Detection of Circular Trading

Aman Panwar
CS20BTECH11004
Department of Computer Science

Amulya Tallamraju
AI20BTECH11003
Department of Aritificial Intelligence

Pranav K Nayak
ES20BTECH11035
Department of Engineering Science

Taha Adeel Mohammed
CS20BTECH11052
Department of Computer Science

Vikhyath Sai Kothamasu
CS20BTECH11056
Department of Computer Science

***Abstract*—Detection of Circular Trading using Node2Vec**

I. Introduction

This report details the problem of circular trading, the nature of the dataset on which enquiries are made, the algorithm used to detect circular trades within the dataset, and a summary of our results.

II. Problem Statement

A. Circular Trading

Circular trading is a fraudulent practice where a company creates fake transactions with another entity or itself to manipulate financial statements or inflate revenues. This deceptive technique involves using shell companies, subsidiaries or other third parties to buy and sell goods or services between them multiple times, making it seem like there is genuine business activity happening.

The practice is based on the premise that a company's share price is directly dependent on the trading volume. High trading volumes signal to the market that the company is expected to implement change that will increase its share price, such as new management or a new product launch. Investors, when they suspect that such a change is happening, invest in order to get ahead of the growth in stock price. This leads to the stock price itself going up, and the company begins to trade at a higher valuation.

Circular trading attempts to fraudulently recreate this trading volume by sending circular trades across shell companies and entities controlled by the company attempting to defraud investors. This causes an artificial inflation in stock price, since the signal received by investors about the stock trend is not based in reality.

B. Approach

We have extracted potential clusters of circular traders by applying the Node2Vec algorithm to find vector embeddings of each of the graph nodes, followed by the DBSCAN algorithm to find clusters in the graph.

III. Description of the Dataset

The dataset for this assignment is contained in `Iron_dealers_data.csv`. It is a table of invoices, with each row containing a single invoice for a single transaction. Each invoice is listed as a 3-tuple of `Seller ID, Buyer ID, Value`, with value being the amount transacted. The dataset contains 130,535 such entries.

IV. Algorithm Used

In order to find communities of circular traders, we first need to represent them as a graph, followed by performing analyses on the graph to find clusters where trading happens frequently within the cluster.

A. Generating A Directed Multigraph

We begin by first taking the sales data and constructing what is known as a *Sales Flow Graph*, a directed multigraph with each vertex corresponding to a single dealer (Buyer/Seller), and edges between vertices weighted by the value of the transaction between them. The code for conversion of the dataset to the graph can be seen in Figure 1. We are then

```
df = pd.read_csv('Iron_dealers_data.csv')
df.head()

# Create edge-weighted graph from DataFrame
G = nx.MultiDiGraph()
for i, row in df.iterrows():
    seller = row['Seller ID']
    buyer = row['Buyer ID']
    value = row['Value']
    G.add_edge(seller, buyer, weight=value)
print(G)
```

```
MultiDiGraph with 799 nodes and 130535 edges
```

Fig. 1. Python code to generate Sales Flow Graph

left with a directed multigraph, but for Node2Vec to work, we need it to be a undirected graph.

B. Generating an Undirected Graph

For generating edges between any two vertices u and v, we look at all the 2- and 3-cycles that exist in the multigraph between u and v. For each cycle $c \in C$, we compute a score for c according to the following rule:

$$\text{let average cycle amount be } \bar{c}$$

$$\text{score(c)} = \begin{cases} \bar{c} & \text{if}(max(c) - min(c))/min(c) < 0.1, \\ \frac{\bar{c}}{100} & \text{if}(max(c) - min(c))/min(c) > 0.1 \end{cases}$$

Using this score, we compute the weight between vertices u and v as

$$Weight_{u,v} = \sqrt{\Sigma_{c \in C} \text{score}(c)^2}$$

Thus, we now have a weighted, undirected graph representing dealers in the dataset, with edges representing their level of association. We can see the distribution of edge weights in the graph below:

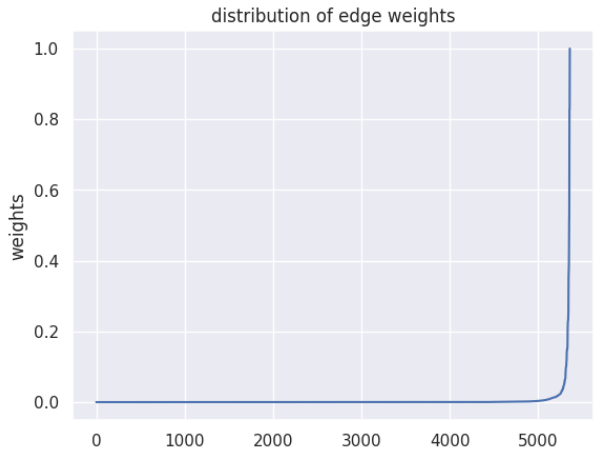

Fig. 2. Distribution of edge weights across the dataset

The following visualization of the undirected graph shows how few of the nodes have heavy weights along their edges. Since very few edges have a very high weight, it is highly likely that these edges are involved in circular trading.

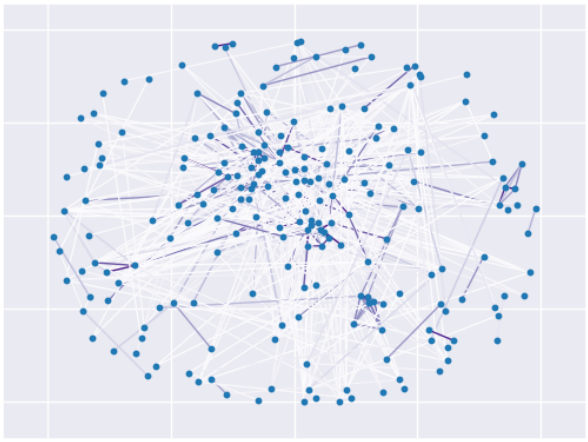

Fig. 3. The undirected graph visualized

C. Generating Embeddings

We then pass the undirected graph into Node2Vec so that we can get vector embeddings that can then be passed into an unsupervised learning model to perform clustering. Refer to Figure 4 for the code to invoke the model.

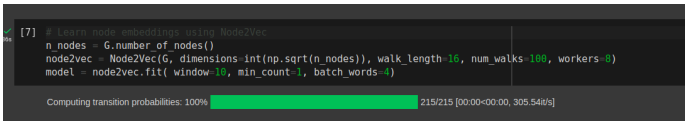

Fig. 4. Python code to invoke the Node2Vec model

D. Dimensionality Reduction

The embeddings generated by Node2Vec have a dimensionality equal to the square root of the number of nodes in the undirected weighted graph. This can be very high, leading to our unsupervised learning problem suffering from the *Curse of Dimensionality*, where each vector encodes so much information that it is considered equidistant from every other vector. Do deal with this, and extract meaningful clusters from the data, we perform dimensionality reduction on the embeddings' space, through the use of t-SNE. Refer to Figure 5 for the Python code to perform dimensionality reduction.

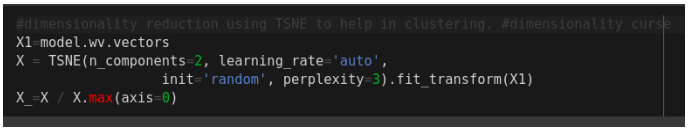

Fig. 5. Dimensionality Reduction via t-SNE

E. Clustering with DBSCAN

Once the data has been processed and its dimensionality sufficiently reduced to where information can be extracted from it without too much noise getting in the way, we perform DBSCAN on the embeddings to find all the clusters that it can. DBSCAN uses cosine similarity to group vectors together, and the resultant clusters can be seen in Figure 8.

```
# clustering with DBScan
clusters = DBSCAN(eps=0.0325, min_samples=4).fit(X_)
```

