ALGORITHMIC PREDICTION AND INTELLIGENT GRAPH ANALYSIS OF GITHUB NETWORK DYNAMICS USING MACHINE LEARNING

SOCIAL NETWORKS AND INFORMATICS

FINAL PROJECT REPORT

By

Suryakumar Selvakumar

110975883

1. Introduction

In my project, the objective is to predict future links/relationships between users in a large GitHub Network and analyze the network dynamics within this complex social network using a number of Machine Learning techniques, metrics, and plots. Various ML algorithms such as Logistic Regression, Random Forest, SVM, and XGBoost will facilitate the analysis of user interactions and relationships in the GitHub Network, which will be represented by a Directed Graph. NetworkX will help explore various aspects of network analysis such as node centrality and node degree, and SHAP values will help find feature dependence, feature importance, and so on. These tools will help gain rich insights about the GitHub Network and how its structure influences relationships between users.

Human societies operate as a collective in the form of social systems and structures. As such, there are endless numbers of social networks all around us, and they inevitably affect how we connect with other people and how information travel between us. Prediction of human behavior and understanding of the dynamics of these networks can be greatly helpful in many ways. A good example is a Social Media Platform, where content or friends suggested to a user can be made more relevant if the user and the network they operate in are analyzed and interpreted. Another example is Cybersecurity, where network insights can help identify malicious users or suspicious behavior. This project is a similar endeavor in which machine learning and graph analysis are used to peek under the veneer of a GitHub Social Network, revealing crucial patterns and facilitating the prediction of user interactions.

2. Dataset

The GitHub Social Network Dataset used for this project was curated by the Multi-Scale Attributed Node Embedding (MUSAE) Project by using the data fetched from GitHub's public API back in 2019. It was obtained for usage in this project from the Stanford Large Network Dataset Collection which collected this dataset as part of the Stanford Network Analysis Project (SNAP) initiative.

The dataset represents a network of GitHub developers where the nodes in the graph represent developers who have starred a number of 10 repositories and edges in the graph represent the follower relationships between the developers. The location, employer, email-address, and repositories starred were used to derive the graph vertex features.

Find the statistics of the GitHub Social Network below:

Graph Features	Feature values
Directed	No.
Node features	Yes.
Edge features	No.
Node labels	Yes. Binary-labeled.
Temporal	No.
Nodes	37700
Edges	289003
Density	0.001
Transitivity	0.013

Explanation of the features and their values:

- *Directed* – The value of no indicates that graph is undirected. This makes sense because follower relationships between developers in the graph are mutual, meaning both developers follow each other. If the graph contained only nodes (developers) with the property that showed only their following data, then the network would be a fully directed graph with the directed edges leading to each developer's following.

- *Node features* – The value of yes indicates that the nodes in the graph are derived by means of using specific relevant features. These features as given above are the email addresses, employers, starred repositories, and locations of the developers. These features could be used for binary node classification, target prediction or analysis of the network.

- *Edge features* – The value of no indicates that the edges in the graph do not have additional features. In this case, since the edges represent the mutual follower relationships, we can conclude that the edges only mean that the developers follow each other and nothing more.

- *Node labels* – The value of yes indicates that each node in the graph has an associated label assigned to it and the value Binary-labeled indicates that the labels are there are two classes of nodes in the graph. In this case, these classes are Machine Learning Developers and Web Developers.

- *Temporal* – The value of no indicates that the dataset does not include any features that are related to time, for example, details such as when two developers followed each other are not included.

- *Nodes* – The value of 37,700 indicates that the graph contains 37,700 nodes, i.e., developers.

- *Edges* – The value of 289,003 indicates that the graph contains 289,003 edges, i.e., the mutual follower relationships that exist between the developers.

- *Density* – Density denotes the ratio of number of actual edges in a graph to the number of maximum possible edges. The value of 0.001 indicates that the graph is very sparsely connected.

- *Transitivity* – It can be interpreted as the clustering tendency of the graph, measuring the likelihood of a node's neighbors also being connected to each other to form a triadic closure. The value of 0.013 indicates that the number of triadic closures are quite low in the network. Another way to interpret it is that there could be 99.987% more triadic closures in the network.

3. Methods of Implementation

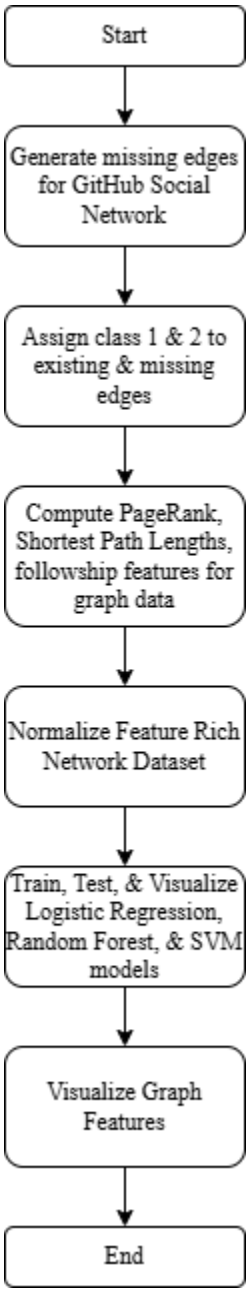
A GitHub Social Network Dataset, with 37,700 nodes corresponding to developers and 289,003 edges corresponding to follower relationships, curated by the Multi-Scale Attributed Node Embedding (MUSAE) Project was obtained from the Stanford Large Network Dataset Collection. The initial step involved pre-processing the raw GitHub Network data to build a directed graph. This graph was then used to generate missing links/edges, which were then used alongside the existing edges to introduce a binary classification problem. Various NetworkX functions were used on the graph to calculate centrality measures such as Page-Rank and Edge Betweenness, find shortest path lengths, and extract followship features for all the source and target nodes. Using these features, training of the aforementioned ML models were carried out to predict the missing links/edges, thereby predicting user relationships. An in-depth analysis on the Social Network was performed using Network plots, Heatmaps, Scatterplots, Histograms, etc., to visualize insightful hidden details such as Edge/Node Betweenness, Degree Distribution, Feature Correlation, Feature Distribution, Feature Relationships, and so on. Additionally, SHAP (SHapley Additive exPlanations) values were computed, which facilitated an even more in-depth analysis of the graph features by way of Feature importance, influence, and dependence, found using Force Plots, Summary Plots, Dependence Plots, etc. This approach gave me insights into the most influential network features and how different models perform on social network data.

Methods of implementation can be divided into:

1. Dataset Preprocessing
2. Feature Extraction
3. Dataset Normalization

4. Model Training & Evaluation
5. Social Network Analysis

Find the flow-chart that visualizes the methodology below:

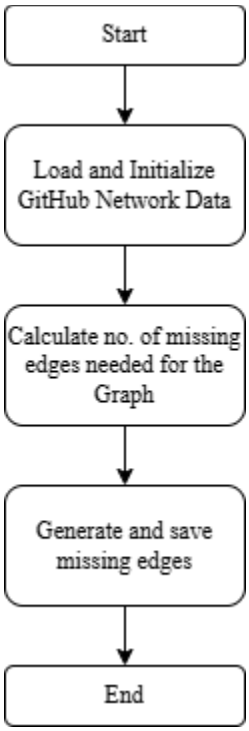


3.1. Dataset Preprocessing

The GitHub Network dataset contains 289,003 edges connecting 37,700 nodes. In order to perform link prediction, we need a negative class that represents the missing edges in the network.

Then, the existing edges and missing edges together can be combined to form a binary classification problem. However, the missing edges needed for this task do not exist in the original GitHub network dataset. Fortunately, they can be generated using a combination of NetworkX functions such as `number_of_nodes()`, `shortest_path_length()`, and `NetworkXNoPath()`. Thus, a proportion of missing edges equal to existing edges are generated and saved for addition to the graph later.

Flowchart visualization of the section:



3.1.1. Load and Initialize GitHub Network data

Here, the GitHub data is loaded and initialized to be used to generate missing edges. Programmatic implementation:

```
git_net_data = open('./data/musae_git_edges.csv')

next(git_net_data)

graph = nx.read_edgelist(git_net_data,
create_using=nx.DiGraph(), delimiter=',', nodetype=int)
```

The GitHub Network data are loaded in the `git_net_data` variable. The `next()` method skips the first line in the loaded dataset and moves the current position to the second line in the data. We do this to skip the header which contains the column names.

NetworkX's `read_edgelist()` returns a directed graph from the given `git_net_data` edges, created using `DiGraph()` to ensure the returned graph is a directed graph, with `nodetype=int` ensuring the nodes are represented as integers.

```
graph = nx.relabel.convert_node_labels_to_integers(graph,
first_label=0)
```

The function `convert_node_labels_to_integers()` converts all the node labels to integers starting from 0, reaching all the way to 37,699 nodes. This helps in the missing edges generation process as later we fetch two random nodes from the range of the graph's nodes, so we ensure that the random numbers fetched would correspond to actual nodes. This also acts as a simple procedure to ensure consistency and increase readability for the training dataset which will soon be derived from this data.

```
print(graph)
print(nx.is_directed(graph))
print(nx.density(graph))
DiGraph with 37700 nodes and 289003 edges
True
0.0002033439101558534
```

General graph info about nodes and edges, its directed property value, and density are printed. Our graph is in fact a directed graph with 37,700 nodes and 289,003 edges. The density of the graph is calculated as follows,

Density = Actual Edges / Total no. of edges

= 289,003 / (37,700 x (37,700 - 1)) = 0.0002033...

This extremely low value of 0.02% indicates that the graph is extremely sparse, i.e., loosely connected.

3.1.2. Calculate no. of missing edges needed for the Graph

The missing edges that are to be generated to create a binary classification problem, need to be in the correct proportion. So we use certain NetworkX functions to find the number/proportion of missing edges to be generated. Programmatic Implementation:

```
N = graph.number_of_nodes()
E = graph.number_of_edges()
```

We retrieve the number of nodes and edges of the graph using NetworkX's `number_of_nodes()` & `number_of_edges()` function.

```
total_possible_edges = N * (N - 1) / 2
```

The total number of possible edges in a directed graph is given by N * (N - 1) / 2 and so we use it to calculate the number for our graph.

```
proportion_existing = E / total_possible_edges
```

We calculate the proportion of existing edges by dividing the number of existing edges by the total number of possible edges.

```
suggested_missing_edges = total_possible_edges *
proportion_existing
print(suggested_missing_edges)
289003
```

To perform link prediction, we must ensure that we have an equal amount of existing and missing edges representing a positive and negative class in our dataset to train our model and have them predict the correct classes. Otherwise, any ML models trained on this data would be biased toward one class. Therefore, we set the no. of missing edges to be 289003, which is the same as the no. of existing edges.

