Assignment 2 for FIT5212, Semester 1, 2020

Student Name: SAI TEJA POTHNAK

Student ID: 30323460

Task 1

Recommender Systems

Collaborative Filtering

This is a technique to filter out the products that customer might prefer based on the behaviour by similar customers.When we talk about collaborative filtering for recommender systems we want to solve the problem of our original matrix having million dimensions.

Instead we can use matrix factorisation to mathematically reduce dimensionality of original matrix into smaller matrix i.e. All users by all items into all items by some taste dimensions and all users into some taste dimensions.

If we could represent user as a vector of their taste value and at the same time represent item as a vector of their taste value, we can quite easily make a recom/mendation.

This can also help us find connections between users who have no specific items in common but share common tastes.

Implicit Data

Implicit Data is the type of feedback data that is generated depending on user's behaviour with no specific actions from the user. For example, how many times a user has visited a site, time spent by the user on particular product, items that are purchased by the user. This kind of data is gathered through user interaction and contains lots of data. The downside is that most of this data is noisy and not always apparent what it means.

In [ ]:

```
#importing libraries for the task
import implicit
from sklearn import metrics
import scipy.sparse as sparse
from scipy.sparse import csr_matrix
import pandas as pd
import numpy as np
import itertools
from itertools import chain
```

- About the Data Sets :

1. `Train Dataset` : It contains set of interactions between user id and item id. i.e. if a user interacts with an item, there will be record as rating in the dataset.
2. `Test Dataset` : This dataset contains list of users where each user is provided with a list with 100 item candidates. This is later used for comparision after generating recommended items from the model and final top 10 items are recommended.
3. `Validation Dataset` : This dataset is similar to Test dataset and is concatenated with train dataset and used for model construction and tuning.

In [ ]:

```
trainData =pd.read_csv('train_data.csv') #reading train dataset
```

In [ ]:

```
validationData=pd.read_csv('validation_data.csv') #reading test dataset
```

In [ ]:

```
testData=pd.read_csv('test_data.csv') #reading the test data csv file
```

In [ ]:

```
trainData.shape #shape of training dataset
```

In [ ]:

```
validationData.shape #shape of validation dataset
```

In [ ]:

```
testData.shape #shape of test dataset
```

In [ ]:

```
trainData.head() #looking at the first 5 rows of train data
```

In [ ]:

```
validationData.head() #looking at first 5 rows of validation data
```

In [ ]:

```
testData.head() #looking at first 5 rows of test data
```

Alternating Least Squares

Alternating Least Squares (ALS) is a recommendation model we will use to fit our data and recommend items to each user.

Methodology : In Als, optimization process is performed for every iteration till the matrix arrives closer to factorized representation of original data.

Initially, random values for users(U) and items(I) are assigned and using least squares method iteratively such that say in first instance it optimises U and fixes V and vice versa until both weights are optimised to reach the best approximation of original matrix. It also contains a regulariser term 'lambda' to reduce overfitting.

This algorithm minimises the loss functona and finds out the optimised user and item vectors. Hence, by performing alternative iteration and optimisation process we generate one matrix with user vectors and one with item vectors that can be later used find similarities and also suggest item recommendations to the user.

Parameters :

1. **Confidence :** It is calculated using magnitude of the feedback. More the user spent time with the product more is the confidence.
2. Confidence is calculated by `c = 1 + alpha * r ( where alpha is rate at which the confidence increases and r is the rating information)`
3. `lambda` is the regularisation term as discussed earlier
4. `factors` is the number of latent factors to compute the model

Bayesian Personalized Ranking

It provides user with ranked list of items. In this process, instead of taking single item as training data, it takes pair of items. The optimization is performed on rank of the user. For example, let's say there are two items i1, i2. user interacted with i1 and not with i2. This approach gives i1 a positive sign anf prefers item i1 over i2. Moreover,bayesian approach maximises the posterior probability for each user. It assumes the user interacted item pair independent of every other item pair.

