# Graph Classification of Twitch user networks

Project Report

Presented to

Prof. Katerina Potika

Department of Computer Science

San Jose State University

In Partial Fulfillment

Of CS286 Course Requisites

By

Atharva Khadilkar - 016654092

Dania Jaison - 016690791

Mrunal Zambre - 016679182

May 2023

# Table of Contents

# Introduction

The advent of online gaming platforms has transformed the gaming landscape, enabling millions of players to connect, interact, and compete with each other in virtual environments. Twitch, one of the leading live streaming platforms for gamers, allows users to broadcast their gameplay and build communities around their favorite games. Understanding the gaming preferences of Twitch users can provide valuable insights for game developers, marketers, and the gaming community as a whole. In this project, we aim to analyze the ego-nets of Twitch users to classify them as either single-game players or multi-game players.

# Problem Statement

The main objective of this project is to develop a binary classification model to predict whether a Twitch user plays a single game or multiple games based on their ego-net. Single-game players are those who predominantly focus on a particular game, while multi-game players engage with multiple games. The hypothesis is that users playing a single game will have a dense network around the egos, while users playing multiple games will have a sparse network. This classification task is essential for understanding user preferences and tailoring game recommendations, targeted advertisements, and community engagement strategies to different player segments.

# Dataset Overview

The dataset at hand comprises the ego-nets of Twitch users who participated in the partnership program in April 2018. Each ego-net represents a Twitch user's network of friendships, where nodes represent users and links represent friendships. [1] The ego-net is a social graph that captures the relationships between Twitch users who interacted within the partnership program. The dataset contains 127,094 ego-nets, each labeled with a binary classification indicating whether the ego user plays a single game or multiple games. The label 0 indicates that the users of the graph play multiple games, while 1 indicates that they play a single game. The graphs are all undirected, and there is no additional information provided about the nodes and edges. We considered a different version of the same dataset, which we retrieved from [2]. This dataset contains the following 3 files:

1. twitch_egos_A: adjacency matrix for all graphs, where each line corresponds to (row, col) resp. (node_id, node_id).
2. twitch_egos_graph_indicator: column vector of graph identifiers for all nodes of all graphs, where the value in the i-th line is the graph_id of the node with node_id i.
3. twitch_egos_graph_labels: class labels for all graphs in the dataset, the value in the i-th line is the class label of the graph with graph_id i.

