Indian Institute of Technology Hyderabad

Fraud Analytics (CS6890)

Assignment: 1 | Identify clusters using (Node2Vec Embedding, Spectral, and GCN) embeddings

Name	Roll Number
Shreesh Gupta	CS23MTECH12009
Hrishikesh Hemke	CS23MTECH14003
Manan Patel	CS23MTECH14006
Yug Patel	CS23MTECH14019
Bhargav Patel	CS23MTECH11026

Contents

1 | Problem statement

- The objective of this project is to identify clusters within a dataset representing financial transactions. Each row in the dataset represents a payment transaction and includes information about the sender, receiver, and the transaction value. The dataset is rich in relational information, making it suitable for exploring cluster structures using advanced embedding techniques.
- Specifically, the project aims to leverage three distinct embedding methods for clustering analysis:
  1. **Node2Vec Embedding:** Node2Vec is a popular algorithm used for generating embeddings from graph data. In this context, the financial transactions dataset can be represented as a graph where nodes represent entities (senders and receivers) and edges represent transactions. Node2Vec will be applied to create vector representations (embeddings) of nodes based on their network neighborhood, capturing the structural information and relationships between nodes.
  2. **Spectral Embedding:** Spectral clustering is a technique that uses the eigenvalues of a similarity matrix derived from the dataset to perform clustering. Spectral embedding transforms the data into a lower-dimensional space using eigenvectors corresponding to the largest eigenvalues of the similarity matrix. This method is effective for identifying clusters with non-linear structures and is suitable for high-dimensional data like financial transaction datasets.
  3. **Graph Convolutional Network (GCN) Embedding:** GCNs are a type of deep learning model designed to operate on graph-structured data. They can learn node embeddings by aggregating information from neighboring nodes in the graph. GCNs are well-suited for capturing complex relational patterns and are particularly effective for clustering tasks in graph data. The GCN embedding approach will be applied to capture the intricate relationships and patterns present in the financial transaction graph.
- The problem statement involves applying these three embedding techniques to the financial transaction dataset and evaluating their effectiveness in identifying meaningful clusters. Clustering aims to group similar transactions together based on sender-receiver relationships and transaction values, uncovering hidden patterns and structures within the data. The clusters generated will be analyzed and interpreted to gain insights into the underlying transaction dynamics, potential fraud detection, or anomaly identification.

2 | Description of the data set

- The dataset titled "Payments" consists of transaction data between various entities, recorded across 130,535 entries. It comprises three primary integer-valued columns:
  1. **Sender:** This column identifies the sender of the payment. Each sender is represented by a unique integer identifier.
  2. **Receiver:** Similar to the Sender column, this column contains integer identifiers for the receivers of the payments, indicating the recipient of each transaction.
  3. **Amount:** This column represents the monetary value of each transaction in integer format. It specifies the amount transferred from the sender to the receiver.

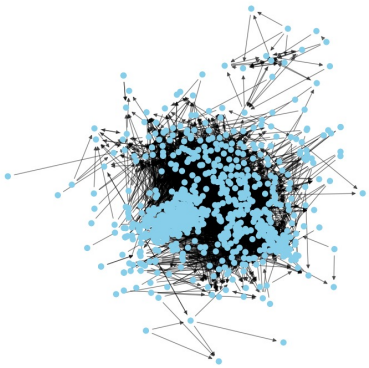


Figure 2.1: Dataset Visualization

- The memory usage for this dataset is approximately 3.0 MB, indicating a moderately sized dataset that is manageable for various data processing and analysis tasks.
- The data is entirely non-null, suggesting complete records across all three fields. The datatype for all columns is integer (int64), which supports efficient handling and processing of the dataset.
- This dataset is pivotal for analyzing financial transactions between entities, potentially aiding in understanding patterns such as the flow of money, identifying high transaction volumes between specific entities, and more.

3 | Algorithm

3.1 | Node2Vec

Node2Vec is designed for generating node embeddings that capture the network's topology while also considering a node's neighborhood's structure. The algorithm leverages the flexibility of random walks to explore diverse neighborhoods and uses these walks to train a **Word2Vec** model.

3.1.1 | Key Parameters:

- **dimensions:** The number of dimensions of the embedding space.
- **walk-length:** The number of steps in each random walk.
- **num-walks:** The number of walks to start at each node
- **p** and **q:** Parameters which control the random walk process. **p** (return parameter) controls the likelihood of immediately revisiting a node in the walk, while **q** (in-out parameter) differentiates between inward and outward nodes in terms of walk transitions.

3.1.2 | Transition Probabilities:

- The transition probabilities between nodes during the walks are defined as follows:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v,x) \in E \\ 0 & \text{otherwise} \end{cases}$$

Where:

(π_{vx}) represents the unnormalized transition probability between nodes (v) and (x),
(Z) is a normalizing constant ensuring probabilities sum to 1 over all choices for (x).

The probabilities (π_{vx}) are computed based on the edge weights and the parameters (p) and (q) :

$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

Where $(w - vx)$ is the weight of the edge from (v) to (x), and $(\alpha_{pq}(t,x))$is given by:

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } x = t \\ 1 & \text{if } x \neq t \text{ and } x \in N(t) \\ \frac{1}{q} & \text{if } x \notin N(t) \end{cases}$$

3.1.3 | Random Walk Simulation:

- Random walks are simulated based on the precomputed transition probabilities. Each walk begins at a selected node and sequentially transitions to a neighboring node based on the probabilities, creating paths that reflect the local network structure.

3.1.4 | Model Training:

- After simulating the walks, they are used to train a Word2Vec model. This model learns vector representations for each node that are useful for machine learning tasks on graphs such as node classification or clustering.

3.1.5 | Implementation Details:

- The class **Node2Vec** includes methods to precompute transition probabilities, simulate random walks, and fit the Word2Vec model. The embeddings produced encapsulate both local and global structural information about the nodes, potentially improving performance on downstream tasks.

3.1.6 | Node2Vec Model Flow chart:

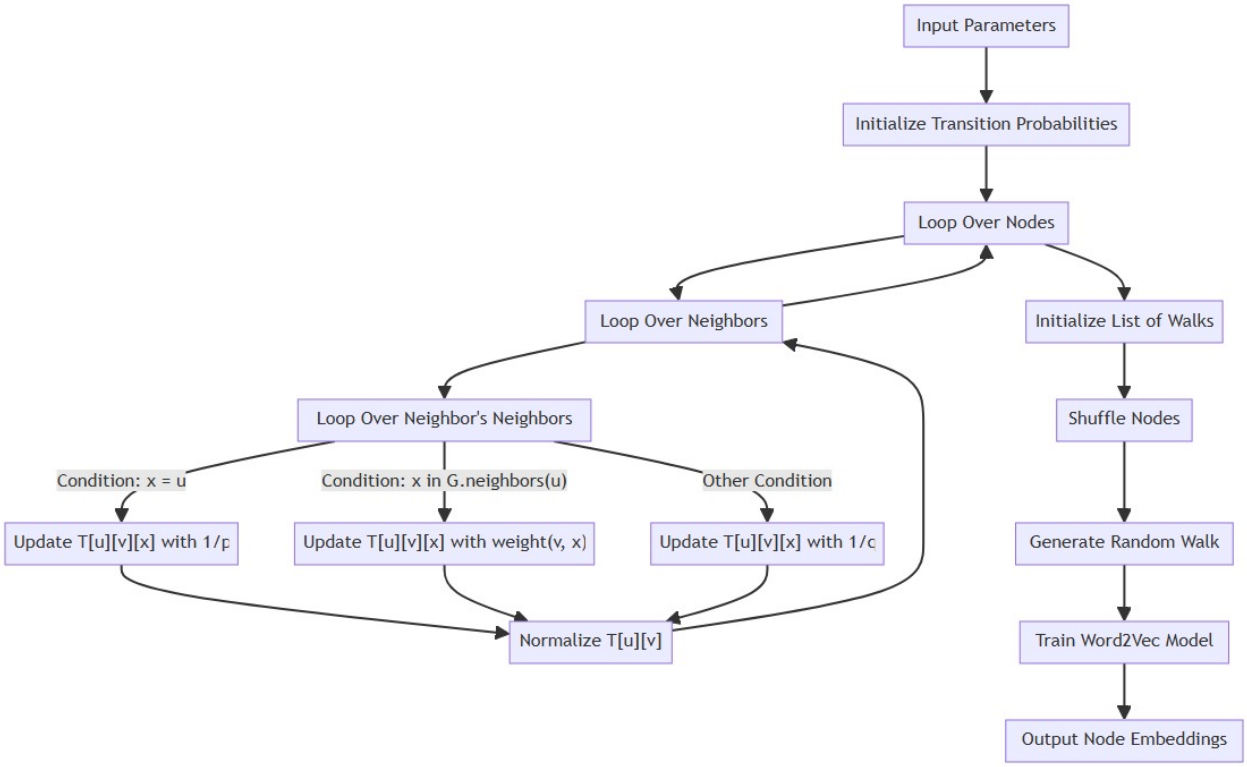


Figure 3.1: Node2Vec Flow chart

3.1.7 | DBSCAN Clustering Hyperparameter Tuning:

