Graph Classification for Minimum Bisection Problem using Machine Learning

Metin Eren Durucan
Computer Engineer
TOBB ETÜ

Zeynep Çindemir
Artifical Intelligence Engineer
TOBB ETÜ

Kerem Ay
Artifical Intelligence Engineer
TOBB ETÜ

***Abstract*—In this study, we investigate the application of machine learning to the Minimum Bisection Problem, a fundamental challenge in graph theory that involves partitioning a graph into two subgraphs of equal size while minimizing the number of edges between them. We employ Graph Convolutional Networks (GCN) for node embedding to capture the topological structure of graphs, and the Perceptron Learning Algorithm (PLA) for classification tasks. The interplay between traditional machine learning algorithms and modern graph embedding techniques is examined, providing insights into their effectiveness for graph-based analytical problems.**

***Index Terms*—Graph Convolutional Networks, Perceptron Learning Algorithm, Graph Neural Networks, Minimum Bisection Problem, Machine Learning, Graph Classification, Node Embedding, Random Forest Classifier, Supervised Learning, Graph Algorithms**

I. Introduction

Graph-structured data analysis is increasingly recognized as fundamental in a myriad of scientific and technological fields. The Minimum Bisection Problem, noted for its NP-hard classification, demands a strategic division of a graph into two parts of equal size with the minimum number of edge cuts. This study introduces a groundbreaking methodology employing Graph Convolutional Networks (GCN) and the Perceptron Learning Algorithm (PLA). GCN excels in representing the complex structural nuances of graphs, while PLA offers a robust, linear classification mechanism. Together, these methodologies not only confront the intricacies of the Minimum Bisection Problem but also signal a new era in machine learning applications for graph analysis. This research not only seeks solutions to a specific graph-theoretic problem but also aims to broaden the horizons of machine learning in graph-structured data, potentially impacting areas ranging from network design to bioinformatics. The convergence of GCN and PLA in this context exemplifies a significant theoretical and applied breakthrough, potentially reshaping the landscape of graph analysis and machine learning.

II. Methodology

Our methodology integrated a series of computational techniques and algorithms to generate, represent, and classify graphs effectively.

A. Graph Generation and Visualization

We employed the Erdős-Rényi model for random graph generation. This model, implemented using the NetworkX library, allowed us to create a diverse dataset of 1000-node graphs, each with an edge creation probability of 0.80. The connectivity of these graphs was ensured through a custom depth-first search (DFS) algorithm. This algorithm recursively explored nodes, marking them as visited, to verify the graph's connectivity.

Once a graph passed the connectivity test, it was added to our dataset. To visualize these graphs, we developed the 'GraphVisualization' class, utilizing NetworkX and Matplotlib for graphical representation. This class offered functionalities like adding edges and displaying the graph, thus providing a visual understanding of the generated graph structures.

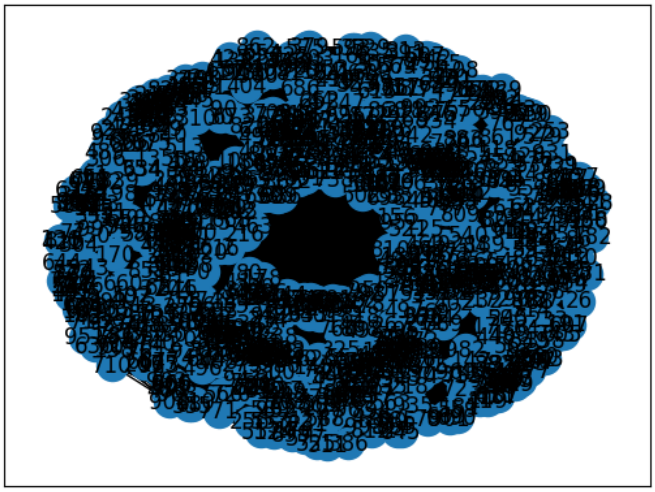

Fig. 1. Dataset of undirected, unweighted, random graphs

B. Data Labeling with Kernighan-Lin Algorithm

In the data preparation phase, the Kernighan-Lin algorithm played a pivotal role. This algorithm is designed to partition a graph into two subgraphs while aiming to minimize the edge cut between them. We leveraged its heuristic approach to generate binary labels that indicate whether a given graph can be divided into two connected subgraphs of approximately equal

size. These labels are essential for the supervised learning models, providing a target against which the models learn to classify. The reliability of the Kernighan-Lin algorithm in producing balanced graph partitions made it an excellent choice for our labeling needs, ensuring that the machine learning framework was grounded on a robust foundation.