3.1.3. Generate and Save Missing Edges

Now, we generate the missing edges for the graph through a set of specific steps. Two nodes are selected at random from the graph, and checked whether there is already an edge between them, if not then we check if the edge to be added is trivial or non-trivial, with the edge getting added if it is non-trivial or if it is fully missing. This process is done for all the generated edges and they are added to a set to avoid duplicate values. Finally, we save the missing edges set. Programmatic Implementation:

```
start_time = time.perf_counter()

edge_dict = {edge: 1 for edge in graph.edges()}

missing_edges_set = set()

max_attempts = 1000000

attempts_made = 0
```

We start a timer to count the time it takes to generate the missing edges. We fetch the existing edges from the graph and store them in a dictionary for faster lookup. A

`max_attempts` variable with the number of attempts to be made for the missing edges generation is setup to avoid a potential infinite loop and finally an `attempts_made` variable to keep track of the no. of attempts that have been made.

```
while len(missing_edges_set) < suggested_missing_edges and
attempts_made < max_attempts:

    if attempts_made % 50 == 0:

        print(f"Iteration: {attempts_made}, Length of
Missing edges:
     {len(missing_edges_set)}, Edges left:
{suggested_missing_edges -    len(missing_edges_set)}")

    attempts_made += 1
```

For the loop condition, the while loop will run the number of edges in the missing edges set is lesser than the suggested no. of missing edges and if the attempts that have been made are lesser than the max_attempts that were set. To monitor the progress of the generation process, every 50 iterations, we print the iteration number, the current number of missing edges that have been generated, and the missing edges that are left to generate.

```
    node_a, node_b =
random.sample(range(graph.number_of_nodes()), 2)

    if (node_a, node_b) not in edge_dict and (node_b,
node_a) not in edge_dict:

        try:

            if nx.shortest_path_length(graph,
source=node_a, target=node_b) > 2:

                missing_edges_set.add((node_a, node_b))

        except nx.NetworkXNoPath:

            missing_edges_set.add((node_a, node_b))
```

First, we generate two random nodes that are within the range of graph by getting its number of nodes using `number_of_nodes()` method and getting random numbers from within the range of total number of nodes using `random.sample()` and `range()`. Then, we check if both edge combinations between the two returned nodes, `node_a => node_b` and `node_b => node_a` already exists. If a direct edge does

not exist, we check if the shortest path length from `node_a` to `node_b`, found using `nx.shortest_path_length()` function is greater than 2, if it is, then we add that as a missing edge to the `missing_edges_set`. If the previous condition is false, then we also check if there is no path that exists between the 2 nodes, and if yes, it is added to the set. These checks help to avoid trivial connections between two nodes as direct neighbors or nodes that are just one step away do not contribute much to the link prediction task as these are obvious predictions and the model does not learn much from the missing edges data.

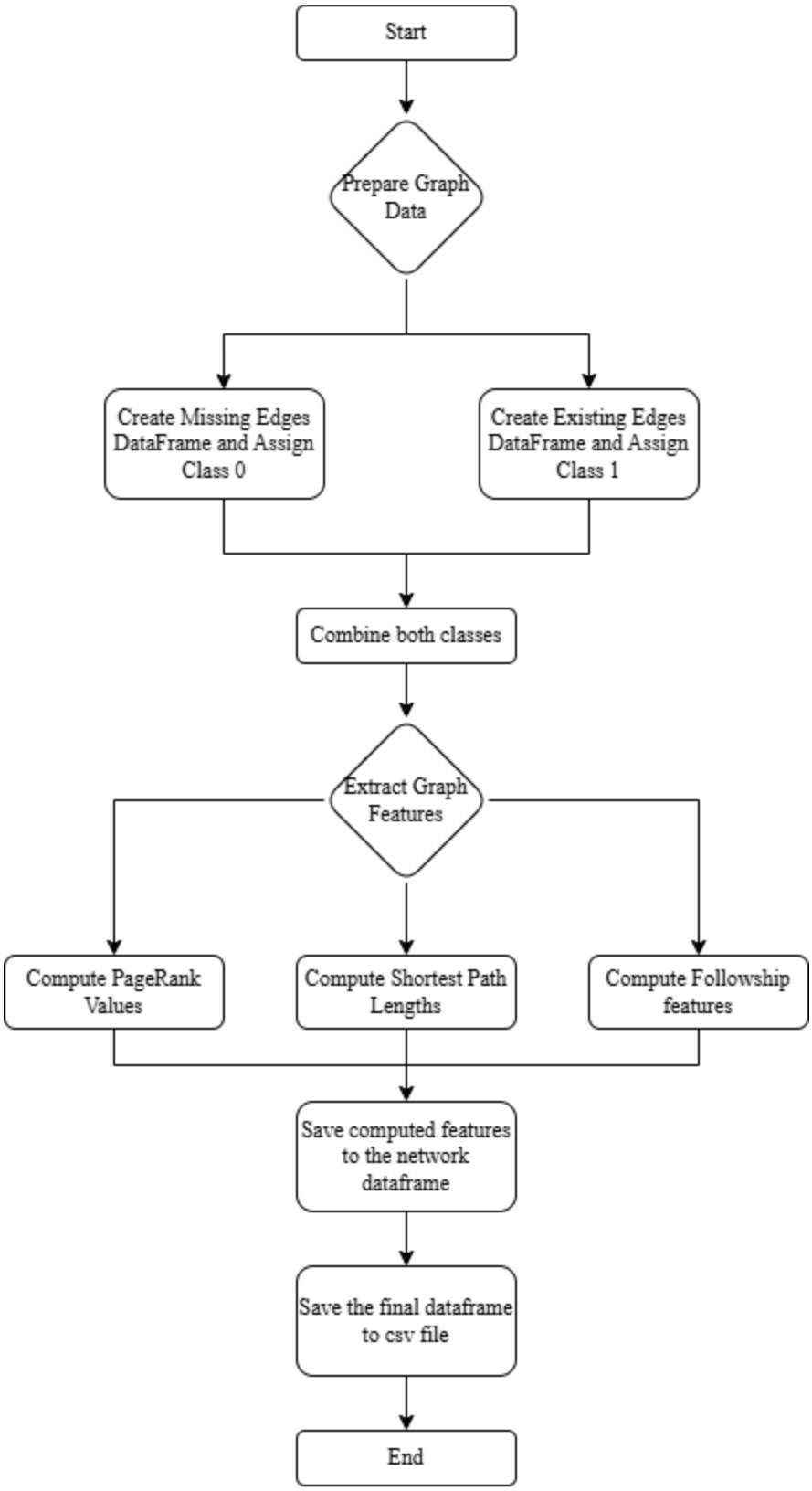
```
loop_ending_time = time.perf_counter()

print('Time taken:', loop_ending_time - start_time,
'Seconds')

with open('./data/github-net-missing-edges-set.p', 'wb') as
z:

    pickle.dump(missing_edges_set, z)

len(missing_edges_set)

289003
```

The timer is stopped after the loop finishes running, meaning the missing edges have been generated. The time taken is printed and the set containing the missing edges is saved in a pickle file. Finally, checking the number of missing edges in the set returns the number as 289003 which means an equal number of missing edges have been successfully generated.

3.2. Feature Extraction

This step involves generating features for the network that are to be used to train the ML models for the link prediction task. The models will learn whether the link (edge) between the nodes exists (positive class) or if it is missing (negative class) from these features. Before feature generation however, we combine the generated missing edges and the existing edges and assign classes 0 and 1 respectively. Then once our dataset is ready, the features such as PageRank, shortest path, predecessors (followers), successors (following), and mutual predecessors / successors are computed and saved.

Find the flow-chart that visualizes this section:


Start
Prepare Graph Data
Create Missing Edges DataFrame and Assign Class 0
Create Existing Edges DataFrame and Assign Class 1
Combine both classes
Extract Graph Features
Compute PageRank Values
Compute Shortest Path Lengths
Compute Followship features
Save computed features to the network dataframe
Save the final dataframe to csv file
End

3.2.1. Prepare Graph Data

The graph data is prepped for feature extraction in this step. This involves splitting the data into their appropriate classes, with the node pairs with existing edges being assigned a class of 1 and the node pairs with missing edges being assigned a class of 0. After class assignment, both classes are combined. Programmatic Implementation:

```
file_path = open('./data/github-net-missing-edges-set.p',
"rb")

missing_edges_set = pickle.load(file_path)

type(missing_edges_set)

set
```

We load the missing edges set

```
df_negative_edges = pd.DataFrame(list(missing_edges_set),
columns=['Source', 'Target'])
print(df_negative_edges.shape)
df_negative_edges.head(3)
(289003, 2)
     Source    Target
0    21385     32676
1    26650     27849
2    19264     17030
```

We convert the missing edges set into a pandas dataframe and assign the source and target nodes to their own columns. The shape and structure of the dataframe is printed to ensure that it is of the proper format.

```
df_positive_edges =
pd.read_csv('./data/musae_git_edges.csv')
df_positive_edges = df_positive_edges.rename(columns =
{'id_1':'Source','id_2':'Target'})
df_positive_edges = df_positive_edges.drop_duplicates()
print(df_positive_edges.shape)
df_positive_edges.head(3)
```

Similarly, the existing edges are loaded into a dataframe, source and target columns are created, and the format of the data is printed for verification.

```
df_positive_edges['Class'] = 1
df_positive_edges.head(7)
```

```
     Source    Target    Class
0    0         23977     1
1    1         34526     1
2    1         2370      1
3    1         14683     1
4    1         29982     1
5    1         21142     1
6    1         20363     1
```

The existing edges are assigned a Class of 1 which can be seen above from the printed data.

```
df_negative_edges['Class'] = 0
df_negative_edges.head(7)
     Source    Target    Class
0    21385     32676     0
1    26650     27849     0
2    19264     17030     0
3    26544     20283     0
4    27487     37391     0
5    27856     20484     0
6    24158     26785     0
```

The missing edges are assigned a Class of 0 which can be seen above from the printed data.

```
df_combined = pd.concat([df_positive_edges,
df_negative_edges])
print(df_combined.shape)
df_combined.iloc[289001:289006]
(578006, 3)
          Source    Target    Class
289001    25879     2347      1
289002    25616     2347      1
0         21385     32676     0
1         26650     27849     0
2         19264     17030     0
```

Both classes of data are combined. The combined dataframe now has 578006 samples with the Source, Target, and Class columns containing both classes of data in contiguous format.

