TEAM

Raghav Chugh	|	18UCS195
Harshit Malhotra	|	18UCS207
Lakshay Bhagtani	|	18UCS132
Divyansh Singh	|	18UCS127

EDGE WEIGHT PREDICTION

AI PROJECT REPORT

OVERVIEW

In this project, we perform edge weight detection in a weighted signed network, specifically a social signed network using Gaussian Naive Bayes Algorithm.

INTRODUCTION

Weighted signed networks (WSNs) are networks in which edges are labeled with positive and negative weights. WSNs can capture like/dislike, trust/distrust, and other social relationships between people. In this paper, we consider the problem of predicting the weights of edges in such networks.

A signed social network (SSN) is a network where edges may be labeled as being “positive” or “negative”. For instance, if a vertex u dislikes a vertex v, there may be an edge with a “negative” edge label whereas if u likes v, the same edge labeled would be “positive”. However, in the real-world, people may like or dislike one another with varying levels of intensity. Person A might dislike B a little bit, but dislike C a lot more. Or person A may trust B a little bit, but trust C a lot more. Or person A may disagree with B a little bit, but disagree with C a lot more. All of these concepts (liking, trusting, agreeing) are different and not necessarily symmetric, yet they all can be captured via (directed) weighted signed networks (WSNs)

Video Demo :

https://drive.google.com/file/d/1rcg2_s58kTyg3dZh314g-l1loxePUVM-/view?usp=sharing

DATASETS USED

Bitcoin OTC trust weighted signed network

Dataset information

This is who-trusts-whom network of people who trade using Bitcoin on a platform called Bitcoin OTC. Since Bitcoin users are anonymous, there is a need to maintain a record of users' reputation to prevent transactions with fraudulent and risky users. Members of Bitcoin OTC rate other members in a scale of -10 (total distrust) to +10 (total trust) in steps of 1. This is the first explicit weighted signed directed network available for research.

Dataset statistics	
Nodes	5,881
Edges	35,592
Range of edge weight	-10 to +10
Percentage of positive edges	89%

Wikipedia Requests for Adminship (with text)

Dataset information

For a Wikipedia editor to become an administrator, a request for adminship (RfA) must be submitted, either by the candidate or by another community member. Subsequently, any Wikipedia member may cast a supporting, neutral, or opposing vote.

We crawled and parsed all votes since the adoption of the RfA process in 2003 through May 2013. The dataset contains 11,381 users (voters and votees) forming 189,004 distinct voter/votee pairs, for a total of 198,275 votes (this is larger than the number of distinct voter/votee pairs because, if the same user ran for election several times, the same voter/votee pair may contribute several votes).

This induces a directed, signed network in which nodes represent Wikipedia members and edges represent votes. In this sense, the present dataset is a more recent version of the Wikipedia adminship election data. However, there is also a rich textual component in RfAs, which was not included in the older version: each vote is typically accompanied by a short comment (median/mean: 19/34 tokens). A typical positive comment reads, *"I've no concerns, will make an excellent addition to the admin corps",* while an example of a negative comment is, *"Little evidence of collaboration with other editors and limited content creation."*

Network statistics	
Nodes	10,835
Edges	159,388
Triangles	956,428

GOALS

1. Import the provided datasets.
2. Perform data preprocessing and cleaning.
3. Construct a graph using the provided dataset .
4. Extract the required features from the provided dataset.
5. Train the Naive bayes classifier on the Final dataset.
6. Use this Model to make predictions on new Nodes.

SPECIFICATIONS

Python Libraries used in this project

networkx - Used to Implement Graphs in python

sklearn - Used for Implement Gaussian Naive Bayes Algorithm in python

Maplotlib - Used to Visualize our datasets

DATASET INSIGHTS

```
df.shape
```

```
(104554, 3)
```

```
df.head()
```

	source	target	rating
0	5988	5849	0.437
1	5989	9263	-0.237
2	5989	7930	0.167
3	5989	8172	0.242
4	5989	8609	0.211

MILESTONES

Methods Applied For Feature Extraction

Based on the degree of the node pair, we consider either

A) The following attributes as features between a pair of nodes,

- Signed shortest path length between the nodes

```
def spl(node1, node2):
  w = G[node1][node2]["weight"]
  Flag=False
  try:
    G.remove_edge(node1,node2)
    nx.shortest_path_length(G, source=node1, target=node2)
    Flag = True
  except Exception as e:
    G.add_edge(node1,node2,weight = w)
    Flag = False
  ans = nx.shortest_path_length(G, source=node1, target=node2)
  if Flag:
    G.add_edge(node1,node2,weight=w)
  return ans
```

- Impact of presence of high degree nodes in the shortest path

```
def n_high_degree(node1, node2):
  w = G[node1][node2]["weight"]
  flag = False
  try:
    G.remove_edge(node1,node2)
    nx.shortest_path(G, source=node1, target=node2)
    Flag = True
  except Exception as e:
    G.add_edge(node1,node2,weight=w)
    Flag = False
  n=0
  for ele in nx.shortest_path(G, source=node1, target=node2):
    if G.degree[ele] > percentile65:
      n+=1
  if Flag:
    G.add_edge(node1,node2,weight=w)

  return n
```

