P2 Practical Course

Semester Project
Data augmentation techniques applied to SpectralMix Algorithm
Based on the work of Ylli Sadikaj, Yllka Velaj, Sahar Behzadi, Claudia Plant

Name: Zoheir El Houari
Matriculation Number: a12044027

Abstract:
Graph-based data representations have gained significant attention due to their ability to capture complex relationships in various domains. Spectral clustering, a popular technique, leverages the eigenvalues and eigenvectors of graph Laplacians to partition nodes into clusters. However, traditional spectral clustering methods often overlook node attributes, which can provide valuable information for improving clustering performance.

In this project, we explore spectral clustering for attributed multi-relational graphs. The goal is to enhance the clustering accuracy by incorporating both structural information and node attributes. Specifically, we investigate two graph data augmentation techniques: feature-oriented augmentation and structure-oriented augmentation. By integrating these methods, we aim to achieve more robust and interpretable clustering results.

1 - Introduction:
Spectral clustering has become a pivotal method for analyzing complex networks and attributed multi-relational graphs. The SpectralMix algorithm, a recent advancement in this domain, has shown promising results by integrating spectral methods with feature mixing to improve clustering accuracy. However, the robustness and performance of such algorithms can be further enhanced through effective data augmentation techniques. This report explores the implementation of both feature-oriented and structure-oriented augmentation techniques to evaluate their impact on the clustering accuracy of the SpectralMix algorithm.

2 - Problem Statment and Motivation:
The primary objective of this study is to investigate whether commonly used data augmentation techniques can improve the clustering accuracy of the SpectralMix algorithm when applied to attributed multi-relational graphs. Given the intricate nature of these graphs, which include both structural and attribute-based information, it is crucial to employ augmentation methods that can potentially enhance the algorithm's ability to generalize and accurately identify clusters.

3 - Proposed methodology:

In this project, we implemented various augmentation techniques categorized into feature-oriented and structure-oriented methods. These techniques were integrated into the SpectralMix algorithm to assess their impact on clustering accuracy.

3.1 - Feature-Oriented Augmentation Techniques:

Feature-oriented augmentation techniques focus on manipulating node attributes to introduce variability and enhance model robustness. Generally, given an input graph ***G = (A, X)***, a feature-oriented graph data augmentation operation that focuses on performing transformations on the node feature matrix ***X***.

3.1.1 - Feature Masking Augmentation:

Feature Masking involves randomly masking a subset of the node features to create augmented data. This helps in making the model robust against missing or noisy features.

Mathematically, given a feature matrix $X \in \mathbb{R}^{n \times d}$, where ***n*** is the number of nodes and ***d*** is the feature dimension, we can define a masking matrix $M \in \{0,1\}^{n \times d}$ where each element is sampled from a Bernoulli distribution with probability ***p***:

$$M_{i,j} \sim Bernoulli(p)$$

The augmented feature matrix $X^{'}$ can be obtained as:
$X^{'} = X \odot M$ Where $\odot$ donates the element-wise multiplication.
The new augmented graph would be defined as $G = \left(A, X^{'}\right)$.

```python
def feature_masking_augmentation(self, node_attributes):
    print("########## feature masking Augmentation ##########")
    # Copy the node_attributes for augmentation
    masked_features = np.copy(node_attributes)
    # generate a mask from bernoulli distribution
    success_probablity = 0.2
    mask = np.random.choice([0, 1], size=node_attributes.shape, p=[success_probablity, 1-success_probablity])
    # perform the hadamard product of the node_attributes and the mask
    masked_features = np.multiply(masked_features, mask)
    print("#### node_attributes were masked ####")
    return  masked_features
```

Figure 1: Feature Masking Implementation

3.1.2 - Feature Shuffling Augmentation: Feature Shuffling involves randomly changing the contextual information through switching rows and columns in the feature

matrix, to disrupt the original feature arrangement while retaining the overall feature distribution.

