# A Graph-based Approach to Classifying GitHub Developers: Web Developer vs. Machine Learning Developer

Annabelle LUO
MSc DSBA 2022-2023
ruijie.luo@student-cs.fr

Konstantina TSILIFONI
MSc DSBA 2022-2023
konstantina.tsilifoni@student-cs.fr

Namrata TADANKI
MSc DSBA 2022-2023
namrata.tadanki@student-cs.fr

Yasmina HOBEIKA
MSc DSBA 2022-2023
yasmina.hobeika@student-cs.fr

**Link to GitHub Repository:** https://github.com/NamrataTadanki/A-Graph-based-Approach-to-Classifying-GitHub-Developers

## Abstract

*Node classification in graphs is an important task in performing Social Network Analysis to identify influential nodes and understand the complex behavior of social networks. In our project, we explore the performance of different classifier algorithms like Logistic Regression, Random Forest, and XGBoost to perform binary node classification on graphs using graph-based features as well as node embeddings extracted using DeepWalk and Node2Vec. We also explore the performance of Graph Convolutional Networks and Graph Attention Networks to solve the same problem as they can better capture the structural information of the graph. Results demonstrate that the logistic regression model outperforms random forest and XGBoost and embeddings extracted from Node2Vec gave more promising results. Graph Neural Networks, although did not generate the optimal results, have a lot of potential and future scope.*

## 1. Introduction / Motivation

Graphs have become a popular tool for representing complex problems in a variety of fields because of their ability to capture the relationships and interactions between different entities in a complex system. Among the many graph-based machine learning tasks, node classification has emerged as a crucial problem in recent years.

Node classification is the task of assigning labels or categories to nodes in a graph based on their attributes and the attributes of their neighboring nodes. Node classification is important to understand the structure of the graph network, predicting node behavior, and gaining insight into the complex behavior of a large graph. This task has a wide range of applications, especially in the context of social network analysis.

In social network analysis, node classification can be used to identify influential nodes and communities in the network and is a powerful tool for understanding the structure and dynamics. It provides valuable insights for applications like marketing influence maximization, fraud detection, and recommendation systems.[1]

## 2. Problem Definition

Our project aims to perform a binary classification problem to classify developers on the GitHub social network as either web developers or machine learning developers. The GitHub social network can be represented as a graph network, where each node represents a developer on the platform and the edges represent mutual follower relationships between them.

Given a graph dataset of 37700 developers on the GitHub social network, which was collected by public API in 2019 sourced from Snap Stanford[2], where each node represents developers who starred in at least 10 repositories and an edge between two users indicates that they follow each other on GitHub. The dataset also includes the attributes of each user, such as their number of followers, number of repositories, and programming language preferences. There are also vertex features associated with each node giving additional information about the developer. These features were extracted based on geographical location, specific repositories in which the developer was featured, the employer, and the developer's e-mail address.

The node classification problem can be formalized as a binary classification problem. The classification aims to maximize the accuracy of the predictions which measures how many developers have been correctly identified as being machine developer or web developer.

To achieve our objective, we plan to use different techniques, including Node2Vec and SkipGram, to extract node embeddings that capture the relationships between the nodes beyond just graph features.

In addition to extracting node embeddings, we plan to use machine learning models such as logistic regression, XGBoost, and random forest to classify the nodes based on their embeddings and their inherent features. These models have been shown to be effective in similar node classification tasks and are commonly used in machine learning applications.

However, we also plan to implement more advanced Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs) to solve the node classification problem. These models take into account the local and global structure of the graph and have shown superior performance compared to traditional machine learning models in graph-based applications.

## 3. Related Work

Identifying influential nodes in social networks is an important task. Apart from conventional machine learning techniques, Graph Neural Network models are gaining popularity in solving the problem of classification. Kou et al. (2023) proposed a graph multi-head attention regression model for identifying influential nodes in social networks[3]. Existing Graph Convolution

Network (GCN)-based models treat problems as a binary classification task, which makes it less practical. However, when treated as a regression problem with a multi-head attention mechanism that considers the difference in neighbor importance during feature aggregation, the proposed model was found to outperform baseline methods.

Semi-Supervised Classification with Graph Convolutional Networks (GCNN)[4] is a graph convolutional neural network model proposed by Kipf and Welling in 2017. This model exploits spectral graph theory to generalize convolutional neural networks from regular grid-like data to graph-structured data, allowing it to learn node representations based on their local graph neighborhood. GCNNs have become a popular approach in graph-based machine learning and have been applied to various tasks such as link prediction, graph classification, and recommender systems.