- Ratio of structurally balanced path between and

```
from itertools import islice
def k_shortest_paths(G, source, target, k):
    return list(
        islice(nx.shortest_simple_paths(G, source, target), k)
    )

def r_balanced(node1,node2):
  n=0
  k=0
  for path in k_shortest_paths(G, node1, node2, 10):
    n+=1
    neg=0
    for u,v in zip(path[0:],path[1:]):
      if G[u][v]['weight'] == -1:
        neg+=1
    if neg%2 == 0:
      k+=1
  return float(k)/n
```

- Ratio of mutual neighbors with same sign

```
def r_mutual(node1,node2):
  k = 0
  n = 0
  for ele in nx.common_neighbors(G,node1,node2):
    if G[node1][ele]['weight']*G[node2][ele]['weight']>0:
      k+=1
    n+=1
  if n==0:
    return 0
  return float(k)/n
```

B) The following attributes,

- Node centrality scores

```
centrality_dict = nx.algorithms.centrality.degree_centrality(G)
clustering_dict = nx.algorithms.cluster.clustering(G)
degree_dict = {}
for node in G.nodes():
  degree_dict[node] = G.degree(node)
```

```
def centrality(node):
  return centrality_dict[node]
```

- Clustering coefficient of nodes and

```
def cluster_coeff(node):
  return clustering_dict[node]
```

- Propensity to form edges of particular sign

```
def propensity(node):
  List = [edge for edge in G.edges(node)]
  n = len(List)
  k=0
  for edge in List:
    if G[edge[0]][edge[1]]['weight']>0:
      k+=1
  if n == 0:
    return 0
  return float(k)/n
```

Note that :

- In the n_high_degree function we use the 65th Percentile Degree of our dataset to label a node as a ‘high degree’ node.
- In the r_balanced function we have taken the top 10 shortest paths between the node pairs to avoid overutilization of computational resources and to prevent the consideration of all the paths in structural balance.

Data Preprocessing

1. Scaling the dataset

We scale all the columns using `MinMaxScaler()`.

```
from sklearn.preprocessing import MinMaxScaler
```

```
min_max_scaler = MinMaxScaler()
min_max_scaler.fit(dataset[column_names[2:12]])
dataset[column_names[2:12]] = min_max_scaler.transform(dataset[column_names[2:12]])
```

2. Categorize the data

We make all positive edges weights 0 and all negative edge weights 1.

```
dataset['rating'] = dataset['rating'].apply(lambda x: 0 if x == 1 else 1)
```

3. Drop duplicate pairs

We remove all (b,a) pairs if an edge from (a,b) is already present in the dataset, to convert the data from directed to an undirected graph.

```
df.loc[pd.DataFrame(np.sort(df[['source','target']],1),index=df.index).drop_duplicates(keep='first').index]
df.head()
```

Data Preparation

We apply all the functionalities we added above and Prepare our data for analysis.

```
L = len(G.edges())
for i, edge in zip(tqdm(range(L), ncols = 100, desc ="Progress"),G.edges()):
  dataset = dataset.append({'node_1':edge[0],
                  'node_2':edge[1],
                  'node_centrality1' : centrality_dict[edge[0]],
                  'node_centrality2' : centrality_dict[edge[1]],
                  'propensity1': propensity(edge[0]),
                  'propensity2': propensity(edge[1]),
                  'cluster_coeff1' : clustering_dict[edge[0]],
                  'cluster_coeff2' : clustering_dict[edge[1]],
                  'shortest_path_length' : spl(edge[0],edge[1]),
                  'n_high_degree' : n_high_degree(edge[0],edge[1]),
                  'r_balanced': r_balanced(edge[0],edge[1]),
                  'r_mutual': r_mutual(edge[0],edge[1]),
                  'rating' : G[edge[0]][edge[1]]['weight']
                  }, ignore_index=True )
```

Algorithm Applied

Naive Bayes Classifier

Naive Bayes Classifiers are based on the Bayes Theorem. One assumption taken is the strong independence assumptions between the features. These classifiers assume that the value of a particular feature is independent of the value of any other feature. In a supervised learning situation. Naive Bayes classifiers need a small training data to estimate the parameters needed for classification.

Gaussian Naive Bayes

When working with continuous data, an assumption often taken is that the continuous values associated with each class are distributed according to a normal (or Gaussian) distribution. The likelihood of the features is assumed to be :

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Training the Models

We train two models `modelA` and `modelAB` using features of A and both A and B respectively.

For node pairs in which both nodes have a degree lesser than 10, we use `modelAB,` otherwise we use `modelA` to predict.