Mathematically, given a feature matrix $X \in \mathbb{R}^{n \times d}$, where ***n*** is the number of nodes and ***d*** is the feature dimension, the augmented feature matrix $X^{'}$ can be obtained as:

$X^{'} = P_r X P_c$ Where P_r and P_c are row-wise permutation matrix and column-wise permutation matrix, respectively. have exactly one entry of 1 in each row and each column and 0 elsewhere.

```python
def feature_shuffling_augmentation(self, node_attributes):
    print("########## feature shuffling Augmentation ##########")
    # Feature shuffling augmentation by randomly shuffling the rows and columns of the node_attributes matrix
    mixed_features = np.copy(node_attributes)
    # genarete PR which is the row-wise permutation matrix
    PR = np.random.permutation(np.eye(node_attributes.shape[0]))
    # generate PC which is the column-wise permutation matrix
    PC = np.random.permutation(np.eye(node_attributes.shape[1]))
    # Apply the row-wise permutation to the node_attributes
    mixed_features = np.dot(np.dot(PR, node_attributes),PC)
    print("#### node_attributes were shuffled ####")

    return  mixed_features
```

Figure 2: Feature Shuffling Implementation

3.1.3 Feature Propagation Augmentation:

Feature propagation uses a personalized PageRank (PPR) diffusion matrix to incorporate topological information into node attributes, enhancing structural awareness.
However, this method doesn't work with the current datasets, since the node features are only categorical.

```python
def feature_propagation_augmentation(self, node_attributes, adj):
    print("########## Feature Propagation Augmentation ##########")
    # Feature propagation augmentation by encoding topological information into the node_attributes
    augmented_features = np.copy(node_attributes)

    # Compute the transition matrix T from the adjacency matrix A
    degree_matrix = np.diag(np.sum(adj, axis=1))
    inv_degree_matrix = np.linalg.inv(degree_matrix)
    transition_matrix = np.dot(inv_degree_matrix, adj)

    # Compute the personalized PageRank diffusion matrix
    alpha = 0.85
    identity_matrix = np.eye(adj.shape[0])
    ppr_matrix = alpha * np.linalg.inv(identity_matrix - (1 - alpha) * transition_matrix)

    # Propagate the node attributes using the personalized PageRank matrix
    augmented_features = np.dot(ppr_matrix, augmented_features)
    print("#### Features were propagated ####")

    return augmented_features
```

Figure 3: Feature Propagation Implementation

3.2 - Structure-Oriented Augmentation Techniques:

Structure-oriented augmentation techniques modify the graph structure to introduce variability in the connections between nodes. Generally, given an input graph ***G = (A, X).*** A structure-oriented Graph data augmentation operation focuses on augmenting the adjacency matrix ***A*** of the input graph. We summarize the representative ones as follows:

3.2.1 Edge Perturbation Augmentation:

Edge Perturbation Augmentation modifies the edges in the graph by adding or removing edges based on a perturbation rate. This method introduces variability in the graph structure, which can help the model learn to be less sensitive to minor structural changes.

Mathematically, given an adjacency matrix (or a list of adjacency matrices for multi-relational graphs). Edge perturbation keeps the original node order and rewrites a part of the entries in the given adjacency matrices, which can be defined as follows:

$A^{'} = A \oplus C$ where ***C*** is the corruption matrix and $\oplus$ denotes the XOR (exclusive OR) operation. Commonly, the corruption matrix C is for example obtained by sampling from a Bernoulli distribution $C_{ij} \sim Bernoulli(p)$

```python
def edge_perturbation_augmentation(self, adj_matrices,):
    print("########## Edge Perturbation Augmentation ##########")
    adj_augmented_list = []

    for  adj in adj_matrices:
        # Copy the current adjacency matrix
        adj_augmented = np.copy(adj)

        # Generate a mask for adding or removing edges
        num_nodes = adj.shape[0]
        edge_mask = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.9, 0.1])

        # Add or remove edges based on the mask (XOR operation)
        adj_augmented = np.logical_or(adj_augmented, edge_mask).astype(int)

        # Add the augmented adjacency matrix to the list
        adj_augmented_list.append(adj_augmented)

    # Convert list of augmented adjacency matrices to numpy array
    adj_augmented_matrices = np.array(adj_augmented_list)

    return adj_augmented_matrices
```

Figure 4: Edge Perturbation Implementation

3.2.2 Graph Rewiring Augmentation:

Graph Rewiring Augmentation adjusts the graph structure by selectively adding and removing edges based on attribute similarity among nodes. This technique leverages node attributes to ensure that the graph's structure reflects the underlying similarities between nodes.