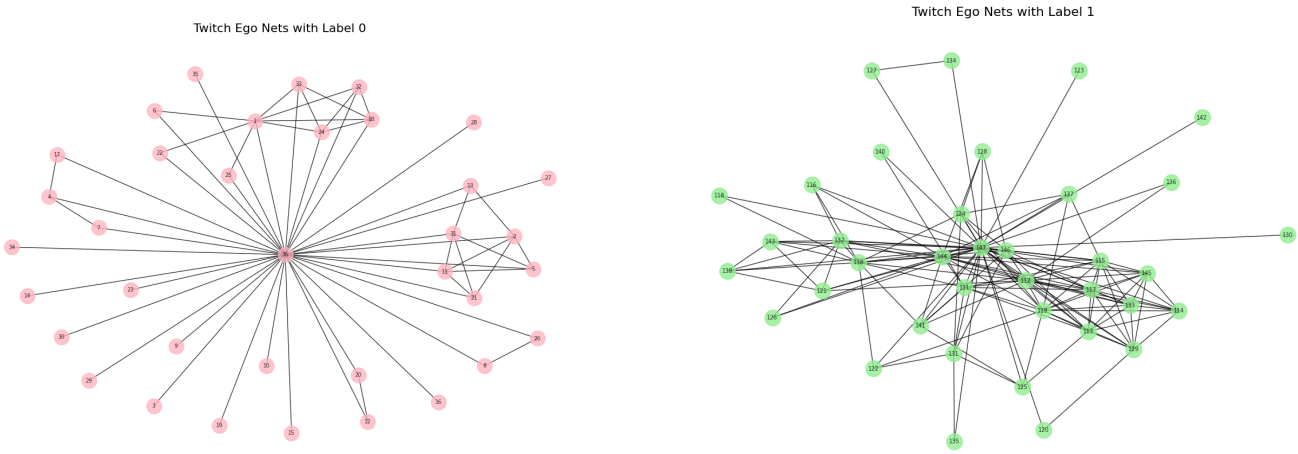
## Data Pre-processing

As the dataset was in the form of text files with edges and labels, it first needed to be pre-processed. The file "twitch_egos_A" contained the sparse (block diagonal) adjacency matrices for all graphs, where each line corresponded to (row, col). Since our dataset contains 127,094 graphs in total, we referred to the file "twitch_egos_graph_indicator" to find which nodes and edges correspond to which graphs. This file contains column vectors of graph identifiers for all nodes of all graphs, where the value in the i-th line is the graph_id of the node with node_id 'i'. From this file, we first extracted how many nodes belonged to each graph and then which nodes were connected through edges. Finally, we got the graph labels (target variable) from the file "twitch_egos_graph_labels" where the i-th line gave the label of the i-th graph. All this information was stored in a new CSV file.

Since the dataset contained 127,094 graphs, and our models would not be able to handle so much data, we decided to choose a subset of the dataset for our experiments. So, we randomly sampled a 1000 graphs belonging to class 0 and 1000 graphs belonging to class 1, and used these 2000 graphs as our final dataset.

## Visualization of the Data

In order to better understand the data and to test the hypothesis previously stated that "users playing a single game will have a dense network around the egos, while users playing multiple games will have a sparse network", we visualized some of the graphs belonging to both classes from the dataset. It was seen from the visualizations that graphs with label 0 were in fact sparser than the graphs with label 1. We used the NetworkX library to create and visualize the graphs as follows:




# Algorithms

To accomplish the classification task, we explored five machine learning approaches and evaluated their performance in predicting the game-playing behavior of Twitch users. The evaluation metrics were Accuracy and F1 scores. The chosen approaches are as follows:

**Random Forest (RF):** Random Forest is a type of supervised ensemble machine learning algorithm that works by constructing multiple decision trees at training time and outputting the class that is the mode of the classes (classification) of the individual trees. It is also capable of handling high-dimensional feature spaces and non-linear decision boundaries, which are often present in graph classification problems, which makes it a great choice for our problem. We considered the number of trees as 400 for all algorithms.

**Logistic Regression (LR):** Logistic regression is a classification algorithm commonly used in machine learning to classify data into two or more classes. The logistic regression classifier learns a set of weights for each feature, which are used to compute a score for each graph. It is mainly a simple and interpretable classification algorithm that can be used for binary classification problems, which makes it a good choice for graph classification problems with two classes.

**Support Vector Machine (SVM):** Support Vector Machines are a popular classification algorithm used in machine learning, which works by finding a hyperplane that best separates the data into different classes, while maximizing the margin between the hyperplane and the closest data points. It is a powerful classification algorithm that can handle both linear and non-linear decision boundaries, as well as high-dimensional feature spaces, which makes it

well-suited for graph classification problems. We used the kernel ‘linear’ for density features, and ‘rbf’ for the rest of the experiments.

**Graph Convolutional Network (GCN):** GCNs have been developed as a way to generalize convolutional neural networks (CNNs) to graph-structured data. GCNs work by performing convolutions on the graph using a filter, which is learned through training. The filter takes into account the features of the neighboring nodes in the graph, allowing the network to capture local patterns and relationships between nodes. Since we did not have any node or edge features in our dataset, GCNs were a good choice for learning hidden relationships.