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import f1_score, confusion_matrix
from sklearn.metrics import accuracy_score
```

```
modelA = GaussianNB()
modelAB = GaussianNB()
```

```
modelA.fit(X_trainA, y_train)
```

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

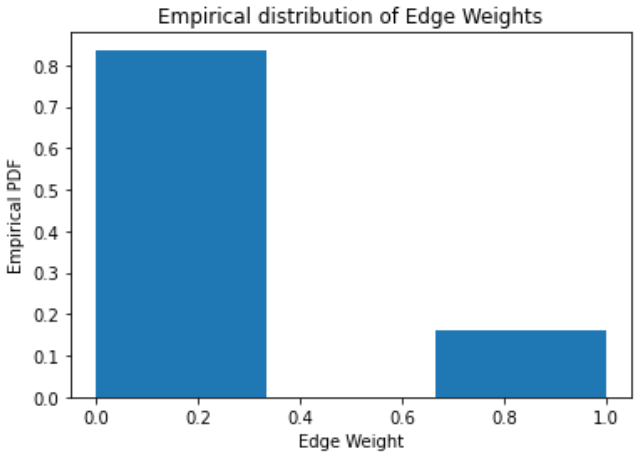
```
modelAB.fit(X_trainAB, y_train)
```

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

Insights

- **Empirical Distribution of Our Edge Weights**

  For this step, we take help of python library **`matplotlib`**



- **Description of the dataset**

  For this step, we take help of python library **`pandas`**

	node_centrality2	cluster_coeff2	propensity2	shortest_path_length	n_high_degree	r_balanced	r_mutual	rating
count	101886.000000	101886.000000	101886.000000	101886.00000	101886.000000	101886.000000	101886.000000	101886.000000
mean	0.128161	0.130538	0.831920	0.41472	0.561117	0.778873	0.687677	0.162760
std	0.142084	0.101821	0.183326	0.07378	0.107101	0.215226	0.353767	0.369149
min	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000
25%	0.033546	0.080882	0.767606	0.40000	0.600000	0.700000	0.500000	0.000000
50%	0.089457	0.114967	0.888889	0.40000	0.600000	0.800000	0.818182	0.000000
75%	0.169329	0.157505	0.958333	0.40000	0.600000	1.000000	1.000000	0.000000
max	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	1.000000	1.000000

Inferences

We infer from the above visualization that

- We are working on a skewed dataset so it becomes important to refer to metrics like Precision and Recall.
- Hence, there are a high number of positive edges.
- There are a high number of balanced paths due to the presence of a high number of positive edges.

Results

`modelA` gives an accuracy of 81.59% whereas `modelAB` gives us an accuracy of 85.05%, but a combined prediction model gives us an accuracy of 87.04%.

```
modelA.score(X_testA, y_test)
```

```
0.8159719950271543
```

```
modelAB.score(X_testAB, y_test)
```

```
0.8505529019171628
```

```
y_pred = np.array([])
```

```
for cols in test_data.values:
  if degree_dict[cols[0]]<10 and degree_dict[cols[1]]<10:
    y_pred = np.append(y_pred, modelA.predict(cols[8:12].reshape(1, -1)))
  else:
    y_pred = np.append(y_pred, modelAB.predict(cols[2:12].reshape(1, -1)))
```

```
accuracy_score(y_test,y_pred)
```

```
0.8704547536478441
```

We infer that:

- For node pairs with higher degrees, features of a) are enough to predict the presence of a positive or a negative edge link.
- But for node pairs with low degree, we require features of a) as well as b) to predict the presence of a positive or a negative edge link.

Conclusion

Metrics : We used the following attributes to predict the presence of a positive or a negative link between two nodes :

A.

- Signed shortest path length
- Presence of high degree nodes
- Ratio of structurally balanced path
- Ratio of mutual neighbors with same sign, or

B.

- Node centrality scores
- Clustering coefficient of nodes
- Propensity to form edges of particular sign.

Effectiveness : We show that the above features can be used to calculate unknown weights in WSNs with higher precision. In conjunction with other features defined for WSNs, we show that our prediction engine is able to perform suitably well as compared to previous methods.

REFERENCES

[1] S. Kumar, F. Spezzano, V.S. Subrahmanian, C. Faloutsos. Edge Weight Prediction in Weighted Signed Networks. IEEE International Conference on Data Mining (ICDM), 2016.

[2] Robert West, Hristo S. Paskov, Jure Leskovec, and Christopher Potts: Exploiting Social Network Structure for Person-to-Person Sentiment Analysis. *Transactions of the Association for Computational Linguistics*, *2(Oct):297–310, 2014*

CONTRIBUTORS

- **Harshit Malhotra 18ucs207**
  - Understanding the nature of problem statement and listing project requirements
  - Gathering required datasets and Information around the problem
  - Creating graph from the given data

- **Raghav Chugh 18ucs195**
  - Creating methods for extracting features from graph
  - Making the final dataset with all the required features

- **Lakshay Bhagtani 18ucs132**
  - Preprocessing the final dataset
  - Using Naive Bayes Algorithm to make predictions on the final dataset

- **Divyansh Singh 18ucs127**
  - Using data visualization to gain valuable insights from the data
  - Implementing the final interface function from Edge Weight Prediction
  - Preparing the final project report and video demo

SOURCE CODE

Github

https://github.com/raghavchugh21/AI.coursework/blob/main/Assignment.ipynb