The **eps** parameter determines the maximum distance between two samples for them to be considered as in the same neighborhood. The aim is to find the 'eps' value that results in the highest number of clusters without considering outliers.

Preprocessing:

- **Standardization:** The feature set **X** (presumed to be a collection of embeddings) is standardized using **StandardScaler**. This normalization ensures that each feature contributes equally to the distance computation, crucial for distance-based algorithms like DBSCAN.

$$X' = \frac{X - \mu}{\sigma}$$

  Where X' is the scaled data, μ is the mean, and σ is the standard deviation.

Hyperparameter Tuning:

- **Exploring eps Values:** The **eps** values are explored over a range from 0.1 to 5, divided into 50 intervals. For each value of **eps**, the DBSCAN algorithm is executed to compute the number of clusters formed.

DBSCAN Execution:

- DBSCAN is run with the current **eps** and a fixed 'min-samples' of 3 (minimum points to form a cluster).
- The number of clusters is determined by counting unique labels, excluding noise points (labeled as '-1').

Visualization:

- **Plotting:** A line plot is generated to visualize the relationship between **eps** values and the number of clusters formed. This helps in visually identifying the **eps** value that maximizes the number of meaningful clusters.

Results:

- **Optimal Parameters:** The script outputs the best **eps** value and the corresponding number of clusters. This **eps** value is deemed optimal based on the criterion of maximizing the number of clusters.
- By adjusting the **eps** parameter based on this analysis, the DBSCAN algorithm can be finely tuned to capture the natural clustering structure of the dataset more effectively.

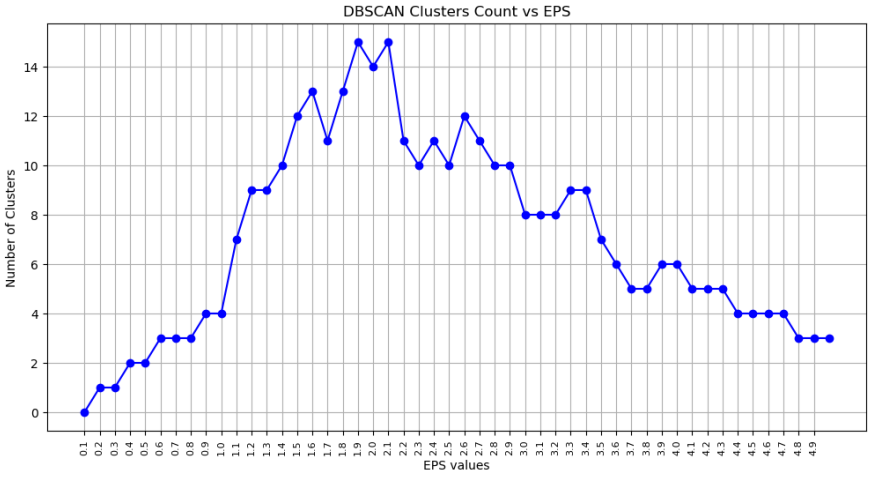
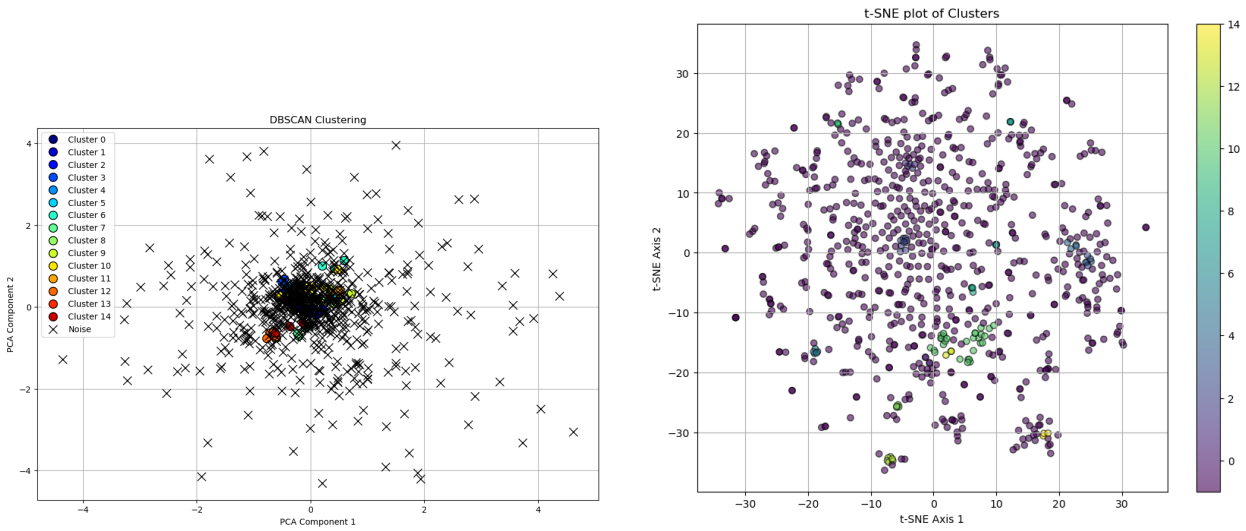


Figure 3.2: DBSCAN Clusters Count vs EPS



(a) DBSCAN Clustering

(b) tSNE plot of cluster

Figure 3.3: DBSCAN Results

3.1.8 | Clustering Evaluation Metrics

- The effectiveness of the DBSCAN clustering algorithm is evaluated using three different metrics, each providing insights into different aspects of the clustering quality.
- **Silhouette Score:**
  - **Range:** -1 to 1
  - **Interpretation:** A higher score indicates that clusters are compact and well-separated compared to other clusters. A score close to 1 denotes that clusters are dense and well-separated. A score near 0 indicates overlapping clusters, and a negative score suggests that samples might have been assigned to the wrong clusters.
- **Calinski-Harabasz Index:**
  - **Interpretation:** A higher score is better. It signifies that the clusters are dense and well-separated. This index is particularly useful for datasets with clusters of roughly equal size.
- **Davies-Bouldin Index:**
  - **Range:** 0 to Infinity
  - **Interpretation:** A lower index indicates better clustering. This index evaluates how much the clusters are separated and how compact they are, with lower values showing better clustering performance.
- These metrics collectively provide a comprehensive view of the clustering performance, helping to understand the strengths and weaknesses of the applied clustering approach. The scores are computed for the DBSCAN algorithm's results, reflecting how well it has identified dense and distinct clusters within the dataset.

3.1.9 | K Means

- A function **KMeansClassFinder** that is designed to determine the optimal number of clusters for a dataset using the Elbow Method, a popular heuristic in KMeans clustering. The function takes two parameters: **X**, which is the dataset to be clustered, and **n-classes**, which defines the range for the number of clusters to test (from 1 to n-classes - 1).

  WCSS Calculation:

  - It first initializes an empty list **wcss** to store the within-cluster sum of squares (WCSS). WCSS is a measure of the compactness of the clusters, with lower values generally indicating better clustering.

