, we want to embed the nodes in the graph into a lower dimensional space in which nodes with similar betweenness centrality are co-located. Ultimately, the goal is to embed large and complex networks, to make this algorithm as useful as possible in real word scenarios. Current algorithms are very efficient in calculating the betweenness centrality for small and simple networks. Estimations are not useful in these cases. In real world graphs such as social networks, it is often the case that a low subgroup of the nodes have a high betweenness centrality while most have very few shortest path running through them [5]. This means the most important task at hand is to identify exactly this group of nodes.

2 Algorithm

The pseudo code for the algorithm itself is summarised in algorithm 1. The core elements of this algorithm are comprised of two biased random walks and a modified SkipGram which is summarised in algorithm 2. The input parameters that are needed for the algorithm are summarised and explained in table 1.

First of all, a matrix representation Φ from $\mathcal{U}^{|V| \times d}$ has to be initialised. This object is simultaneous the output and it is the target of the training's process. This is followed by randomly sampling $\mathcal{A} * |V|$ nodes from the whole sample of nodes present in the graph $G(V, E)$. Next, for every node in the resulting sample, two random walks are performed, with which the matrix Φ is optimised via a SkipGram model. These biased random walks are explain in section 2.1, while the SkipGram is substantiated in section 2.2. Finally, the algorithm return an embedding space representation Φ for all nodes in the graph $G(V, E)$.

Algorithm 1: BetweennessApproximator($G, e, w, d, \gamma, \eta, t, \mathcal{A}$)

```
Input  : graph $G(V, E)$
         epochs $e$
         window size $w$
         embedding size $d$
         walks per node $\gamma$
         learning rate $\eta$
         walk length $t$
         approximating factor $\mathcal{A}$
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$.
1  Initialisation: Sample $\Phi$ from $\mathcal{U}^{|V| \times d}$
2  for $i = 0$ to $e$ do
3      $O(|V||E|)$ = Shuffle($V$)
4      $O'(|V||E|)$ = Sample $\mathcal{A}$% of nodes from $O(|V||E|)$
5      for all $v_i \in O'(|V||E|)$ do
6          $\mathcal{W}_{v_i} = RandomWalk(G, v_i, \gamma, t)$
7          sample one $w_i \in O(|V||E|)$
8          $\mathcal{W}_{w_i} = RandomWalk(G, w_i, \gamma, t)$
9          SkipGram($\Phi, \mathcal{W}_{v_i}, \mathcal{W}_{w_i}, w$)
10     end for
11 end for
```

2.1 Biased random walk

The biased random walk is based on the random walk which is part of the deepwalk algorithm and was first introduced in 2014 [6]. Like the name suggests, this algorithm performs a random walk on the input graph. All the visited nodes are then stored into an array. An example of a specific walk with this algorithm can be seen in figure 1. In general, the biased random walk takes a node as input, and chooses a random neighbour of this node. A specific node is chosen with a probability proportional to the number of neighbours this neighbouring node has. This means, the higher the degree of a neighbouring node is, the more likely it is for this node to be chosen. Walking backwards is however not possible, which means a sequence of $[v_i, v_j, v_i]$ would never occur. This guarantees that the algorithm simulates a depth first approach. This walk is subsequently repeated

Symbol	Description
$G(V, E)$	Graph object with V nodes and E vertices. This represents the graph where the dimensional reduction into the embedding space should occur.
e	Number of epochs/times the optimisation of the embedding space is performed.
w	Window size for the SkipGram algorithm. Represents the mixture of centralities in the training's process.
d	Dimension of the embedding space.
t	Walk length of the random walks.
γ	Biased random walk per node in each random walk step.
η	Learning rate with respect to the SkipGram algorithm
$\mathcal{A}$	Percentage of the total nodes, where a random walk is performed.

Table 1: Summary of all parameters used in the algorithm.