While Graph Neural Networks extract their own node embeddings by aggregating information from neighboring nodes, another popular method for generating graph embeddings is DeepWalk[5]. DeepWalk generates node sequences through random walks and then applies a neural network to learn low-dimensional representations of these nodes. The embeddings are optimized to maximize the likelihood of observing the random walk sequences, which preserves the structural information of the graph.

## 4. Methodology

### 4.1 Preprocessing

In the dataset being used, there are in total 9739 ML developers and 27961 non-ML developers, indicating a highly imbalanced distribution in the nodes. Apart from this, the graph itself is quite large with 37700 nodes and 289003 edges. Therefore, we subsampled the graph to create a smaller, balanced sub-graph to address the class imbalance problem and also alleviate the heavy computation.

After sub-sampling, the training graph consists of 13000 nodes and 129505 edges with a balanced distribution of each class of developers. The density of the training set has increased to 0.0015 compared to the complete graph. There are however presence of degree 0 nodes, which are mostly developers who are machine developers. We decided to keep these nodes so as to preserve useful information.

In addition to the training graph, we also created a testing graph consisting of 13,000 nodes and 129,505 edges. However, unlike the training graph, the testing graph is imbalanced, with only 25% of the nodes identified as machine learning developers. This imbalance allows us to evaluate how well the trained model generalizes to new, unbalanced datasets. By testing the model on this imbalanced testing graph, we assess its ability to accurately classify nodes in the presence of class imbalance.

## 4.2 Feature Extraction and Engineering

### 4.2.1 Graph-based Features

We implemented several different graph features to help better classify the nodes. These include:

1. Degree Centrality: This measures the number of edges connected to a node.
2. Eigenvector Centrality: This measures the importance of a node based on the importance of its neighbors.
3. PageRank: This measures the importance of a node based on the importance of its incoming edges.
4. Clustering Coefficient: This measures the extent to which a node's neighbors are also connected to each other.
5. Betweenness Centrality: This measures the extent to which a node lies on the shortest paths between other nodes in the network.

However, among the aforementioned features, betweenness centrality takes extremely long (~3-4 hours) to compute, and did not significantly improve the results. Therefore, we used degree centrality, eigenvector centrality, page rank and clustering coefficient as additional features.

### 4.2.2 Subsampling for Features (Limitations)

To compute betweenness centrality and clusters, we faced the challenge of high computational time, which could impact the performance of our algorithms. To address this issue, we attempted to use a bootstrapping approach to subsample the graph and calculate betweenness centrality on a subset of nodes. However, despite multiple attempts, the results did not significantly improve the algorithm's performance. This suggests that the bootstrapping approach was not effective in capturing important graph characteristics.

We also explored sub-sampling based on node similarity, but this further reduced the number of nodes, potentially leading to the loss of valuable information.

### 4.2.3 Community-detection based Features

To identify nodes that are similar to each other and extract some properties of connectivity from the graph, we implemented the Greedy Modularity Algorithm for community detection. This algorithm successfully generated 265 communities that served as a feature to indicate the community to which each node belongs. However, due to the significant computation time of up to 3 hours, we opted to use clustering coefficient instead.

Nodes with high clustering coefficients are more likely to belong to densely connected communities within the network and can provide useful insights into the class of the node. Thus, we leverage this feature to supplement our analysis and potentially improve the accuracy of our classification models.

### 4.2.4 Principal Component Analysis

Since the node information has 4004 columns, we attempted to perform Principal Component Analysis (PCA) to reduce the dimensionality and generate new features that capture the most significant information.

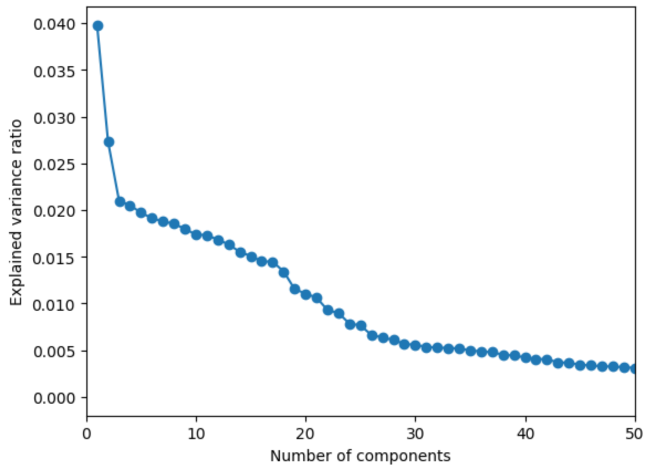

Fig.1. Explained variance vs number of components

From the plot we find the elbow point at around 25 components, which explains more than 80% of the variance. We opted to use these 25 principal components instead of the original 4004 features. The resulting models maintained comparable performance while significantly reducing the execution time by more than half.