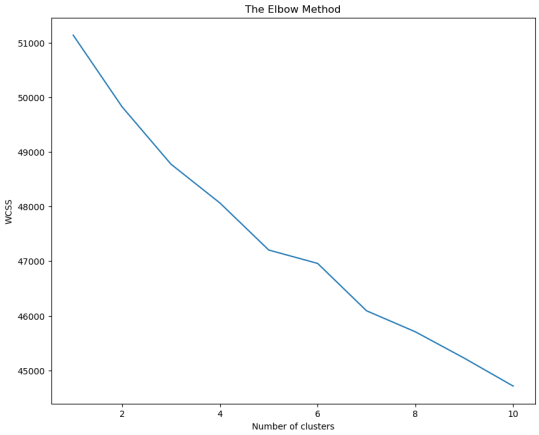
  KMeans Iteration:

  - The function then iterates over the specified range of cluster numbers. For each possible number of clusters, it initializes a KMeans clustering with parameters **init='k-means++'** for efficient centroid placement, **max-iter=300** for a maximum number of iterations, **n-init=10** for multiple initial centroid seeds, and a fixed **random-state=0** for reproducibility. It then fits the KMeans algorithm to the dataset **X** and appends the inertia (WCSS) of the resulting model to the **wcss** list.

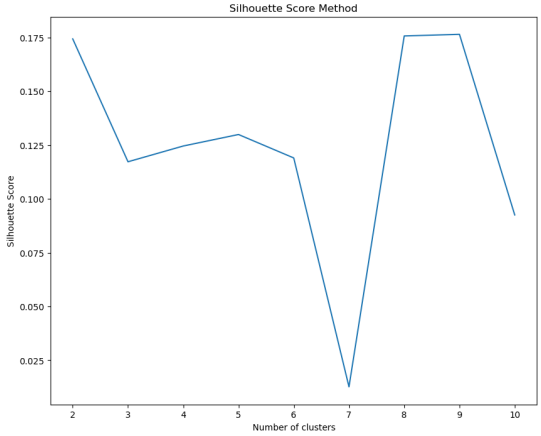
  Elbow Plotting:

  - After calculating WCSS for each cluster number, the function plots these values against the number of clusters using matplotlib. The plot is sized 10x8 inches and is labeled appropriately, including a title 'The Elbow Method'. The elbow point in the plot, where the rate of decrease in WCSS sharply shifts, can indicate the optimal number of clusters.

  This approach allows for a visual inspection of the WCSS to identify the best cluster count based on the Elbow Method.



(a) Elbow method



(b) Silhouette score method

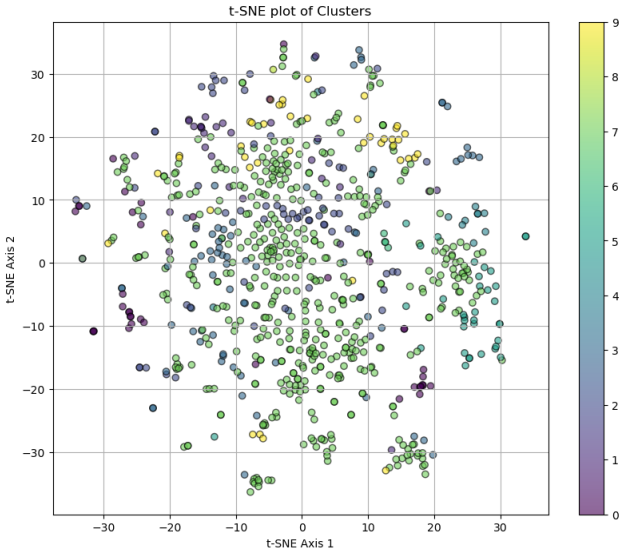


Figure 3.5: tSNE plot of cluster

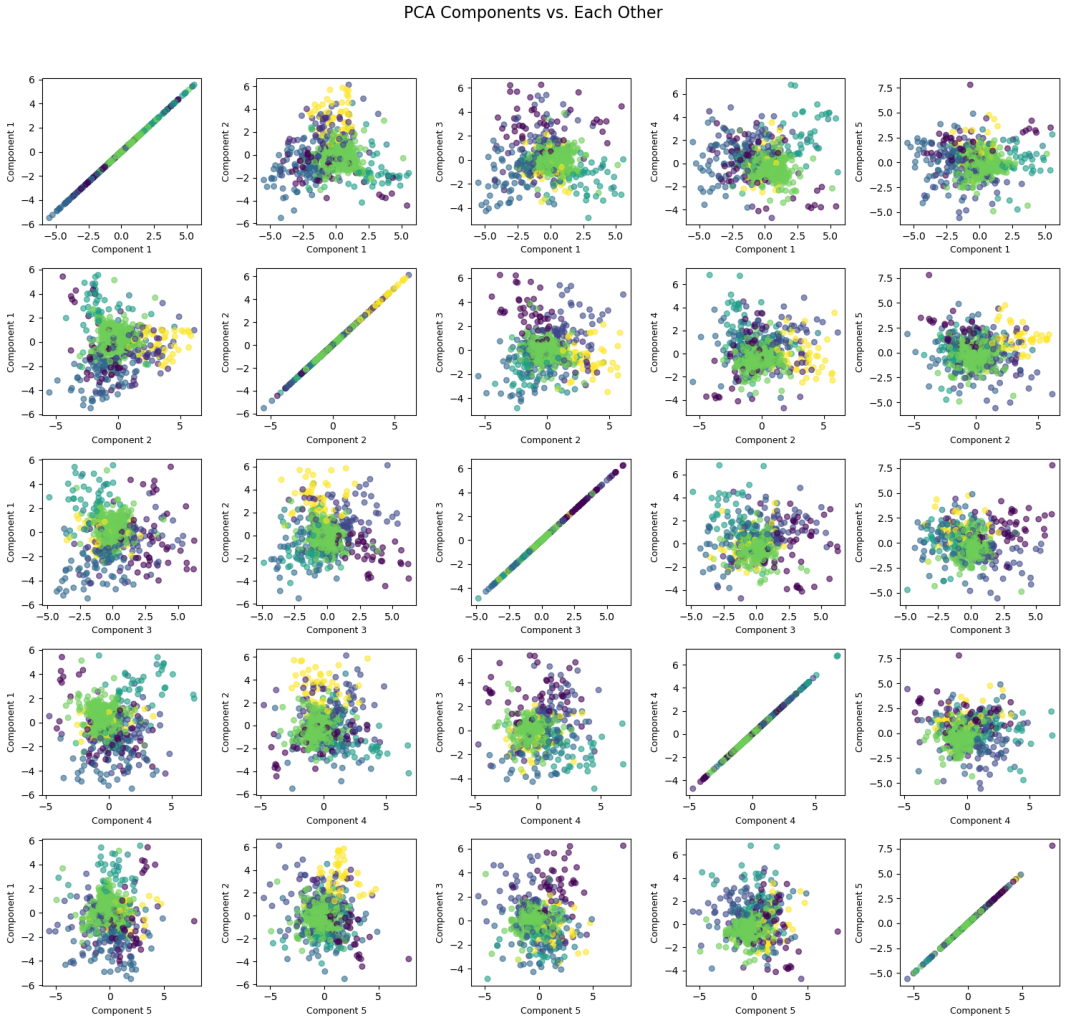


Figure 3.6: PCA Components VS each other

3.1.10 | Evaluation Metric

- **Hierarchical Clustering and Dendrogram Visualization** Hierarchical clustering using the Ward linkage method. The Ward method minimizes the variance of the clusters being merged, making it well-suited for identifying clusters that are relatively compact and equal in size.