γ times. Finally, the algorithm returns an ordered list of the nodes that appeared in all these γ walks combined. The nodes are ordered by the frequency they appeared. Finally, to illustrate how exactly this algorithm works, a step by step walk through is presented:

1. Do a biased random walk from input node v_i: $w_1 = [v_{a_1}, v_{a_2}, ..., v_{a_t}]$;
2. repeat this γ times: $w_\gamma = [v_{x_1}, v_{x_2}, ..., v_{x_t}]$;
3. order the nodes from all γ walks by how frequently they appeared in total and return the result: $\mathcal{W}_{v_i} = [v_{\text{most\_frequent}}, ..., v_{\text{least\_frequent}}]$

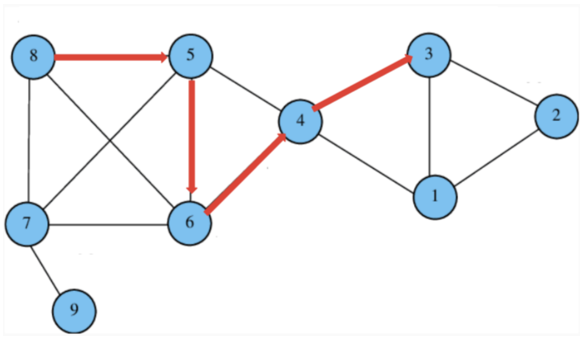


Figure 1: Example of a biased random walk on a graph with walk length of $t = 4$ and output $[5, 6, 4, 3]$.

2.2 SkipGram

Originally, the SkipGram algorithm was developed as a language modelling tool, that maximises cooccurrences among the words that appear within a window, w, in a sentence. It can be seen that Algorithm 2 iterates over all possible collocations in random walk that appear within the window w. In this modified approach, this algorithm has been adapted to handle two independent random walks as described in section 2.1. Instead of maximising the likelihood function of cooccurrence of nodes in a window on one random walk, the target node of the optimisation are the nodes in the secondary random walk. In essence, nodes with similar frequencies within the two random walks will be trained to maximise the likelihood function of cooccurrence, moving them closer together in the embedding space. Finally, within the training's process, the Softmax function is utilised.

Algorithm 2: SkipGram($\Phi, \mathcal{W}_{v_i}, w$)

```
1 for all $v_j \in \mathcal{W}_{w_i}$ do
2     for all $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ do
3         $J(\Phi) = -\log(Pr(u_k|\phi(v_j))$
4         $\Phi = \Phi - \alpha \frac{\partial J}{\partial \Phi}$
5     end for
6 end for
```

2.3 Betweenness centrality estimation with neural networks.

Lastly, after the training's process is completed, to test if the embedding has been successful, the betweenness centrality is estimated using a fully connected neural network (see appendix B). This takes as input the embedding space and as output a estimation of the betweenness centrality. It is important to note that for the training's process of this neural network, the betweenness centrality has to be calculated at least for the training's dataset. Furthermore, the mean absolute error of the whole test dataset is calculated. However, as the algorithm tries to identify the most important nodes in the network, the mean absolute error for the 3 nodes with the highest betweenness centrality is calculated as-well. The fully connected neural network is a simple network with one hidden layer with 64 hidden nodes.

3 Experimental results

The experimental results are summarised in 3 separate sections. First of all, section 3.1 describes how the algorithm performs on a Barabási–Albert model. This is followed by applications to different real world graphs in section 3.2. Finally the algorithm has also been applied to Watts Strogatz random graphs in section 3.3.