**Graph Attention Networks (GAT):** The GAT architecture uses a self-attention mechanism to allow nodes to attend over their neighbors' features, thereby enabling the model to weigh the importance of the neighboring nodes in a dynamic manner. The main idea behind GAT is to compute node embeddings by aggregating the features of their neighbors using a learned attention mechanism that assigns different importance weights to different neighbors. It can also handle graphs with varying sizes and structures, which was necessary for our dataset.

## Python Libraries used

1. **NetworkX**: A library for the creation, manipulation, and study of complex networks. It was used for creation and visualization of graphs, and for computing densities and centralities.
2. **Gklearn**: A library for graph machine learning based on scikit-learn API. It was used for computing weisfeiler-lehman kernels.
3. **Karateclub**: A library for unsupervised machine learning on graphs. It was used for calculating node and graph embeddings [4].
4. **Pytorch and Pytorch Geometric**: A machine learning library for creating and training neural networks, with support for efficient computation on GPUs. It was used for the implementation of GCN and GAT.
5. **Sklearn**: A machine learning library that provides tools for data mining and data analysis, including classification, regression, clustering, and dimensionality reduction.
6. **Numpy**: A library for numerical computing.
7. **Pandas**: A library for data manipulation and analysis.

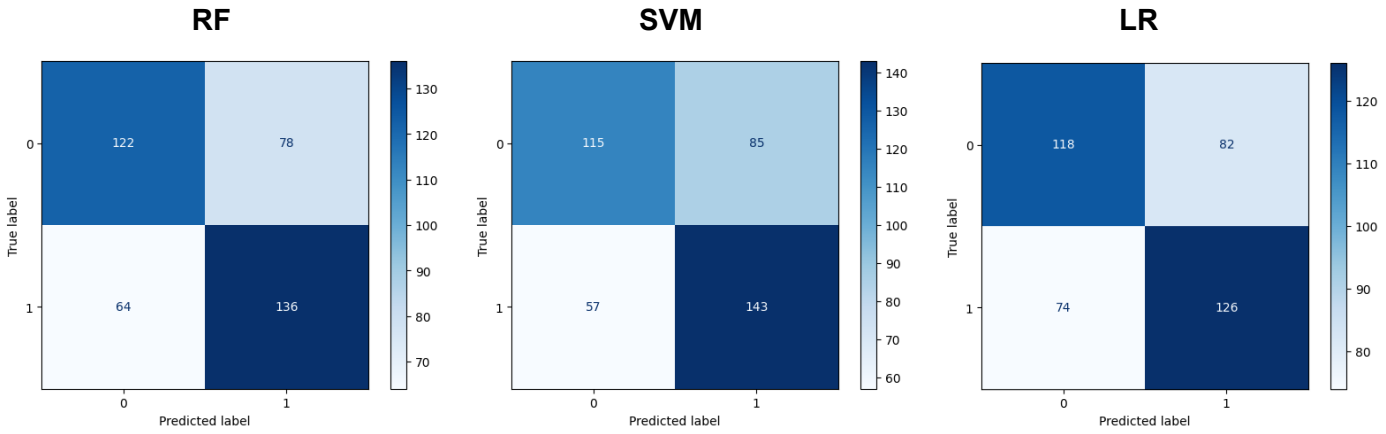
## Experiments: Traditional Models

For the traditional machine learning models, namely Random Forest, SVM and Logistic Regression, we used four different sets of features as inputs to the models.

## 1. Number of Nodes, Number of Edges, Density

In this approach, the number of nodes and edges and the density of all the graphs was calculated. According to the hypothesis [1] that denser graphs correspond to label 1, while sparser graphs belong to label 0, we tried to predict the graph labels using the three ML models. We saw that SVM and Logistic Regression performed best with accuracies of 65%. Considering F1 scores, Logistic Regression appears to perform better than SVM, with more equal scores. The results and confusion matrices were as follows:

| Model | Accuracy | F1 score of class 0 | F1 score of class 1 |
|---|---|---|---|
| Random Forest | 0.61 | 0.60 | 0.62 |
| SVM | 0.65 | 0.62 | 0.67 |
| Logistic Regression | 0.65 | 0.63 | 0.66 |