3.2.2. Extract Graph Features

After dataset preparation, we move on to feature computation. We use our dataset to create a directed graph. Using the graph and various NetworkX functions, we compute features such as

PageRank, Shortest Path lengths, follower/following counts and so on. These features are added to the same network dataset and saved for further use. Programmatic Implementation:

```
graph = nx.from_pandas_edgelist(df_combined[['Source',
'Target']], source='Source',  target='Target',
create_using=nx.DiGraph())
```

Similar to the first step in 3.1.1. Load and Initialize GitHub Network data, we create a directed graph using our network data.

```
page_rank_values = nx.pagerank(graph, alpha=0.85)

df_combined['Page_Rank_Source'] =
df_combined['Source'].apply(lambda node:
page_rank_values.get(node, 0))

df_combined['Page_Rank_Target'] =
df_combined['Target'].apply(lambda node:
page_rank_values.get(node, 0))
```

We use the `pagerank()` method from NetworkX to compute the pagerank values. PageRank is a type of algorithm that measures the importance of a node in a network by evaluating its relationship with other nodes in the network by checking the connections it receives. `alpha=0.85` sets the damping factor, which adds some randomness to the pagerank values by making it so that 85% of the time, a user follows the available links but 15% of the time, they switch to a random page. This adds user behavior into consideration while computing the pagerank values making for more realistic calculations. Then we apply the function to compute pagerank values for the source and target nodes and save them in their own columns `Page_Rank_Source` and `Page_Rank_Target`.

```
def get_shortest_path(source, target):
    try:
        if graph.has_edge(source, target):
            graph.remove_edge(source, target)
            distance = nx.shortest_path_length(graph,
source=source, target=target)
            graph.add_edge(source, target)
```

```
        else:
            distance = nx.shortest_path_length(graph,
source=source, target=target)
        return distance
    except nx.NetworkXNoPath:
        return -1
df_combined['Shortest_Path'] = df_combined.apply(lambda
row: get_shortest_path(row['Source'], row['Target']),
axis=1)
```

This is the function we use to get the shortest path lengths between the source and target nodes. We make use of a try-except block to get the shortest path when it exists and return a value of -1 when there is no path. First, we check if there exists an edge between the source and target nodes, and if so, then we remove it temporarily, compute shortest path distance between source and target using `nx.shortest_path_length()`, and then re-add the edge. This is done to avoid generating shortest path values for trivial connections in the network. Now, the shortest path values will at least be greater than or equal to 2. We add another condition to get the shortest path values for source and target nodes that are not connected by an edge. Then, the except block uses `nx.NetworkXNoPath`, to check for a "no path exists" exception and return -1 if that is the case. Finally, we apply our function to the dataset containing source and target nodes to compute shortest path lengths and these values are saved to the dataframe in the column `Shortest_Path`.

```
def extract_followship_features():
    source_followers, source_following, target_followers,
     target_following, mutual_followers, mutual_following =
     [], [], [], [], [], []

    for idx, row in df_combined.iterrows():
        source_predecessors =
        set(graph.predecessors(row['Source']))
        source_successors =
        set(graph.successors(row['Source']))
```

```
            target_predecessors =
            set(graph.predecessors(row['Target']))

            target_successors =
            set(graph.successors(row['Target']))

            source_followers.append(len(source_predecessors))

            source_following.append(len(source_successors))

            target_followers.append(len(target_predecessors))

            target_following.append(len(target_successors))

            mutual_followers.append(len(source_predecessors.
            intersection(target_predecessors)))

            mutual_following.append(len(source_successors.
            intersection(target_successors)))


        return source_followers, source_following,
         target_followers, target_following,
         mutual_followers, mutual_following


followship_features = extract_followship_features()

df_combined['Source_Followers'],
df_combined['Source_Following'],
df_combined['Target_Followers'],
df_combined['Target_Following'],
df_combined['Mutual_Followers'],
df_combined['Mutual_Following'] = followship_features
```

The function to calculate followship features for the source and target nodes. The lists to store the followship features are initialized at the start of the function. We use NetworkX functions `predecessors()` and `successors()` to get the followers and following nodes for the source and target nodes and the returned nodes get saved in a set to avoid duplicates. The length of these sets are then computed using `len()` and the length values are added to their respective lists that were previously initialized. We also compute the mutual followers and following for source and target nodes using `intersection()` and add those numbers to those lists as well. At function end, all these lists containing the counts for each followship type across source and target nodes are returned. At last, we call

this function to compute all the values and save them to their respective columns in the dataframe.

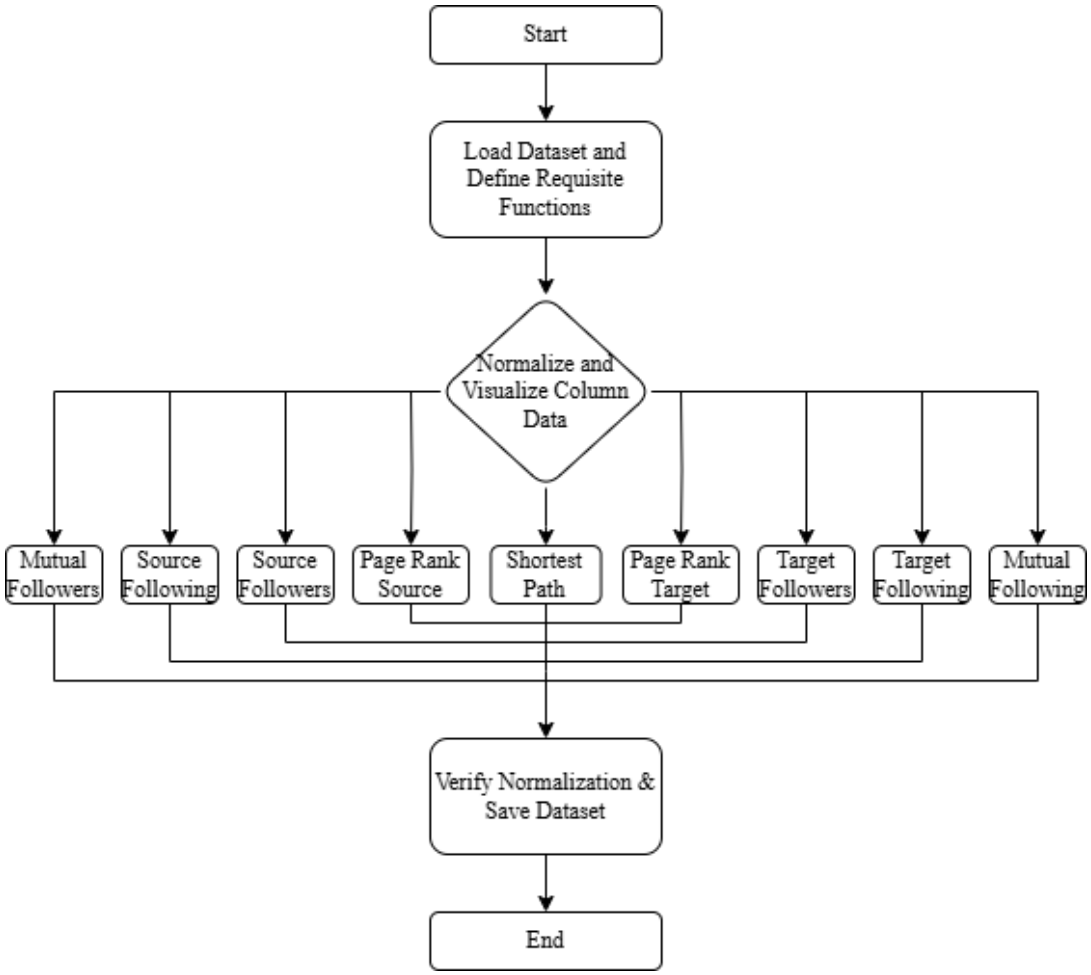
```
df_combined.head(10)
```

```
df_combined.to_csv('./data/github-full-dataset.csv')
```

The final dataframe is visualized to check if all values have been properly computed. Once verified, we then save the dataframe to a csv file for future use.

3.3. Dataset Normalization

We have all the features we need to train the ML model. However, before training can be carried out, the dataset has to be normalized. Normalization of data is a standard procedure in the Machine Learning Space that can lead to benefits such as a uniform scale of data, better model performance, bias reduction, and so on. Thus, we normalize our data, visualize the normalizations and save the normalized dataset. Find the flow-chart that visualizes this section below:



3.3.1. Load Dataset and Define Requisite Functions

To begin the normalization, we load our feature-rich dataset with newly computed features such as PageRank, Shortest Path Lengths, and Followship features. Functions needed to perform the normalization and visualize the normalized column data are defined. Programmatic Implementation:

```
df = pd.read_csv('./data/github-full-dataset.csv')

df = df.drop(columns=['Unnamed: 0'])

df.head(10)

Source Target Class Page_Rank_Source Page_Rank_Target

0      23977  1     0.000012         0.000027

Shortest_Path Source_Followers Source_Following

3             6                6

Target_Followers    Target_Following    Mutual_Followers

26                  16                  0

Mutual_Following

0
```

The previously saved dataset is loaded. A column `unnamed` appears in the dataset when saving or reloading dataframes to or from a CSV file, thus this is a precautionary step to remove it. We see a sample of the dataset once before starting the normalization process.

```
def normalize_column(data_series):

    return (data_series -
     data_series.min())/(data_series.max()        -
     data_series.min())
```

The function that takes in a column's data and normalizes it is defined. First, a data_series or a column is given as a parameter, the `min()` and `max()` functions return the minimum and maximum values in the column. In the numerator, the minimum value is subtracted from all the values in the column. In the denominator, the minimum value is subtracted from the maximum value. Then, the numerator value is divided by the denominator value. This calculation is known as Min-Max Normalization.

```
def plot_detailed_histogram(input_data, plot_title,
x_axis_label, y_axis_label='Frequency',
plot_color='skyblue'):

    plt_edge_color = "black"

    plt.figure(figsize=(10, 6))

    plt.hist(input_data, bins=50, color=plot_color,
     edgecolor=plt_edge_color)

    plt.title(plot_title)

    plt.xlabel(x_axis_label)

    plt.ylabel(y_axis_label)

    plt.grid(True, linestyle='--', alpha=0.6)

    plt.show()
```

This function is the one that will be used to plot the newly normalized data. The function takes the column data to be visualized, title of the plot, x, y – axis labels, & plot color as parameters. Matplotlib's functions are used to set the details of the plot such as the edge color, figure size, title, labels, grid properties, etc. and `show()` prints the created plot.