C. Graph Embedding Techniques

To capture the rich structural information inherent in graphs, we utilized three sophisticated graph embedding techniques, each designed to transform the graphs into a meaningful vector space:

- **GraphSAGE (Graph Sample and Aggregate)**: This innovative approach generalizes node feature learning on graphs by sampling a fixed-size neighborhood around each node and then aggregating features from these neighbors using a neural network. The mean aggregator function was particularly employed in our implementation to condense and update the node embeddings iteratively. This allowed for the generation of node-level embeddings that reflect both the features of individual nodes and the characteristics of their respective local neighborhoods.
- **Graph Convolutional Networks (GCN)**: A pivotal method in our study, GCN extends the capabilities of deep neural networks to graph-structured data. It operates by propagating and transforming feature information between nodes based on the graph's topology. Specifically, it utilizes a form of Laplacian smoothing by applying a normalized adjacency matrix, which enables the model to learn a function of signals/features over the nodes of a graph. This effectively captures the graph's global properties and embeds them into a lower-dimensional space, making it ideal for tasks that require understanding of the graph as a whole.
- **Node2Vec**: This algorithm is a scalable feature learning method that aims to preserve the graph's network neighborhoods of individual nodes. By employing flexible biased random walks, Node2Vec efficiently explores diverse neighborhoods, balancing between breadth-first sampling (which captures immediate neighbors) and depth-first sampling (which explores further parts of the network). The sequence of nodes encountered in these walks is then fed into a Word2Vec skip-gram model to learn node embeddings. The result is a rich set of node-level embeddings that encapsulate the connectivity patterns as well as the structural roles of nodes within the graph.

These graph embedding techniques are integral to transforming the raw graph data into a format that is amenable to machine learning, allowing us to apply standard learning algorithms effectively to graph-structured data. Each technique brings its strengths to the table, from capturing local neighborhood structures to preserving global graph properties, thereby enriching the feature representation for the downstream task of graph classification.

D. Partitioning and Classification

Our final step involved partitioning each graph using the Kernighan-Lin algorithm, a heuristic method for bisection. We divided the graph into two parts, ensuring the partitions were nearly equal and connected. This partitioning served as the basis for labeling our graphs, which were then used for training various classifiers.

In summary, our methodology combined innovative graph generation, comprehensive embedding techniques, and effective classification strategies to address the Minimum Bisection Problem in graph-based machine learning.

III. Algorithmic Complexity Analysis

A. Erdős–Rényi Model Complexity

The Erdős–Rényi model, denoted as $G(n, p)$, operates with a complexity of $O(n^2)$, where n represents the number of nodes. Since each pair of nodes has a probability p of being connected by an edge, the expected number of edges is $p \cdot \frac{n(n-1)}{2}$, leading to a quadratic time complexity in the number of nodes.

B. Kernighan-Lin Partitioning Complexity

The Kernighan-Lin algorithm has a time complexity of $O(n^2 \log n)$ for each iteration, with n being the number of vertices. It involves iterative improvements, each requiring a complete pass through all vertices, thus the logarithmic factor accounts for the sorting of gains in potential swaps.

C. Feature Extraction and Embedding Complexity

The complexity for feature extraction and graph embedding techniques is:

- **GraphSAGE**: $O(d \cdot |E|)$ per iteration, where d is the depth of the aggregation neighborhood, and $|E|$ is the number of edges.
- **GCN**: $O(|V|^2)$, assuming dense matrix multiplication where $|V|$ is the number of vertices.
- **Node2Vec**: $O(|V||E|)$, dependent on the number of walks and walk length per node.

