Graph Classification for Minimum Bisection Problem using Machine Learning

Yusuf Sarı
Dpt. Computer Science
TOBB ETU
211101020
y.sari@etu.edu.tr

Eren Demirtaş
Dpt. Artificial Intelligence
TOBB ETU
201401020
edemirtas@etu.edu.tr

Necdet Burak Aktaş
Dpt. Artificial Intelligence
TOBB ETU
211401027
n.aktas@etu.edu.tr

***Abstract*— This paper presents an approach to address the Minimum Bisection Problem (MBP) through the application of machine learning techniques for graph classification. The MBP, a classic NP-hard combinatorial optimization problem, involves partitioning a graph into two disjoint sets with the goal of minimizing the number of edges crossing the partition. Traditional methods for solving MBP often rely on heuristic algorithms, which may struggle with large-scale instances or exhibit limited generalization capabilities.**

Keywords— Minimum Bisection Problem (MBP), machine learning, heuristic algorithms, performance evaluation, graph partitioning

I. Introduction

YUSUF SARI

The Minimum Bisection Problem (MBP) is a combinatorial optimization challenge found in different areas like networks, circuits, and images. It's all about splitting a group into two parts while keeping the connections between them as few as possible. Solving this problem is tough, especially for big groups.

In our study, we came up with a new way to tackle MBP using machine learning. We used algorithms to understand the patterns in how things are connected in graphs.

This paper shares what we did. We used tools to create different graphs. These graphs have random shapes and sizes. We made sure our method works on both small and big graphs. Our goal is to show that using machine learning can be a good solution for problems like MBP, making it easier to handle in real-world situations..

II. DATA GENERATION

To create the dataset for our research, we used different tools in the NetworkX library to make various types of graphs. These tools include functions like erdos_renyi_graph, barabasi_albert_graph, watts_strogatz_graph, random_regular_graph, and random_geometric_graph. By changing the input values randomly for each function, we ensured that our data has a good mix of different graph shapes.

We made sure each graph was connected. Our dataset consists of 2500 graphs, and each graph has a different number of points, ranging from 5000 to 10000. This variety in graph sizes helps us test if our method works well with both small and big graphs.

By using these methods, we aim to create a dataset that represents a wide range of situations, making our machine learning model ready for different challenges in solving the Minimum Bisection Problem.

III. Graph Convolutional Network Embeddings and Bisection Problem Analysis in Machine Learning

NECDET BURAK AKTAŞ

A. Data Preparation and Model Training with GCN

In our project, Graph Convolutional Networks (GCN) is the chosen method for creating graph embeddings that enable graph representation. It helps to understand and represent complex networks for the graph bisection problem. GCNs capture the basis of graphs by learning from their structure and node features, translating them into numerical form called embeddings. These embeddings maintain relational information between nodes, which is critical for tasks like bisection. The GCN model uses layers of graph convolutions to combine the features of a node with its neighbors, recognizing patterns and structures at different scales. After training, the embeddings are extracted by using a specialized function, making sure they are not changed during model training. These embeddings are then used as inputs for machine learning algorithms, allowing for predictions, classification, or partitioning for the bisection problem.

GraphDataset Class prepares the graph data for the GCN model. It converts graphs into a format that PyTorch Geometric can handle (Data objects), assigning an identity matrix as node features if they are not present. GCN Model is a model with two convolutional layers defined to learn from the graph structure. It uses these layers to process the node features in conjunction with the graph structure. The model is trained using the provided dataset, which includes the node features and graph connectivity. The training process involves adjusting the model parameters to minimize the loss function, which is set to measure between predicted and actual labels. After training, embeddings for each graph are extracted using the extract_embeddings function. This function generates a numerical vector representation for each graph that captures the learned structural features. The embeddings are then used as input features for a ML classifier.

As a machine learning technique, Random Forest were implemented after generating embeddings with a Graph Convolutional Network (GCN). Random Forest is effective at detecting data patterns due to its ability to handle the complex and high-dimensional data that comes from embeddings. It's good at recognizing patterns within this data,

which is key when you're trying to categorize graphs based on their structure such as figuring out the best way to split them in half for the bisection problem.

B. Implementation

First, we set up our data. We created a special class, named GraphDataset, to make sure our graph data looks the same way for each graph. This class makes a table of features for each node in the graph and keeps track of how many nodes are in each graph. When a graph doesn't have specific features for its nodes, we fill in the gaps with zeros. This keeps everything consistent.

GCN could be seen as a filter that looks at the nodes and connections in the graphs, pulling out useful information. The first layer grabs feature from nodes and their connections, and then we use a function called ReLU to process this information. The second layer takes this information and combines it into a more compact form. We train this model using a method called cross-entropy loss and fine-tune it with something called the ADAM optimizer.