3.1 Barabási–Albert model

The Barabási–Albert model is an algorithm for generating random graphs. The main characteristic of these graphs is that these networks are scale-free. In the generating process, a preferential attachment mechanism is implemented, so that new edges attach to nodes with high degrees more frequently. This has the advantage that it mimics some real world graphs such as the Internet, citation networks and most social networks. These types of networks contain few nodes with unusually high degree and betweenness centrality, which are called hubs. Figure 2 shows an example of the embedding space that resulted from an embedding of a Barabási–Albert model with 300 nodes and 3 connections per node into a 2 dimensional space. Figure 2b shows the loss per epoch, which decreases over time. Intuitively, it can already be seen that this has been very successful. Nodes with high betweenness centrality are isolated in to bottom right corner, while unimportant nodes are clustered together on the upper left side. Moreover, with higher centrality, nodes are more isolated from the majority of unimportant nodes. In other words, hubs are clearly distinguishable from all other nodes, using this embedding method. In fact, it turned out that the best accuracy for this model was reached with an embedding into 2 dimensions most of the time. Table 2 - 4 describe results on this model with different parameters.

Number of nodes in graph	Number of connections	Absolute error test set	Absolute error top 3 nodes in test set	Absolute error top node in test set
300	1	0.007	0.019	0.068
300	2	0.007	0.022	0.035
300	3	0.014	0.003	0.005
300	4	0.006	0.011	0.008
300	5	0.008	0.003	0.004
300	6	0.007	0.016	0.011
300	7	0.009	0.007	0.001

Table 2: The model tested on a Barabási–Albert model with 300 nodes and different number of connections. The hyperparameters for the embedding are: $e = 10$, $\gamma = 5$, $t = 14$, $w = 1$, $d = 2$, $\eta = 0.01$ and $\mathcal{A} = 1$

Number of nodes in graph	Number of connections	Absolute error test set	Absolute error top 3 nodes in test set	Absolute error top node in test set
30	4	0.065	0.028	0.023
60	4	0.018	0.053	0.035
90	4	0.005	0.014	0.024
120	4	0.006	0.005	0.006
150	4	0.004	0.002	0.003
300	4	0.006	0.011	0.008
1000	4	0.005	0.033	0.034
3000	4	0.004	0.021	0.006

Table 3: The model tested on a Barabási–Albert model with 4 connections per node and differing number of nodes. The hyperparameters for the embedding are: $e = 3 - 20$, $\gamma = 5$, $t = 14$, $w = 1$, $d = 2$, $\eta = 0.01$ and $\mathcal{A} = 1$

$\mathcal{A}$	Time for the training's process in s	Absolute error test set	Absolute error top 3 nodes in test set	Absolute error top node in test set
1	114	0.001	0.005	0.004
0.8	91	0.005	0.005	0.008
0.6	68	0.005	0.019	0.027
0.4	45	0.006	0.015	0.028
0.2	22	0.007	0.013	0.018

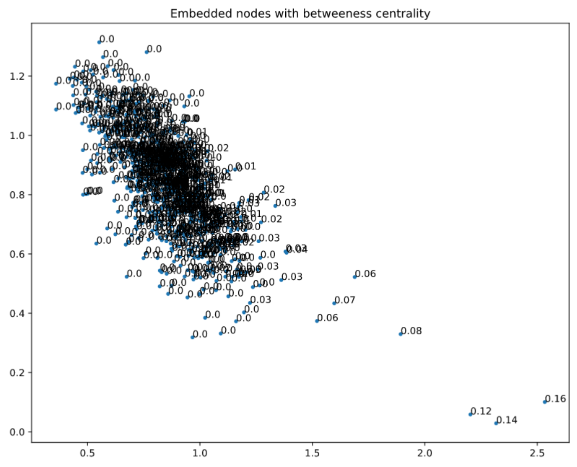
Table 4: The model tested on a Barabási–Albert model with 3 number of connections per node and 1000 nodes in total. The hyperparameters for the embedding are: $e = 5$, $\gamma = 5$, $t = 14$, $w = 1$, $d = 2$ and $\eta = 0.01$, while the approximating factor $\mathcal{A} = 1$ is being changed.