  Function plot-dendrogram:

  - **Parameters:**
    - **X:** A data matrix where samples are rows and features are columns.
  - **Process:**
    1. **Linkage Computation:** The linkage() function computes hierarchical clusters using the 'ward' method, which is a common approach for hierarchical agglomerative clustering (HAC). This method is particularly effective for creating clusters by minimizing the total within-cluster variance. At each step, the pair of clusters with the minimum between-cluster distance are merged.
    2. **Dendrogram Plotting:** The resulting linkage array, which contains the hierarchical clustering information, is used to plot a dendrogram. The dendrogram illustrates how each cluster is composed by drawing a U-shaped link between a non-singleton cluster and its children.
  - **Visualization Settings:**
    - **orientation='top':** This places the root at the top, and items are displayed from top to bottom.
    - **distance-sort='descending':** This sorts the distances between pairs of clusters in descending order, aiding in the visualization of the most distinct clusters.
    - **show-leaf-counts=True:** This sorts the distances between pairs of clusters in descending order, aiding in the visualization of the most distinct clusters.
  - **Plot Customization:** The plot is adjusted to a size of 10x7 inches to ensure all elements of the dendrogram are clearly visible, making it suitable for detailed analysis or presentations.

  The function returns the linkage matrix, which can be used for further analysis or for plotting additional dendrograms with different configurations.

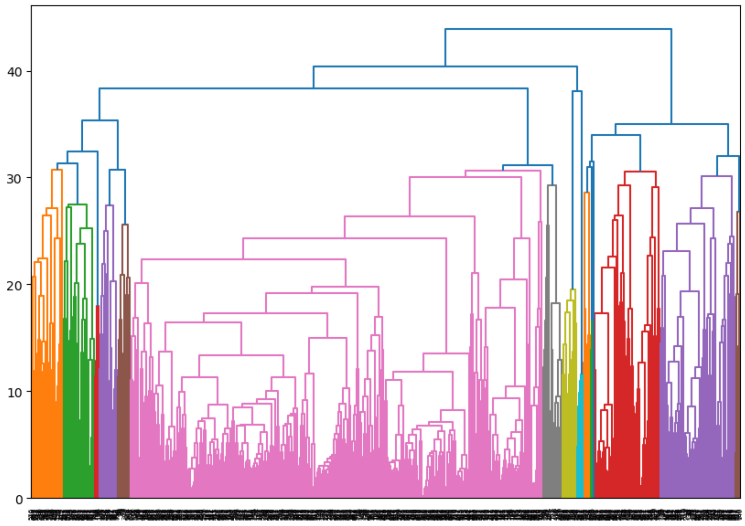
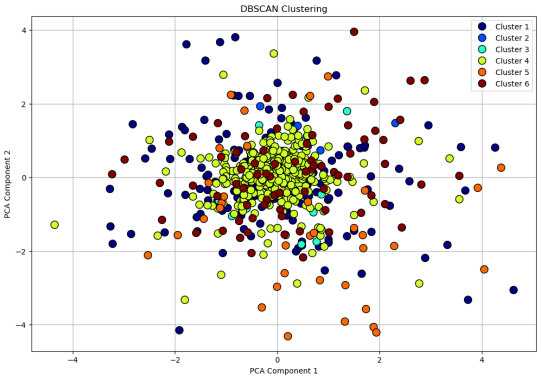
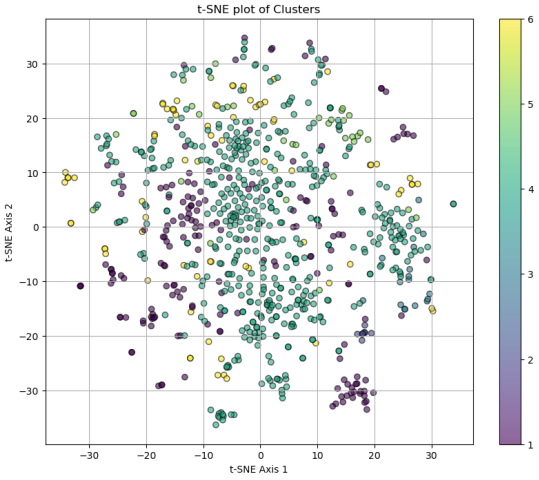


Figure 3.7: Hierarchical Clustering Visualization



(a) DBSCAN Clustering



(b) tSNE plot of cluster

3.2 | GCN

1. **Feature Matrix Creation:**
   - The script creates a feature matrix where each feature is the degree of a node. It collects the degrees of nodes in a sorted order, converts them into a PyTorch tensor of floats, and adds an extra dimension to make it a column vector.
   - This is done with the line **features = torch.tensor([G.degree(node) for node in sorted(G.nodes)], dtype=torch.float).unsqueeze(1).**
2. **Edge Index Tensor:**
   - An edge index tensor is created from the list of edges in the graph **G**. The tensor is converted to a long data type and then transposed to ensure the shape matches PyTorch Geometric's expectations (two rows where each column is an edge and rows represent source and target nodes respectively).
   - This tensor is made contiguous for performance reasons in memory, using **edge-index = torch.tensor(list(G.edges), dtype=torch.long).t().contiguous()**.
3. **Graph Data Object:**
   - A **Data** object (assuming from PyTorch Geometric) is instantiated with the node features **x** and the edge index **edge_index**. This object conveniently packages the graph data for processing in graph neural networks.
   - The setup is demonstrated with **data = Data(x=features, edge_index=edge_index)**.
4. **Function to Remap Edge Indices:**
   - The function **remap_edges** takes an edge index tensor and a dictionary mapping old node indices to new ones. It iterates through each edge, updating indices based on the mapping, ensuring the edge indices align with new or reordered node indices.
   - This function is critical in cases where node indices may have been changed or reordered and is applied to the graph data with **data.edge_index = remap_edges(data.edge_index, node_to_index)**.
5. **Output:**
   - Finally, the script prints the **data** object to display its structure and contents, which include the node features and the possibly remapped edge indices.

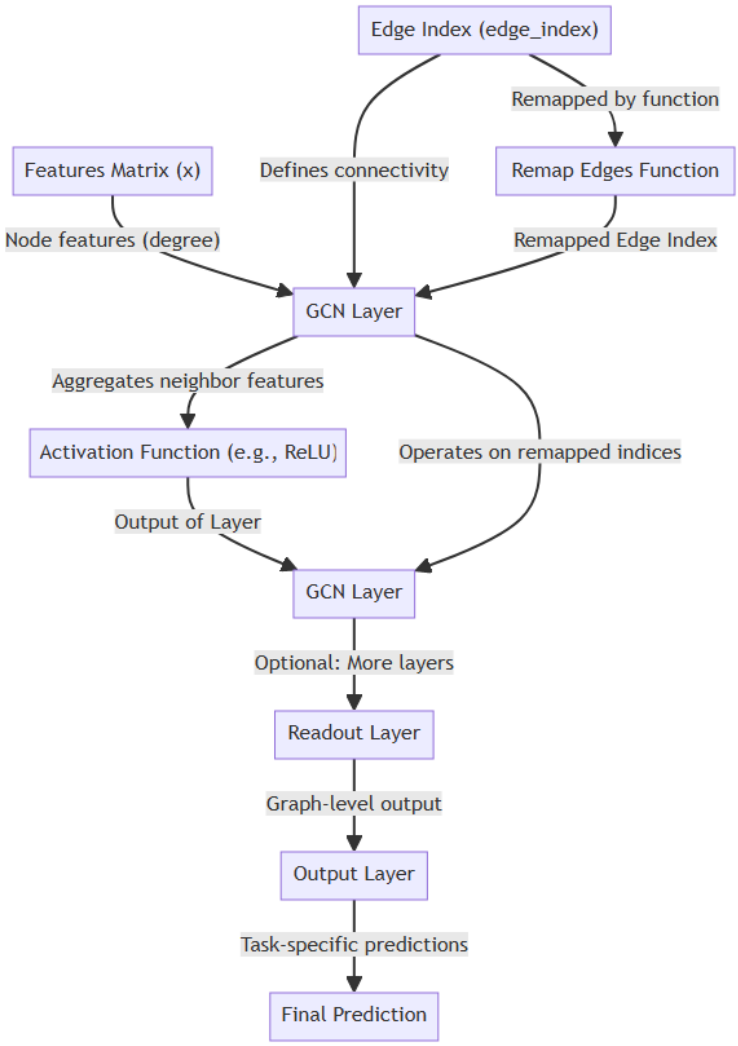
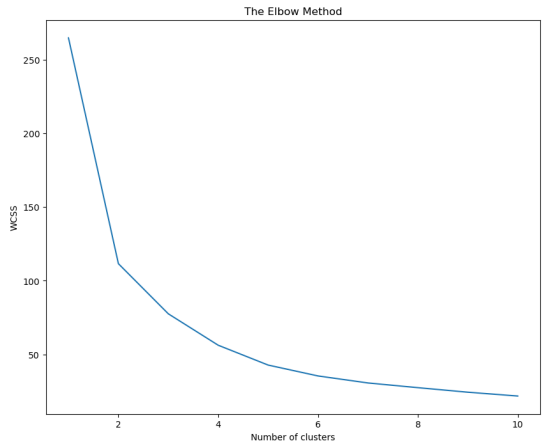
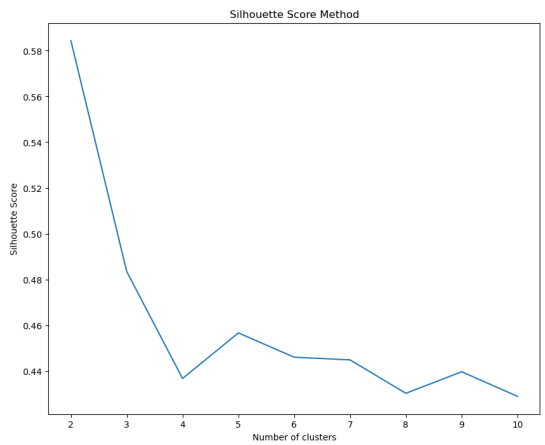