3.3.2. Normalize and Visualize Column Data

We apply the normalization and visualization functions to each of the dataset's columns Page Rank Source, Page Rank Target, Shortest Path, Source Followers, Source Following, Target Followers, Target Following, Mutual Followers, and Mutual Following contiguously. Programmatic Implementation:

```
df['Page_Rank_Source'] =
normalize_column(df['Page_Rank_Source'])

plot_detailed_histogram(df['Page_Rank_Source'],
'Distribution of Page Rank (Source)',   'Page Rank
(Source)', plot_color='teal')


df['Page_Rank_Target'] =
normalize_column(df['Page_Rank_Target'])
```

```
plot_detailed_histogram(df['Page_Rank_Target'],
'Distribution of Page Rank (Target)', 'Page Rank (Target)',
plot_color='purple')


df['Shortest_Path'] = normalize_column(df['Shortest_Path'])

plot_detailed_histogram(df['Shortest_Path'], 'Distribution
of Shortest Path', 'Shortest Path', plot_color='red')


df['Source_Followers'] =
normalize_column(df['Source_Followers'])

plot_detailed_histogram(df['Source_Followers'],
'Distribution of Source Followers', 'Source Followers',
plot_color='blue')


df['Source_Following'] =
normalize_column(df['Source_Following'])

plot_detailed_histogram(df['Source_Following'],
'Distribution of Source Following', 'Source Following',
plot_color='green')


df['Target_Followers'] =
normalize_column(df['Target_Followers'])

plot_detailed_histogram(df['Target_Followers'],
'Distribution of Target Followers', 'Target Followers',
plot_color='orange')


df['Target_Following'] =
normalize_column(df['Target_Following'])

plot_detailed_histogram(df['Target_Following'],
'Distribution of Target Following', 'Target Following',
plot_color='magenta')
```

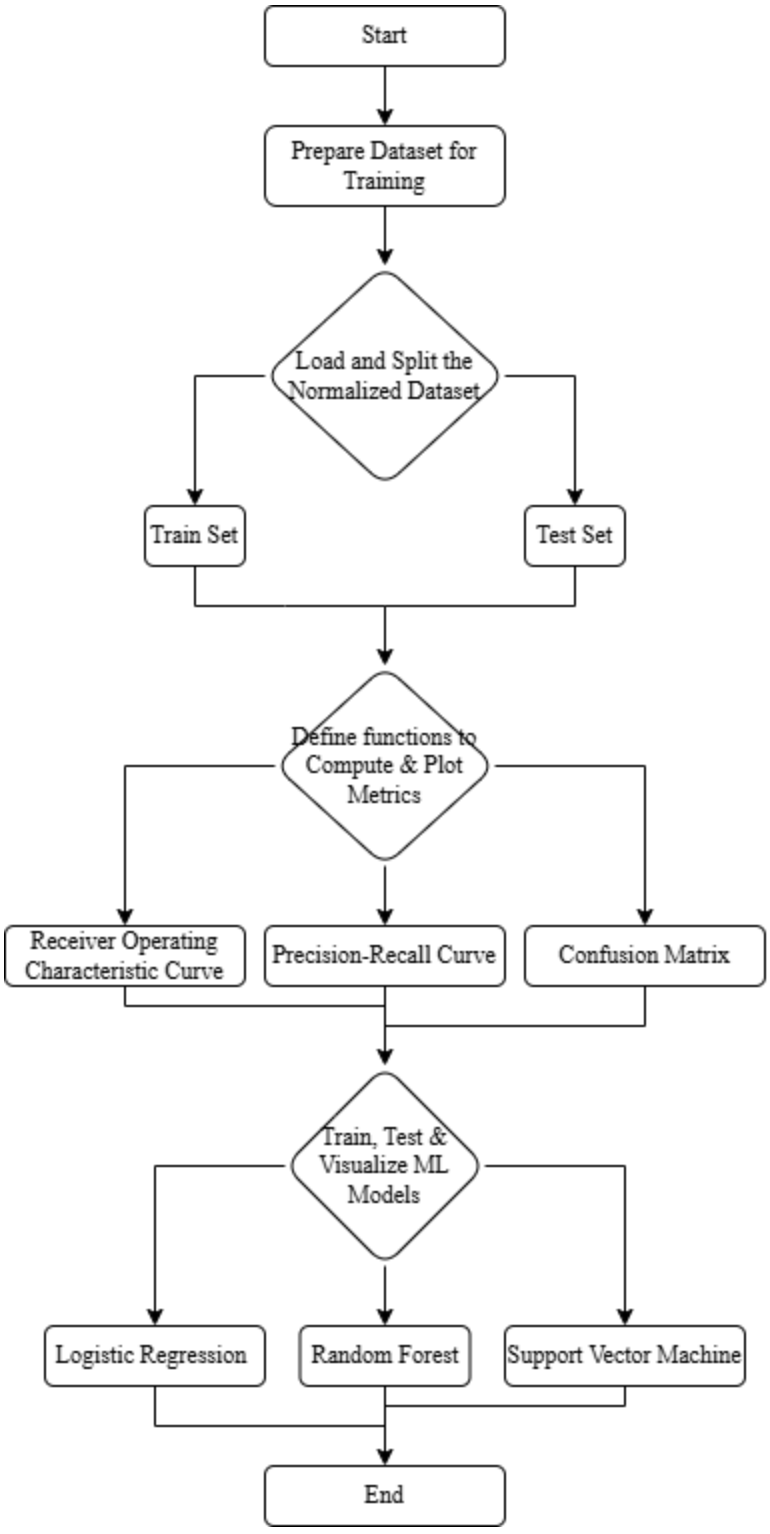
```
df['Mutual_Followers'] =
normalize_column(df['Mutual_Followers'])

plot_detailed_histogram(df['Mutual_Followers'],
'Distribution of Mutual Followers', 'Mutual Followers',
plot_color='brown')


df['Mutual_Following'] =
normalize_column(df['Mutual_Following'])

plot_detailed_histogram(df['Mutual_Following'],
'Distribution of Mutual Following', 'Mutual Following',
plot_color='grey')
```

The normalization and visualization functions are applied to the `Page_Rank_Source`, `Page_Rank_Target`, `Shortest_Path`, `Source_Followers`, `Source_Following`, `Target_Followers`, `Target_Following`, `Mutual_Followers`, and `Mutual_Following` columns and we supply the appropriate title, labels, and color.

3.4. Model Training & Evaluation

The GitHub Network Dataset has been pre-processed, the missing links were added, all the required features have been computed and added as well, and finally all the features were normalized. Our dataset is now almost ready to be used to train the ML models – Logistic Regression, Random Forest Classifier, and Support Vector Machine. Before training, we split the dataset into train and test sets and define the functions needed to plot the metrics that are to be used for model evaluation. We then train the models and evaluate each of them.

Find the flow-chart of the section below:


Start
Prepare Dataset for Training
Load and Split the Normalized Dataset
Train Set
Test Set
Define functions to Compute & Plot Metrics
Receiver Operating Characteristic Curve
Precision-Recall Curve
Confusion Matrix
Train, Test & Visualize ML Models
Logistic Regression
Random Forest
Support Vector Machine
End

3.4.1. Prepare Dataset for Training

We load our feature-rich normalized dataset and prepare it to be used for Training. This involves splitting the dataset into Train and Test sets. For our task, we use 70% of the dataset to train the models and the remaining data is used for testing. The train and test data are converted to cuDF format to be used for cuML models. Programmatic Implementation:

```
df = cudf.read_csv('./data/github-normalized-dataset.csv')
df = df.drop(columns=['Unnamed: 0'])
```

The `cudf.read_csv()` loads our GitHub Network dataset into a GPU accelerated DataFrame. Similar to a previous step, we drop the `unnamed` column that may have been created when loading the data to/from a csv file.

```
g = nx.from_pandas_edgelist(df.to_pandas()[['Source',
'Target']], source='Source', target='Target',
create_using=nx.DiGraph())
```

We create a directed graph like before with `nx.from_pandas_edgelist()` and our Source and Target nodes.

```
df_x = df.drop(columns=['Source', 'Target', 'Class'])
df_y = df[['Class']]
```

The DataFrame is being split into features and target. The columns that are not needed for training such as `Source`, `Target`, `Class` are dropped using `df.drop()` and the remaining columns are stored in `df_x`. The target, i.e., the label to be predicted as 0 or 1 is stored in `df_y`.

```
x_train, x_test, y_train, y_test =
train_test_split(df_x.to_pandas(), df_y.to_pandas(),
test_size=0.3, random_state=42)
x_train = cudf.DataFrame.from_pandas(x_train)
x_test = cudf.DataFrame.from_pandas(x_test)
y_train = cudf.DataFrame.from_pandas(y_train)
y_test = cudf.DataFrame.from_pandas(y_test)
```

The `train_test_split()` function from scikit-learn is provided the features and target converted to pandas data as input, the `test_size` is set to 30% and the random state is set to 42, meaning the data will always be sorted in this fixed order which ensures that we can reproduce any obtained results. The function then runs and returns the `x_train` set which contains training features that make up 70% of all features, `x_test` set which contains testing features that make up 30% of all features, `y_train` which contains training labels corresponding to `x_train`, `y_test` which contains testing labels corresponding to `x_test`. Finally, all four of these subsets are converted back to cuDF format from regular pandas DF format to facilitate training of cuML models.