3.2 Real world graphs

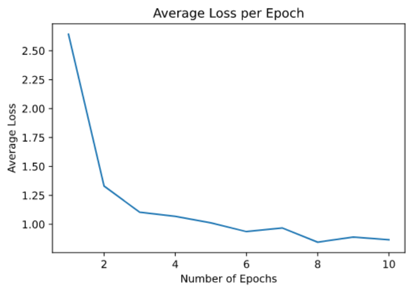
Now that the model hast been tested on a scale free synthetic network, this section will show the results of the application to real world networks.

3.2.1 Collaboration network of Arxiv General Relativity category

The first network the algorithm has been tested on is the Arxiv GR-QC (General Relativity and Quantum Cosmology) collaboration network. This dataset has been sampled from e-print arXiv and within its scope, scientific collaborations between authors papers submitted to General Relativity and Quantum Cosmology category are included. Authors who co-authur a paper have an undirected edge between each other. If the paper is co-authored by k authors this generates a completely connected (sub)graph on k nodes. Moreover, the dataset itself covers papers from January 1993 to April 2003,



(a) Embedding space after training. The blue dots represent the nodes, while the numbers represent the exact betweenness centrality.



(b) Loss of the likelihood function from the training's process

Figure 2: Result of the embedding of a Barabási–Albert graph with 300 nodes in total and 3 connections added per node into a 2 dimensional space.

which means that this is a very comprehensive list with 5242 nodes and 14496 edges. Results are presented here with the absolute loss:

- 0.004 for the whole test set;
- 0.020 for the top 3 central nodes in the test set;
- 0.021 for the top 1 central node in the test set.

With these result, hubs are clustered separately.

3.2.2 Social circles: Facebook

The algorithm has also been tested on a dataset provided by Facebook. It consists of 'circles' (or 'friends lists') from Facebook. The data was gathered from individual survey participants using the Facebook app. This network includes 4039 and 88234 edges. Results are presented here with the absolute loss:

- 0.007 for the whole test set;
- 0.063 for the top 3 central nodes in the test set;
- 0.092 for the top 1 central node in the test set.

Again, with these result, hubs are clustered separately, this can also be when looking at figure 3. Here nodes with a high centrality are very clearly separated from the rest of the nodes.

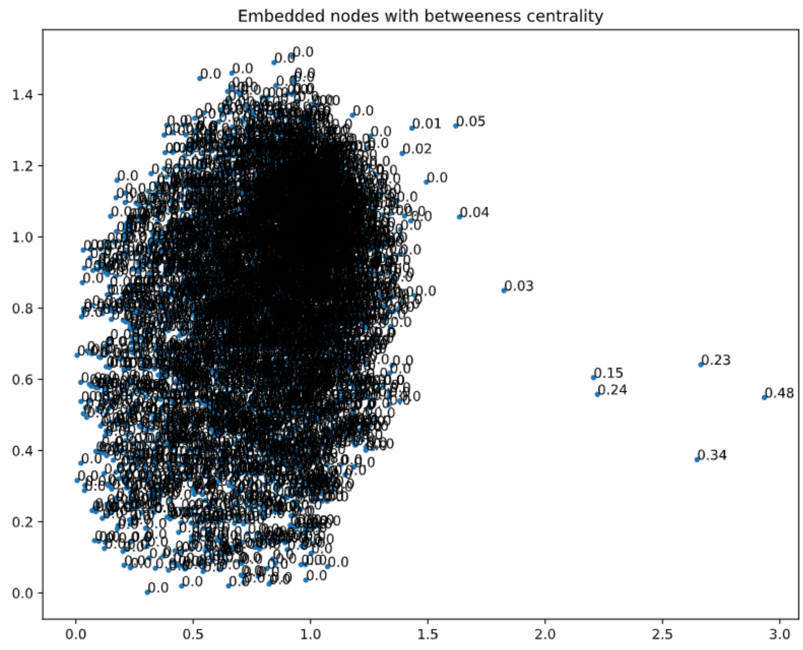


Figure 3: Embedding space after training of the Facebook dataset. Blue dots represent the embedded nodes, while the numbers represent the exact betweenness centrality.