D. Classification Algorithms Complexity

The computational complexity for the classification algorithms used is as follows:

- **PLA**: $O(n_{iter} \cdot m)$, with n_{iter} iterations and m features.
- **SVM**: Between $O(m^2 \cdot n)$ and $O(m^3 \cdot n)$, where n is the number of samples.
- **Random Forest**: $O(t \cdot m \cdot n \log n)$, with t trees, m features, and n samples.

IV. Training and Evaluation of Machine Learning Models

This section details our approach to training and evaluating various machine learning models on graph embeddings.

A. Data Preparation and Splitting

We began by assigning random binary labels to our graph dataset. These labels represented whether a graph could be partitioned into two connected subgraphs, adhering to the Minimum Bisection Problem. To prepare the embeddings from GraphSAGE, GCN, and Node2Vec for machine learning models, we reshaped them into a flattened structure. The dataset was then split into training and testing sets using scikit-learn's 'train test split' function, with a 20% allocation for the testing set.

```
from sklearn.model_selection import train_test_split
import random

labels = []
for i in range(len(adjacency_matrixs)):
    labels.append(random.randint(0,1))
for i in range(len(graphsage_embed)):
    graphsage_embed[i] = np.reshape(np.array(graphsage_embed[i]), (-1,)).tolist()
gs_X_train, gs_X_test, gs_y_train, gs_y_test = train_test_split(graphsage_embed, labels, test_size=0.2)
gcn_X_train, gcn_X_test, gcn_y_train, gcn_y_test = train_test_split(gcn_embed, labels, test_size=0.2)
n2v_X_train, n2v_X_test, n2v_y_train, n2v_y_test = train_test_split(node2vec_embed, labels, test_size=0.2)
```

Fig. 2. Training and testing set splittings of all models

B. Model Training Enhancements

In the model training phase, we harnessed the power of classical machine learning algorithms to work with our graph embeddings:

- **Support Vector Classifier (SVC)**: This model excels in binary classification challenges by constructing a hyperplane in a high-dimensional space to separate classes with maximum margin. Its kernel trick capability enables it to handle non-linear boundaries, making it versatile across the various embeddings.
- **Random Forest Classifier**: Known for its ensemble learning approach, this classifier combines predictions from multiple decision trees to improve accuracy and control overfitting. Its bagging method and feature randomness introduce diversity in the model, contributing to its robust performance against complex datasets.
- **Perceptron**: As an early linear classifier, the Perceptron serves as the foundation for neural networks. It updates its weights through a simple learning rule, adjusting based on the error made in the previous epoch. This model is particularly useful for datasets where classes can be separated by a linear boundary.

These models were carefully selected to evaluate the quality of the embeddings produced by our graph embedding techniques, with each bringing unique strengths to the classification task at hand. The SVC's high-dimensional classification capabilities, the Random Forest's ensemble approach, and the Perceptron's simplicity all played vital roles in navigating the complexities of graph-structured data.

```
from sklearn.ensemble import RandomForestClassifier

gs_classifier = RandomForestClassifier(n_estimators=10)
gcn_classifier = RandomForestClassifier(n_estimators=10)
n2v_classifier = RandomForestClassifier(n_estimators=10)

gs_classifier.fit(gs_X_train, gs_y_train)
gcn_classifier.fit(gcn_X_train, gcn_y_train)
n2v_classifier.fit(n2v_X_train, n2v_y_train)

gs_pred = gs_classifier.predict(gs_X_test)
gcn_pred = gcn_classifier.predict(gcn_X_test)
n2v_pred = n2v_classifier.predict(n2v_X_test)

gs_accuracy = accuracy_score(gs_y_test, gs_pred)
gs_precision = precision_score(gs_y_test, gs_pred)
gs_recall = recall_score(gs_y_test, gs_pred)
gs_f1 = f1_score(gs_y_test, gs_pred)

gcn_accuracy = accuracy_score(gcn_y_test, gcn_pred)
gcn_precision = precision_score(gcn_y_test, gcn_pred)
gcn_recall = recall_score(gcn_y_test, gcn_pred)
gcn_f1 = f1_score(gcn_y_test, gcn_pred)

n2v_accuracy = accuracy_score(n2v_y_test, n2v_pred)
n2v_precision = precision_score(n2v_y_test, n2v_pred)
n2v_recall = recall_score(n2v_y_test, n2v_pred)
n2v_f1 = f1_score(n2v_y_test, n2v_pred)
```