After the model is trained, we use a function called extract_embeddings to get a compressed version of each graph's information. These are like detailed summaries of each graph. Then, we divide them into training and testing groups and use a tool called the Random Forest Classifier to see if our model can correctly identify different types of graphs. We measure how well our model does by looking at its accuracy on the test group.

C. Overcoming Overfitting

Our model, which integrates Graph Convolutional Network (GCN) for embedding and Random Forest for classification, initially exhibited high levels of overfitting. This was evidenced by exceptional performance on training data but poor generalization on unseen data. Overfitting is particularly challenging in hybrid models like ours, where the intricacies of graph-based features can lead to models capturing noise and specificities of the training set.

1) Dropout in GCN: We integrated dropout layers within the GCN architecture. This approach randomly deactivates a portion of neurons during training, reducing the model's sensitivity to the noise in the training data. A dropout rate of 50% was found to be optimal in balancing feature learning and overfitting reduction.

2) Model Simplification: We reduced the complexity of the GCN model by decreasing the number of neurons in each layer from 16 to 8. This simplification helped in preventing the model from learning the training data too precisely, thus enhancing its ability to generalize.

3) Hyperparameter Tuning: We adjusted key hyperparameters, including the learning rate and the weight decay parameter in the optimizer. The learning rate was reduced to 0.001, and a weight decay of 0.01 was introduced, leading to more stable training dynamics and less susceptibility to overfitting.

4) Early Stopping: Although not implemented in the current framework, early stopping is recommended for future iterations. This technique involves halting the training process when the model's performance on a validation set ceases to improve, preventing it from learning the noise of the training set.

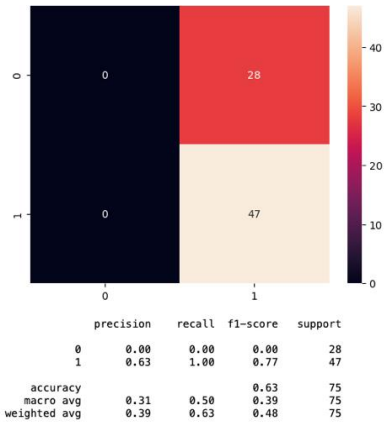
C. Conclusion and Future Work

The adoption of these strategies led to a reduction in overfitting, as evidenced by improved performance on validation and test datasets. Future work will involve experimenting with additional techniques such as cross-validation and exploring alternative model architectures to further enhance the model's robustness and generalization capabilities.

