VE444 Project Report

Group 18

Dec. 2020

Ge Jintian	517021911142
Ye Roushuang	516370910241
Xia Binyu	517370910012

1 Introduction

Nowadays, relationships between people and people become more and more important. Sometimes, if there are two groups, we may be interested in whether these two groups can be connected in some ways. For example, assume that there are two classes in JI, class No.1 and class No.2. Then, all students in class 1 will form a group, while all students in class 2 will form another group. Usually, some students in class 1 should have already known some students in class 2. Now, if the class assistant wants to introduce students in class 1 to students in class 2, how can he increase the probability that two students who are introduced to each other will become friends? If we can find out some potential friendships between students in class 1 and students in class 2, then it will be easier to make sure that two students who are introduced to each other will become friends. Such a technique can be applied on many other fields. For instance, in online communication platform, it can be used to recommend possible friends to someone.

This problem can be solved by using knowledge from network. From the perspective of network, we use vertices to represent individuals and use edges to indicate whether two individuals have a certain relationship. This relationship depends on the type of network. For instance, vertices in social network stand for people and edges will indicate that two connected people are friends, as shown in Figure 1. Then, this problem can be abstracted into a general problem in network: assume there are two networks which are of the same type, but only some individuals in one network are connected to some individuals in another network. Then, according to these edges, we want to find out all possible edges between these two networks. We call this as "merging two networks together according to existing edges".

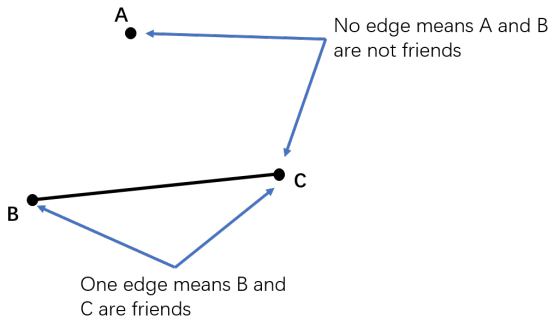

Figure 1: Social network

The problem is described in detail in Figure 2. There are two networks of the same type, A and B. We can see that there are only a few edges between A and B. The problem is that how can we merge A and B together. During this process, if vertex a_1 and b_1 have potential relationship (indicated by a virtual line), we will capture it and link them together. If two vertices don't have a potential relationship, like node a_2 and node b_2, we will try to avoid linking them together. Our project aims at building an algorithm to solve such a problem. We will use node embedding method to provide information for machine learning, and use machine learning to solve this problem.

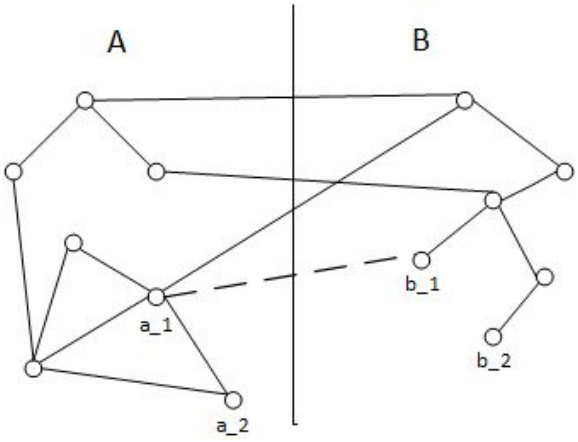

Figure 2: Network description

We plan to evaluate our algorithm using a dataset from Facebook. This dataset will be described clearly later. The dataset contains a whole network. We plan to divide this network into two separated networks. We will break all edges between these two networks and remember them. Then we will try to merge these two networks together and see if we restore original edges successfully. The evaluation method will be described in detail in Algorithm Proposal part.

2 Problem Definition

2.1 Two Isolated Networks

Our project aim to merge two graphs or networks together or predict the new edge after two networks are introduced to one another.

The problem comes from a very common situation: fellowship of two group. For example, in Figure. 3 shows, Group 1 is a group of girls and group 2 is a group of boys. The edge within the group shows a friendship between two people. The edge between groups shows whether two nodes know each other and the positive sign means they are also friends while negative sign shows they are not.