User specific log likelihood function is given with following assumptions:

1. Users are independent to each other.
2. Item pair association is independent to one another.

Model Development

Three Models are used and compared here :

1. Alternative Least Squared with only Training Data

2. Alternative Least Squares with both Training and Validation Data

3. Bayesian Personalized Ranking method with both Training and Validataion Dataset

Model 1 :Fitting the ALS Model with only Training Data:

In [ ]:

```
data1=trainData #storing training data into a new variable
```

In [ ]:

```
data1.shape #shape of final dataset
```

In [ ]:

```
users1=list(np.sort(data1.user_id.unique())) #creating a list of all unique users from data
items1=list(np.sort(data1.item_id.unique())) #creating a list of all unique items from data
rating1=list(data1.rating) #creating a list of all ratings from dataset
```

In [ ]:

```
# creating rows and columns for our matrix
rows1=data1.user_id.astype(int)
cols1=data1.item_id.astype(int)
```

Constructing a sparse matrix containing items engaged for the user with rows as user and columns as items

In [ ]:

```
#constructing a sparse matrix containing items engaged for the user with rows as user and c
data_sparse_user1=sparse.csr_matrix((rating1,(rows1, cols1)), shape=(len(users1),len(items1
```

Constructing a csr matrix where rows of the matrix are the items and columns of the matrix as items

In [ ]:

```
#constructing a sparse matrix containing items engaged for the user with rows as user an
data_sparse_item1=sparse.csr_matrix((rating1,(cols1,rows1)), shape=(len(items1),len(users1)
```

In [ ]:

```
data_sparse_user1.shape #shape of user_item matrix
```

In [ ]:

```
data_sparse_item1.shape #shape of item_user matrix
```

In the datset when rating =1 means there is an interaction between user and item.

When rating is 0, it can be considered as negative item that means preference (Piu)=0 and Confidence (Ciu=1) is assumed for all the negative items.

Negative items can also passed with high confidence value whch can indicate that user disliked the item

In [ ]:

```
alpha = 35 #The rate in which we'll increase our confidence in a preference with more inter
data_1 = (data_sparse_item1 * alpha).astype('double')
```

In [ ]:

```
model1 = implicit.als.AlternatingLeastSquares(factors=5, regularization=0.15, iterations=3
```

In [ ]:

```
model1.fit(data_1,show_progress=True) #fitting our data to our model.
```

Methodology for recommending top 10 items for each user. Same methodology for all the three models.

`Steps` :

1. All the users from test data are stores `user_id`
2. A nested list `item_list` is created to store all the item corresponding to each user.
3. `.recommend()` this is an inbuilt package in implicit.als to recommend items for each user.

   Its parameters are:

   `userid` : The userid to calculate recommendations for.{stored as `user_id` in this case}

   `user_items` : A sparse matrix of shape(number_users,numer_items).{stored as `data_sparse_user` in our case}

   `N` : Number of results to return {Size of all unique items ids in our case}

   It returns : List of itemid and corresponding score as a tuple.

   For each user we generate the set of item recommendations by giving parameters as above. Fetching only the list of recommended item ids from the tuple for each user.
4. From the nested loop of recommended items for each user, we compare with the item list{ `item_list` } from test data and

   append only the items that meet this condition. Finally, only top 10 from this new list are taken and stored as a nested

   list `top_recommendations` where each inner list contains top 10 recommendations for each user.
5. `chain.fom_iterable` is performed on the nested lists `user_lists` and `top_recommendations` to generate a new single

   list `final users` and `top_10_recommendations` .
6. A dataframe `recommendations_dataframe` is created with above two lists where each row corresponds to a user and a

   recommended item.
7. This dataframe `recommendations_dataframe` is conveted into a csv file and used for final submissions.