Mathematically, given an adjacency matrix ***A*** and node attributes ***X***, graph rewiring augmentation modifies the graph by iteratively removing the least similar edges and adding the most similar non-edges. The similarity between node attributes is measured using cosine similarity. Formally, the cosine similarity between two attribute vectors x_i and x_j is defined as follows:

$$cosinesimilarity(x_i, x_j) = \frac{x_i \cdot x_j}{\| x_i \| \cdot \| x_j \|}$$

The graph rewiring process can be described step-by-step as follows:

1. **Identify edges to remove**:
    - Compute the cosine similarity for each existing edge in the adjacency matrix.
    - Sort the edges in ascending order of similarity.
    - Remove the edge with the lowest similarity.
2. **Identify pairs to add an edge**:
    - Compute the cosine similarity for each pair of nodes that do not have an edge between them.
    - Sort these non-edges in descending order of similarity.
    - Add an edge between the pair with the highest similarity.

This process is repeated for a specified number of iterations *(num_rewires)*, to gradually rewire the graph.

```python
def cosine_attribute_similarity(self,attr1, attr2):
        """Compute cosine similarity between two attribute vectors."""
        return np.dot(attr1, attr2) / (np.linalg.norm(attr1) * np.linalg.norm(attr2))


    # Class GraphRewiring was used instead of the following function for better computational efficiency
    def graph_rewiring_augmentation(self,adj_matrix, node_attributes, num_rewires):
        """Rewire the adjacency matrix based on attribute similarity."""
        n = adj_matrix.shape[0]
        print("########## Graph Rewiring Augmentation #########")
        for _ in range(num_rewires):
            # Identify edges to remove
            edges = np.transpose(np.nonzero(adj_matrix))
            edge_similarities = [(i, j, self.cosine_attribute_similarity(node_attributes[i], node_attributes[j])) for i, j in edges]
            edge_similarities.sort(key=lambda x: x[2])  # Sort by similarity (ascending)

            # Remove the least similar edge
            if edge_similarities:
                i, j, _ = edge_similarities[0]
                adj_matrix[i, j] = adj_matrix[j, i] = 0

            # Identify pairs to add an edge
            non_edges = [(i, j) for i in range(n) for j in range(i + 1, n) if adj_matrix[i, j] == 0]
            non_edge_similarities = [(i, j, self.cosine_attribute_similarity(node_attributes[i], node_attributes[j])) for i, j in non_edges]
            non_edge_similarities.sort(key=lambda x: x[2], reverse=True)  # Sort by similarity (descending)

            # Add the most similar non-edge
            if non_edge_similarities:
                i, j, _ = non_edge_similarities[0]
                adj_matrix[i, j] = adj_matrix[j, i] = 1
        print("#### Graph was rewired ####")
        return adj_matrix
```

Figure 5: Graph Rewiring Implementation

3.2.3 Graph Diffusion Augmentation:

Graph Diffusion Augmentation enhances the graph structure by propagating information across the graph based on the proximities between nodes. This technique uses diffusion processes to transform the adjacency matrix, adepting at capturing complex relationships and influences among nodes.

Given an adjacency matrix ***A***, graph diffusion augmentation modifies the graph by applying specific diffusion processes. Two widely used diffusion methods are Personalized PageRank (PPR) and Heat Kernel, each controlled by a distinct parameter.

1. Personalized PageRank (PPR)

Personalized PageRank diffuses information with a bias towards the starting node, retaining a preference for specific nodes. It is mathematically defined as:

$$A^{PPR} = \alpha(I - (1 - \alpha)T)^{-1}$$

where:

- ***α*** is the teleport probability,
- ***I*** is the identity matrix,
- ***T*** is the transition matrix derived from ***A***.