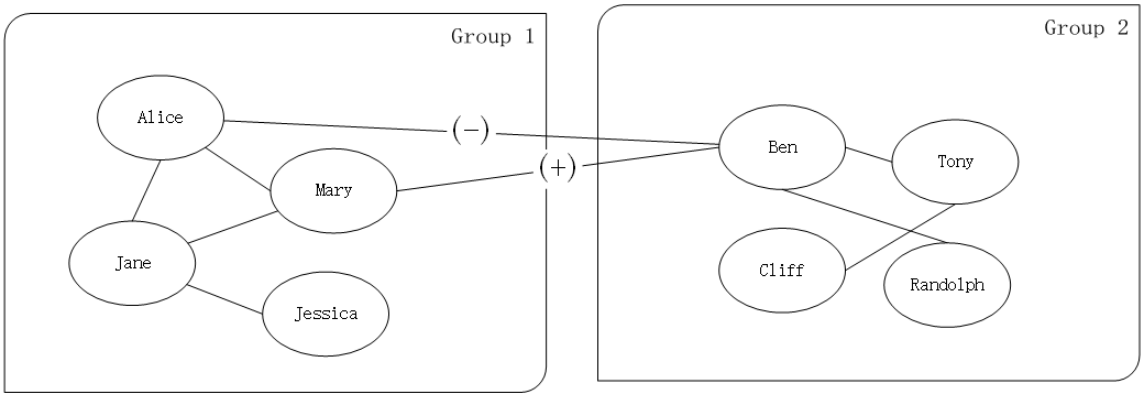

Figure 3: An Example of Fellowship

Usually, fellowship starts from a very few relationship between some persons of each group. In this example, Alice and Mary originally know Ben; what's more, Mary and Ben are good friends. They organize the event of fellowship, hoping introduce their friends to other people. A natural question will be raised that which pairs have a higher probability to form friendship.

Therefore, the new friendship to predict is the main concern of our project and two isolated networks are the basic prerequisite of our project.

2.2 Known Edges Between Groups

A common question will be whether two completely isolated networks is applicable for our algorithms. We consider about the question and notice that if two networks are totally isolated, namely not connected, the node embedding algorithms may not work properly since the features of two networks are trained independently so the dimension of each graph can hardly fit in the another one. Therefore, we require two edges have a minimum known edges between them to help us normalize the embedded features.

3 Related works

3.1 Efficient Estimation of Word Representations in Vector Space. (Mikolov, 2013)

This paper introduces the famous "Word2Vec" Skip-gram model originally designed for word embedding by implicitly factorizing a matrix of shifted pointwise mutual information of word co-occurrence statistics. This model is proved to have a high quality with good accuracy and low computational cost with very large data sets. Since the model inspires several algorithms focus on neighborhood-preserving node embedding, it can help us have a better understanding of the node embedding and make it easier for us to furtherly learn other related algorithms and models.

3.2 Node2vec: Scalable Feature Learning for Networks. (Grover, 2016)

This paper introduces "Node2Vec" – a more familiar algorithmic framework for us for learning continuous feature representations of nodes in networks. Node2Vec gives a mapping of nodes to a low-dimensional space of features maximizing the likelihood of preserving network neighborhoods of nodes. We can learn a flexible notion of nodes' network neighborhood and a biased random walk procedure, which helps efficiently explores diverse neighborhoods. After reading the paper, we can better capture the diversity of connectivity patterns in the networks and have a deeper understanding of nodes representations, which will help us advance in node classification, link prediction and other possible problems facing us in the project.

3.3 Line: Large-scale Information Network Embedding. (Tang, 2015)

This paper provides another network embedding method "Line" to solve the problem of embedding very large real world information networks into low-dimensional vector spaces, which is useful in node classification, link prediction and network visualization. And the method is suitable for arbitrary types of networks, including undirected, directed, weighted an so on, which is proved to be effective on various information networks such as social networks, language networks and citation networks. The paper gives us a reference to the solutions of our project and it can also be used as a comparison to experiment our own algorithm.