3.4.2. Define Functions to Compute & Plot Metrics

To evaluate the trained ML models, we use three metrics - Receiver Operating Characteristic (ROC) Curve, Precision-Recall Curve, and Confusion Matrix. These metrics are computed for each of the model's performance and their respective curves are plotted as well. We define functions to compute & visualize these metrics for each of the models. Programmatic Implementation:

```
def plot_roc_curve(actual_labels, predicted_probs,
model_name="Model"):
```

This function takes in the actual labels from the dataset, the model's predictions of the labels, and the model name and produces the Receiver Operating Characteristic (ROC) curve Plot.

```
actual_labels =
 actual_labels.to_pandas().values.ravel()

predicted_probs =
 predicted_probs.to_pandas().values.ravel()
```

The labels which are currently in cuDF format are converted back to pandas and flattened to a single dimension (1D) as scikit-learn's `roc_curve` metric does not support cuDF data.

```
false_positive_rate, true_positive_rate,
 thresholds_list = roc_curve(actual_labels,
 predicted_probs)

auc_score = auc(false_positive_rate,
 true_positive_rate)
```

The `roc_curve()` method takes the actual labels and predicted labels as parameters and returns the False Positive Rate (FPR) which is the rate of the number of negatives falsely

classified as positives, True Positive Rate (TPR) which is the rate of positives correctly classified, and thresholds list which is the list of probability thresholds for ROC points. The `auc()` method takes the FPR & TPR parameters and returns the Area Under the Curve score which is a metric that summarizes the model's overall performance.

```
plt.figure(figsize=(8, 6))

plt.plot(false_positive_rate, true_positive_rate,
 color='#1f77b4', lw=2, label=f'{model_name} (AUC =
 {auc_score:.2f})')

plt.plot([0, 1], [0, 1], color='grey', linestyle='--',
 lw=2)
```

Matplotlib is used to configure the properties of the plot such as the figure size, labels, range, style and so on.

```
plt.grid(color='grey', linestyle=':', linewidth=0.5)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate (FPR)', fontsize=12,
 fontweight='bold')

plt.ylabel('True Positive Rate (TPR)', fontsize=12,
 fontweight='bold')

plt.title(f'ROC Curve - {model_name}', fontsize=14,
 fontweight='bold')
```

Grid lines are added for better interpretation, x and y-axis limits & labels for the plot are set, and title of the plot is defined.

```
plt.fill_between(false_positive_rate,
 true_positive_rate, alpha=0.1, color='blue')
```

`fill_between()` fills the plot area within the curve so we could see the AUC score better.

```
annotation_step = max(1, len(thresholds_list) // 8)

for idx in range(0, len(thresholds_list),
 annotation_step):

    plt.text(false_positive_rate[idx],
      true_positive_rate[idx],f"{
```

```
            thresholds_list[idx]:.2f}",   fontsize=10,
            color='black')

    plt.legend(loc="lower right", fontsize=10)

    plt.show()
```

The list of ROC thresholds are used to add annotation steps which ensures our plot's data is distributed at regular intervals. These intervals are also labeled with their appropriate values. The position for the legend is set with `legend()` and `show()` shows the plot.

```
def plot_pr_curve(true_labels, predicted_probs,
model_name="Model"):

    true_labels = true_labels.to_pandas().values.ravel()

    predicted_probs =
     predicted_probs.to_pandas().values.ravel()
```

This function visualizes the Precision-Recall curve using the true labels and predicted labels. Similar to previous function, the data is converted to pandas 1D arrays for calculation of the metric.

```
    precision_vals, recall_vals, threshold_vals =
precision_recall_curve(true_labels, predicted_probs)

    avg_precision = average_precision_score(true_labels,
predicted_probs)
```

The `precision_recall_curve()` returns the Precision which is the ratio of correct predictions out of all positive predictions, Recall which is the ratio of correct prediction out of all actual positives, and Thresholds corresponding to precision-recall points.

```
    plt.figure(figsize=(8, 6))

    plt.plot(recall_vals, precision_vals, color='#ff7f0e',
     lw=2, label=f'{model_name} (Avg Precision =
     {avg_precision:.2f})')

    plt.xlabel('Recall', fontsize=12, fontweight='bold')

    plt.ylabel('Precision', fontsize=12, fontweight='bold')

    plt.title(f'Precision-Recall Curve - {model_name}',
     fontsize=14, fontweight='bold')
```

```
plt.grid(color='grey', linestyle=':', linewidth=0.5)
```

Matplotlib is used to configure the properties of the plot such as the figure size, labels, range, style, title and so on.

```
plt.fill_between(recall_vals, precision_vals,
 alpha=0.1, color='orange')

plt.legend(loc="lower left", fontsize=10)

plt.show()
```

`fill_between()` fills the plot area within the curve so we could see the AUC score better. The position for the legend is set with legend() and show() shows the plot.

```
def plot_confusion_matrix(y_test, predicted_values,
model_name):
```

Like before, the function takes in the actual labels and predicted labels to compute the Confusion Matrix.

```
y_test_flat = y_test.to_pandas().values.ravel() if
 hasattr(y_test, 'to_pandas') else y_test.ravel()

predicted_values_flat =
 predicted_values.to_pandas().values.ravel() if
 hasattr(predicted_values, 'to_pandas') else
 predicted_values.ravel()
```

Convert test labels and predicted labels data to 1D numpy arrays.

```
conf_matrix = confusion_matrix(y_test_flat,
 predicted_values_flat)

C_host = conf_matrix.get() if hasattr(conf_matrix,
 'get') else conf_matrix
```

Using scikit-learn's `confusion_matrix()` method to generate the confusion matrix. The Confusion Matrix shows the performance of the model by comparing true labels and predicted labels, providing True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN).

```
plt.figure(figsize=(8, 6))
```

```
sns.heatmap(
    C_host, annot=True, cmap='YlGnBu', fmt='g',
      linewidths=2, linecolor='black',
    annot_kws={"size": 15, "weight": "bold"},
      cbar_kws={'shrink': 0.75}
)
plt.xlabel('Predicted Labels', fontsize=14,
 fontweight='bold', color='darkblue')
plt.ylabel('True Labels', fontsize=14,
 fontweight='bold', color='darkblue')
plt.title(f'Confusion Matrix - {model_name}',
 fontsize=16, fontweight='bold', color='darkred')
plt.xticks(fontsize=12, weight='bold', rotation=0,
 color='darkgreen')
plt.yticks(fontsize=12, weight='bold', rotation=0,
 color='darkgreen')
plt.tight_layout()
plt.show()
```

The figure size for the matrix is set. Seaborn's heatmap() is used to generate the confusion matrix color-coded to reflect the sample size of each cell. Matplotlib's pyplot is used to set labels & tilts for the x and y-axis, title for the plot, and the layout as well.

3.4.3. Train, Test & Visualize ML Models

The ML models Logistic Regression, Random Forest, and Support Vector Machine are trained on the training set and evaluated on the testing set, both derived from the GitHub Network Dataset. The Classification Report, Confusion Matrix Plot, Receiver Operating Characteristic Curve Plot, Precision-Recall Curve Plot are generated for each of the model to assess the effectiveness of the link prediction task. Programmatic Implementation:

Logistic Regression – First model to be trained is the Logistic Regression model. It is a supervised learning algorithm, mainly used for Classification tasks. It is a simple statistical algorithm that checks the capability of a sample's data in a dataset to be used to predict what class that sample belongs to.

```
logistic_regression_model_file = './data/github-logistic-
regression-model.pkl'
```

We setup the model's save path.

```
if not model_exists(logistic_regression_model_file):

    print("Initiating Logistic Regression model
     training...")

    penalty_options = ['l1', 'l2']

    regularization_values = np.logspace(0, 4, 10)

    optimal_params = None

    optimal_model = None

    highest_score = -1

    start_time = time.time()
```

It is checked whether the model already exists before starting the training. `penalty_options` sets the regularization parameter of the model. Here, we provide two options, L1 – Lasso Regularization, which has the effect of eliminating some feature coefficients. L2 – Ridge Regularization, which has the effect of shrinking the feature coefficients rather than eliminating them. `regularization_values` provides a range of values for the parameter C, which increases the regularization when low and decreases it when high. Other parameters to store the best parameters, model and highest score are initialized and timer to measure the training time is started.

```
    for reg_penalty in penalty_options:

        for reg_strength in regularization_values:

            lr_model =
            LogisticRegression(penalty=reg_penalty,
            C=reg_strength)

            lr_model.fit(x_train, y_train)

            test_score = lr_model.score(x_test, y_test)
```

This code iterates over all combinations of regularization types and values, training the linear regression model on each one, and testing against the test sets for each one.

```
            if test_score > highest_score:
```

```
                highest_score = test_score

                optimal_params = {'penalty': reg_penalty,
                'C': reg_strength}

                optimal_model = lr_model
```

Each score is checked against the highest score and the model and its parameters are saved as the best if it has a better score than the previous highest score set by another model.

```
    end_time = time.time()

    print('Time taken for Logistic Regression training:',
          round(end_time - start_time, 2), 'seconds')

    print('Optimal parameters identified:', optimal_params)

    print('Highest accuracy score:', highest_score)

    pickle.dump(optimal_model,
     open(logistic_regression_model_file, 'wb'))
```

The timer ends after the training concludes. The time taken, optimal parameters, and highest accuracy are printed and the best model is saved using the file path that was initialized.

```
else:

    print("Logistic Regression model already exists.
     Loading model from file.")

    optimal_model =
     pickle.load(open(logistic_regression_model_file,
     'rb'))
```

If the original check for whether the file doesn't exist fails, then that means a previously trained model exists, so we simply load that model.

```
predicted_values = optimal_model.predict(x_test)

print(classification_report(y_test.to_pandas(),
predicted_values.to_pandas()))
```

We use the model to predict the class on the test set. The predictions and actual values are used to generate a classification report.

```
plot_confusion_matrix(y_test, predicted_values, "Logistic
Regression")

plot_roc_curve(y_test, predicted_values,
model_name="Logistic Regression")

plot_pr_curve(y_test, predicted_values, model_name="Logistic
Regression")
```

Similarly, the predictions and actual values from the test set are used to compute and plot the Confusion Matrix, Receiver Operating Characteristic Curve, & Precision-Recall Curve.