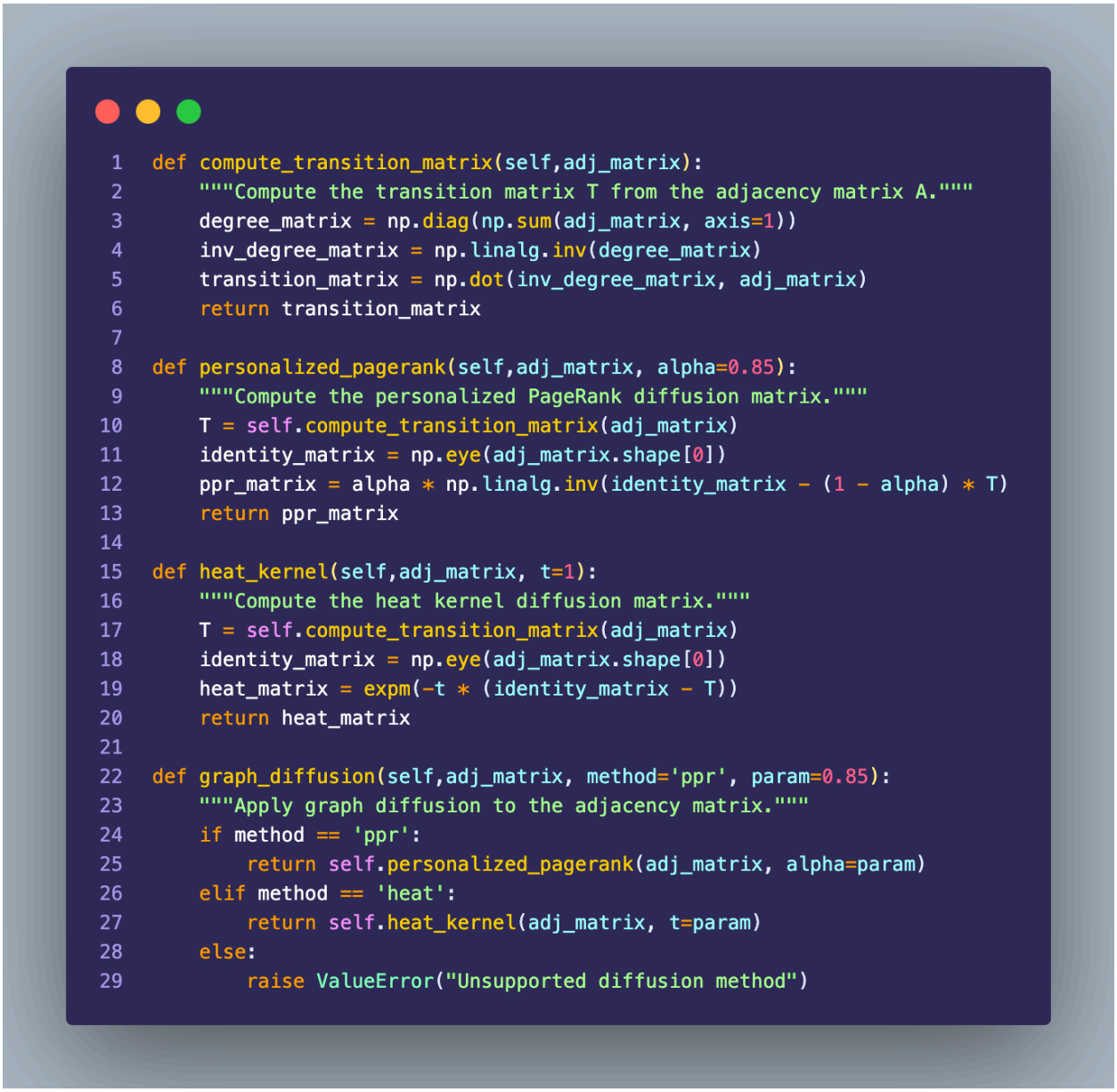
2. Heat Kernel

The Heat Kernel method diffuses information similarly to heat dispersing over time. It is represented as:

$$A^{heat} = e^{-(I-T)t}$$

where:

- ***t*** is the diffusion time,
- ***exp*** denotes the matrix exponential,
- ***I*** is the identity matrix,
- ***T*** is the transition matrix derived from ***A***.



```python
def compute_transition_matrix(self,adj_matrix):
    """Compute the transition matrix T from the adjacency matrix A."""
    degree_matrix = np.diag(np.sum(adj_matrix, axis=1))
    inv_degree_matrix = np.linalg.inv(degree_matrix)
    transition_matrix = np.dot(inv_degree_matrix, adj_matrix)
    return transition_matrix

def personalized_pagerank(self,adj_matrix, alpha=0.85):
    """Compute the personalized PageRank diffusion matrix."""
    T = self.compute_transition_matrix(adj_matrix)
    identity_matrix = np.eye(adj_matrix.shape[0])
    ppr_matrix = alpha * np.linalg.inv(identity_matrix - (1 - alpha) * T)
    return ppr_matrix

def heat_kernel(self,adj_matrix, t=1):
    """Compute the heat kernel diffusion matrix."""
    T = self.compute_transition_matrix(adj_matrix)
    identity_matrix = np.eye(adj_matrix.shape[0])
    heat_matrix = expm(-t * (identity_matrix - T))
    return heat_matrix

def graph_diffusion(self,adj_matrix, method='ppr', param=0.85):
    """Apply graph diffusion to the adjacency matrix."""
    if method == 'ppr':
        return self.personalized_pagerank(adj_matrix, alpha=param)
    elif method == 'heat':
        return self.heat_kernel(adj_matrix, t=param)
    else:
        raise ValueError("Unsupported diffusion method")
```