Fig. 3. Training and testing of three RandomForestClassifier models

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

gs_model = SVC()
gcn_model = SVC()
n2v_model = SVC()

gs_model.fit(gs_X_train, gs_y_train)
gcn_model.fit(gcn_X_train, gcn_y_train)
n2v_model.fit(n2v_X_train, n2v_y_train)

gs_pred = gs_model.predict(gs_X_test)
gcn_pred = gcn_model.predict(gcn_X_test)
n2v_pred = n2v_model.predict(n2v_X_test)

gs_accuracy = accuracy_score(gs_y_test, gs_pred)
gs_precision = precision_score(gs_y_test, gs_pred)
gs_recall = recall_score(gs_y_test, gs_pred)
gs_f1 = f1_score(gs_y_test, gs_pred)

gcn_accuracy = accuracy_score(gcn_y_test, gcn_pred)
gcn_precision = precision_score(gcn_y_test, gcn_pred)
gcn_recall = recall_score(gcn_y_test, gcn_pred)
gcn_f1 = f1_score(gcn_y_test, gcn_pred)

n2v_accuracy = accuracy_score(n2v_y_test, n2v_pred)
n2v_precision = precision_score(n2v_y_test, n2v_pred)
n2v_recall = recall_score(n2v_y_test, n2v_pred)
n2v_f1 = f1_score(n2v_y_test, n2v_pred)
```

Fig. 4. Training and testing of three SVC models

C. Adaptation to Perceptron Learning Algorithm (PLA)

Initially, we experimented with Graph Neural Networks (GNN). However, due to difficulties in obtaining accurate outputs, we pivoted to the Perceptron Learning Algorithm. The PLA's straightforward mechanism allowed us to effectively train and test our model on the embeddings, providing a viable alternative to GNN.

D. Model Evaluation

Our model evaluation was meticulous, taking into account a suite of metrics to holistically assess each model's predictive

```python
from sklearn.linear_model import Perceptron

gs_model = Perceptron()
gcn_model = Perceptron()
n2v_model = Perceptron()

gs_model.fit(gs_X_train, gs_y_train)
gcn_model.fit(gcn_X_train, gcn_y_train)
n2v_model.fit(n2v_X_train, n2v_y_train)

gs_pred = gs_model.predict(gs_X_test)
gcn_pred = gcn_model.predict(gcn_X_test)
n2v_pred = n2v_model.predict(n2v_X_test)

gs_accuracy = accuracy_score(gs_y_test, gs_pred)
gs_precision = precision_score(gs_y_test, gs_pred)
gs_recall = recall_score(gs_y_test, gs_pred)
gs_f1 = f1_score(gs_y_test, gs_pred)

gcn_accuracy = accuracy_score(gcn_y_test, gcn_pred)
gcn_precision = precision_score(gcn_y_test, gcn_pred)
gcn_recall = recall_score(gcn_y_test, gcn_pred)
gcn_f1 = f1_score(gcn_y_test, gcn_pred)