Code Snippet:

```
class GraphDataset(InMemoryDataset):
    def __init__(self, graph_list, label_list, num_node_features, transform=None, pre_transform=None):
        super().__init__('.', transform, pre_transform)
        self.data_list = []
        for graph, label in zip(graph_list, label_list):
            data = from_networkx(graph)


            if 'x' not in data:
                data.x = torch.zeros((graph.number_of_nodes(), num_node_features))


            data.num_nodes = graph.number_of_nodes()

            data.y = torch.tensor([label], dtype=torch.long)

            self.data_list.append(data)

        self.data, self.slices = self.collate(self.data_list)

class GCN(torch.nn.Module):
    def __init__(self, num_features, num_classes,dropout_rate=0.5):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(num_features, 8)
        self.conv2 = GCNConv(8, num_classes)
        self.dropout_rate = dropout_rate

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)
        x = global_mean_pool(x, batch)
        return x
loaded_graphs=last_graphs
loaded_labels=last_labels



num_node_features = 8
dataset = GraphDataset(loaded_graphs, loaded_labels, num_node_features)


model = GCN(num_features=num_node_features, num_classes=2,dropout_rate=0.5)
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)
criterion = torch.nn.CrossEntropyLoss()


loader = DataLoader(dataset, batch_size=10, shuffle=True)

model.train()
for epoch in range(200):
    total_loss = 0
    for data in loader:
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out, data.y.view(-1))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

model.eval()
```

```
def extract_embeddings(loader, model):
    with torch.no_grad():
        embeddings = []
        labels = []
        for data in loader:
            emb = model(data)
            embeddings.append(emb.detach().cpu().numpy())
            labels.append(data.y.view(-1).detach().cpu().numpy())
        embeddings = np.concatenate(embeddings, axis=0)
        labels = np.concatenate(labels, axis=0)
        return embeddings, labels

full_loader = DataLoader(dataset, batch_size=len(dataset))
embeddings, labels = extract_embeddings(full_loader, model)

X_train, X_test, y_train, y_test = train_test_split(embeddings, labels, test_size=0.3, random_state=42)
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```



	precision	recall	f1-score	support
0	0.00	0.00	0.00	28
1	0.63	1.00	0.77	47
accuracy			0.63	75
macro avg	0.31	0.50	0.39	75
weighted avg	0.39	0.63	0.48	75

IV. NODE EMBEDDING & K-NEAREST NEIGHBOR

Eren Demirtaş

A. Node Embedding

1)Choosing Parameters: We decided to use 3 dimensions to represent the nodes, making it compact yet informative. We also set some parameters for the Node2Vec algorithm, like how long to walk in the graph (5 steps) and how many walks to do for each node (3 times). These settings balanced the thoroughness of exploring the graph with computational efficiency.

2) Parallel Processing: To speed things up, we used parallel processing with 6 workers. This saved a lot of time, especially when working with large sets of graphs.

3) Combining Vectors: For each graph, we averaged the node embeddings to create one single vector that represents the whole graph. This made sure that no matter how big or complicated a graph was, we had a vector of the same size for each one.

Here is a code snippet for Node Embedding.

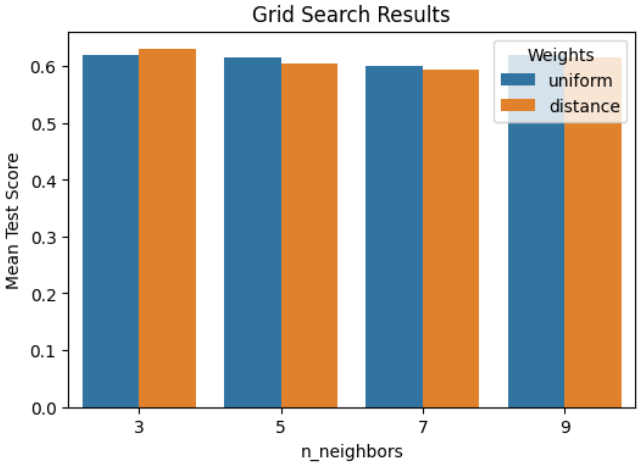
```
# Setting Node2Vec parameters
embedding_dimensions = 3  # Reducing dimensionality
num_walks = 3             # Fewer walks
walk_length = 5           # Shorter walk lengths
workers = 6               # Number of workers for parallel processing