### 4.2.5 Helper Functions

We created several helper functions to facilitate the workflow and readability of our code. These include:

1. Compute_network_characteristics: It describes various characteristics of a preferred graph, including number of nodes, number of edges, minimum degree, maximum degree, mean degree, median degree, graph density, type of graph.
2. Grid search function: It performs grid search for the model with defined parameter gird, given the train features and train labels.
3. Prediction function: It produces a classification report. It uses the trained model to make predictions on the test set and then evaluates the performance of the model by computing various metrics such as accuracy, F1 score, and area under the ROC curve. The function also plots the ROC curve and precision-recall curve, and computes the optimal threshold for classification using the test set. Finally, the function generates a confusion matrix,

displays a classification report, and plots the feature importance scores for logistic regression models.

## 4.3 Algorithms

### 4.3.1 Logistic Regression

We first implemented a basic logistic regression model with the parameter multi_class='ovr'. This specifies the multi-class strategy being used to perform the classification. In our case, it is set to a one-vs-rest (ovr) strategy, which means that the algorithm treats each class as a binary classification problem, and then combines the results.

We implemented the model on two types of features - the inherent node attributes in the dataset and using the computed graph-based features.

### 4.3.2 Random Forest

We also implemented a Random Forest Classifier that uses the inherent node feature vector as well as the computed graph-based feature vector as inputs to the model. The classifier was instantiated with 100 tree estimators, gini criterion to measure the quality of the split and other default parameters. We attempted to perform grid-search to optimize the hyperparameters but it was computationally expensive.

### 4.3.3 XGBoost

Similar to the above approach, we also implemented a XGBoost classifier. We applied grid-search to try and optimize the important hyperparameters like learning rate, number of estimators, and maximum depth. The XGBoost classifier was instantiated with a learning rate of 0.1, 100 estimators and a maximum depth of 3 to perform the node classification problem.

### 4.3.4 Node2Vec and DeepWalk with Logistic Regression

We also tried to use Node2Vec and DeepWalk to extract node embeddings and combine them with logistic regression to perform node classification. The first step was to use Node2Vec or DeepWalk to generate embeddings for each node in the graph.

To obtain the DeepWalk embeddings, we used the Word2Vec implementation from the gensim library to learn representations of nodes in a graph. The parameters for the model were set as follows: the number of random walks to perform is 300, the length of each walk is 8, the embedding size is 32, and the window size is 6.

To obtain the Node2Vec embeddings, the Node2Vec library was used. The dimensions of the embedding vectors are set to 32, with each random walk having a length of 8 and generating a total of 200 walks. The workers parameter is set to 1 for Windows compatibility.

The resulting node embeddings were then used as features in a logistic regression model to predict the class label of each node.

### 4.3.5 Graph Neural Networks (GNNs)

We implemented 2 types of graph neural network models - Graph Convolutional Network (GCN) and Graph Attention Network (GAT). In order to implement these networks, we first converted our NetworkX graph into a DGL graph representation and generated train, validation, and test masks.

For the Graph Convolutional Network, we introduced a GCN class that takes the number of input features, 16 hidden channels, and number of output classes as the input parameters. The GCN model has four graph convolution layers, each followed by a ReLU activation function. The GraphConv layers perform a linear transformation of the input features using the graph structure to propagate information between nodes. In the forward pass, the model applies each graph convolution layer followed by the activation function to the input node features. The output of the final layer is the predicted class label for each node in the graph. The model is trained by minimizing cross-entropy loss and Adam optimizer with a learning rate of 0.005.

Graph Attention Networks are built on the idea and try to learn the importance of each neighbor based on the Attention mechanism. We introduced a GAT class that takes the number of input features, 64 hidden channels, and number of output classes, and number of attention heads as the input parameters. The model was instantiated with 8 attention heads, cross entropy loss, Adam optimizer, and a learning rate of 0.01.

## 5. Evaluation

To evaluate the performance of our algorithms, we adopted two distinct approaches depending on the algorithm being evaluated. The first approach centered around the assessment of the basic classifier models; logistic regression, random forest and XGBoost, while the second approach was focused on the implemented Graph Neural Networks.

Although we used a balanced sub-graph to perform the training, in the validation set we maintained the same imbalance ratio as that of the original graph and included nodes and edges that were not part of the training set. This approach allowed us to tackle both the imbalanced data and the size of the data issue, which proved to be computationally demanding given the resources at our disposal.