Fig. 6. Calling DBSCAN in Python

V. Results

A. Distribution of Points in a Cluster

The number of points in a cluster is a good indicator that the clustering worked, since clusters should not have too many or too few points within them. The figure below confirms this.

```
label_dict = {}
for i, x in enumerate(X_):
    key = clusters.labels_[i]
    label_dict.setdefault(key, [])
    label_dict[key].append(x)


print('number of point in cluster:')
for x in label_dict:
    if x  == -1: continue
    print(f"cluster {x} :\t{len(label_dict[x])}")
```

```
number of point in cluster:
cluster 1 :     4
cluster 0 :     4
cluster 11 :    5
cluster 2 :     5
cluster 9 :     5
cluster 3 :     4
cluster 4 :     4
cluster 10 :    4
cluster 5 :     4
cluster 12 :    5
cluster 6 :     4
cluster 7 :     4
cluster 8 :     4
```

Fig. 7. Counts of points within each cluster

Here we can see that most points are evenly distributed across clusters.

B. Visualizing the Clusters

Below we find the clusters, the result of our DBSCAN visualized on a graph. Note that DBSCAN will generate slightly different results every time it is run.

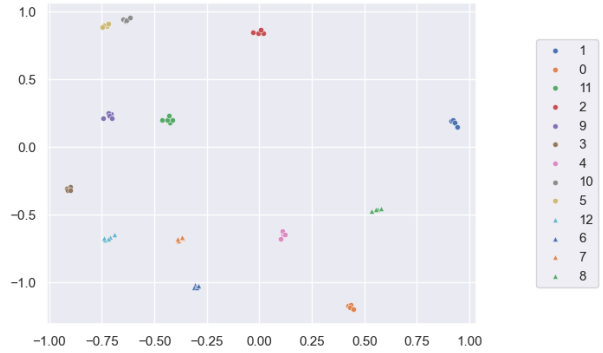

Fig. 8. The clusters

C. Nodes within a Cluster

In order to extract the dealer IDs from their embeddings in order to find out which dealers are grouped into clusters, we present the following code and output, which lists the cluster number and the IDs of those dealers that are grouped into them. These clusters are the suspected groups for circular trading.

```
print('Nodes in clusters:')
label_W_nodes= {}
for x, l in zip(data_tsne.node_ids, data_tsne.cluster_id):
    label_W_nodes.setdefault(l ,[])
    label_W_nodes[l].append(x)

for l in label_W_nodes:
    if l==-1:continue
    print(f'cluster {l}: {label_W_nodes[l]}')
```

```
Nodes in clusters:
cluster 1: ['1007.0', '1038.0', '1050.0', '1534.0']
cluster 0: ['1245.0', '1108.0', '1220.0', '1141.0']
cluster 11: ['1138.0', '1419.0', '1158.0', '1324.0', '1317.0']
cluster 2: ['1356.0', '1149.0', '1254.0', '1251.0', '1113.0']
cluster 9: ['1035.0', '1073.0', '1126.0', '1471.0', '1431.0']
cluster 3: ['1330.0', '1381.0', '1250.0', '1043.0']
cluster 4: ['1004.0', '1172.0', '1136.0', '1104.0']
cluster 10: ['1133.0', '1159.0', '1305.0', '1640.0']
cluster 5: ['1114.0', '1067.0', '1065.0', '1155.0']
cluster 12: ['1210.0', '1626.0', '1258.0', '1057.0', '1048.0']
cluster 6: ['1355.0', '1214.0', '1215.0', '1314.0']
cluster 7: ['1318.0', '1376.0', '1319.0', '1370.0']
cluster 8: ['1076.0', '1075.0', '1580.0', '1510.0']
```

Fig. 9. Dealers in a Cluster

VI. References

- Priya Mehta, Sanat Bhargava, M. Ravi Kumar, K. Sandeep Kumar, Ch. Sobhan Babu, "Representation Learning on Graphs to Identifying Circular Trading in Goods and Services Tax"