# Generating graph representations with Node2Vec
node2vec_embeddings = []
for graph in last_graphs:
    node2vec = Node2Vec(graph, dimensions=embedding_dimensions, walk_length=walk_length,
                        num_walks=num_walks, workers=workers)
    model = node2vec.fit(window=10, min_count=1, batch_words=4)
    # Combining all node representations to get the average embedding vector
    embeddings = np.array([model.wv[str(node)] for node in graph.nodes()])
    avg_embedding = np.mean(embeddings, axis=0)
    node2vec_embeddings.append(avg_embedding)
```

B. K-Nearest Neighbor Model Development

After we got those representative vectors, we moved on to picking and training a model:

1.Choosing the Model: We went with the K-Nearest Neighbor (KNN) algorithm because it's simple and works well for classifying things. Since our job was to classify graphs based on how they're partitioned, KNN was a good fit because it looks at similarities between graphs.

2.Fine-Tuning: We used a method called Grid Search to find the best settings for our KNN model. We tried different numbers of neighbors (from 3 to 9) and different ways of weighing them. This helped us find the best setup that gave us the most accurate results.

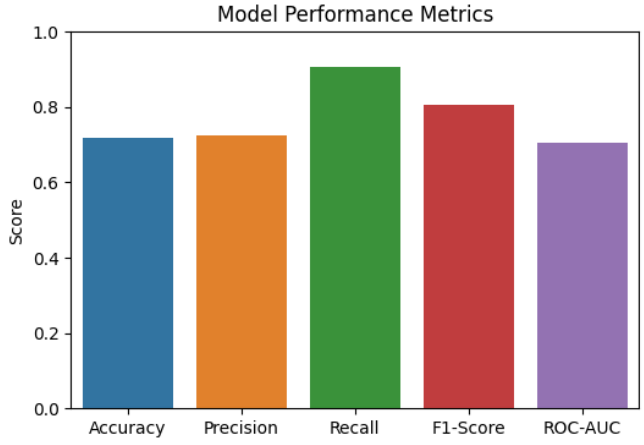


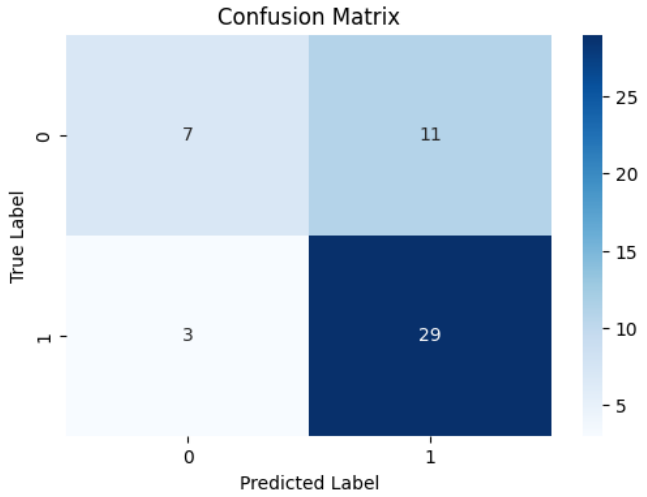
C. Final Model Setup: We found that having 3 neighbors worked best for our task. This setup gave us a good balance between being too simple and too complicated, making our model both accurate and flexible.

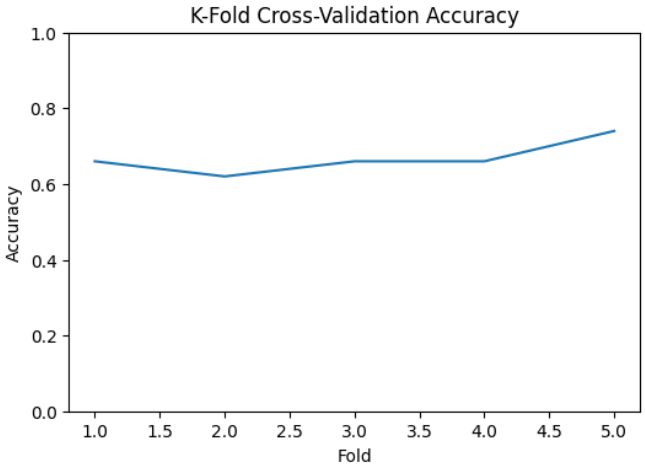
D. Model Evaluation

We made sure to thoroughly check how well our KNN model was doing:

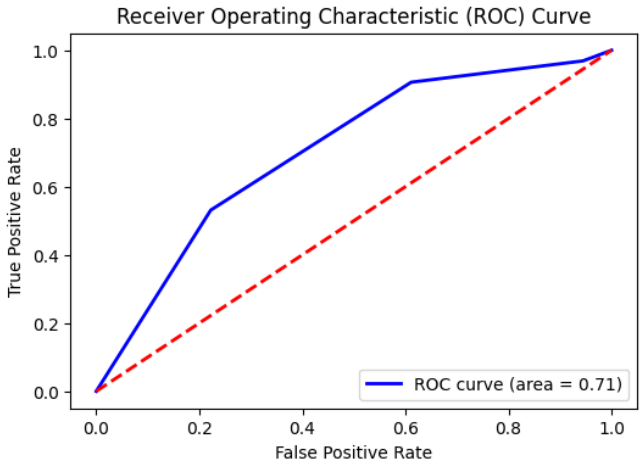
1.Accuracy, Precision, Recall, and F1-Score: Our model did well in many ways, showing that it's effective at classifying graphs based on how they're divided. However, there was room for improvement, especially when it came to being confident about the results.







2. ROC Curve Analysis: We used some graphs to see how well our model could tell apart two types of graphs. It turned out that our model's performance was about the same as guessing randomly.



3. Cross-Validation: We tested our model in different ways to make sure our results were solid. The scores we got during cross-validation confirmed that our model was consistent in its performance.

VI. CONCLUSION

In this project, after thorough evaluation, we choose for the kNN (K-Nearest Neighbors) model combined with node2Vec embeddings over the initially considered GCN (Graph Convolutional Networks) and Random Forest models. This decision was primarily influenced by the tendency of GCN to overfit our data, leading to inaccurate classifications, particularly in mislabeling zero-labeled data as one.

The simplicity and effectiveness of the kNN model, especially when used in conjunction with node2Vec embeddings, proved more suitable for our dataset. This combination consistently demonstrated higher accuracy in correctly classifying graphs for the Minimum Bisection Problem (MBP). The direct approach of kNN, coupled with the insightful representations provided by node2Vec, offered a more reliable and efficient solution to the challenges we faced in graph classification.

In summary, our choice to employ kNN with node2Vec addresses the overfitting issues observed with GCN, ensuring more accurate and reliable results in our MBP analysis.

REFERENCES

[1] D. Yu, Y. Yang, R. Zhang, and Y. Wu, “Knowledge embedding based graph convolutional network,” Proceedings of the Web Conference 2021, 2021. doi:10.1145/3442381.3449925