In [ ]:

```
user_id1=list((testData.user_id.unique())) #storing all the unique user ids as a list

item_list1=[] #empty list
for user in user_id1: #iterating through user_id list
    item_list1.append(testData[testData['user_id']==user]['item_id'].tolist()) #storing all



total_items_recommend1=[] #empty list
user_lists1=[] #empty list
for user in user_id1: #iterating through each user id
    top1=model1.recommend(user,data_sparse_user1,N=len(testData['item_id'].unique().tolist(
    total_items_recommend1.append([i[0]for i in top1]) #storing only item ids
    user_lists1.append([user]*10) #10 entries of each user is created.

top_recommendations1=[] # empty list
for i in range(len(total_items_recommend1)): #iterating through length of recommendations g
    temp=[] #empty list
    for j in total_items_recommend1[i]: #for all the recommended items for each user
        if j in item_list1[i]: #if that recommended item is present in item list of that us
            temp.append(j) # append that item to new list

    top_recommendations1.append(temp[:10]) # append first 10 items satisfying the condition



final_users1=list(chain.from_iterable(user_lists1)) #creating a flattened iterable of users
top_10_recommendations1=list(chain.from_iterable(top_recommendations1)) #creating a flatten

#creating a dataframe containing final user ids and item ids
recommendations_dataframe1=pd.DataFrame({'user_id':final_users1,'item_id':top_10_recommenda

#writing the dataframe into a csv file
# recommendations_dataframe1.to_csv('sample_solution_data.csv',index=False)
recommendations_dataframe1[1:11]
```

Model 2

2. Fitting the ALS Model with only Training Data and Validation Data

Concatenating training and validation Datasets

It can be noticed from shape of train dataset that its size is smaller compared to test dataset and validation dataset.

It can also be inferred that trainData set merely has only 1 as rating value, hence using only this dataset wouldn't help building a generalised model and results in overfitting as the labels(ratings) of trained data is highly biased. On the other side, validation dataset has combinations of both 0's and 1's. To achieve a generalised model with lesser overfitting, both training and validation dataset are merged.

Proceeding to concatenate both training dataset and validation dataset to achieve a bigger dataset and significantly get a better model.

In [ ]:

```
data2=pd.concat([trainData,validationData],ignore_index=True) #concatenating both train and
```

In [ ]:

```
data2.shape #shape of final dataset
```

In [ ]:

```
#Data Exploration
```

In [ ]:

```
data2.rating.value_counts() #displaying value counts of ratings in dataset
```

Dropping duplicates

In [ ]:

```
data2 = data2.drop_duplicates()
data2=data2.reset_index(drop=True)
data2.shape
```

In [ ]:

```
users2=list(np.sort(data2.user_id.unique())) #creating a list of all unique users from data
items2=list(np.sort(data2.item_id.unique())) #creating a list of all unique items from data
rating2=list(data2.rating) #creating a list of all ratings from dataset
```

In [ ]:

```
# creating rows and columns for our matrix
rows2=data2.user_id.astype(int)
cols2=data2.item_id.astype(int)
```

Constructing a sparse matrix containing items engaged for the user with rows as user and columns as items

In [ ]:

```
#constructing a sparse matrix containing items engaged for the user with rows as user and c
data_sparse_user2=sparse.csr_matrix((rating2,(rows2, cols2)), shape=(len(users2),len(items2
```

Constructing a csr matrix where rows of the matrix are the items and columns of the matrix as items

In [ ]:

```
#constructing a sparse matrix containing items engaged for the user with rows as user an
data_sparse_item2=sparse.csr_matrix((rating2,(cols2,rows2)), shape=(len(items2),len(users2)
```

In [ ]:

```
data_sparse_user2.shape #shape of user_item matrix
```

In [ ]:

```
data_sparse_item2.shape #shape of item_user matrix
```

Training the ALS Model with Combined Data

In [ ]:

```

```

In [ ]:

```
alpha = 35 #The rate in which we'll increase our confidence in a preference with more inter
data_2 = (data_sparse_item2 * alpha).astype('double')

model2 = implicit.als.AlternatingLeastSquares(factors=5, regularization=0.15, iterations=30

model2.fit(data_2,show_progress=True) #fitting our data to our model.
```