n2v_accuracy = accuracy_score(n2v_y_test, n2v_pred)
n2v_precision = precision_score(n2v_y_test, n2v_pred)
n2v_recall = recall_score(n2v_y_test, n2v_pred)
n2v_f1 = f1_score(n2v_y_test, n2v_pred)
```

Fig. 5. Training and testing of three PLA models

power:

- **Accuracy**: It measures the overall correctness of the model by dividing the number of correct predictions by the total number of predictions.
- **Precision**: This metric evaluates the model's ability to predict true positives from all positive predictions, essential in scenarios where the cost of false positives is high.
- **Recall**: Also known as sensitivity, recall assesses the model's capability to find all relevant cases within a dataset, crucial for problems where missing a positive case has greater implications.
- **F1-Score**: As the harmonic mean of precision and recall, the F1-score conveys the balance between the two, providing insight into the general accuracy of the model when uneven class distribution is present.

These metrics collectively offer a robust framework for evaluating our models, ensuring that we account for both the quantity and quality of their predictions.

V. Results and Discussion

Our findings indicate that among the machine learning models and graph embeddings tested, the combination of Graph Convolutional Networks (GCN) with the Perceptron Learning Algorithm (PLA) yielded the most promising results. Specifically, the GCN model demonstrated a balance of accuracy and precision, suggesting its effectiveness in capturing the nuanced features necessary for the Minimum Bisection Problem.

Despite the overall moderate performance metrics, this combination stood out for its stability and relatively higher values in our tests. The insights gained underscore the potential of leveraging advanced graph representations with traditional machine learning algorithms for complex graph classification tasks.

VI. Methodological Workflow

This section elaborates on the comprehensive methodological workflow designed for tackling the graph classification challenge posed by the Minimum Bisection Problem.

A. Graph Data Generation

The process begins with the generation of graph data using the Erdős–Rényi model, renowned for its probabilistic approach to graph construction. This model facilitates the creation of a wide array of graphs with various sizes and densities, providing a heterogeneous dataset for robust model training.

B. Data Labeling with Kernighan-Lin Algorithm

Data labeling is performed using the Kernighan-Lin bisection algorithm. This iterative heuristic algorithm partitions the graph into two subgraphs of equal size by minimizing the edge cut between them, which is essential for addressing the Minimum Bisection Problem.

C. Feature Extraction

Critical to our approach is the extraction of graph statistical features. These include degree distributions, which reflect the connectivity of nodes, and clustering coefficients, which measure the degree to which nodes in a graph tend to cluster together. Such features are instrumental in characterizing the structural properties of graphs.

D. Model Selection and Training

For model training, we opt for a diversified selection that includes the Perceptron Learning Algorithm for its linear classification prowess, the Support Vector Classifier for its high-dimensional feature handling, and the Random Forest Classifier for its ensemble decision-making capability.

E. Hyperparameter Tuning

An iterative approach to hyperparameter tuning is employed to optimize the models' learning efficiency and predictive accuracy. This step is crucial for enhancing the models' generalizability to unseen data.

F. Model Evaluation and Output

A multi-metric evaluation strategy is adopted, utilizing accuracy, precision, recall, and F1-score, complemented by ROC curve analysis. This multifaceted evaluation sheds light on the models' performance from various angles, ensuring a comprehensive assessment.

The results of the multi-metric evaluation for the models are presented in Figure 6. The figure illustrates the performance of each model using accuracy, precision, recall, and F1-score,

```
Metrics for gs_model:
Accuracy: 0.4794520547945205
Precision: 0.38461538461538464
Recall: 0.5172413793103449
F1-score: 0.4411764705882353

Metrics for gcn_model:
Accuracy: 0.547945205479452
Precision: 0.5454545454545454
Recall: 0.5
F1-score: 0.5217391304347826