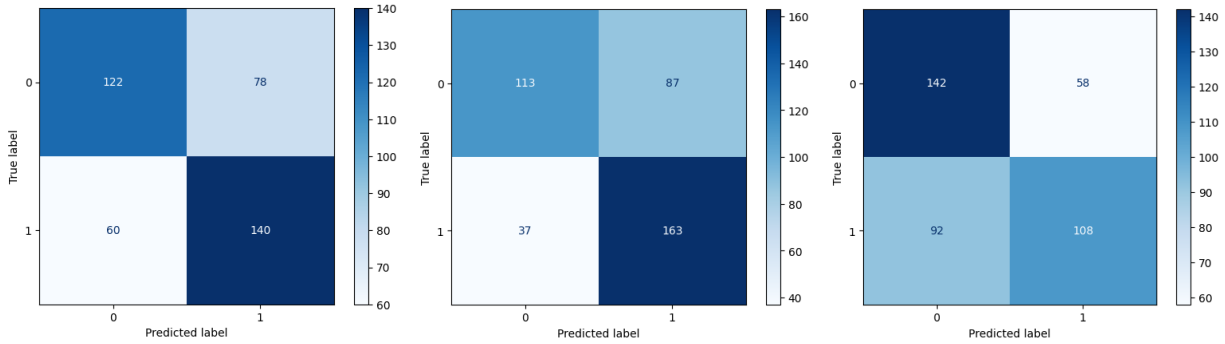

## 2. Weisfeiler-Lehman Kernels

For another approach, we calculated the Weisfeiler-Lehman kernels for all the graphs in our dataset, using the "gklearn" library [3].The resulting features were measures of similarity between each graph. We split the dataset into 80-20 split for training and testing. The train dataset was of the size 1600 x 1600 and the test dataset was of the size 400 x 1600. We saw that SVM gave us the best accuracy using the similarity features. However, it also gave us a lot of false positives (label 0). So, in that case, Random Forest was better at distinguishing

between the two classes, with equally good F1 scores for both classes. The results and confusion matrices were as follows:

| Model | Accuracy | F1 score of class 0 | F1 score of class 1 |
|---|---|---|---|
| Random Forest | 0.66 | 0.64 | 0.67 |
| SVM | 0.69 | 0.65 | 0.72 |
| Logistic Regression | 0.62 | 0.65 | 0.59 |

**RF** **SVM** **LR**



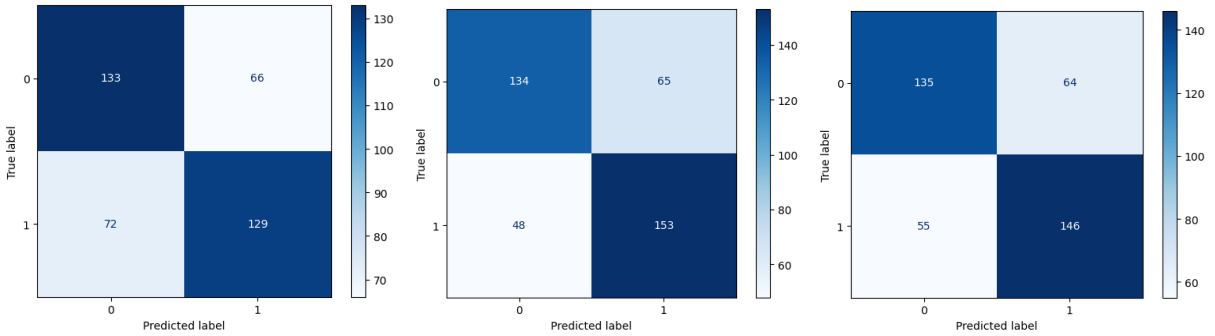

## 3. Feather Graph Embeddings

Next, we tried to use graph embedding techniques to get features [4]. One such technique was Feather Graph embeddings, which uses characteristic functions of node features with random walk weights to describe node neighborhoods [5]. Pooling with mean pooling is then used to get graph level statistics from these node features. These embedding features allowed SVM to give us the highest accuracy of 72%, with good F1 scores for each label. In fact, all the models were effective in distinguishing between the two classes, and no bias was present towards either label in the results. The results and confusion matrices were as follows:

| Model | Accuracy | F1 score of class 0 | F1 score of class 1 |
|---|---|---|---|
| Random Forest | 0.66 | 0.66 | 0.65 |
| SVM | 0.72 | 0.70 | 0.73 |

| | | | |
|---|---|---|---|
| Logistic Regression | 0.70 | 0.69 | 0.71 |

**RF** **SVM** **LR**

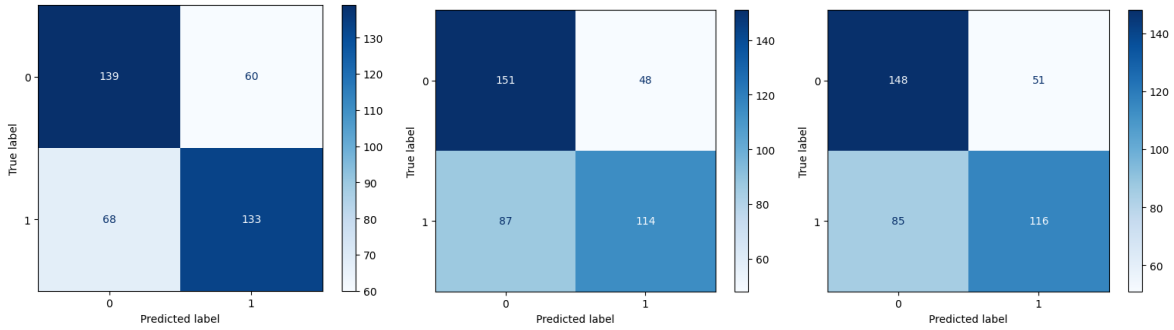



## 4. Graph2Vec Embeddings

Another technique for getting graph embeddings was using Graph2Vec. This program generates Weisfeiler-Lehman tree characteristics for graph nodes [6]. A graph - feature co-occurrence matrix is deconstructed using these characteristics to build graph representations. We expected this technique to give us similar results as our 2nd approach of computing and using WL kernels directly. We saw similar accuracy score results, but the best model in this approach was Random Forest, which had equal F1 scores for both classes as well. SVM and LR performed exactly the same, with equal accuracies and F1 scores. The results and confusion matrices were as follows:

| **Model** | **Accuracy** | **F1 score of class 0** | **F1 score of class 1** |
|---|---|---|---|
| Random Forest | 0.68 | 0.68 | 0.68 |
| SVM | 0.66 | 0.69 | 0.63 |
| Logistic Regression | 0.66 | 0.69 | 0.63 |

**RF** **SVM** **LR**




# Experiments: Generating Graphs for Neural Networks.

## Generating Node Embeddings for node feature matrices.

To create pytorch graphs as inputs to the Neural Networks, we needed a node feature matrix as an attribute to the existing graph. The node feature matrix was created using the following methods.

### 1. Using Closeness Centrality and Betweenness Centrality as node embeddings.

Due to the absence of node features in the original dataset, in the first approach for Graph Neural Networks we used Closeness and Betweenness centralities as node features to generate node feature matrix [7]. Since the betweenness captures the information flow and closeness captures a node's ability to reach other nodes we used these together to create a feature matrix to describe the importance of a node in a graph. The generated features were of dimensions 2 X 1 for each node.