3.3 Watts Strogatz graph

Finally, the algorithm was tested on a synthetic network, that is not scale free. These types of networks do not mimic real world graphs. A type of synthetic graph that fits these criteria are graphs created with the Watts Strogatz model. These graphs take two parameters: the total number direct neighbours in a circle and a probability that a edge is randomly reassign to a different end-node. Results are summarised in table 5.

4 Discussion

To start off with, the experimental results of the Barabási–Albert model have been summarised in tables $2-4$. Table 2 shows the accuracy of the algorithm on a graph with the same amount of nodes, but where the amount of edges per nodes is varied. Two things immediately stand out. Firstly, the absolute error on the whole test set is constant. The reason for this is that the vast majority of nodes in a scale free network have an extremely small betweenness centrality. This means that if the fully connected neural network predicted a centrality of 0 for every node in the network, the accuracy would still be very high. This is different for the errors for the most central nodes in the network. Accuracy here is only low if the embedding has been able to isolate those nodes. Secondly, the higher the number of connections per node are, the more accurate the model gets in accurately identifying

Number of nodes in graph	Probability of rewiring	Absolute error test set	Absolute error top 3 nodes in test set	Absolute error top node in test set
300	0.001	0.010	0.016	0.000
300	0.005	0.033	0.139	0.192
300	0.01	0.025	0.115	0.127
300	0.05	0.017	0.026	0.038
300	0.1	0.014	0.029	0.045
300	0.3	0.013	0.017	0.024

Table 5: The model tested on an Watts Strogatz model with 300 nodes and 6 start edges to its direct neighbours. However, probabilities are varied. The hyperparameters for the embedding are: $e = 10$, $\gamma = 5$, $t = 16$, $w = 1$, $d = 2$, $\eta = 0.01$ and $\mathcal{A} = 1$

nodes with a high centrality. This is both expected and important. The higher the average degree is, the more hubs occur in the network, which again makes it easier to cluster these together (easier to find cooccurrences). This is important because the time complexity of calculating the betweenness centrality exactly is proportional to the number of edges. Approximating the betweenness centrality with this method does not have this property and can thus accurately be used to estimate centralities for huge and strongly interconnected graphs. Similarly, when varying the number of nodes instead of the number of connections, the model gets more accurate with complexity of the network. Table 3 shows that the algorithm gets accurate only after 90 nodes are present in the network. Finally reducing calculation time can be done in two ways. On way is to reduce the amount of training's epochs e, and the other way is to reduce the percentage of nodes that is iterated through in each epoch $\mathcal{A}$. The results of this can be seen in table 4. It can be seen that by decrease $\mathcal{A}$ accuracy and time decrease too. For a complex enough network, choosing $\mathcal{A} = 0.8$ still produces accurate results while greatly reducing calculation times.

Applications to real world networks in section 3.2 have also been successful. Accuracy on both tested networks is not ideal, however hubs were still accurately identified. Rather than accurately approximating the betweenness centrality, in these real world networks, the algorithm was only able to identify if a specific node is a hub or not.

Finally, from table 5 it can clearly be seen that a limitation of this algorithm is applications to non scale free networks. For the case of no rewired edges, the centrality of each node is the same, which is the reason for the high accuracy. However, once edges are rewired, accuracy drops, as in these types of graphs, no hubs from.

All in all, it can be said that generally the algorithm is successful for huge and complex real world (scale free) graphs, while applications to small, simple and non scale free graphs is limited.

References

[1] Shivaram Narayanan. “The betweenness centrality of biological networks”. PhD thesis. Virginia Tech, 2005.

[2] K-I Goh et al. “Betweenness centrality correlation in social networks”. In: *Physical Review E* 67.1 (2003), p. 017101.