Figure 6: Graph Diffusion Implementation

3.2.4 Graph Sampling Augmentation:

Graph Sampling Augmentation selects a subset of nodes or edges from the original graph to create a new, smaller graph. The augmented graph is obtained by using edge-based sampling (can be other methods ex: traversal-based sampling).

Mathematically, graph sampling can be represented as:

$$G^{'} = \{A^{'}, X^{'}\} = \{A[idx, idx], X[idx, :]\}$$

where ***idx*** is a list of indexes to select the given elements (i.e., rows and columns) from ***A*** and ***X,*** however, I must mention that in the implementation. I did a slight modification which improved performance, so instead of sampling from the feature matrix ***X***, I instead used the entirety of the feature matrix.

In general, the goal of graph sampling is to find augmented graph instances from the input graphs that best and their underlying linkages

```python
def graph_sampling_augmentation(self,adj_matrices, features, sample_rate=0.5):
        random.seed(0)

        num_nodes = features.shape[0]
        sampled_adj_matrices = []

        for adj in adj_matrices:
            # Flatten the adjacency matrix and get the indices of the edges
            edge_indices = np.transpose(np.nonzero(adj))
            # Determine the number of edges to sample
            num_edges = edge_indices.shape[0]
            num_sampled_edges = int(sample_rate * num_edges)

            # Sample the edges
            sampled_indices = np.random.choice(num_edges, num_sampled_edges, replace=False)
            sampled_edge_indices = edge_indices[sampled_indices]

            # Create a new adjacency matrix for the sampled edges
            sampled_adj = np.zeros_like(adj)
            sampled_adj[sampled_edge_indices[:, 0], sampled_edge_indices[:, 1]] = 1
            sampled_adj[sampled_edge_indices[:, 1], sampled_edge_indices[:, 0]] = 1
            sampled_adj_matrices.append(sampled_adj)

        # Determine the nodes that are included in the sampled edges
        sampled_nodes = np.unique(sampled_edge_indices.flatten())
        self.node_attr_augmented = features

        # Adjust the sampled adjacency matrices to reflect the new node indices
        self.adj_augmented = [adj[np.ix_(sampled_nodes, sampled_nodes)] for adj in sampled_adj_matrices]
         # update number of nodes
        self.nodes = (sampled_nodes.shape[0])
        self.atts = len(self.node_attr_augmented[0])
        self.test_ids = self.test_ids[self.test_ids < self.nodes]

        return self.adj_augmented, self.node_attr_augmented
```

Figure 7: Graph Sampling Implementation

3.2.5 Node Dropping Augmentation:

Node Dropping Augmentation removes a subset of nodes and their associated edges from the graph. Specifically, a set of nodes $V^{'} = \left\{ v_i \in V \right\}$ will be dropped from the input graph, together with their associated edges $E^{'} = \left\{ e_i \in E \right\}$.
Mathematically, this augmentation that's to be applied to the adjacency matrix can be formulated as follows: $A^{'} = \{ V / V^{'}, E / E^{'} \}$.