In [ ]:

```
user_id2=list((testData.user_id.unique())) #storing all the unique user ids as a list

item_list2=[] #empty list
for user in user_id2: #iterating through user_id list
    item_list2.append(testData[testData['user_id']==user]['item_id'].tolist()) #storing all



total_items_recommend2=[] #empty list
user_lists2=[] #empty list
for user in user_id2: #iterating through each user id
    top2=model2.recommend(user,data_sparse_user2,N=len(testData['item_id'].unique().tolist(
    total_items_recommend2.append([i[0]for i in top2]) #storing only item ids
    user_lists2.append([user]*10) #10 entries of each user is created.

top_recommendations2=[] # empty list
for i in range(len(total_items_recommend2)): #iterating through length of recommendations g
    temp=[] #empty list
    for j in total_items_recommend2[i]: #for all the recommended items for each user
        if j in item_list2[i]: #if that recommended item is present in item list of that us
            temp.append(j) # append that item to new list

    top_recommendations2.append(temp[:10]) # append first 10 items satisfying the condition



final_users2=list(chain.from_iterable(user_lists2)) #creating a flattened iterable of users
top_10_recommendations2=list(chain.from_iterable(top_recommendations2)) #creating a flatten

#creating a dataframe containing final user ids and item ids
recommendations_dataframe2=pd.DataFrame({'user_id':final_users2,'item_id':top_10_recommenda

#writing the dataframe into a csv file
recommendations_dataframe2.to_csv('sample_solution_data.csv',index=False)
recommendations_dataframe2[1:11]
```

Observations :

ALS on trainData Vs ALS on trainData + validationData

There will be a significant increase in accuracy

Model 3.

Bayesian Personalised Ranking on trainData + validationData

In [ ]:

```
# Building the Bayesian Personalized Ranking
model3 = implicit.bpr.BayesianPersonalizedRanking(factors=5, regularization=0.1, iterations
alpha_val = 35 # Setting the scaling parameter alpha
data_3 = (data_sparse_item2 * alpha_val).astype('double') # Configuring the data and conver
model3.fit(data_3) # fitting the bayesian personalized ranking model .
```

In [ ]:

```
user_id3=list((testData.user_id.unique())) #storing all the unique user ids as a list

item_list3=[] #empty list
for user in user_id3: #iterating through user_id list
    item_list3.append(testData[testData['user_id']==user]['item_id'].tolist()) #storing all



total_items_recommend3=[] #empty list
user_lists3=[] #empty list
for user in user_id3: #iterating through each user id
    top3=model3.recommend(user,data_sparse_user2,N=len(testData['item_id'].unique().tolist(
    total_items_recommend3.append([i[0]for i in top3]) #storing only item ids
    user_lists3.append([user]*10) #10 entries of each user is created.

top_recommendations3=[] # empty list
for i in range(len(total_items_recommend3)): #iterating through length of recommendations g
    temp=[] #empty list
    for j in total_items_recommend3[i]: #for all the recommended items for each user
        if j in item_list3[i]: #if that recommended item is present in item list of that us
            temp.append(j) # append that item to new list

    top_recommendations3.append(temp[:10]) # append first 10 items satisfying the condition



final_users3=list(chain.from_iterable(user_lists3)) #creating a flattened iterable of users
top_10_recommendations3=list(chain.from_iterable(top_recommendations3)) #creating a flatten

#creating a dataframe containing final user ids and item ids
recommendations_dataframe3=pd.DataFrame({'user_id':final_users3,'item_id':top_10_recommenda

#writing the dataframe into a csv file
# recommendations_dataframe3.to_csv('sample_solution_data.csv',index=False)
recommendations_dataframe3[1:11]
```

Achieved accuracy for BPR is which is significantly low. ALS algorithm is a better choice in this case.

Main advantages of ALS are :

1. Implicit datasets are usually sprase and ALS optimisation technique is more efficient approach compared to other techniques such as SGD.
2. It is very easy to parallelize.