[3] Aisan Kazerani and Stephan Winter. “Can betweenness centrality explain traffic flow”. In: *12th AGILE international conference on geographic information science*. 2009, pp. 1–9.

[4] Ulrik Brandes. “A faster algorithm for betweenness centrality”. In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[5] Francesco Buccafurri et al. “Measuring betweenness centrality in social internetworking scenarios”. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2013, pp. 666–673.

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

A Betweenness Approximate python implementation

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import collections

class b_approx:
    def __init__(self, G, epochs, gamma, t, w, dim, eta, centralities = None, approx = 1):
        self.G = G
        self.approx = approx
        self.epochs = epochs
        self.gamma = gamma
        self.nodelist = np.array(list(G))
        self.t = t
        self.w = w
        self.dim = dim

        #Initialising embedding matrix
        self.phi = np.random.rand(len(self.nodelist),self.dim)
        self.psi = np.random.rand(self.dim,len(self.nodelist))
        self.avg_loss = [] #average loss for each epoch
        self.loss = [] #average loss for each epoch
        self.x = np.identity(len(self.nodelist)) #One hot encoded input vectors
        self.eta = eta
        self.centralities = centralities

    def embedding_training(self):
        for a in range(self.epochs): #Goes through all epochs
            self.loss = [] #loss for each timestep
            np.random.shuffle(self.nodelist)
            acutal_nodes = self.nodelist[:int(len(self.nodelist)*self.approx)]
            for v in acutal_nodes: #Goes through every node
                self._randomwalk(v)
                self._skipgram()
            #Calculate the mean of the loss of all random walks that happen in this epoch
            self.avg_loss.append(np.mean(self.loss))
            print(a + 1)

        range_list = np.array(range(self.epochs))
        plt.plot(range_list + 1, self.avg_loss)
        plt.xlabel('Number of Epochs')
        plt.ylabel('Average Loss')
        plt.title('Average Loss per Epoch')
        plt.show()
        #This computes the embedded vectors
        self.embedding = np.dot(self.x, self.phi)

    #This function return the random walks as described, for and inputnode v
    def _walker(self, v):
        walk = []
        v_start = v
        for _ in range(self.gamma):
            v = v_start
            neighbours = np.array(list(self.G.neighbors(v)))
            for _step in range(self.t-1):
                v_prev = v
                node_degree = {}
                for node in neighbours:
                    node_degree[node] = len(list(self.G.neighbors(node)))
                node_degree = {k: v / total for total in (sum(node_degree.values()),)
                    for k, v in node_degree.items()}
                v = np.random.choice(list(node_degree.keys()),
                    p = list(node_degree.values()))
                neighbours = np.array(list(self.G.neighbors(v)))
                neighbours = np.setdiff1d(neighbours, v_prev)
                if len(neighbours) == 0:
                    v = v_start
                    neighbours = np.array(list(self.G.neighbors(v)))
                walk.append(v)
        return collections.Counter(walk)


    #Do the random walk and safe all visited nodes in the list walk
    def _randomwalk(self, v):
        self.main_walk = list(dict(sorted(dict(self._walker(v)).items(),key
            = lambda x:x[1])).keys())
        self.rev_walk = list(dict(sorted(dict(self._walker(np.random.choice(self.nodelist)))
            .items(), key = lambda x:x[1])).keys())



    #Skipgram as described in the lecture
    def _skipgram(self):
        N = max(len(self.main_walk), len(self.rev_walk))
        for u in range(N):
            self.window_elements = self.main_walk[min(max(0,u-self.w-1),
                len(self.main_walk) - 1):min(len(self.main_walk),u+self.w)] #Computes the nodes fo
            target = self.rev_walk[min(u, len(self.rev_walk)-1)]
            self._forward(target)
            self._loss()
            self._backward(target)