3.4 Attributed Social Network Embedding. (Liao, 2018)

Embedding network data into a low-dimensional vector space performs well for node classification and entity retrieval by leveraging network structure. But for our proposal, such model and its related methods fail to sufficiently help us address the problems in our project since we obviously need more detailed information besides only network structure to find out the certain relationships among different individuals. This paper, in the focus of social networks, leverages not only the network structure but also the rich information about social actors (nodes) such as the user profiles of friendship networks – that is focusing on the attribute information of social networks to improve network embedding. In real world networks as well as the networks in our project, nodes often have attributes, and nodes with similar attributes are often more likely to be connected. Learning attributed social network embedding from this paper can enlighten us on the solutions of our project.

3.5 Multi-Scale Attributed Node Embedding. (Rozemberczki, 2019)

This paper presents many network embedding algorithms while delivering their own attributed algorithms, both pooled and multi-scale. It gives a useful method for a diverse range of applications, including latent feature identification across disconnected networks with similar attributes by capturing attribute-neighborhood relationships over multiple scales. The method both follows an approach similar to skip-gram while leveraging the attributes of nodes and the algorithms are proved to be outperforming comparable models on social networks. So the methods provided by this paper are worthy of study and reference for us to address our project problems..

4 Data Sets

4.1 Social circles: Facebook

4.1.1 Description

This dataset consists of 'circles' (or 'friends lists') from Facebook. Facebook data was collected from survey participants using this Facebook app. The dataset includes node features (profiles), circles, and ego networks.

Facebook data has been anonymized by replacing the Facebook-internal ids for each user with a new value. Also, while feature vectors from this dataset have been provided, the interpretation of those features has been obscured. For instance, where the original dataset may have contained a feature "political=Democratic Party", the new data would simply contain "political=anonymized feature 1". Thus, using the anonymized data it is possible to determine whether two users have the same political affiliations, but not what their individual political affiliations represent.

Data is also available from Google+ and Twitter.

4.1.2 Feature

As Table 1 shows, the network is not directed and the nodes are assigned with certain features from the their description. From our perspective, this dataset is good for training for multi-class node classification, link prediction, community detection and network visualization.

Nodes:	4039
Edges:	88234
Nodes in largest WCC:	4039
Edges in largest WCC:	88234
Nodes in largest SCC:	4039
Edges in largest SCC:	88234
Average Clustering Coefficient:	0.6055
Number of Triangles:	1612010
Diameter:	8
Average Degree:	43.7

Table 1: Feature of Facebook Large Page-Page Network

There are 223 features in total, these features cover different aspects of users, like the birthday (a vague period of time), education, languages, work, etc. Some of the nodes are shown below:

```
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000010000000000000000000000000000000
0000000000000000000000000000000001000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000
2000000000000000000000000000000000000000010000000000000000000000
0000001010100000000000000000000010000000000000100000000000000000
0001000000000000000001000000000001000000001000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000
3000000010000001000000000000000000000010000000000000000000000000
```

Figure 4: Caption

Following is a part of the features table:

```
6 birthday;anonymized feature 6
7 birthday;anonymized feature 7
13 education;concentration;id;anonymized feature 13
14 education;concentration;id;anonymized feature 14
15 education;concentration;id;anonymized feature 15
176 work;location;id;anonymized feature 129
177 work;location;id;anonymized feature 173
178 work;location;id;anonymized feature 174
```

6 and 7 are a part of birthday features, it shows whether a person is born in a certain periods. This help our program to learn about their ages.

13 to 15 shows the region of education a person takes. This features can indicate the major or other similar features for a node that can be a very import key for learning their relations.

176 to 178 shows the work locations for each nodes, these features take the geography reason into consideration and help us know better about these nodes.

All other features are similar binary features about different aspect of nodes. Though these features have been anonymized, they can still help us build up the model and algorithms we need.

5 Proposal Algorithms

5.1 Model Construction

Given two isolated groups (the connection between the nodes from each group is unknown), we want to predict whether a edge exist given two nodes from one of each groups. At first, we will use node embedding algorithm to estimate vertices features in two networks. Then, we use edges between these two networks to train our prediction machine learning algorithm, which is a neural network. Finally, we will use the trained machine learning algorithm to predict whether there is a potential edge between two vertex in these two networks. If so, we will link them together. By repeating this process, finally we can merge two networks together.