Figure 3.9: GCN Flow chart

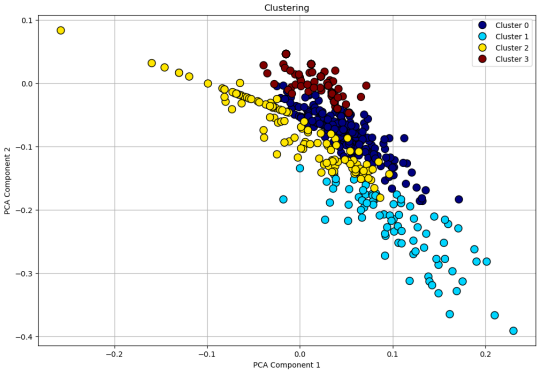
3.2.1 | K Means



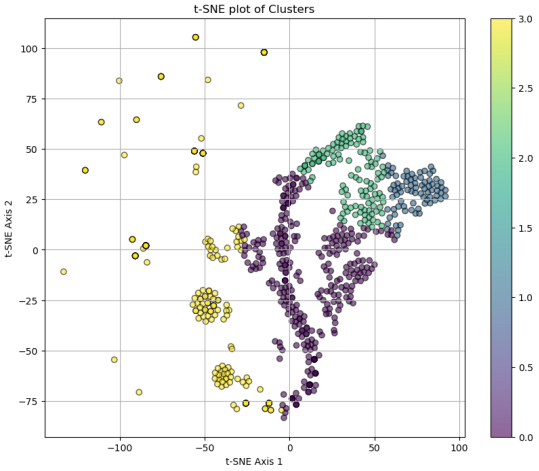
(a) Elbow Method



(b) Silhouette Scores Method



(a) Clustering



(b) tSNE plot of Clusters

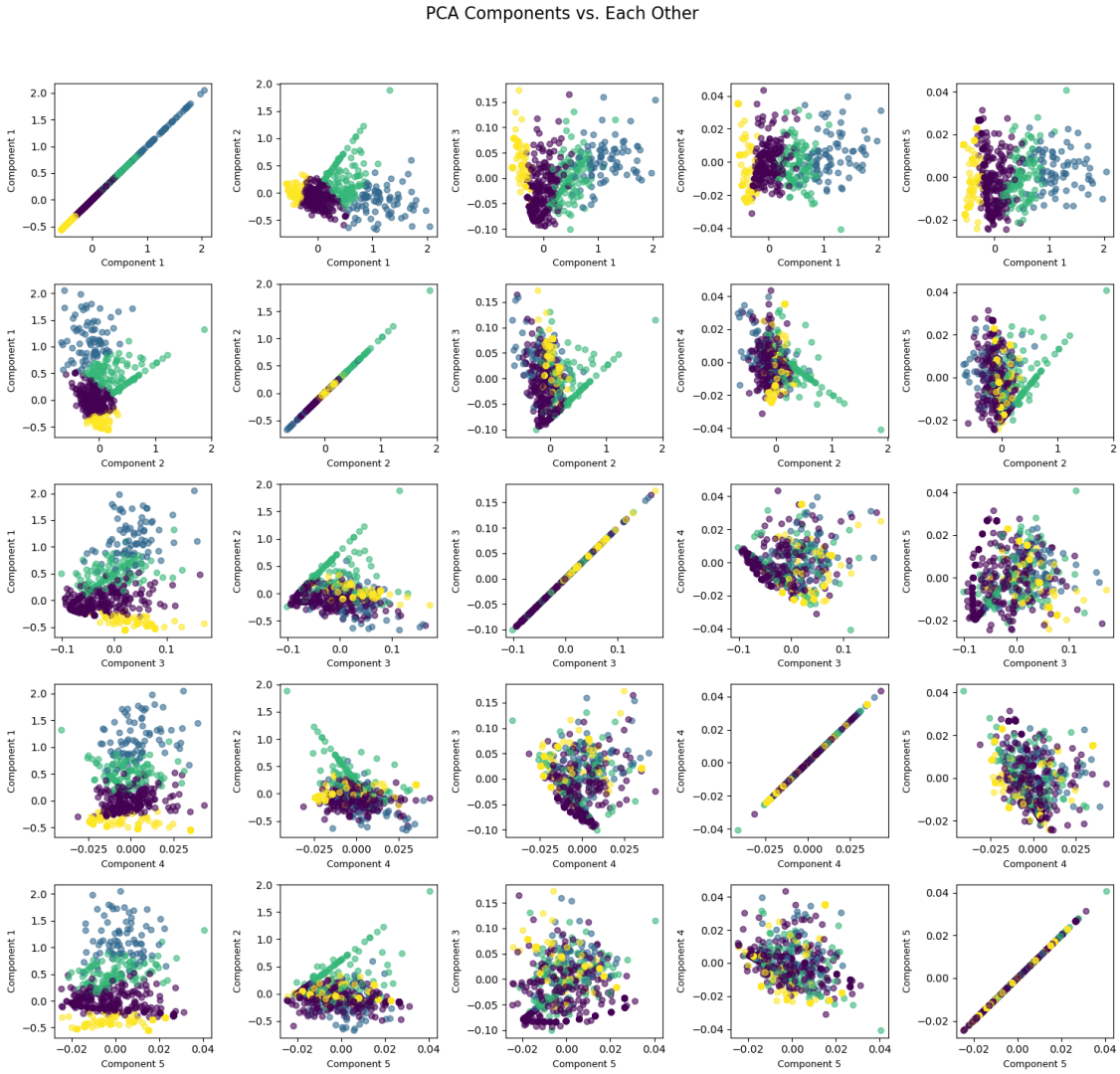
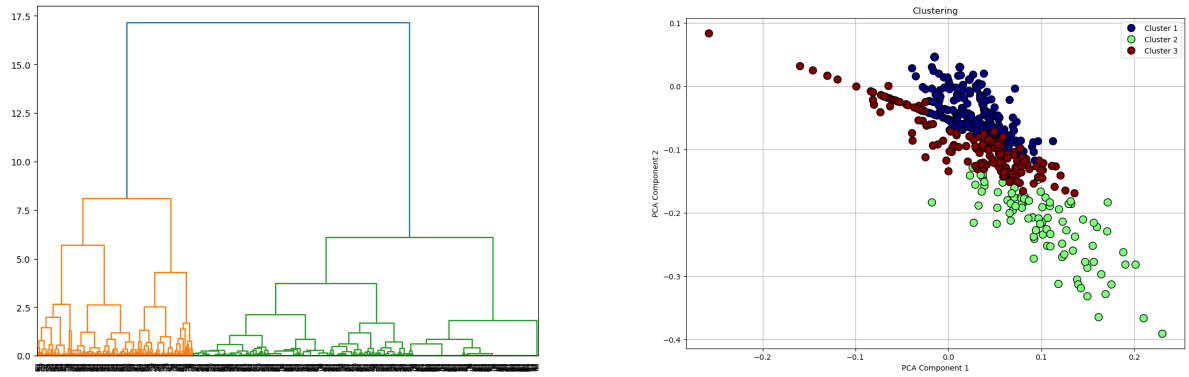