### 2. Using GraphWave for node embeddings.

The second approach used to generate node embeddings was GraphWave which is a structural node embedding technique. GraphWave is a graph-based machine learning algorithm for learning node embeddings. The algorithm learns a set of wavelet-like functions to encode the graph structure into fixed-length representations [8]. Since the task at hand was Graph classification we chose GraphWave to find structural features. We defined the walk length to be 20. For each node the generated features was a vector of dimension 400 X 1.

### 3. Using Role2vec for node embeddings.

The third approach used to generate node embeddings was Role2vec which is also a structural node embedding technique. Role2vec leverages the concept of node roles, which represent the

local connectivity patterns of nodes in the graph [8]. We defined the walk length to be 20. For each node the generated features was a vector of dimension 128 X 1.

### Using node feature matrices to generate pytorch graphs.

Since the existing data was in csv data format, we generated networkx graphs from the csv file. The networkx graphs were then used to generate embeddings described in the previous section. Using the embeddings generated we created a node feature matrix for each node in the graph. The networkx graphs were then converted into pytorch geometric graphs using utility function from the pytorch library. Since these graphs were devoid of node features matrix, we set the attribute for each graph manually using the node features we generated previously. The created pytorch graphs were then put together in a list and loaded into the model using the pytorch dataloader.

## Experiments: GCN

For GCN we created a Graph Convolutional Network having 6 total layers. The input layer was a GCNConv layer. The input dimensions of this layer were the dimension of the node feature matrix by 64. This was followed by 2 GCNConv layers with normalization layers in between each of them. The final output layer was a linear layer which output one of two predicted classes.

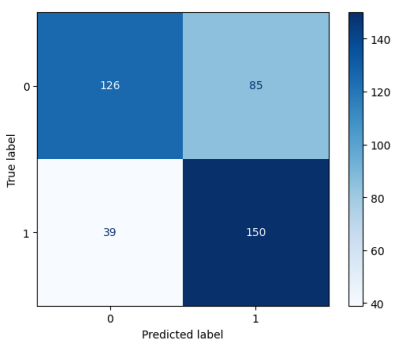
The parameters used for GCN were:
Batch size = 64,
Epochs = 75,
Optimizer : Adam,
Pooling Layer : Mean Pooling Layer.
Learning Rate : 0.008.

Following are the results for the experiments using GCN.

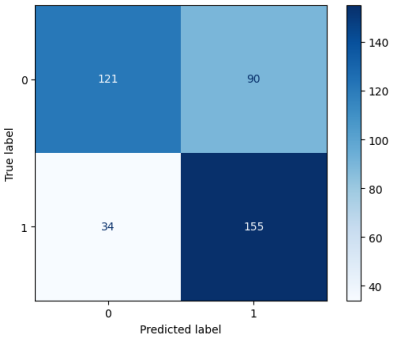
| Embeddings used | Accuracy | F1 score for class 0 | F1 score for class 1 |
|---|---|---|---|
| Centralities | 0.69 | 0.67 | 0.71 |
| GraphWave | 0.69 | 0.69 | 0.69 |
| Role2vec | 0.68 | 0.67 | 0.69 |

The confusion matrices for each of the embedding techniques used are listed below.

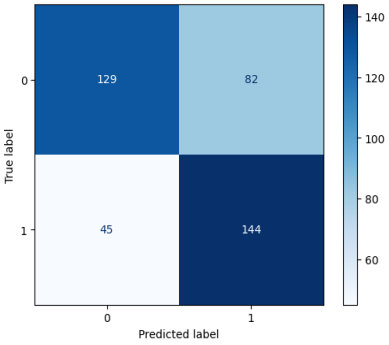
Centralities




GraphWave




Role2vec




# Experiments: GAT

For the GAT model, we created a Graph Attention Network with a total of 4 layers. The input layer was a GATConv layer with 2 input channels and a specified number of hidden channels (hidden_channels = 128) and 4 attention heads. This was followed by three additional GATConv layers, each taking the output of the previous layer as input. The final layer was a linear layer that outputted one of two predicted classes.

