Outlier Identification via Spectral Clustering

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*—This report documents our implementation of outlier detection using spectral clustering on dealer transaction data.**

I. Introduction

Within any dataset, anomalies are those points whose behaviour and properties deviate to a non-trivial degree from the norm defined by the dataset. Detecting these anomalies is an important problem, one that can improve our understanding of the dataset by discarding those values that throw off the reported behaviour of the dataset. Identifying them can also help when our job is to specifically search for outliers, since many problems, ranging from identifying credit card fraud in the financial sector to diagnosing malignant tumours in healthcare, explicitly look for outliers as their solutions.

II. Problem Statement

We proceed to identify outliers in dealers from their transaction data. We search for these outliers through the use of spectral clustering, grouping dealers whose behaviour is deemed similar by the algorithm. Any anomalies whose financial behaviour deviates from the norm could potentially be committing financial crimes.

III. Description of the Dataset

The dataset contains 1999 rows, with each row corresponding to a dealer. Each dealer has 10 features that define them. It is these features which we use to determine our clusters. The following are some metrics we used to determine similarity of features across the dataset.

- The Pearson Correlation Matrix

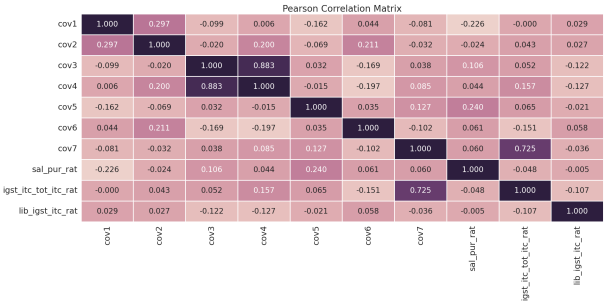

Fig. 1. The Pearson Correlation Matrix

- The Kendall Correlation Matrix

Fig. 2. The Kendall Correlation Matrix

- The Spearman Correlation Matrix

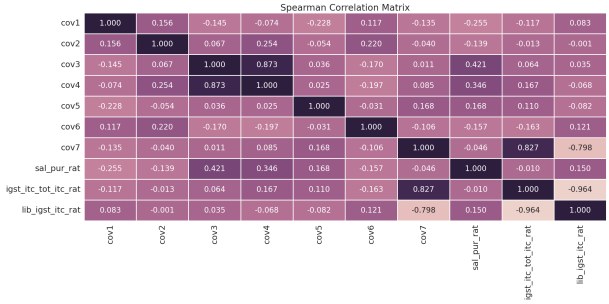

Fig. 3. The Spearman Correlation Matrix

- The Cosine Similarity Matrix

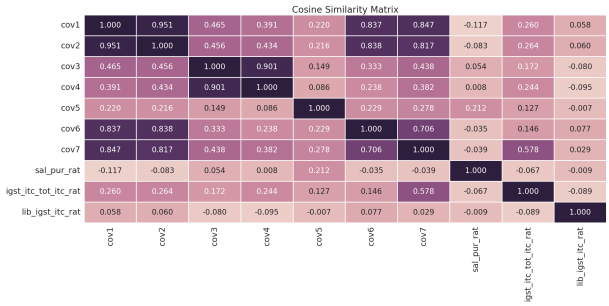

Fig. 4. Caption

We also computed the Gaussian Distributions of each feature in the dataset:

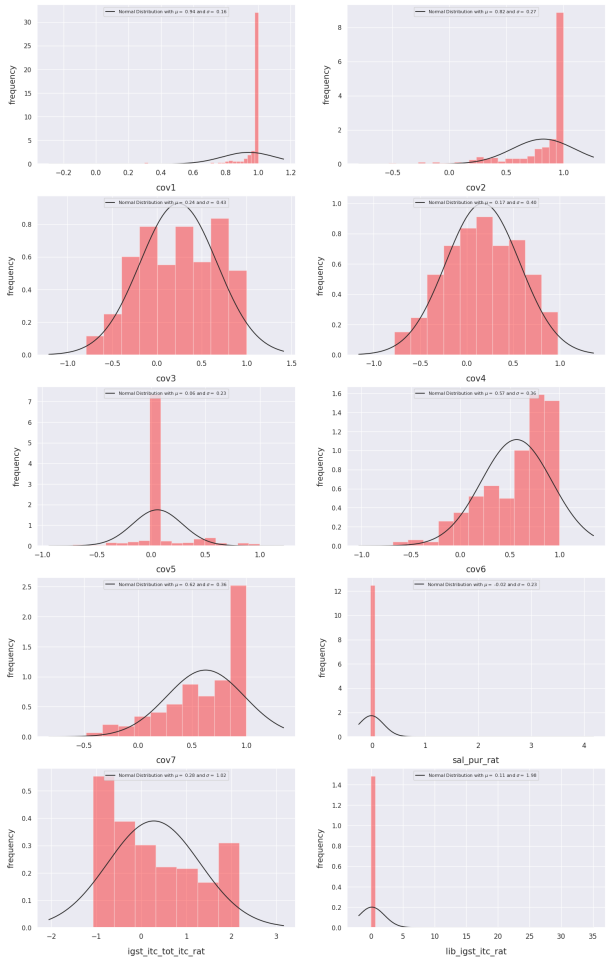

Fig. 5. Gaussians for each Feature

IV. Algorithm Used

A. Normalizing the Data

We normalized the data within the dataset using `StandardScaler`. We also performed Principal Component Analysis on the dataset as a means of dimensionality reduction, to help with the final visualizations.

```
scaler = StandardScaler()
scaled_df = scaler.fit_transform(df)

pca = PCA(n_components=2)
principal_components = pca.fit_transform(scaled_df)
print(principal_components.shape)

(300, 2)
```

Fig. 6. Normalizing the data and performing PCA

B. Constructing the Adjacency Matrix and the Laplacian

We meed to determine the number of clusters we want spectral clustering to compute. To determine this, we compute, for a given similarity measure, the Adjacency Matrix, a matrix that indicates whether two nodes in the graph are similar enough to count as adjacent. We then compute the Laplacian from this Adjacency Matrix. Using the Laplacian, we compute its eigenvectors and sort them in ascending order. We then compute the differences between adjacent eigenvalues, called the eigengaps. We then select the index of that eigenvalue that leads to the largest eigengap. That index is often the value that gives us a stable number of clusters.

1) Using Euclidean Norm Similarity: We compute the Gaussian Norm Similarity for every pair of vectors on the grap, i.e. for any pair $\bar{u}, \bar{v}$, we compute the value of $||\bar{u} - \bar{v}||_2^2$. If this value is greater than some preset threshold k, we count them as being adjacent and set the adjacency matrix element indexed by u, v, i.e. $A[u, v]$ to be 1. Below is a figure indicating the different adjacency results with different values for our threshold k. Note that visualization has been achieved by performing PCA on the dataset first.

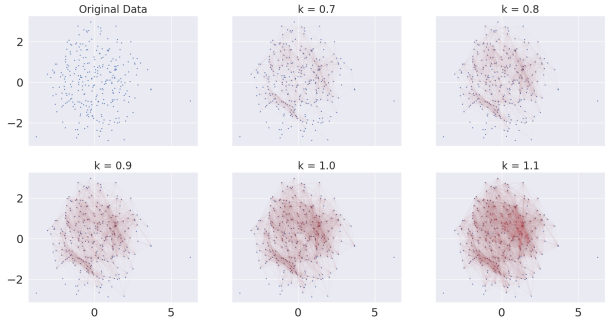

Fig. 7. Norm based adjacency visualizations for different k's

2) Using k-Nearest Neighbours: We perform the same process as above, except we now instead of Euclidean similarity we use k-Nearest Neighbours as our criterion of similarity. We plot below the visualizations of adjacency density for different values of k.

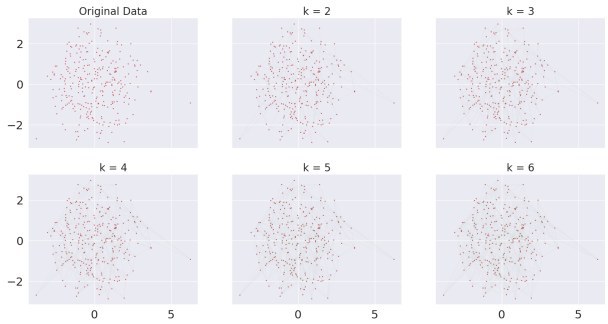

Fig. 8. k-NN based adjacency visualizations for different k's

3) Using Weighted Connections: For this representation of the adjacency graph, we treat the graph as fully connected and give adjaceny meaning by assigning every edge $(\bar{u}, \bar{v})$ a weight corresponding to the Gaussian similarity function $s(\bar{u}, \bar{v}) = e^{-\frac{||\bar{u}-\bar{v}||_2^2}{\sigma^2}}$, σ being a hyperparameter that determines the spread of the neighbourhood.