Algorithm 1: Merging Algorithm

Input: $Net_1(V_1, E_1)$,$Net_2(V_2, E_2)$:Network to be merged
$C_i(v_1, v_2), v_1 \in V_1' \subset V_1, v_2 \in V_2' \subset V_2$: The known connectivity between some nodes from Net_1, Net_2.

Output: $C_o(v_1, v_2), v_1 \in V_1 - V_1', v_2 \in V_2 - V_2'$: The predicted connectivity between the rest nodes from Net_1, Net_2.

$F(Net_1)$ =**Embedding** Net_1;//Feature of Net_1
$F(Net_2)$ =**Embedding** Net_2;//Feature of Net_2
M =**Machine Learning** $F(Net_1), F(Net_2), C_i$ //The trained model
for $v_i \in V_1 - V_1'$ **do**
  for $v_j \in V_2 - V_2'$ **do**
    $C_0(v_i, v_j)$=**Predict Connectivity**$F(Net_1, v_1), F(Net_2, v_2), M$
  endfor
endfor

This pseudo code explains our model. The challenges here are node embedding algorithm and machine learning algorithm. When these two parts are finished, the remaining is easy to accomplish. In the rest part of this section, we will introduce the node embedding algorithm we design and the machine learning algorithm we implement.

5.2 Node Embedding algorithm

We design our node embedding algorithm according to the homophily of group. For each node in the network, we set its attributes as $z = [z_1, z_2, z_3, ..., z_n]$. Similar to the course slide, the similarity between two nodes as is dot product, which is $similarity = z_i^T \cdot z_j$. The homophily of group means that nodes connected together have some attributes in common, so our goal is to capture these common attributes based on the connections. To realize this, we define the similarity to be the indicator of connection. Concretely, if two nodes are connected, their similarity should be 1. Otherwise, their similarity should be 0. Our node embedding algorithm will train the node attributes so that their products will converge to the true similarity.

To realize this, we define our loss function as the square of distance between the estimated value and the true value. We denote $y_{i,j}$ as whether there is an edge between node i and node j. Then our loss function J is:

$$J = \sum_{v,u \in V} (z_i^T z_j - y_{i,j})^2$$

This loss function represents how our estimation value differs from the true value. Therefore, our goal becomes to minimize this loss function. To achieve this goal, we apply gradient descent to each attribute. At first we need to differentiate the loss function. Let's denote the kth attribute of node i as $z_i^{(k)}$, namely $z_i = [z_i^{(1)}, z_i^{(2)}, z_i^{(3)}, ..., z_i^{(n)}]$. The gradient for kth attribute when we meet node i and node j is:

$$\frac{\partial J}{\partial z_i^{(k)}} = (z_i^T z_j - y_{i,j}) z_j^{(k)}$$

When the gradient shows if the loss function is moving up or down in a certain scenery. Since the loss function we derived above has only one extreme, which is the global minimum, we can find this point so that the whole loss function will be minimized. To achieve this, we apply gradient descent algorithm. We use the the gradient to update the parameters. Namely, we update it as

$z_i^{(k)} = z_i^{(k)} - \frac{\partial J}{\partial z_i^{(k)}}$. In this case, if the gradient is larger than 0, we will reduce $z_i^{(k)}$ and if the gradient is less than 0, we will increase $z_i^{(k)}$. This idea is shown in figure 5. In the figure, we can understand that we should increase z if $\frac{\partial J}{\partial z} < 0$ and decrease z if $\frac{\partial J}{\partial z} > 0$ to find the global minimum point.