The parameters used for training the GAT model were as follows:

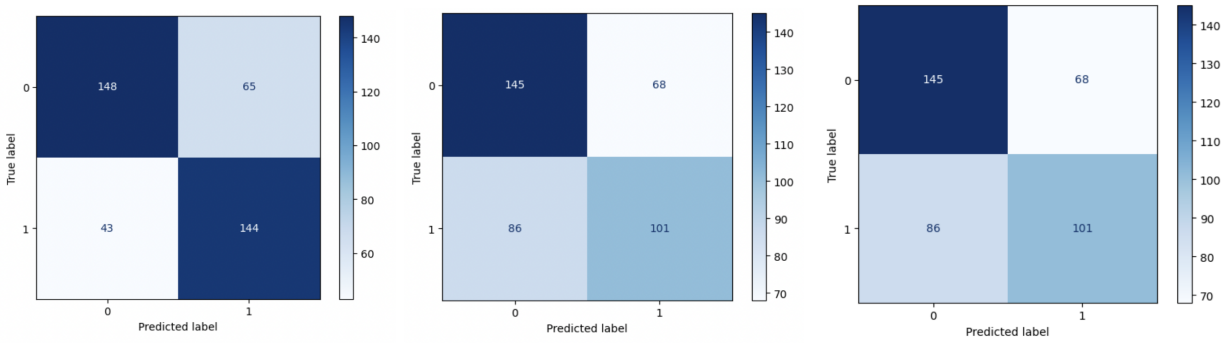
Batch size: 64
Epochs: 75
Optimizer: Adam
Learning Rate: 0.008

Following are the results for the experiments using GAT.

| Embeddings used | Accuracy | F1 score for class 0 | F1 score for class 1 |
|---|---|---|---|
| Centralities | 0.73 | 0.71 | 0.75 |
| GraphWave | 0.60 | 0.63 | 0.61 |
| Role2vec | 0.61 | 0.65 | 0.57 |

The confusion matrices for each of the embedding techniques used are listed below.

Centralities GraphWave Role2vec




## Conclusions

By using different features like density and node centralities, as well as graph embedding techniques which extracted different structural features of the graphs, we were able to achieve great results on the limited dataset. Graph Embeddings proved to be the best features for both types of models, traditional and neural networks. Centralities were also equally good features for the neural networks. The best model was found to be GAT, trained on centralities as node features. It achieved the highest accuracy of 73% and the highest F1 scores of 0.71 and 0.75.

If the dataset was more detailed, with more information about nodes and edges, it might have been possible to get higher classification accuracy. Ensemble models may also be suitable for usage and capable of producing greater outcomes in the future.

# References

[1] Twitch Ego Nets. *SNAP.* (n.d.). Available at: *http://snap.stanford.edu/data/twitch_ego_nets.html*

[2] *Datasets* (2023) *TUDataset*. Available at: https://chrsmrrs.github.io/datasets/docs/datasets/

[3] T. Ma, H. Wang, L. Zhang, Y. Tian, and N. Al-Nabhan, 'Graph classification based on structural features of significant nodes and spatial convolutional neural networks', *Neurocomputing*, vol. 423, pp. 639–650, 2021.

[4] B. Rozemberczki, O. Kiss, and R. Sarkar, 'Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs', in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, 2020, pp. 3125–3132.

[5] B. Rozemberczki and R. Sarkar, 'Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models', *arXiv*, 2020.

[6] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, 'graph2vec: Learning Distributed Representations of Graphs', *CoRR*, vol. abs/1707.05005, 2017.

[7] N. K. Ahmed et al., 'Role-Based Graph Embeddings', IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 5, pp. 2401–2415, 2022.

[8] C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec, 'Learning Structural Node Embeddings via Diffusion Wavelets', in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, London, United Kingdom, 2018, pp. 1320–1329.