Note that for attributed graphs, the corresponding node features will also be dropped simultaneously.

```python
def node_dropping_augmentation(self,adj_matrices, features, drop_rate=0.2):
        num_nodes = adj_matrices[0].shape[0]
        num_drop_nodes = int(num_nodes * drop_rate)

        # Randomly select nodes to drop
        drop_nodes = np.random.choice(num_nodes, num_drop_nodes, replace=False)

        # Create mask to keep nodes that are not dropped
        keep_nodes = np.setdiff1d(np.arange(num_nodes), drop_nodes)

        # Create new adjacency matrices and feature matrix with the remaining nodes
        adj_augmented = []
        for adj in adj_matrices:
            dropped_adj_matrix = adj[np.ix_(keep_nodes, keep_nodes)]
            adj_augmented.append(dropped_adj_matrix)

        node_attr_augmented = features[keep_nodes, :]

        # Update instance variables
        self.adj_augmented = adj_augmented
        self.node_attr_augmented = node_attr_augmented

        # Update number of nodes
        self.nodes = adj_augmented[0].shape[0]
        self.atts = node_attr_augmented.shape[1]

        # Update the test_ids to reflect the new adjacency matrix
        self.test_ids = self.test_ids[self.test_ids < self.nodes]
        # self.val_ids = self.val_ids[self.val_ids < self.nodes]
        # self.train_ids = self.train_ids[self.train_ids < self.nodes]

        return self.adj_augmented, self.node_attr_augmented
```

Figure 8: Node Dropping Implementation

3.2.6 Node Insertion Augmentation:

Node Insertion Augmentation inserts new virtual nodes in order to improve the message-passing or connectivity on the input graph. Node insertion adds an extra set of nodes $V' = \{v_i\}$ and a set of edges $\varepsilon' = \{e_i\}$ between V' and V to the original node set V and edges ε.

Mathematically, this augmentation that's to be applied to the adjacency matrix can be formulated as follows: $A' = \{V \cup V', \varepsilon \cup \varepsilon'\}$.

Since node insertion also requires adding additional edges in the new graph, this Graph data augmentation operation is highly related to graph rewiring. Note that for attributed graphs, the corresponding node features also need to be initialized. In the implementation, The virtual node features are initialized by copying the features from a random existing node.

```python
def node_insertion_augmentation(self, adj_matrices, features, num_virtual_nodes=50):
        num_nodes = adj_matrices[0].shape[0]
        feature_dim = features.shape[1]

        # Initialize new adjacency matrices with increased dimensions
        new_adj_matrices = typed.List()
        for adj in adj_matrices:
            new_adj_matrix = np.zeros((num_nodes + num_virtual_nodes, num_nodes + num_virtual_nodes))
            new_adj_matrix[:num_nodes, :num_nodes] = adj
            new_adj_matrices.append(new_adj_matrix)

        # Initialize new feature matrix with increased dimensions
        new_features = np.zeros((num_nodes + num_virtual_nodes, feature_dim))
        new_features[:num_nodes, :] = features

        for i in range(num_virtual_nodes):
            virtual_node_index = num_nodes + i
            connected_nodes = np.random.choice(num_nodes, size=int(np.sqrt(num_nodes)), replace=False)

            for new_adj_matrix in new_adj_matrices:
                new_adj_matrix[virtual_node_index, connected_nodes] = 1
                new_adj_matrix[connected_nodes, virtual_node_index] = 1
            # # Initialize virtual node features by copying the features from a random existing node
            random_node_index = np.random.choice(num_nodes)
            new_features[virtual_node_index, :] = features[random_node_index, :]
        # Update number of nodes
        self.nodes = new_adj_matrix[0].shape[0]
        self.atts = new_features.shape[1]
        # Update the test_ids to reflect the new adjacency matrix
        self.test_ids = self.test_ids[self.test_ids < self.nodes]

        return new_adj_matrices, new_features
```

Figure 9: Node Insertion Implementation

4. Experiments and Results:

In this study, we aimed to improve the clustering accuracy of the SpectralMix algorithm by exploring various data augmentation techniques. we evaluated the performance of the SpectralMix algorithm on two multi-relational graph datasets, ACM and IMDB. Additionally, we generated a synthetic dataset to further test the algorithm's robustness under a controlled environment.
The augmentation methods applied included feature masking, feature shuffling, feature propagation, edge perturbation, graph diffusion (heat and PPR), graph rewiring, graph sampling, node dropping, and node insertion. The clustering performance was measured using the Adjusted Rand Index (ARI) and Normalized Mutual Information (NMI).