In [ ]:

```

```

Task 2

Node Classification

In [ ]:

```
#Importing the packages required for this task
import pandas as pd
import numpy as np
import networkx as nx
from node2vec import Node2Vec
from gensim.models import Word2Vec  #word2vec
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confus
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import BernoulliNB
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from nltk.corpus import stopwords
import nltk
from sklearn.model_selection import train_test_split
```

About Datasets:

1. `docs.txt` : Contains title information for rach node ID
2. `adjedges.txt` : Contains neighbour nodes of each node in a network. In each row, first item is a node ID and the rest items are nodes that are linked to the first item.
3. `labels.txt` : Contains class labels for each node. Each row represents a node id and its corresponding class label.

Read graph in adjacency list format from path.

In [ ]:

```
G=nx.read_adjlist("adjedges.txt")
```

Node2Vec

- In Node2Vec we learn to map nodes to a low dimensional space of features that maximises the likelihood of preserving network neighbourhood of nodes.
- In node classification task, we are interested in predicting most probable labels of nodes in a network.
- In prediction problem on network, one has to construct a feature vector representation of edges and nodes.

- While this supervised procedure results in good accuracy, it comes at the cost of high training time complexity due to a blow up in the number of parameters that need to be estimated.
- Algorithm should learn node representationss that embed nodes from the same network community close to each other. And also nodes that share similar roles have similar embeddings. This would allow feature learning algorithms to generalise across a wide variety of domains and prediction tasks.
- In summary, `Node2vec` is a semi-supervised learning algorithm for scalable feature learning in networks. By choosing an appropriate notion of a neighborhood, node2vec can learn representations that organise the nodes based on their network roles. We achieve this by developing a family of biased random walks, which efficiently explore neighborhoods of given node.

Initialising `Node2vec` on the citation network with the following parameters.

1. `walk_length` : Number of nodes the algorithm visits in each walk
2. `num_walk` : Number of times the algorithm performs the random walks
3. `dimensions` : dimensions of the vector generated by the algorithm
4. `workers` : number of workers the algorithm uses for parallelisation

In [ ]:

```
# pre-compute the probabilities and generate walks :
node2vec = Node2Vec(G, dimensions=15, walk_length=30, num_walks=20,workers=4)
```

Below, `node2vec.fit` uses the vector generated by node2vec algorithm and runs word2vec algorithm on top of it Its parameters are :

1. `window` : defines total number of words before and after current word used by word2vec algorithm
2. `min_count` : frequency of words that need to be dropped
3. `batch_words` : size of chunk sent to each workermm

In [ ]:

```
#embed the nodes

model = node2vec.fit(window=10, min_count=1, batch_words=4)
```

In [ ]:

```
# importing pickle for saving the model and its reusability.
# import pickle


# pickle_out = open("dict4.pickle","wb")
# pickle.dump(model, pickle_out)
# pickle_out.close()
```

In [ ]:

```
# pickle_in = open("dict3.pickle","rb")
# model= pickle.load(pickle_in)
```

Reading labels dataset and doing some pre processing to create a final dataframe with two columns i.e. nodes and their corresponding labels

In [ ]:

```
labels=pd.read_csv('labels.txt',header=None) #reading labels dataset

labels.columns=['Name'] #giving the name of column

labels[['First','Last']] = labels.Name.str.split(" ",expand=True,) #splitting the column ar

del labels['Name'] #deleting the previous name column

labels.head() # looking at first 5 rows
```

Using the generated model and constructing the vectorised form of each node and storing all node vectors as a list

In [ ]:

```
X=model.wv[labels['First']].tolist()  #generating and storing node vectors as a list
```

Storing all their corresponding labels as a list

In [ ]:

```
y=(labels['Last'].tolist()) #storing labels as a list
```

Splitting the final data (that contains node vectors and their corresponding labels for each node ID) into train, test split set with stratified sampling

In [ ]:

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.8,shuffle=True,strati
```

Comparing and evaluating the performance of different classification algorithms

In [ ]:

```
accuracy_list=[]
models_list=[]
embedding_list=[]
models = [LogisticRegression(),BernoulliNB(),LinearSVC(),RandomForestClassifier()] #storing
classifiers=['Logistic Regression','Bernoulli NB','Linear SVC','Random Forest Classifier']
for i in range(len(models)): #looping over each model from model list
    clf=models[i] # taking each model
    clf.fit(x_train,y_train) #fitting train data on each model
    y_predict=clf.predict(x_test) #making predictions on test set
    accuracy=accuracy_score(y_test,y_predict) #calculating the accuracies
    print('Accuracy achieved by using model : '+classifiers[i]+' on Node Embedding Vectors
    accuracy_list.append(accuracy)
    models_list.append(classifiers[i])
    embedding_list.append('Node Embedding')
```

By doing Node Embedding, among the above 4 classifiers, accuracy is comparitively high for Linear SVC(). Its accuracy is around 0.55.

But, Accuracy achieved seems to be very less when node embedding is performed. Let's leverage the

`docs.txt` file to see if there is any improvement in the accuracy

This filecontains title information of each node and convert into vectors in latent space.

Generate vectors for each node ID by doing Text embedding and seeing if there is any improvement in accuracy

Word2Vec Text Embedding

Reading the docs file and doing some preprocessing to perform further operations

In [ ]:

```
docs=pd.read_csv('docs.txt',header=None,sep=" \n") # reading the text file as dataframe

docs.columns=['Name'] #assigning the column name

docs[['Node ID','text']] = docs.Name.str.split(n=1,expand=True) #splitting the column into

del docs['Name']

docs.head()
```

In [ ]:

```
docs.shape #shape of the file
```

In [ ]:

```
def clean_text(text):
    """
    :param X: a text
    :return: clean tokens of the text
    """
    text=text.lower() # converting to lower case
    tokens= nltk.word_tokenize(text) #building tokens
    tokens=[x for x in tokens if x.isalpha()] #taking only alphabets
    tokens= [word for word in tokens if word not in stopwords.words('english')] #removing s
    return tokens #returning the tokens
```

In [ ]:

```
docs['text']=docs['text'].apply(clean_text) # cleaning the text and converting into tokens
```

In [ ]:

```
docs.head() #first 5 rows
```

In [ ]:

```
model_word=  Word2Vec(docs.text,min_count=1,sg=1) # building the Word2Vec model
```

In [ ]:

```
#vector representaion of one of the words.
model_word.wv['assessing']
```

Creating a new column and and storing the node vectors for each node

In [ ]:

```
docs['node_vectors']=X #storing node vectors
```

Creating a new column and storing the labels for each node

In [ ]:

```
docs['y']=y #storing node labels
```

In [ ]:

```
docs.shape  #shape of the final dataframe
```

Checking for rows if there are any empty tokens

In [ ]:

```
indexes=[] #empty rows
for i in range(len(docs['text'])): #looping over text rows
    if len(docs['text'][i])==0: #if the length of token list is zero
        indexes.append(i) #appending those indexes
print('Number of rows that have empty tokens are :',len(indexes))
```

Deleting all the rows that have empty tokens in tokens column i.e. 'text'

In [ ]:

```
doc=docs[docs['text'].map(len) !=0]  # removing the rows with empty tokens

doc.shape #shape of the new dataframe
```

It can be seen that around 200 rows are dropped.

In [ ]:

```
doc=doc.reset_index(drop=True) #resetting the index
```

Replacing all the tokens with their corresponding vectors generated using Word2Vec and storing it as a new column in the data frame

In [ ]:

```
doc['text_vectors']= doc['text'].apply(lambda x: [model_word.wv[word] for word in x])# Repl
```

Adding up all the vectors to achieve a summed vector that represents each node

In [ ]:

```
doc['text_vector_sum']=doc['text_vectors'].apply(lambda x: sum(x)) #adding up all the vecto
```

Storing all the text vectors as a list and their corresponding labels as lists which are further used for model building and calculate accuracies

In [ ]:

```
X_text=doc['text_vector_sum'].tolist() #taking this vector column as a list
Y_text=doc['y'].tolist() #taking the l=final labels as a list
```

Splitting the final data (that contains text vectors and their corresponding labels for each node ID) into train, test split set with stratified sampling

In [ ]:

```
x_train, x_test, y_train, y_test = train_test_split(X_text, Y_text, test_size=0.8,shuffle=T
```

In [ ]:

```
models = [LogisticRegression(),BernoulliNB(),LinearSVC(),RandomForestClassifier()] #storing
classifiers=['Logistic Regression','Bernoulli NB','Linear SVC','Random Forest Classifier']

for i in range(len(models)): #looping over the models
    clf=models[i] #taking each model
    clf.fit(x_train,y_train) #fitting train dataset
    y_predict=clf.predict(x_test) #making predictions on test data
    accuracy=accuracy_score(y_test,y_predict) #calculating the accuracy
    print('Accuracy achieved by using model : '+classifiers[i]+' on Text Embedding Vectors
    accuracy_list.append(accuracy)
    models_list.append(classifiers[i])
    embedding_list.append('Text Embedding')
```

There is an improvement in the accuracy compared to Node Embedding. Linear SVC() accuracy achieved by doing text embedding is around 70 %

Node embedding + Text Embedding

Concatenating vectors generated through node embedding and text embedding for each node ID and use this final vector for model development and see if there is an improvement in accuracy.

In [ ]:

```
final=[] #empty list
for i in range(len(doc['node_vectors'])): #iterating over length of dataframe column
    final.append(np.array(doc['text_vector_sum'][i].tolist()+doc['node_vectors'][i])) #conc
```

Storing this concatenated vector list as a column of dataframe. This column represents possible vector representation of each node in latent space.

In [ ]:

```
doc['node_text']=final #storing this concatenated vector list as a column of dataframe
```

Storing all the concatenated node+text vectors as a list and their corresponding labels as lists which are further used for model building and calculate accuracies

In [ ]:

```
X_final=doc['node_text'].tolist() #taking this vector column as a list

Y_final=doc['y'].tolist() #taking the l=final labels as a list
```

Splitting the final data (that contains concatenated node+text vectors and their corresponding labels for each node ID) into train, test split set with stratified sampling

In [ ]:

```
x_train, x_test, y_train, y_test = train_test_split(X_final, Y_final, test_size=0.8,shuffle
```

Comparing and evaluating the performance of different classification algorithms

In [ ]:

```
models = [LogisticRegression(),BernoulliNB(),LinearSVC(),RandomForestClassifier()] #storing
classifiers=['Logistic Regression','Bernoulli NB','Linear SVC','Random Forest Classifier']

for i in range(len(models)): #looping over the models
    clf=models[i] #taking each model
    clf.fit(x_train,y_train) #fitting train dataset
    y_predict=clf.predict(x_test) #making predictions on test data
    accuracy=accuracy_score(y_test,y_predict) #calculating the accuracy
    print('Accuracy achieved by using model : '+classifiers[i]+' on Concatenated Node + Tex
    accuracy_list.append(accuracy)
    models_list.append(classifiers[i])
    embedding_list.append('Node + Text')
```

There is an improvement in the accuracy compared to Node Embedding. Linear SVC() accuracy achieved by concatenating Node + Text embedding is around 75 %

Constructing a Data Frame to compare Embedding Types and their accuracies on various models

In [ ]:

```
table={'Model Name':models_list,'Embedding Type':embedding_list,'Accuracy':accuracy_list} #
```

In [ ]:

```
accuracy_dataframe=pd.DataFrame(table) # converting dictionary to dataframe
```

In [ ]:

```
accuracy_dataframe # displaying the dataframe
```

In [ ]:

```
accuracy_dataframe[accuracy_dataframe.Accuracy==accuracy_dataframe['Accuracy'].max()]
```

From the above dataframe it can be seen that Linear SVC() performed better than other three classifier in terms of accuracy and it can be noticed that highest accuracy is achieved on a model where data was from combining both Node vectors and Text vectors

In [ ]:

```
#the End
```