Figure 3.12: GCN PCA vs each other

3.2.2 | Hierarchical Clustering



(a) (b)

Figure 3.13: Hierarchical Clustering Visualization

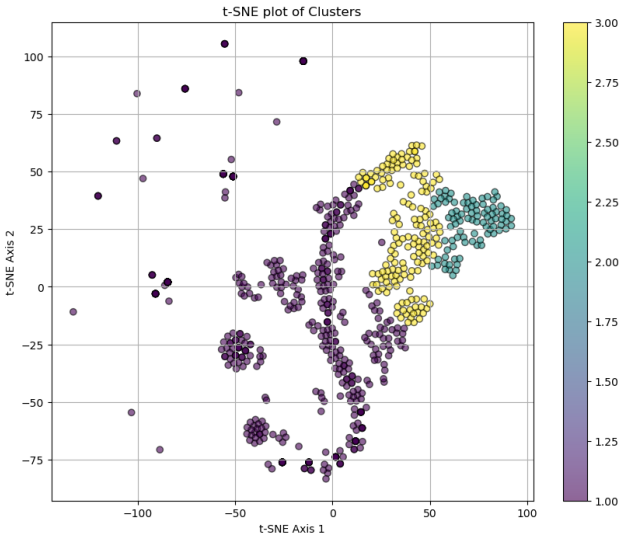


Figure 3.14: tSNE plot of cluster

3.3 | Spectral

- The provided Python script performs spectral graph analysis using NetworkX and NumPy libraries. Here's a detailed explanation of each part of the script:

  1. **Adjacency Matrix Creation:** The script starts by converting a graph **G** into its adjacency matrix **A** using NetworkX's **to_numpy_array** function. The adjacency matrix **A** represents the connections between nodes, where each entry **A[i, j]** is **1** if there is an edge between node **i** and node **j**, and **0** otherwise.
  2. **Degree Matrix Calculation:** Next, it computes the degree matrix **D**, which is a diagonal matrix where each diagonal entry **D[i, i]** is the sum of the **i**-th row of the adjacency matrix **A**. This represents the number of connections (degree) of each node.
  3. **Laplacian Matrix Computation:** The Laplacian matrix **L** is calculated as **D - A**. The Laplacian matrix is a fundamental matrix in graph theory used for various applications like

clustering, network analysis, and dimensionality reduction. It reflects the structure of the graph in terms of its connectivity.

4. **Eigenvalues and Eigenvectors:** The script then calculates the eigenvalues and eigenvectors of the Laplacian matrix using the **eigh** function, which is optimized for symmetric matrices like **L**. The eigenvalues are sorted in increasing order since the Laplacian matrix is positive semi-definite, meaning all eigenvalues are non-negative.
5. **Spectral Embedding:** To reduce the graph into a lower-dimensional space for tasks like visualization or clustering, the script selects a subset of the smallest eigenvalues (skipping the smallest trivial zero eigenvalue which represents the overall connectivity of the graph). It then uses the corresponding eigenvectors to form a **spectral_embedding**. The number of dimensions for the embedding is set to 64, but this can be adjusted based on the specific application or the graph's size and complexity.
6. **Output:** The script originally intended to print the smallest eigenvalues and the spectral embedding, though the actual printing of the embedding is commented out in the code provided. The user can uncomment these lines to see the embedding or modify the print statements to focus on different aspects of the output, such as specific eigenvectors or dimensions of the embedding.

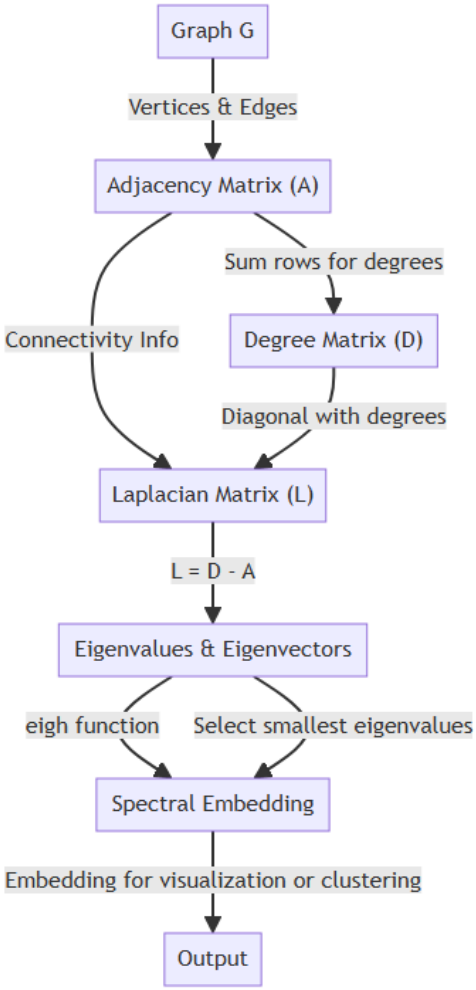
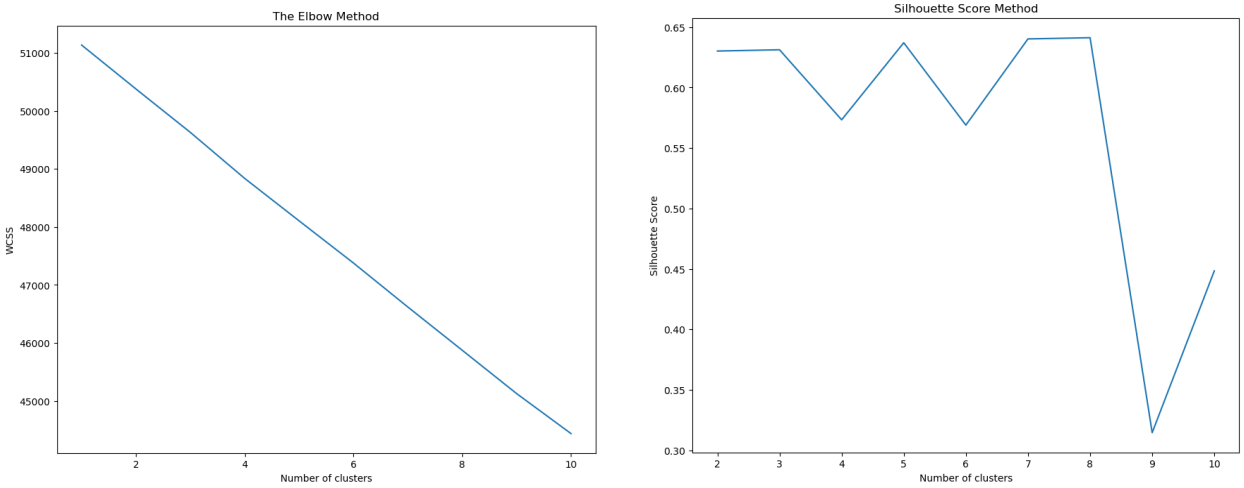