Experimental Setup

1. Datasets:
    - ACM: A dataset with academic paper metadata.
    - IMDB: A dataset containing movie information.
    - RANDOM Synthetic Dataset: To create a controlled environment for testing, we generated a random graph using the ***generate_synthetic_dataset*** function.
2. Augmentation Techniques:
    - No Augmentation: The baseline without any augmentation.
    - Feature Masking: Randomly masking a portion of node features.
    - Feature Shuffling: Randomly shuffling the features of nodes.
    - Feature Propagation: uses a (PPR) diffusion matrix to incorporate topological information into node attributes (Implemented by can’t be applied to the current datasets)
    - Edge Perturbation: Randomly adding or removing edges.
    - Graph Diffusion (Heat): Applying heat diffusion to smooth node features.
    - Graph Diffusion (PPR): Using Personalized PageRank for feature smoothing.
    - Graph Rewiring: Rewiring edges to enhance graph connectivity.
    - Graph sampling: Sampling subset that best preserves desired properties by keeping a portion of nodes
    - Node Dropping: Randomly drops a subset of nodes based on a perturbation rate
    - Node Insertion: Increases the input graph dimensions by adding new virtual nodes.
3. Synthetic Data Generation:
    - The ***generate_synthetic_dataset()*** function was implemented to create a synthetic dataset for additional testing. This function takes inputs (num_nodes=2000, num_features=1200, num_relations=2, p=0.75) to generates random adjacency matrices from a Bernoulli distribution and categorical node attributes, ensuring the generated data can comprehensively test the SpectralMix algorithm's performance.

5. Results:

This comprehensive evaluation of various data augmentation methods was applied to the ACM and IMDB datasets and a synthetic dataset, ensuring robustness in the performance assessment of the SpectralMix algorithm. The summarized results provide a clear insight into the effectiveness of each augmentation method on these datasets.

The algorithm was executed five times; the optimal results are presented in the table below.

Datasets	***ACM***		**IMDB**		**Synthetic Dataset**	
	NMI	*ARI*	*NMI*	*ARI*	*NMI*	*ARI*
Base SpectralMix	***0.71***	***0.77***	***0.16***	***0.16***	***0.0440***	***-0.0011***
Feature-Oriented Methods Applied on SpectralMix						
Feature Masking	*0.68*	*0.74*	*0.15*	*0.15*	*0.0414*	*-0.0026*
Feature Shuffling	*0.70*	*0.77*	*0.16*	*0.16*	*0.0431*	*-0.0020*
Structure-Oriented Methods Applied on SpectralMix						
Graph Rewiring	*0.70*	*0.76*	*0.16*	*0.16*	***0.0466***	***0.0002***
Graph Diffusion ppr	*0.68*	*0.75*	*0.15*	*0.15*	***0.0498***	***0.0014***
Graph Diffusion Heat	*0.68*	*0.75*	*0.15*	*0.15*	*0.0485*	*-0.0004*
Edge Perturbation	*0.67*	*0.74*	*0.15*	*0.15*	*0.0457*	*-0.0007*
Node Insertion	*0.70*	*0.77*	*0.16*	*0.16*	*0.0452*	*-0.0011*
Node Dropping	*0.64*	*0.70*	*0.01*	*0.01*	*0.0455*	*-0.0010*
Graph Sampling	***0.75***	***0.81***	*0.15*	*0.15*	*0.0460*	*-0.0002*

6. Analysis:

From the results, it is clear that the choice of data augmentation method has a significant impact on the performance of the SpectralMix model.

1. **Feature-Oriented Methods**:
    - **Feature Masking**: Shows a slight decrease in performance compared to no augmentation.

- **Feature Shuffling**: Maintains performance on the IMDB dataset and shows a minimal decrease on the ACM dataset.

2. **Structure-Oriented Methods**:
   - **Graph Rewiring**: Maintains performance on both datasets, indicating robustness to minor structural changes.
   - **Graph Diffusion (PPR and Heat)**: Both methods show similar performance, slightly lower than no augmentation.
   - **Edge Perturbation**: Results in a slight decrease in performance, likely due to the introduction of noise.
   - **Node Insertion**: Maintains performance, suggesting that adding virtual nodes can be beneficial or neutral.
   - **Node Dropping**: Shows a significant decrease in performance on the IMDB dataset, indicating sensitivity to missing nodes.
   - **Graph Sampling**: Performs the best among all methods, improving performance on the ACM dataset and maintaining it on the IMDB dataset. This suggests that sampling can effectively preserve essential structural information while reducing complexity.
3. **Synthetic dataset:**
   - Provides a benchmark to evaluate the effectiveness of these data augmentation methods in a controlled environment.