Random Forest – Random Forest is another Supervised Machine Learning Algorithm that is also used for classification tasks. This algorithm makes use of decision trees to provide accurate predictions with any given dataset.

```
random_forest_model_file = './data/github-random-forest-
model.pkl'
```

Path to save or load the model to/from is set.

```
if not model_exists(random_forest_model_file):

    print("Training Random Forest model...")

    y_train_flat = y_train.to_pandas().values.ravel()

    y_test_flat = y_test.to_pandas().values.ravel()
```

We check to ensure that the model does not already exist before we start training. We flatten the train and test data to 1D arrays.

```
    num_trees = [50, 100, 150]

    max_tree_depths = [9, 12, 15, 24]

    min_samples_splits = np.random.randint(100, 150, 2)

    min_samples_leaves = np.random.randint(20, 30, 2)

    best_params = None

    best_model = None

    best_score = -1

    start_time = time.time()
```

We set a range for the number of tress the model could have, the depths for the trees, minimum no. of samples required to split an internal node, minimum no. of samples required to be at a leaf node. Timer is started to measure time taken for training.

```
for trees in num_trees:
    for depth in max_tree_depths:
        for split in min_samples_splits:
            for leaf in min_samples_leaves:
                rf_model = RandomForestClassifier
                (n_estimators=trees, max_depth=depth,
                     min_samples_split=split,
                     min_samples_leaf=leaf)
                rf_model.fit(x_train, y_train_flat)
                accuracy = rf_model.score(x_test,
                y_test_flat)
```

All combinations of parameters are iterated through and used to train a model, which is tested against test sets.

```
                if accuracy > best_score:
                    best_score = accuracy
                    best_params = {
                        'n_estimators': trees,
                        'max_depth': depth,
                        'min_samples_split': split,
                        'min_samples_leaf': leaf
                    }
                    best_model = rf_model
```

We check if the accuracy of the trained model is better than the accuracy of the previous model and save that as the best model along with its parameters. This is done iteratively for all combinations until the best model and parameters are found and saved.

```
end_time = time.time()
print('Time taken to train Random Forest model:',
 round(end_time - start_time, 2), 'seconds')
```

```
    print('Best parameters found:', best_params)

    print('Best accuracy score:', best_score)

    pickle.dump(best_model, open(random_forest_model_file,
     'wb'))
```

The timer is stopped and time taken, the best parameters, the best score are printed, and the best model is saved.

```
else:

    print("Random Forest model already exists. Loading from
     file.")

    best_model = pickle.load(open(random_forest_model_file,
     'rb'))
```

On the other hand, if the model already exists in the path defined, then it is loaded as the best model.

```
predicted_values = best_model.predict(x_test)

print(classification_report(y_test.to_pandas(),
predicted_values.to_pandas()))

plot_confusion_matrix(y_test, predicted_values, "Random
Forest")

plot_roc_curve(y_test, predicted_values, model_name="Random
Forest")

plot_pr_curve(y_test, predicted_values, model_name="Random
Forest")
```

The trained random forest model is used to predict the classes of the test set. Using the predictions, the classification report, roc, pr, and conf matrix plots are generated.

Support Vector Machine – This is a machine learning algorithm that has the special capability of adaptability and the ability to focus on target feature's different classes, making it very suitable for classification tasks.

```
svm_model_file = './data/github-support-vector-machine-
model.pkl'
```

The path for the model file is set.

```
if not model_exists(svm_model_file):
    print("Training Support Vector Machine (SVM) model...")
    kernel_types = ['rbf', 'linear']
    regularization_values = [0.001, 0.01, 0.1, 1]
    best_params = None
    best_model = None
    best_accuracy = -1
    start_time = time.time()
```

Training is started if the model does not exist. The kernel is a function that is used for data-mapping and here, two types are included. Regularization values within a range are given which will help to prevent overfitting. Then variables to store best parameters and model accuracy are initialized. Timer to measure the training time is started.

```
    for kernel in kernel_types:
        for regularization in regularization_values:
            svm_model = SVC(kernel=kernel,
               C=regularization)
            svm_model.fit(x_train, y_train)
            accuracy = svm_model.score(x_test, y_test)
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_params = {'kernel': kernel, 'C':
                    regularization}
                best_model = svm_model
    end_time = time.time()
    print('Time taken to train SVM model:', round(end_time
     - start_time, 2), 'seconds')
    print('Best parameters:', best_params)
    print('Best accuracy score:', best_accuracy)
    pickle.dump(best_model, open(svm_model_file, 'wb'))
```

A structure similar to the previous two models is followed. We iterate through the parameters, train a model variation, test it, and save the model, its parameter and accuracy as the best if the accuracy is higher than the previous model.

```
else:

    print("SVM model already exists. Loading from file.")

    best_model = pickle.load(open(svm_model_file, 'rb'))
```

We load the model if it is already present in the path.

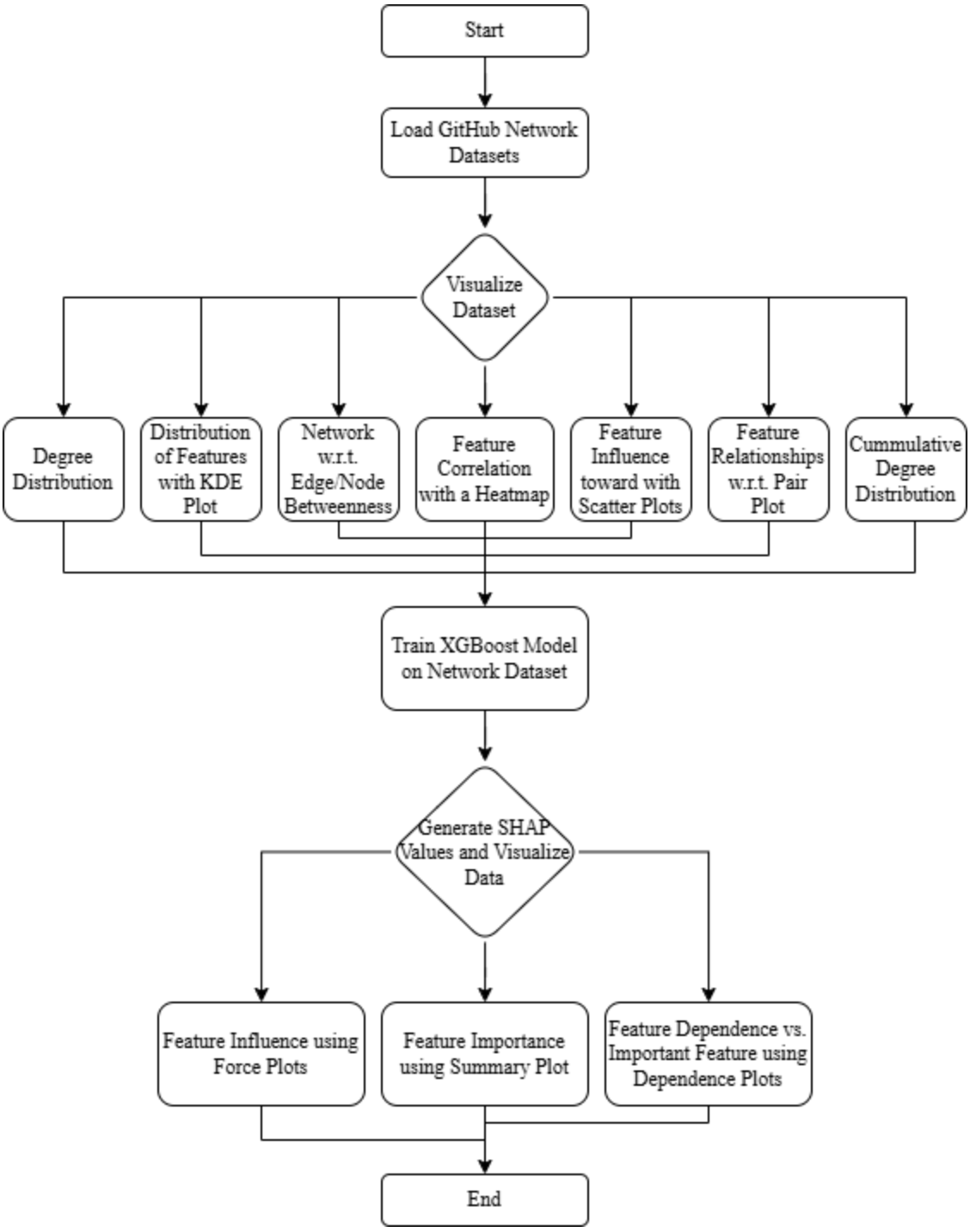
```
predicted_values = best_model.predict(x_test)

print(classification_report(y_test.to_pandas(),
predicted_values.to_pandas()))

plot_confusion_matrix(y_test, predicted_values, "SVM")

plot_roc_curve(y_test, predicted_values, model_name="SVM")

plot_pr_curve(y_test, predicted_values, model_name="SVM")
```

The SVM model is used to predict the class on the test set. The predictions and actual values are used to generate a classification report, and to compute and plot the Confusion Matrix, Receiver Operating Characteristic Curve, & Precision-Recall Curve.

3.5. Social Network Analysis

The Link Prediction task was successfully completed with the training and evaluation of the previous three ML models. The other motive of this project to analyze the GitHub network and gain insights into its structure, features, working, and so on. Towards this end, we utilize various methods of visualizing data such as Node/Edge Betweenness, Feature – Correlation, Influence, Distribution, Relationships, and Degree – Distribution, Cumulative Distribution. After these visualizations, we further analyze the network by training an XGBoost Model on the data and using this model to generate SHapley Additive exPlanations (SHAP) values and using these values to compute the Feature – Influence, Importance, and Dependence. These visualizations provide a holistic understanding of the GitHub Network and the importance and impact of its features.