### 5.1 Evaluation of classifier models

The main metric of evaluation we considered was the F1-score since it is the ideal evaluation metric in the case of imbalanced data. However, we used other evaluation metrics to assess the model's performance:

1. Receiver Operating Characteristic (ROC) curve
2. Area Under the ROC Curve (AUC)
3. Precision-Recall Curve
4. F1-score
5. Confusion Matrix

To compute the ROC curve, we first predicted the probabilities of the test and train data using the trained model. Then we plotted the ROC curve, and computed the area under the curve (AUC) for the test data. A higher AUC indicates better performance of the model.

We also plotted a precision-recall curve and calculated the F1-score, which is the harmonic mean of precision and recall, and indicates how well the model balances between precision and recall. We reported the best threshold for the F1-score and the corresponding F1-score and presented with the following visualization.

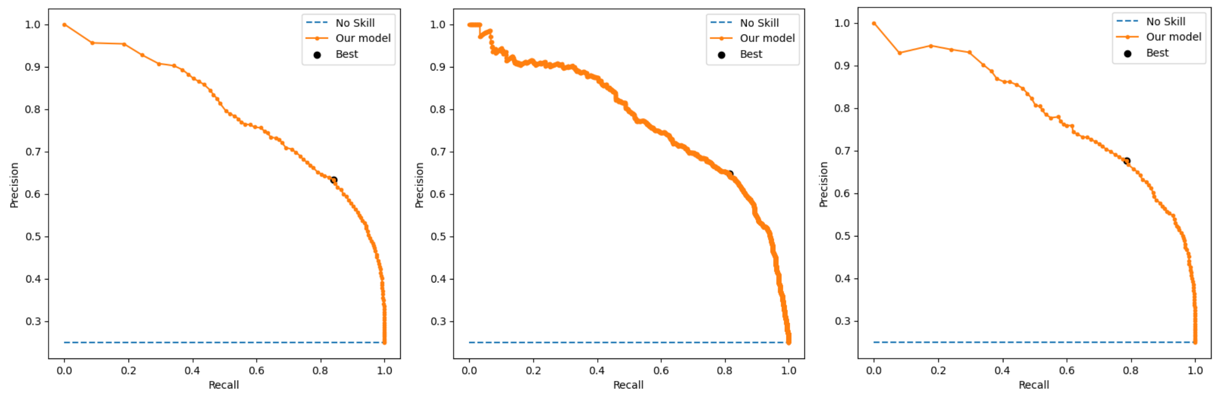

Fig.2. Precision-Recall curve for Logistic Regression, Random Forest, and XGBoost respectively

We also computed the confusion matrix for each algorithm and based on the confusion matrix, we computed the accuracy and F1-score for the training and test data. With this approach we could see exactly where our models show weakness in its predictions.

Finally, using a loop we optimized the threshold of the model, by iterating over a range of possible thresholds to obtain the best accuracy and F1-score for the test data. In the end, the threshold resulting in the highest F1-score on the testing set was kept and the final predictions were made using that threshold.

### 5.2 Evaluation of Graph Neural Networks

To evaluate the GNNs, we used cross-entropy loss and accuracy.

## 6. Conclusion

The project used various graph machine learning methods, including logistic regression, random forest, XGBoost, DeepWalk, Node2Vec and Graph Neural Networks  to classify Github users. Principal components were used to reduce computational time without sacrificing performance. The precision score for all three algorithms was 0.86, with logistic regression slightly outperforming the others with an F1 score of 0.84. Node2Vec outperformed DeepWalk with a precision and F1 score of 0.82.

However, the Graph Neural Networks did not perform so well with both the GCN and GAT producing a test accuracy of around 62%. We believe that the poor performance was due to our inability to tune hyperparameters due to heavy computational requirements and limited resources available at our disposal.

For the future, we believe that different sampling techniques like GraphSMOTE or over/undersampling the graph might help us obtain better results. It would also be interesting to implement feature selection techniques to obtain better results.

## 7. References

*[1] Goldenberg, D. (2021). Social network analysis: From graph theory to applications with python. arXiv preprint arXiv:2102.10014.*

*[2] Leskovec, J., & Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. Retrieved from https://snap.stanford.edu/data/github-social.html*

*[3] Kou, J., Jia, P., Liu, J., Dai, J., & Luo, H. (2023). Identify influential nodes in social networks with graph multi-head attention regression model. Neurocomputing, 530, 23-36. https://doi.org/10.1016/j.neucom.2023.01.078*

*[4] Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. Retrieved from https://arxiv.org/abs/1609.02907 (Accessed February 28, 2023).*

*[5] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). DeepWalk: Online learning of social representations. Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, 701-710.*