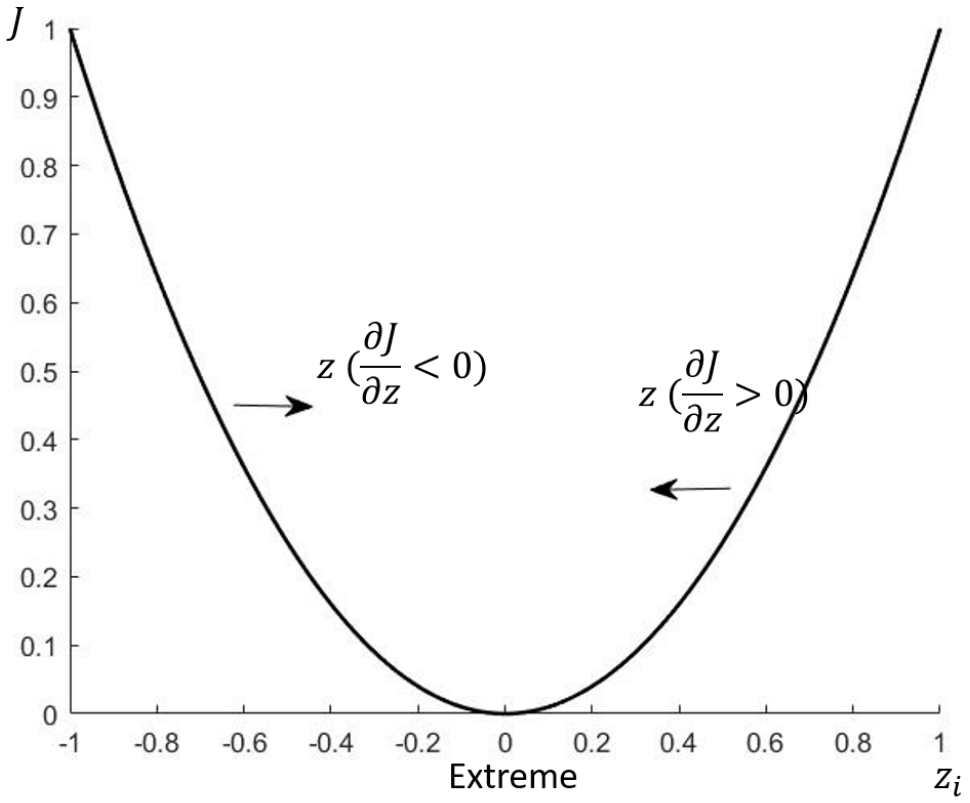

Figure 5: Gradient

Meanwhile, to prevent the gradient explosion problem (which we really meet during implementation), we add a regularization term. Also, we want to control the weight each sample has, so we and some parameters before each terms. Finally, the gradient update equation becomes:

$$\frac{\partial J}{\partial z_i^{(k)}} = \alpha(z_i^T z_j - y_{i,j}) z_j^{(k)} + \lambda z_i^{(k)}$$

Then, we will go through the whole dataset and apply gradient descent to each node pair (i, j) we meet. Originally we use the cross_entropy loss function instead of this square error loss function. However, during our experiment, we find that this loss function interacts better with our machine learning algorithm. So, we decide to use the square error loss function to train our node attributes. The whole process is described in the following pseudo code:

Algorithm 2: Node Embedding Algorithm

Input: $Net(V, E)$:Network to be merged
n:number of attributes to generated;
α:learning rate;
λ:regularization weight.

Output: $Attri^{|V|\times n}$:The embedded features of each node.

$Attri = random((|V| \times n))$;
for $v_i \in V$ **do**
  for $v_j \in V/\{v_j\}$ **do**
    $y_{i,j} = E.is\_edge(v_i, v_j)$;
    $z_i = Attri[v_i]$;
    $z_j = Attri[v_j]$;
    $gradient_i = \alpha(z_i^T z_j - y_{i,j})z_j + \lambda z_i$;
    $gradient_j = \alpha(z_i^T z_j - y_{i,j})z_i + \lambda z_j$;
    $Attri[v_i] -= gradient_i$;
    $Attri[v_i] -= gradient_i$;
  endfor
endfor
return $Attri$;

5.3 Machine learning algorithm

The next part we should do is to implement a machine learning algorithm to predict potential edges. Since it is a logistic regression problem, we choose to use a classical neural network with binary cross entropy loss as its loss function. We decide to use two dense layers to fully connect input layer and output layer. The concrete parameters should be adjusted according to the dataset.

6 Experiment

In this section, we will verify our algorithm based on the dataset we introduced before. And we will adjust our parameters according to this specific dataset.

6.1 Parameters Settings

When we look at the features provided by the Facebook, we find that most of them are zeros, and only a small number of them are ones. In most cases, the number of ones will not exceed 10. This is a very sparse vector, and this is why we think that our embedding algorithm will work better than just inputting original features. Since the number of ones will not exceed 10, we set the number of attributes produced by node embedding algorithm to be 10. For learning rate and regularization weight, after multiple experiments, we set them to be both 0.3.