Find the flow-chart that visualizes the section below:



3.5.1. Visualize GitHub Network Features

First, we load the datasets needed for the visualizations, which include the original MUSAE dataset and the feature-rich normalized dataset. We then create a function that prepares a sample of the graph data for our network visualizations as visualizing the entire Network would be

computationally expensive, unfeasible & won't yield the best insights as the data would be too crowded. Then, we move on to the visualizations which involve visualizing the Network with respect to Edge Betweenness & Node Betweenness, Feature Correlation with a Heatmap, Feature Influence toward Class 0 or 1 with Scatter Plots, Distribution of Features with Kernel Density Estimate (KDE) Plot, Feature Relationships with respect to Classes with a Pair Plot, and Degree & Cumulative Degree Distribution of the Network. Programmatic Implementation:

```
data = open('./data/musae_git_edges.csv')

next(data)

graph = nx.read_edgelist(data, create_using=nx.DiGraph(),
delimiter=',', nodetype=int)

print('Graph Details')

print(graph)

Graph Details

DiGraph with 37700 nodes and 289003 edges
```

We load the MUSAE dataset which contains the edges data of the GitHub Network. The first line of the data is skipped because it contains the column headers. Then the data is used to generate a directed graph. The graph details are printed to ensure that our generated graph is ready for analysis.

```
df = pd.read_csv('./data/github-normalized-dataset.csv')

df = df.drop(columns=['Unnamed: 0'])

df.head(10)
```

We also load the normalized feature-rich dataset, drop the unnamed column that appears when loading from a csv file and visualize a few samples to ensure that it is also ready to be analyzed.

```
def prepare_sampled_graph(graph, num_edges=500, seed=42):
```

This function is used to get a sample graph from our directed graph. The default no. of edges it has is set to 500 and the seed value is set to 42 to ensure reproducibility.

```
    sampled_edges = random.sample(list(graph.edges),
     num_edges)

    sampled_graph = nx.DiGraph()
```

```
    sampled_graph.add_edges_from(sampled_edges)
```

The `random.sample()` function fetches the edges of the graph and it returns a random subset of size `num_edges`. Then an empty directed graph is initialized to `sampled_graph` and the sampled edges are added to this graph using `add_edges_from()` method.

```
    sampled_graph =
     nx.convert_node_labels_to_integers(G=sampled_graph,
     first_label=0)

    layout = nx.spring_layout(sampled_graph, seed=seed)

    return sampled_graph, layout
```

The labels of the nodes are converted to consecutive integers for easier interpretation and that layout for the graph is set to `spring_layout`. This layout works very well to visualize graph data as its nature is to have all the nodes push away from each other with only the connections between the nodes pulling them closer, i.e., acting like a spring. This makes each node stand out from one another and the connections between them easier to interpret. Lastly, the sample graph and the layout for it are returned.

Visualization #1 – The first visualization involves visualizing the GitHub Network with respect to the Edge Betweenness of the edges. Edge betweenness of an edge can be described as the importance of the edge. It is based on the shortest paths between nodes that make use of or pass through that edge. The higher the value, more important is the edge.

```
sampled_graph, layout = prepare_sampled_graph(graph)

edge_betweenness =
nx.edge_betweenness_centrality(sampled_graph)

centrality_values = list(edge_betweenness.values())

log_scaled_values = np.log1p(centrality_values)

norm_centrality =
np.array(log_scaled_values)/log_scaled_values.max()
```

We get our sample graph from the previously defined function. The method `edge_betweenness_centrality()` is used to compute the edge betweenness values of edges in the network. A dictionary containing the edges as the key and the edge betweenness values as their values is returned. These values are converted to a list `centrality_values` which are then log-scaled using `np.log1p()` method which is

a normalization pre-step to avoid zero values or values that are too small. Once log-scaled, we then perform the normalization which transforms the values to fall within the range of (0,1).

```
edge_colors = plt.cm.viridis(norm_centrality)

node_degrees = dict(sampled_graph.degree())

node_sizes = [degree * 50 for degree in
node_degrees.values()]

edge_thickness = [0.5 + node_degrees[u] * 0.05 for u, v in
sampled_graph.edges()]

plt.figure(figsize=(10, 8), facecolor='white')

ax = plt.gca()

ax.set_facecolor('white')

ax.grid(False)
```

The colors for the edges are applied based on the normalized edge betweenness values and color scheme viridis is used where lesser values are colored dark, skewing towards purple and higher values are colored light, skewing towards yellow. The degrees of the nodes are obtained using the `degree()` method and stored as a dictionary in `node_degrees`, using which the sizes of the nodes are computed. Here we multiply the node sizes by 50 to make easier to see. The edge thickness is also computed which is actually proportional to the source nodes of each edge in the graph. A base thickness of 0.5 is added to all the nodes to prevent invisible edges, and then the degree value is multiplied by 0.05 to fit the thickness to appropriate range for edge betweenness. Thus, higher the degree of a source node, thicker its edge. Then, for the plot, matplotlib's pyplot is used to set the figure size, its axes is initialized to `ax` using `gca()`, it's color is set to white for better visibility with the viridis color scheme and the grid is disabled so that the network of nodes and their edges can be interpreted better.

```
nodes = nx.draw_networkx_nodes(
    sampled_graph,
    layout,
    node_color=list(node_degrees.values()),
    cmap='cool',
    node_size=node_sizes,
    alpha=0.9,
```

```
    edgecolors='black'
)
```

The `draw_networkx_nodes()` method is used to draw the nodes of our sample graph. The sample graph, its layout, the node degrees, a color map of "cool", computed node sizes, and an alpha of 0.9 for a tinge of transparency are all set as parameters for the method, including the default edge color set to black as a fallback.

```
edges = nx.draw_networkx_edges(
    sampled_graph,
    layout,
    edge_color=edge_colors,
    width=edge_thickness,
    alpha=0.8
)
```

Similarly, `draw_networkx_edges()` method is used to draw the edges of our sampled graph and mostly similar parameters are given with the exceptions of `edge_color` which uses the viridis scheme we defined earlier and `width` which uses our computed list of edge thickness values.

```
plt.title('GitHub Network Plot with Edge Betweenness
Centrality', fontsize=16, fontweight='bold', color='black')
sm = plt.cm.ScalarMappable(cmap='viridis',
norm=plt.Normalize(vmin=0, vmax=log_scaled_values.max()))
sm.set_array([])
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Edge Betweenness Centrality (Log-Scaled)',
color='black', fontsize=12)
cbar.ax.yaxis.set_tick_params(color='black')
plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'),
color='black')
plt.figtext(0.5, -0.05, "Note: Node size & color are
proportional to node degree (number of connections)",
wrap=True, horizontalalignment='center', fontsize=12)
```

```
plt.tight_layout()

plt.show()
```

Finally, we add a colorbar to our plot to help identify what all the colors mean. The title is set, colorbar is initialized with the normalized edge betweenness values, labels and angles for them are set for the colorbar, and layout and text are set for the figure using Matplotlib's Pyplot. Thus, the GitHub Network is visualized with respect to its Edge Betweenness values.

Visualization #2 – The next visualization involves visualizing the GitHub Network with respect to the Node Betweenness of the Nodes. Node betweenness of a node can be described as the importance of a node. It is based on the shortest paths between nodes that make use of or pass through that node. The higher the value, more important is the node.

```
sampled_graph, layout = prepare_sampled_graph(graph)

node_betweenness = nx.betweenness_centrality(sampled_graph)

betweenness_values =
np.array(list(node_betweenness.values()))

log_betweenness_values = np.log1p(betweenness_values)

norm_betweenness = log_betweenness_values /
log_betweenness_values.max()

node_degrees = dict(sampled_graph.degree())

node_sizes = [degree * 50 for degree in
node_degrees.values()]

edge_thickness = [0.5 + node_degrees[u] * 0.05 for u, v in
sampled_graph.edges()]
```

We follow the same approach as before with the only difference being that the node betweenness values are computed as opposed to the edge betweenness.

```
plt.figure(figsize=(10, 8), facecolor='white')

ax = plt.gca()

ax.set_facecolor('white')

ax.grid(False)

nodes = nx.draw_networkx_nodes(
```

```
    sampled_graph,
    layout,
    node_color=norm_betweenness,
    cmap='plasma',
    node_size=node_sizes,
    alpha=0.9,
    edgecolors='black'
)
edges = nx.draw_networkx_edges(
    sampled_graph,
    layout,
    edge_color='gray',
    width=edge_thickness,
    alpha=0.8
)
plt.title('GitHub Network Plot with Node Betweenness
Centrality', fontsize=16, fontweight='bold', color='black')
sm = plt.cm.ScalarMappable(cmap='plasma',
norm=plt.Normalize(vmin=0,
vmax=log_betweenness_values.max()))
sm.set_array([])
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Node Betweenness Centrality (Log-Scaled)',
color='black', fontsize=12)
cbar.ax.yaxis.set_tick_params(color='black')
plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'),
color='black')
plt.figtext(0.5, -0.05, "Note: Node size is proportional to
node degree (number of connections)", wrap=True,
horizontalalignment='center', fontsize=12)
```

```
plt.tight_layout()

plt.show()
```

Same as before, the methods draw_networkx_nodes() and draw_networkx_edges() are used to draw the network's nodes and edges but this time the color of the nodes is set based on the normalized node betweenness values instead of their degrees, with the colormap set to "plasma" instead of "viridis". The remaining properties are customized appropriately for this figure and therefore, the GitHub Network is visualized with respect to its Node Betweenness values.

Visualization #3 – This visualization involves displaying the Feature correlation using a heatmap. Correlation is the degree to which two features are dependent on each other. A positive correlation between two features indicates that when one feature increases in value, the other one also does, a negative correlation indicates that when one feature increases, the other decreases in value, and a neutral one indicates no correlation. Correlation falls in the range [-1, 1].

```
plt.figure(figsize=(12, 8))

corr_matrix = df.drop(columns=['Source', 'Target']).corr()
```

We set the figure size, drop the node columns as they are not features and compute the correlation matrix using the method `corr()` which returns a table containing the correlation coefficients between the features.

```
ax = sns.heatmap(
    corr_matrix,
    vmin=-1, vmax=1, center=0,
    cmap='coolwarm',
    annot=True,
    fmt=".2f",
    square=True,
    linewidths=0.5,
    cbar_kws={"shrink": 0.75},
)
```

We use the `heatmap()` method from Seaborn to generate a heatmap for the correlation matrix. The parameters include the correlation matrix, vmin, vmax, & center which sets the range of the metrics to -1, 1, & 0, a colormap of coolwarm which makes small values a cool color skewing towards blue, large values a hot color skewing towards red, and neutral values a light color closer to white, annotation is set to true to label the cells with their appropriate values, formatting to ensure the values are limited to two decimal places, and so on.

```
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=45,
    horizontalalignment='right',
    fontsize=10,
    fontweight='bold'
)
ax.set_yticklabels(
    ax.get_yticklabels(),
    rotation=0,
    horizontalalignment='right',
    fontsize=10,
    fontweight='bold'
)
plt.title('Correlation Heatmap of Features', fontsize=16,
fontweight='bold')
plt.tight_layout()
plt.show()
```

The angles and text properties for the x and y-axis labels, the title & layout for the plot are set and the plot is shown in the end. Thus, Correlation between the GitHub Network's features are visualized with a Heatmap.