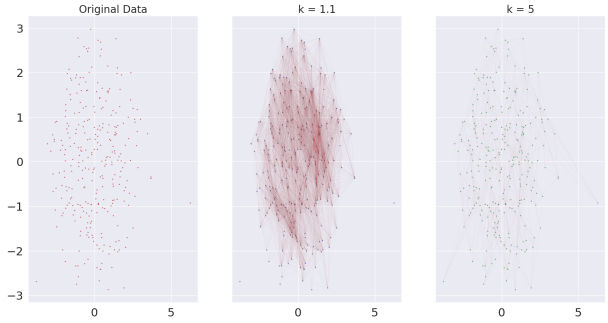

Fig. 9. Gaussian similarity based adjacency visualizations for different k's

C. Computing Eigenvalues/Eigenvectors

In order to compute the eigenvectors for any given Laplacian, we first need to normalize the Laplacian.

```
[ ]  def normalizeLaplacian(L, norm='sym'):
        if norm == 'rw':
            D = np.diag(np.diag(L))
            L_norm = np.linalg.inv(D) @ L
        elif norm == 'sym':
            D = np.diag(np.diag(L))
            D_sqrt = np.sqrt(np.linalg.inv(D))
            L_norm = D_sqrt @ L @ D_sqrt
        return L_norm
```

Fig. 10. Python implementation for normalizing a Laplacian

Following this, we need to compute the eigenvectors, for which we make use of `np.linalg.eigh()`. We also define functions to store the values of the eigengap for any particular index `k` in the eigenvalues list, as well as returning that index of the list that causes the largest eigengap.

```
def gen_eigenvectors(L, k):
    eigenvalues, eigenvectors = np.linalg.eigh(L)
    idx = np.argsort(eigenvalues)[:k]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]
    return eigenvalues, eigenvectors

def eigengap(eigenvalues, k):
    return eigenvalues[k] - eigenvalues[k - 1]

def bestk(eigenvalues, maxk=10):
    eigengaps = [eigengap(eigenvalues, i) for i in range(2, maxk)]
    return np.argmax(eigengaps) + 2
```

Fig. 11. Python implementation for computing eigenvalues, eigengaps, bestK

D. Comparing Cluster Counts for Each Similarity

We then run the logic to count the recommended number of clusters for each of the three different formulations of similarity we computed earlier in the report. We compute the Laplacian for each, followed by computing the eigenvalues, followed by the eigengaps, and finally ending on a count of the number of clusters as the index of the largest eigengap's eigenvalue. As a result, we get the following visualizations:

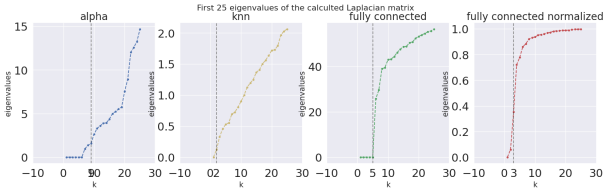

Fig. 12. First 25 eigenvalues across Laplacians

```
Best k for alpha is 9
Best k for knn is 2
Best k for fully connected is 5
Best k for fully connected normalized is 3
```

Fig. 13. Recommended cluster counts per metric

V. Results

After computing the cluster counts, we can now perform spectral clustering, extracting the clusters for each of the types of similarity metrics. Performing this yields the following results:

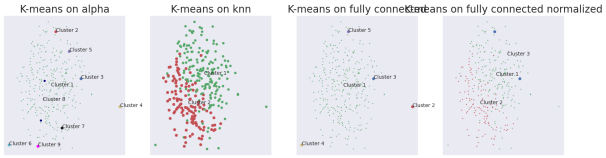

Fig. 14. Spectral Clustering Results across Laplacians

Thus, we have detailed our approach and implementation of performing Spectral Clustering on a dataet to extract outliers.

VI. References

- Ulrike von Luxburg, "A Tutorial on Spectral Clustering", Statistics and Computing, 17 (4), 2007, Springer
- scikit-learn's SpectralClustering Module Documentation