For the neural network, for original features provided by Facebook, the input layer should be 448. So, our neural network has 125, 25, 1 as the neurons in each layers. For the attributes produced by node embedding algorithm, the input is only 20, so the number of neurons is 25, 1, and we remove one layer to avoid overfitting problem.

6.2 Dataset processing

We split the dataset by half and randomly, which is similar to the class arrangement in school. Then we count the edges between the two groups, there are **1263** edges between these two groups. So, we decide to show **900** of them to our neural network and hide the remaining for training. We also randomly choose **900** node pairs between these two groups which have no edge between them to replenish the training set. For testing set, we add **600** more negative samples. So, totally, our training set has **1800** samples, with 50% of them are positive, while our training set has **963** samples and 40% of them are positive. This means that if the prediction accuracy of our machine learning algorithm exceed 60% in testing set (better than all negative), then we can conclude that our neural network has some improvements in prediction.

6.3 Experiment Result

To show that our machine learning algorithm is valid, we input the original features into the neural network. The result is shown in figure 6. In the figure, the blue line stands for training set, and the orange line stands for validation set. The left part is accuracy and the right part is loss.

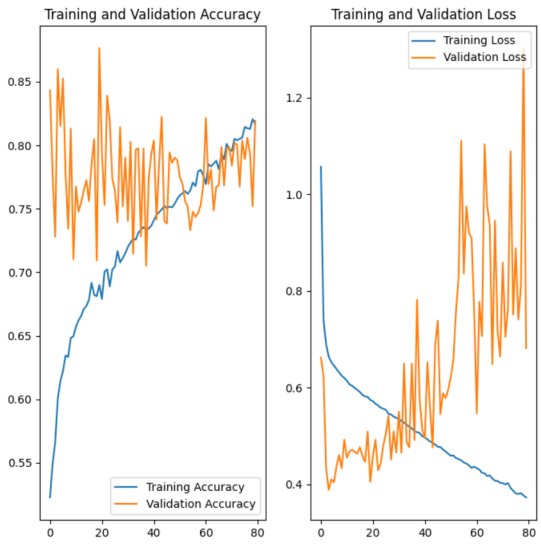

Figure 6: Node prediction based on original features

As we can see from this figure, through the accuracy of training set is improving during training, the accuracy of validation is damping between 0.75 and 0.85. We have run this for 80 epochs, but it still doesn't converge. Therefore, we don't expect to see an improvement in the accuracy of testing set.

To verify our node embedding algorithm, we want to compare it with the original features in our machine learning algorithm. According to our design, if we use the attributes produced by our node embedding algorithm, we should see improvement in accuracy and stability. The result is shown in figure. 7.

Figure 7: Node prediction based on Node Embedding

In this figure, we only train our neural network in 10 epochs and it converges very early. The accuracy is fixed at 0.85. Please see detail analysis in Conclusion section.

7 Conclusion

As we introduce in Experiment section, the accuracy of our original features based prediction damps between 0.75 and 0.85. We think that it is because of the sparse input vector. It is so sparse that our neural network can not find a suitable pattern which meets every situation. Though the accuracy is damping, it is all higher than 0.6, which is the baseline of the testing set. Therefore, we think that our prediction model is successful.

For Node Embedding Algorithm, we expect that it could extract useful information from the network. This means that it should solve the sparse problem, because there are only 10 attributes for each node. The result strongly support our analysis. The algorithm converges so quickly because the input is very simple, and the accuracy of testing set is much stable than the original features based prediction. However, we cann't explain why the accuracy in training set is so low. It should be higher (or at least not lower) than the testing set. Maybe we could consider it as a future work. After all, we think that our project successfully solving the group merging problem we have proposed.

8 Reference

1. Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. In International Conference on Learning Representations, Workshop Track Proceedings, 2013.

2. Liao, L., He, X., Zhang, H. and Chua, T. Attributed social network embedding. IEEE Transactions on Knowledge and Data Engineering, 20(12): 2257-2270, 2018.

3. Grover, A. and Leskovec, J. Node2Vec: Scalable feature learning for networks. In International Conference on Knowledge Discovery and Data Mining, 2016.

4. Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. Line: Large-Scale Information Network Embedding. In International Conference on World Wide Web, 2015.