        #Forward propagation as described in the lecture
    def _forward(self, i):
        self.h = np.dot(self.x[i].T, self.phi)
        self.u = np.dot(self.psi.T, self.h)
        self.y = self._softmax(self.u)


    # calculate the error and EI for updating
    def _loss(self):
        self.EI = np.sum([np.subtract(self.y, j) for j in self.x[self.window_elements]],axis=0)
        self.loss.append((-np.sum([self.u[j] for j in self.window_elements])
            + np.log(np.sum(np.exp(self.u)))))


    #Backward propagation with weight update
    def _backward(self,i):
        self.grad2 = np.outer(self.h,self.EI)
        self.psi -= self.eta * self.grad2
        self.grad1 = np.outer(self.x[i].T, np.dot(self.psi,self.EI.T))
        self.phi -= self.eta * self.grad1

    #Softmax function
    def _softmax(self,x):
        return np.exp(x) / np.sum(np.exp(x))


    def plot(self):
        if self.dim == 2:
            if self.centralities ==None:
                plt.plot(self.embedding[:,0],self.embedding[:,1],'.')
                for i in range(len(self.nodelist)):
                    plt.text(self.embedding[i,0],self.embedding[i,1],str(i))
                    plt.title('Embedded nodes')
                plt.show()
            else:
                plt.plot(self.embedding[:,0],self.embedding[:,1],'.')
                for i in range(len(self.nodelist)):
                    plt.text(self.embedding[i,0],self.embedding[i,1],
                        str(np.round(self.centralities[i],2)))
                    plt.title('Embedded nodes with betweeness centrality')
                plt.show()
        else:
            print("Dimensional error, embeeding dimension of "
                + str(self.dim) + " is not equal to 2")
```

B Machine learning classificaiton model

```
import seaborn as sns
import tensorflow as tf

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import matplotlib.image as mpimg
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing

class ml_model:
    def __init__(self, centralities, model, epochs, eta):
        self.model = model
        self.epochs = epochs
        self.eta = eta

        #Data preprocessing
        df = pd.DataFrame(self.model.embedding)
        df["centralities"] = list(centralities.values())
        df_train = df.sample(frac=0.7, random_state=0)
        df_test = df.drop(df_train.index)

        self.features = df_train.copy()
        self.labels = self.features.pop("centralities")
        self.normalizer = preprocessing.Normalization()
        self.normalizer.adapt(np.array(self.features))

        #Highest values to tests
        df_train = df_test.sort_values('centralities')
        features_test = df_train.copy()
        self.features_test = features_test
        self.labels_test = self.features_test.pop("centralities")

        features_highest = df_train.copy()
        self.features_highest = features_highest.tail(3)
        self.labels_highest = self.features_highest.pop("centralities")


    def evaualte_highest(self):
        score1 = self.dnn_model.evaluate(self.features_test, self.labels_test, verbose=0)
        print("The mse for the test set is: " + str(score1))

        score2 = self.dnn_model.evaluate(self.features_highest, self.labels_highest, verbose=0)
        print("The mse for the highest 3 nodes is: " + str(score2))


    def _build_and_compile_model(self, norm):
        model = keras.Sequential([
            norm,
            layers.Dense(64, activation='relu'),
            layers.Dense(1)
        ])

        model.compile(loss='mean_absolute_error',
                        optimizer=tf.keras.optimizers.Adam(self.eta))
        return model

    def _plot_loss(self, history):
        plt.plot(history.history['loss'], label='loss')
        plt.plot(history.history['val_loss'], label='val_loss')
        plt.xlabel('Epoch')
        plt.ylabel('Error')
        plt.legend()
        plt.grid(True)
        print("The validation loss is: " + str(history.history['val_loss'][-1]))

    def test(self):
        self.dnn_model = self._build_and_compile_model(self.normalizer)
        history = self.dnn_model.fit(
            self.features, self.labels,
            validation_split=0.2,
            verbose=0, epochs=self.epochs)
        self._plot_loss(history)
```