7. Synthetic Dataset Interpretation:

The synthetic dataset provides a valuable benchmark to evaluate the performance of the SpectralMix algorithm under controlled conditions. Here is a detailed interpretation of the results:

1. **Base SpectralMix Performance:**

The base SpectralMix algorithm on the synthetic dataset shows relatively low **NMI** and **ARI** values. This is expected since the synthetic data might not capture the complex structures that are present in real-world datasets like ACM and IMDB.

2. **Feature-Oriented Methods:**
   - **Feature Masking:** It shows a slight decrease in both **NMI** and **ARI** compared to the base SpectralMix. This suggests that masking features introduce some noise, which disrupts the model's ability to cluster effectively.

    - **Feature Shuffling:** Similar to feature masking, feature shuffling results in a minor decrease in **NMI** and **ARI**. This indicates that randomizing features can slightly harm the performance by disrupting the feature relationships.
3. **Structure-Oriented Methods:**
    - **Graph Rewiring:** It shows slightly better performance than the base model, suggesting that small structural changes can positively affect the model's clustering ability.
    - **Graph Diffusion (PPR and Heat):** These methods show the best performance among all methods, indicating their potential to enhance structure by diffusing information more effectively.
    - **Edge Perturbation and Node Insertion:** Both methods maintain or slightly improve performance, showing robustness to structural modifications.
    - **Node Dropping:** This method maintains performance with a slight improvement in both **NMI** and **ARI**, suggesting the synthetic dataset is somewhat resilient to missing nodes.
    - **Graph Sampling:** This method maintains performance with a slight improvement in both **NMI** and **ARI**, highlighting its effectiveness in simplifying the graph while preserving essential information.

8. Conclusion:

Data augmentation methods can have varying effects on the performance of SpectralMix, results underscore the importance of selecting appropriate augmentation techniques based on the dataset characteristics and the task requirements. While some augmentation methods, such as graph sampling and diffusion, enhance performance by preserving or propagating structural information, others, like feature masking and shuffling, introduce noise that can slightly degrade performance. These findings suggest that a nuanced approach to data augmentation, potentially combining multiple augmentation methods or developing dataset-specific augmentation strategies, could achieve a better overall performance

Complexity of Attributed Multi-Relational Graphs:

- The inherent complexity of attributed multi-relational graphs, such as those in the ACM and IMDB datasets, may require more sophisticated augmentation strategies. These strategies need to be specifically tailored to the unique characteristics of the dataset and the clustering objectives. The diverse nature of nodes and relationships in these graphs means that simple augmentations may not sufficiently capture the underlying structure and attribute information.

Augmentation Suitability:

- Not all augmentation techniques are universally suitable for every type of graph data. Their effectiveness can vary significantly based on the graph's structure and the

attributes involved. For instance, while graph diffusion and graph sampling maintained or slightly improved performance, methods like node dropping resulted in a significant performance drop, particularly on the IMDB dataset. This variability underscores the importance of selecting appropriate augmentation techniques that align with the specific properties of the graph.

SpectralMix-Specific Factors:

- The SpectralMix algorithm, despite its robustness, may not benefit from certain types of augmentations. This suggests a need for a deeper integration or modification of the algorithm to leverage augmented data effectively. The mixed results indicate that some augmentations may not align well with the algorithm's design, highlighting the potential necessity for algorithmic adjustments to harness the full potential of augmented data.

9. Final Thoughts

The findings emphasize that while data augmentation holds promise, its application to attributed multi-relational graphs requires careful consideration of the graph's unique structure and attributes as well as the specific clustering tasks. Future work should focus on developing more sophisticated and dataset-specific augmentation strategies, along with potential modifications to algorithms like SpectralMix to better exploit the augmented data. Also, another choice of clustering algorithm could be considered, as KMeans is not the most sophisticated clustering algorithm. By addressing these factors, we can enhance the efficacy of data augmentation techniques and improve the overall performance of graph-based clustering algorithms.

10. GitHub Code Repository:

- **https://github.com/ZoheirELhouari/SpectralMixPython**

11. Reference:

- https://arxiv.org/abs/2403.05198
- https://www.ml.cmu.edu/research/dap-papers/kondor-diffusion-kernels.pdf
- https://arxiv.org/pdf/2202.08235
- https://arxiv.org/pdf/2103.00111
- https://arxiv.org/pdf/2311.01840