5. Rozemberczki, B., Allen, C., Sarkar, R. Multi-Scale Attributed Node Embedding. arXiv preprint arXiv:1909.13021, 2019

Appendix

8.1 README

Please read the following part before running the algorithm.

Our algorithm is a little hard-encoding, which means that we need to change parameters to adjust something. For now, to run the original feature based prediction, just run [python prj.py].

If you want to run the node embedding based prediction, please delete ["""] above [CF] (it is where we define a new instance of the node embedding algorithm) and delete ["""] below [print("3")].

8.2 Prj.py

```
#TODO:  1. implement the node embedding algorithm



import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras import layers
import random
import matplotlib.pyplot as plt

class ANN():
    def __init__(self, input_dim):
        self.input_dim = input_dim
        self.model = self.build_network()
    def build_network(self):

        model = Sequential([
        layers.Dense(25, input_dim = self.input_dim, activation='relu'),
        #layers.Dense(25,activation='relu'),
        layers.Dense(1)
        ])

        model.compile(optimizer = 'sgd',
         loss='binary_crossentropy',
```

```
          metrics = ["accuracy"])

        return model

class Collaborative_filtering():
    def __init__(self, number, mat, feat_num):
        self.map = mat
        self.number = number
        self.feat_num = feat_num
        self.user_feat = np.random.rand(self.number, self.feat_num)
        self.generate_feature()

    def generate_feature(self):
        J = self.Cost_function()
        #print(self.user_feat)
        #print('Before descent. loss is:',J)
        for i in range(self.number):
            for j in range(self.number):
                if i == j: continue
                self.Gradient_Descent(i,j)
                #print(self.user_feat)
        #print('After descent. loss is:',J)
        #print(self.user_feat)

    def Cost_function(self):
        J = 0
        for i in range(self.number):
            for j in range(self.number):
                if i == j: continue
                j_m = 0
                for k in range(self.feat_num):
                    j_m = j_m + self.user_feat[i][k] * self.user_feat[j][k]
                j_m = (j_m-self.map[i][j])*(j_m-self.map[i][j])
                J = J + j_m
        return J

    def Gradient(self, u1, u2):
        j=0
        for i in range(self.feat_num):
            j = j + self.user_feat[u1][i] * self.user_feat[u2][i]
        j = j - self.map[u1][u2]
        #print(j)
        return j

    def Gradient_Descent(self, u1, u2):
        gradient1 = np.zeros(self.feat_num)
```

```
        gradient2 = np.zeros(self.feat_num)
        y = self.map[u1][u2]
        for i in range(self.feat_num):
            gradient1[i] = 0.3*self.Gradient(u1,u2)*self.user_feat[u2][i] + 0.5*self.us
            gradient2[i] = 0.3*self.Gradient(u1,u2)*self.user_feat[u1][i] + 0.5*self.us
        for i in range(self.feat_num):
            self.user_feat[u1][i] -= gradient1[i]
            self.user_feat[u2][i] -= gradient2[i]


edges = np.zeros((347,347))
f = open("facebook/0.edges","r")
line = f.readline()
line = line[:-1]
x = line.split()
edges[int(x[0])-1][int(x[1])-1] = 1
while line:
    line = f.readline()
    line = line[:-1]
    x = line.split()
    if len(x) == 0: break
    edges[int(x[0])-1][int(x[1])-1] = 1
f.close()

features = np.zeros((347,224))
f = open("facebook/0.feat","r")
line = f.readline()
line = line[:-1]
x = line.split()
for i in range(224):
    features[int(x[0])-1][i] = int(x[i+1])
while line:
    line = f.readline()
    line = line[:-1]
    x = line.split()
    if len(x) == 0: break
    for i in range(224):
        features[int(x[0])-1][i] = int(x[i+1])
f.close()

#Begin to split dataset
#0~170, 171~346
store_edges = np.zeros((171,176))
pos_id = np.zeros((1263,2))
count = 0
for i in range(171):
```

```
    for j in range(176):
        k = j + 171