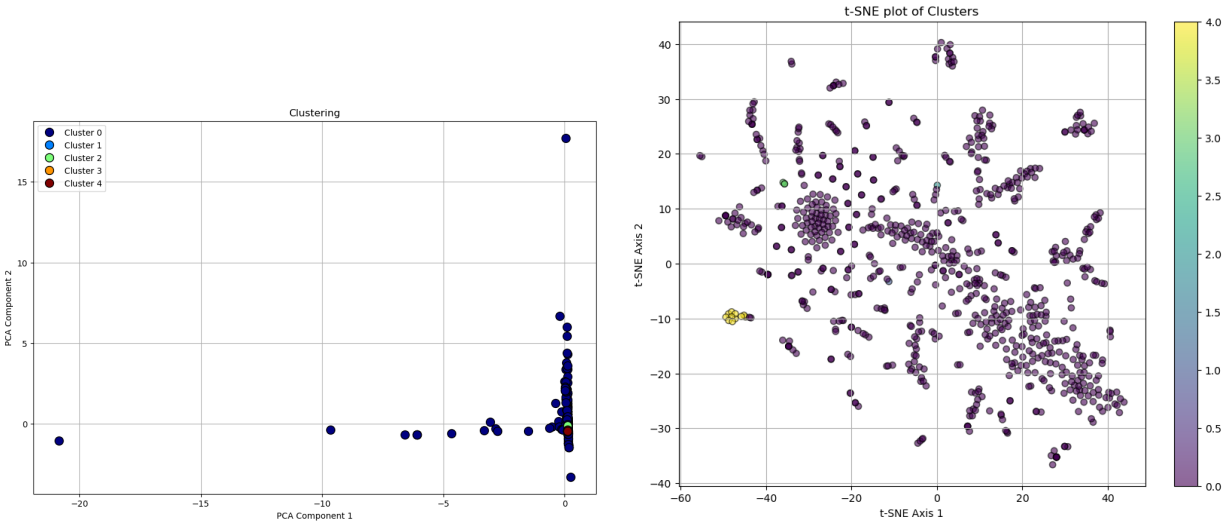
Figure 3.15: Spectral Flow Chart

3.3.1 | K Means



(a) Elbow Method **(b)** Silhouette Score Method

Figure 3.16: K Means



(a) Clustering **(b)** tSNE plot of Cluster

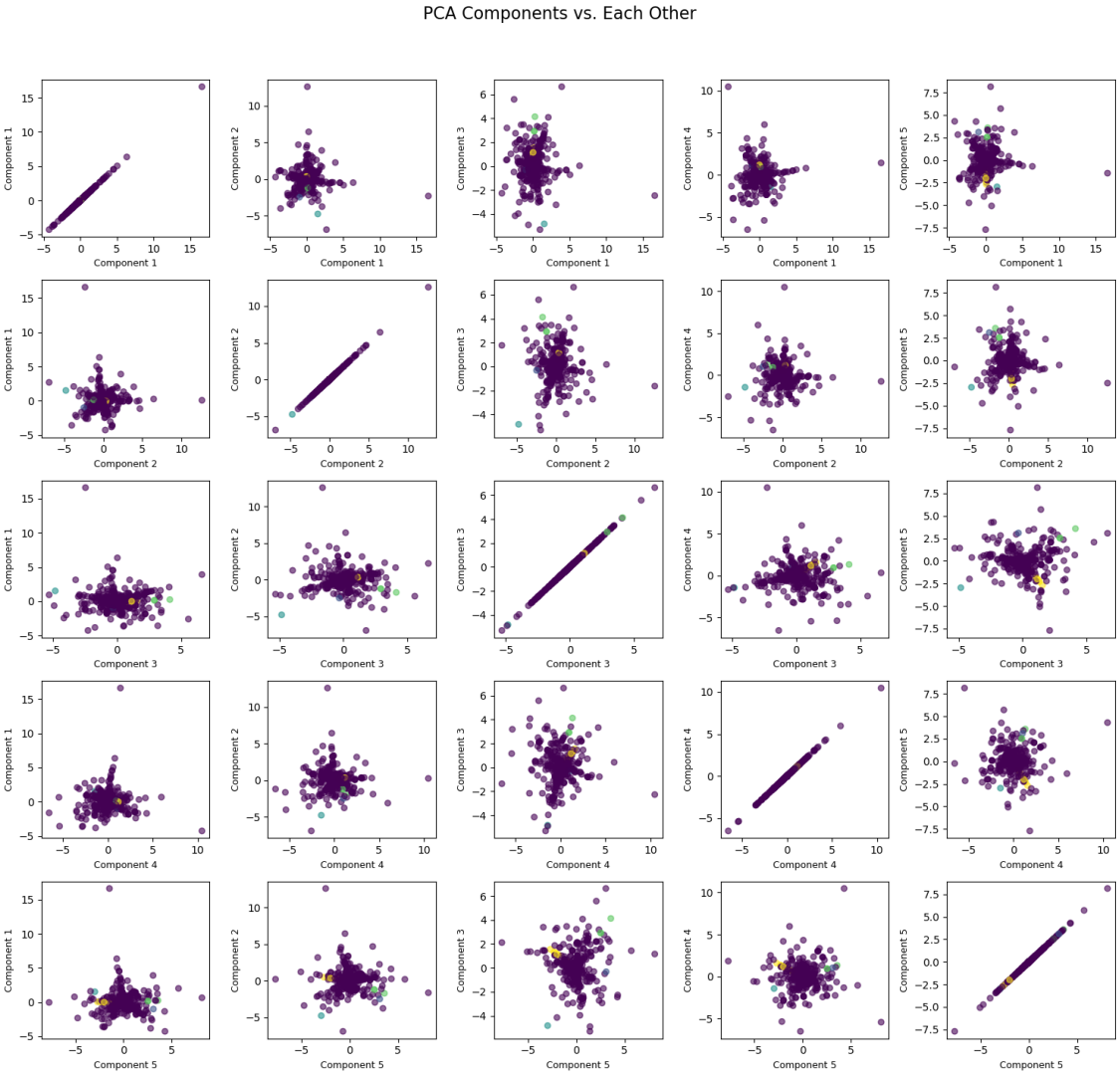
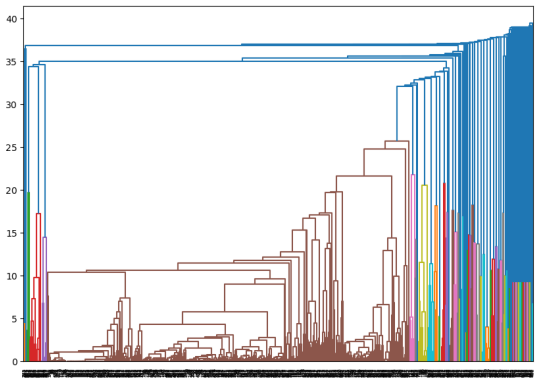
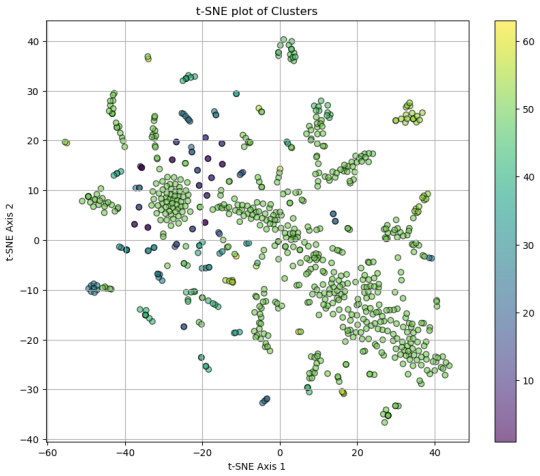