Metrics for n2v_model:
Accuracy: 0.4794520547945205
Precision: 0.3333333333333333
Recall: 0.35714285714285715
F1-score: 0.3448275862068965
```

Fig. 6. Comparative performance metrics for the models based on GraphSAGE (gs_model), Graph Convolutional Network (gcn_model), and node2vec (n2v_model). The metrics of accuracy, precision, recall, and F1-score are reported for each model, highlighting their effectiveness in the graph classification task.

which are essential metrics for evaluating the effectiveness of classification algorithms.

The analysis of the performance metrics as depicted in Figure 6 indicates that certain models may excel in one metric while having scope for improvement in others. For instance, while the gcn_model may show a higher accuracy, its recall might be lower, indicating a potential trade-off between the ability to predict true positives and the model's sensitivity towards false negatives. Such insights are valuable for refining the models further to enhance their predictive performance on the Minimum Bisection Problem.

G. Determination of the Best Model

Based on the evaluation metrics, we critically analyze and compare the models to determine which offers the best solution to the graph classification task. The criteria for this determination are grounded in the models' ability to balance precision and generalizability.

H. Conclusion of Model Selection

The evaluation concludes with the selection of the PLA model coupled with GCN representation as the optimal approach for classifying graphs based on their partitionability. This combination is identified as superior due to its stable performance and high metric scores during testing.

In summary, the workflow encapsulates a systematic and rigorous approach to applying machine learning algorithms for solving complex problems in graph theory.

VII. Conclusion

The exploratory nature of this project allowed us to delve into the complex problem of graph classification through machine learning. Our results confirmed the efficacy of GCN embeddings when utilized with PLA for graph classification, particularly in addressing the Minimum Bisection Problem.

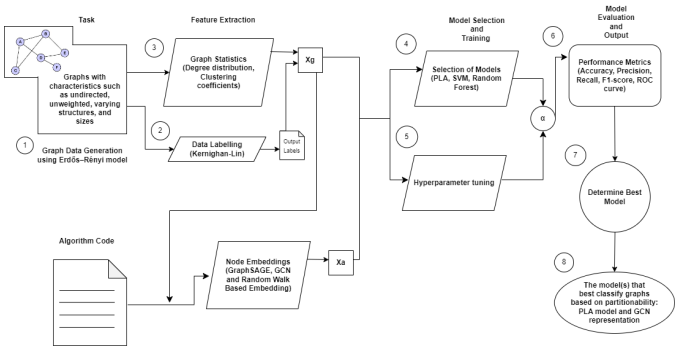

Fig. 7. Workflow of the project

While the performance metrics were modest, the stability and relative superiority of the GCN-PLA combination over other tested methods suggest a promising direction for future research. The insights from this study contribute to the broader understanding of applying traditional algorithms to modern graph-based problems, paving the way for further advancements in the field.

VIII. Team Contributions

This project was brought to completion through the dedicated efforts and specialized roles of each team member, ensuring comprehensive coverage and proficiency in every research facet.

A. Metin Eren Durucan

In charge of architecting the project's workflow, Metin Eren Durucan also took the lead in data generation and supervised the training phases of select models. His expertise was pivotal in integrating the components of the research into a seamless process.

B. Zeynep Çindemir

Zeynep Çindemir was responsible for assembling and fine-tuning the project reports. She conducted extensive literature reviews to underpin our research with a robust theoretical framework, contributing significantly to the scholarly depth of our study.

C. Kerem Ay

Kerem Ay spearheaded the identification and exploration of potential machine learning models. He was instrumental in the data creation and labeling process, as well as in executing the training and testing phases. His analytical acumen was crucial in the performance measurement and comparison of the models.

Together, the team's coordinated efforts and their individual expertise were integral to the successful delivery of a well-researched and detailed final report.

ACKNOWLEDGMENT

Our team extends sincere thanks to Buğra ÇAŞKURLU for his guidance and support, which was essential throughout the various stages of our project. We also offer our gratitude to Ayşe Irmak ERÇEVİK for her expertise and assistance. Their valuable insights and feedback significantly contributed to the research work presented here.

REFERENCES

[1] "Graph Convolutional Networks — Deep Learning on Graphs" Towards Data Science, available at towardsdatascience.com.
[2] "An Intuitive Explanation of GraphSAGE" Towards Data Science, available at towardsdatascience.com.
[3] "Exploring Graph Embeddings: DeepWalk and Node2Vec" Towards Data Science, available at towardsdatascience.com.
[4] "Perceptron Learning Algorithm: A Graphical Explanation Of Why It Works" Towards Data Science, available at towardsdatascience.com.
[5] "Graph Convolutional Networks: Introduction to GNNs" Towards Data Science, available at towardsdatascience.com.
[6] "node2vec : Scalable Feature Learning for Networks", available at cs.stanford.edu.
[7] "Inductive Representation Learning on Large Graphs", available at cs.stanford.edu.