        if edges[i][k] == 1:
            store_edges[i][j] == 1
            pos_id[count][0] = i
            pos_id[count][1] = k
            count = count + 1
print(count)


#Preparing for training dataset
neg_id = np.zeros((5000,2))
idx1 = []
idx2 = []
for i in range(5000):
    m = random.randint(0,170)
    n = random.randint(0,175)
    while (store_edges[m][n] == 1) | m == n | ((m in idx1) & (n in idx2)):
        m = random.randint(0,170)
        n = random.randint(0,175)
    idx1.append(m)
    idx2.append(n)
    neg_id[i][0] = m
    neg_id[i][1] = n + 171


#900 pos and 900 neg for training, 363 pos and 600 neg for validation
#This is for the binary features
training_set = np.zeros((1800,224*2))
train_tag = np.zeros((1800,1))
for i in range(900):
    train_tag[i] = 1
    dx1 = pos_id[i][0]
    dx2 = pos_id[i][1]
    for m in range(224):
        training_set[i][m] = features[int(dx1)][m]
    for m in range(224):
        n = m + 224
        training_set[i][n] = features[int(dx2)][m]
for i in range(900):
    j = i+900
    train_tag[j] = 0
    dx1 = neg_id[i][0]
    dx2 = neg_id[i][1]
    for m in range(224):
        training_set[j][m] = features[int(dx1)][m]
```

```
    for m in range(224):
        n = m + 224
        training_set[j][n] = features[int(dx2)][m]
validation_set = np.zeros((963,224*2))
val_tag = np.zeros((963,1))
for i in range(363):
    j = i+900
    val_tag[i] = 1
    dx1 = pos_id[j][0]
    dx2 = pos_id[j][1]
    for m in range(224):
        validation_set[i][m] = features[int(dx1)][m]
    for m in range(224):
        n = m + 224
        validation_set[i][n] = features[int(dx2)][m]
for i in range(600):
    j = i+900
    k = i + 363
    dx1 = neg_id[j][0]
    dx2 = neg_id[j][1]
    for m in range(224):
        training_set[k][m] = features[int(dx1)][m]
    for m in range(224):
        n = m + 224
        training_set[k][n] = features[int(dx2)][m]

"""
CF = Collaborative_filtering(347, edges, 10)
#This is for CF generated features
training_set = np.zeros((1800,10*2))
train_tag = np.zeros((1800,1))
for i in range(900):
    train_tag[i] = 1
    dx1 = pos_id[i][0]
    dx2 = pos_id[i][1]
    for m in range(10):
        training_set[i][m] = CF.user_feat[int(dx1)][m]
    for m in range(10):
        n = m + 10
        training_set[i][n] = CF.user_feat[int(dx2)][m]
for i in range(900):
    j = i+900
    train_tag[j] = 0
    dx1 = neg_id[i][0]
    dx2 = neg_id[i][1]
    for m in range(10):
```

```
            training_set[j][m] = CF.user_feat[int(dx1)][m]
        for m in range(10):
            n = m + 10
            training_set[j][n] = CF.user_feat[int(dx2)][m]
print("2")
validation_set = np.zeros((963,10*2))
val_tag = np.zeros((963,1))
for i in range(363):
    j = i+900
    val_tag[i] = 1
    dx1 = pos_id[j][0]
    dx2 = pos_id[j][1]
    for m in range(10):
        validation_set[i][m] = features[int(dx1)][m]
    for m in range(10):
        n = m + 10
        validation_set[i][n] = features[int(dx2)][m]
for i in range(600):
    j = i+900
    k = i + 363
    dx1 = neg_id[j][0]
    dx2 = neg_id[j][1]
    for m in range(10):
        training_set[k][m] = features[int(dx1)][m]
    for m in range(10):
        n = m + 10
        training_set[k][n] = features[int(dx2)][m]
print("3")
"""

ML = ANN(224*2)
train_ds = [training_set,train_tag]
val_ds = [validation_set,val_tag]
#AUTOTUNE = tf.data.experimental.AUTOTUNE
#train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
#val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
history = ML.model.fit(
    x = training_set,
    y = train_tag,
    validation_data = (validation_set, val_tag),
    epochs = 80
)
#Plot the training result
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
```

```
loss=history.history['loss']
val_loss=history.history['val_loss']

epochs_range = range(80)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
scores = ML.model.evaluate(validation_set, val_tag, verbose=1)
print('result is :',scores)
```