Figure 3.18: PCA Components vs Each other

3.3.2 | Hierarchical Clustering



(a) Hierarchical Clustering



(b) tSNE plot of Cluster

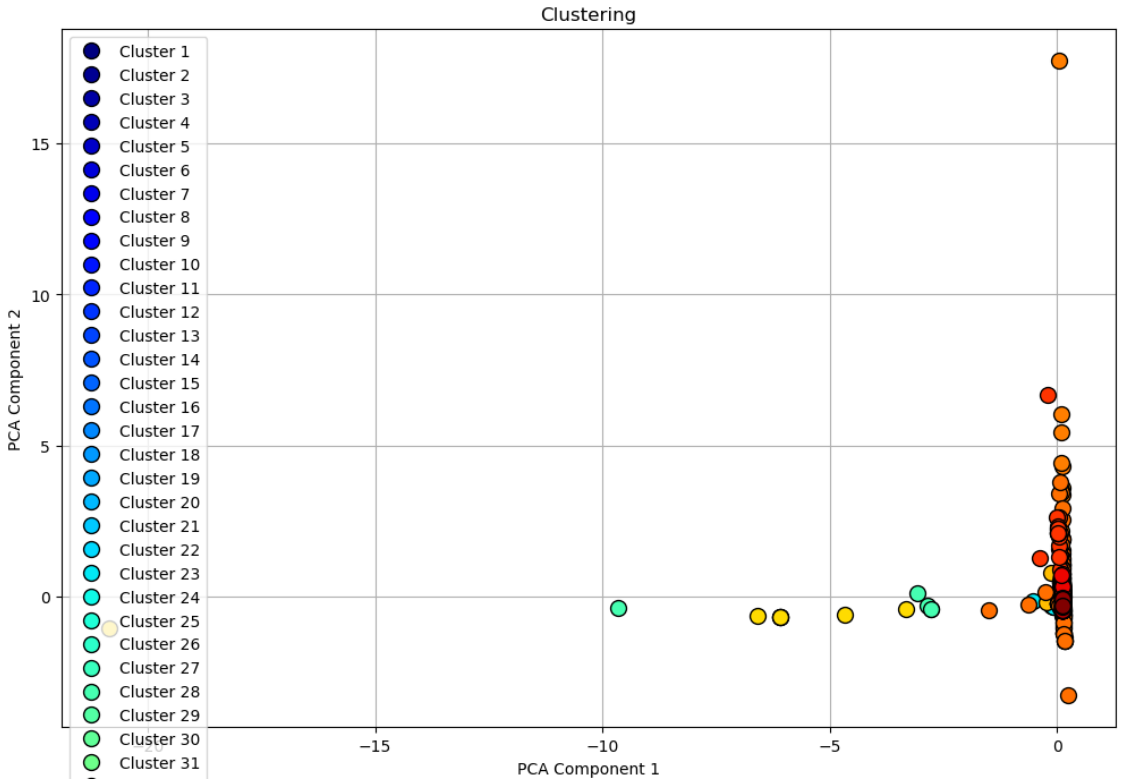


Figure 3.20: Cluster

3.4 | Pseudocode:

3.4.1 | Node2Vec Algorithm

Algorithm 1 Node2Vec Algorithm

procedure NODE2VEC($G, dimensions, walk\_length, num\_walks, p, q, workers$)
  Initialize graph G
  Set hyperparameters $dimensions, walk\_length, num\_walks, p, q$
  Initialize transition probabilities for each node in G
  for each node u in G **do**
    for each neighbor v of u **do**
      for each neighbor x of v **do**
        if $x = u$ **then**
          prob $\leftarrow$ weight$(v, x) \cdot (1/p)$
        else if x is a neighbor of u **then**
          prob $\leftarrow$ weight(v, x)
        else
          prob $\leftarrow$ weight$(v, x) \cdot (1/q)$
        end if
        Store prob in transition probabilities
      end for
      Normalize transition probabilities for v
    end for
  end for
  Generate walks based on transition probabilities
  for $i \leftarrow 1$ to num_walks **do**
    for each node u in G **do**
      $walk \leftarrow$ generate_walk$(u, walk_length)$
      Append $walk$ to walks
    end for
  end for
  Train Word2Vec model on walks
end procedure
function GENERATE_WALK($u, walk\_length$)
  $walk \leftarrow [u]$
  while length of $walk < walk\_length$ **do**
    $current \leftarrow walk$[end]
    Choose $next$ based on transition probabilities from $current$
    Append $next$ to $walk$
  end while
  return $walk$
end function=0

3.4.2 | GCN Algorithm

Algorithm 6 Forward pass of GraphAutoencoder

procedure GCNEncoder($in\_channels, out\_channels$)
  Create GCNEncoder object
  self.conv1 $\leftarrow$ GCNConv($in\_channels, 2 \times out\_channels$, cached $= True$)
  self.conv2 $\leftarrow$ GCNConv($2 \times out\_channels, out\_channels$, cached $= True$)
end procedure
procedure Forward($x, edge\_index$)
  $x \leftarrow$ ReLU(self.conv1($x, edge\_index$))
  return self.conv2($x, edge\_index$)
end procedure
procedure InnerProductDecoder($z, edge\_index, sigmoid = True$)
  $value \leftarrow$ sum($z[edge\_index[0]] \times z[edge\_index[1]]$, axis $= 1$)
  if $sigmoid$ **then**
    return sigmoid($value$)
  else
    return $value$
  end if
end procedure
procedure GraphAutoencoder($in\_channels, out\_channels$)
  Create GraphAutoencoder object
  self.encoder $\leftarrow$ GCNEncoder($in\_channels, out\_channels$)
  self.decoder $\leftarrow$ InnerProductDecoder()
end procedure
procedure Forward($x, edge\_index$)
  $z \leftarrow$ self.encoder($x, edge\_index$)
  return self.decoder($z, edge\_index$)
end procedure=0

3.4.3 | Spectral Algorithm

Algorithm 7 Spectral Embedding Algorithm

Input: Graph G
Output: Spectral embedding matrix S
// Step 1: Adjacency matrix
$A \leftarrow$ nx.to_numpy_array(G)
// Step 2: Degree matrix
$D \leftarrow$ np.diag(A.sum($axis = 1$))
// Step 3: Laplacian matrix
$L \leftarrow D - A$
// Step 4: Eigenvalues and eigenvectors
$eigenvalues, eigenvectors \leftarrow$ np.linalg.eigh(L)
// Print smallest eigenvalues
Print "Smallest eigenvalues:", **len**($eigenvalues$)
// Sorting eigenvalues and selecting dimensions for embedding
$num\_dimensions \leftarrow 64$
$smallest\_eigenvalues \leftarrow$ np.argsort($eigenvalues$)$[1 : num\_dimensions + 1]$
// Generating the spectral embedding
$spectral\_embedding \leftarrow eigenvectors[:, smallest\_eigenvalues]$
Return $spectral\_embedding$ =0

4 | Results

Algorithm	Methods	Metric		
		Silhouette Score	Calinski-Harabasz Index	Davies-Bouldin Index
Node2Vec	Clustering	−0.310846	1.016246	3.628428
	K Means	0.189120	12.580262	2.912012
	Hierarchical Clustering	0.157056	12.977587	4.064141
GCN	K Means	−0.049281	3.503435	11.721809
	Hierarchical Clustering	0.165871	4.690222	11.220474
Spectral	K Means	0.028885	2.791442	1.414690
	Hierarchical Clustering	−0.287804	1.557824	2.704128

4.1 | Observation

- **Silhouette Score:** Node2Vec has the lowest Silhouette Score (-0.310846), which indicates poor cluster separation. Spectral Clustering (0.165871) and Hierarchical Clustering (0.157056) have the highest Silhouette Scores, indicating better cluster separation.
- **Calinski-Harabasz Index:** Node2Vec also has the lowest Calinski-Harabasz Index (1.016246), indicating poor separation between clusters. In contrast, GCN (Hierarchical Clustering) has the highest Calinski-Harabasz Index (12.977587), indicating good separation between clusters.
- **Davies-Bouldin Index:** A lower Davies-Bouldin Index indicates better cluster separation. Here, Node2Vec again has the highest Davies-Bouldin Index (3.628428), whereas Spectral Clustering (1.414690) has the lowest value, indicating the best cluster separation among the four algorithms.

5 | References

[1] Daniel de Roux, Boris R. Pérez, Andres Moreno, Pilar Villamil, César Alfonso Figueroa. (2018). Tax Fraud Detection for Under-Reporting Declarations Using an Unsupervised Machine Learning Approach. DOI: 10.1145/3219819.3219878