Visualization #4 – The visualization involves the Feature Influence toward either Class 0 or Class 1 visualized using Scatter Plots for each Feature. Each plot shows the influence

that each feature has toward the prediction of class 1 or 0. If a feature has more values and higher values for one class than the other, then it is likely to facilitate the prediction for that class more.

```
dims = (15, 12)

fig, axes = plt.subplots(nrows=3, ncols=3, figsize=dims)

sns.set(style="whitegrid")

color_palette = ['red', 'teal', 'purple', 'blue', 'orange',
'green', 'magenta', 'brown', 'grey']
```

First, the dimensions for the figure are set. Then we set the rows and columns of the plot, which is set to a 3x3 matrix of subplots, with each being a scatter-plot of one particular feature for a total of 9 features. We set the grid of the plot to be white, and define a list of colors that will be used for the plot points in each of the scatterplots.

```
sns.scatterplot(x='Shortest_Path', y='Class', data=df,
ax=axes[0][0], color=color_palette[0], s=70, alpha=0.7)

sns.scatterplot(x='Page_Rank_Source', y='Class', data=df,
ax=axes[0][1], color=color_palette[1], s=70, alpha=0.7)

sns.scatterplot(x='Page_Rank_Target', y='Class', data=df,
ax=axes[0][2], color=color_palette[2], s=70, alpha=0.7)

sns.scatterplot(x='Source_Followers', y='Class', data=df,
ax=axes[1][0], color=color_palette[3], s=70, alpha=0.7)

sns.scatterplot(x='Target_Followers', y='Class', data=df,
ax=axes[1][1], color=color_palette[4], s=70, alpha=0.7)

sns.scatterplot(x='Source_Following', y='Class', data=df,
ax=axes[1][2], color=color_palette[5], s=70, alpha=0.7)

sns.scatterplot(x='Target_Following', y='Class', data=df,
ax=axes[2][0], color=color_palette[6], s=70, alpha=0.7)

sns.scatterplot(x='Mutual_Followers', y='Class', data=df,
ax=axes[2][1], color=color_palette[7], s=70, alpha=0.7)

sns.scatterplot(x='Mutual_Following', y='Class', data=df,
ax=axes[2][2], color=color_palette[8], s=70, alpha=0.7)
```

We generate a scatterplot for each of the features from our dataset – `Shortest_Path`, `Page_Rank_Source`, `Page_Rank_Target`, `Source_Followers`, `Target_Followers`, `Source_Following`, `Target_Following`,

`Mutual_Followers`, and `Mutual_Following`. The parameters set for them include the x-axis set to represent the value of the feature, y-axis denoting the classes 0 & 1, data set to use the network dataframe, axes to fix where in the 3x3 matrix the particular plot should be placed, color for the plot points, marker size for plot points, and a slight transparency to see the data clearly through any overlap.

```
for ax in axes.flat:

    ax.set_ylim(-1, 2)

plt.tight_layout()

plt.show()
```

Then for each plot, the y-axis limit is set to a -1 lower-bound and a 2 upper-bound. This helps to add some buffer between our classes 0 and 1 allowing to see the data more clearly. The layout for the plot is set to be tight, this has the effect of adjusting each sub-plot and its details to fit perfectly inside our matrix. Finally, the plot is printed. Thus, Feature Influence toward either a missing link (Class - 0) or an existing link (Class - 1) in the GitHub Network is visualized using Scatter Plots.

Visualization #5 – The Distribution of the Features from our GitHub Network is visualized using a Kernel Density Estimate Plot. This plot shows us the probability density of each of the features and their values, essentially answering which feature's values are most abundant and what value is the most frequent for each feature. The height of a feature's curve in the y-axis gives its density, while the x-axis denotes its value in the curve.

```
plt.figure(figsize=(12, 10))
```

The size of the figure is set.

```
for feature in df.columns.drop(['Class', 'Source',
'Target']):

    sns.kdeplot(df[feature], label=feature, fill=True)
```

In our network dataframe, we drop the Class, Source, and Node Columns as they are not features. We iterate through the remaining features and add them to our `kdeplot()` method from Seaborn, adding also the feature as the label and setting fill to true to color the area under the curve of a feature, which would help in observing correlation with other features.

```
plt.title('Density Plot of Features', fontsize=16,
fontweight='bold')
```

```
plt.xlabel('Feature Values')

plt.ylabel('Density')

plt.legend()

plt.tight_layout()

plt.show()
```

Matplotlib's Pyplot is used to set the remaining properties of the plot such as the title, x and y-axis labels, legend, layout and so on. Then, the plot is shown using `show()`. Thus, the Distribution of the Features in our GitHub Network is visualized using a Kernel Density Estimate Plot.

Visualization #6 – This visualization involves the relationship between each of the features in a pair-wise manner is visualized using a Pair Plot. This shows us how two features are correlated across the classes, giving us an idea of their influence on each other and on the prediction outcome of a class, leading to informational patterns.

```
sns.pairplot(df, vars=['Shortest_Path', 'Page_Rank_Source',
'Source_Followers', 'Target_Followers'], hue='Class',
palette='coolwarm', plot_kws={'alpha': 0.5})

plt.suptitle('Pair Plot of Key Features', fontsize=16,
fontweight='bold', y=1.02)

plt.show()
```

We use the `pairplot()` method from Seaborn to generate our pair plot. Here, we choose four features for our plot – `Shortest_Path`, `Page_Rank_Source`, `Source_Followers`, and `Target_Followers`. The plot data for these features is colored based on class, with class - 0 taking blue and class - 1 taking orange due to the coolwarm palette. The plot points are also made transparent for better visibility. Lastly, the title is set and the pair plot is shown. Thus, the pairwise relationships between multiple features in the GitHub Network are visualized using a Pair Plot.

Visualization #7 – We visualize the Rankings of the nodes in the GitHub Network by way of their Degree Distribution. Degree is the number of connections that a node has with other nodes in the network. In this plot, the nodes with higher degrees are ranked higher and skewed towards the left while the nodes with lower degrees are ranked lower and skewed towards the right. Thus, the y-axis denotes the degree of the nodes and the x-axis denotes the rank.

```
degree_distribution_values = sorted([degree for node,
degree in graph.degree()], reverse=True)
```

We use NetworkX's `degree()` method to get the degrees of all the nodes, store them in a list, and sort them in descending order to rank the nodes from highest to lowest in terms of their degree.

```
plt.figure(figsize=(12, 10))

plt.plot(degree_distribution_values, marker='o',
linestyle='-', color='#1f77b4')

plt.title('Degree Distribution of the Network',
fontsize=16, fontweight='bold')

plt.xlabel('Node Rank', fontsize=12, fontweight='bold')

plt.ylabel('Degree', fontsize=12, fontweight='bold')

plt.grid(True, linestyle='--', alpha=0.6)

plt.show()
```

The figure size, title, labels, and grid styles of the plot are set. The plot is generated using the degree values, with properties such as the linestyle, marker type, and color for the plot data also being set, after which the plot is shown. Thus, Ranking of the Nodes with respect to their Degrees is visualized using a Degree Distribution Plot.

Visualization #8 – Here, we visualize the fraction of nodes with at least a degree of K using a Cumulative Degree Distribution Plot. This plot shows the frequency of high degree nodes in the network.

```
degree_counts = nx.degree_histogram(graph)

degrees = range(len(degree_counts))

cumulative_degrees = [sum(degree_counts[i:]) for i in
degrees]
```

`degree_histogram()` returns a list containing the number of nodes with a certain degree with the indexes of these numbers corresponding to degrees in the network. We initialize a range using the degrees. Cumulative counts of nodes with that degree and degrees higher than that are computed for each degree, for example, for degree 2, the total number of nodes with degree 2 and higher are found and so on.

```
plt.figure(figsize=(12, 6))
```

```
plt.loglog(degrees, cumulative_degrees, marker='o',
color='#ff7f0e')

plt.title('Cumulative Degree Distribution', fontsize=16,
fontweight='bold')

plt.xlabel('Degree (k)', fontsize=12, fontweight='bold')

plt.ylabel('P(X >= k)', fontsize=12, fontweight='bold')

plt.grid(True, linestyle='--', alpha=0.6)

plt.show()
```

We configure the plot properties such as the size, title, labels, grid styles, and so on using Matplotlib. The `loglog()` function makes our plot log-scaled, allowing it to be visualized across many orders of magnitude and then we visualize our plot using `show()`. Thus, the fraction (or count) of nodes with degree greater than or equal to k has been visualized using a Cumulative Degree Distribution plot.

3.5.2. Train XGBoost Model & Analyze Network with SHAP

We move on to the next phase of the GitHub Network analysis. For this analysis, SHapley Additive exPlanations approach has been used. This evaluation tool provides many plots such as Force plots, Summary plots, & Dependence plots to analyze, compare and interpret the features present in our GitHub network and their quirks. However, to generate SHAP values, we need a machine learning model that is able to make predictions using the features in our dataset. For this purpose, we train the XGBoost Model on our GitHub network dataset. It is one of the most efficient and accurate gradient boosting algorithms that can also take advantage of a GPU's computing power. Thus, generating SHAP values using this model is both fast and can result in accurate values. Programmatic Implementation:

```
xgb_model_path = './data/xgboost_model.pkl'

shap_values_path = './data/shap_values.pkl'
```

Paths for the XGBoost model and the SHAP values to be saved or load from is setup.

```
X = df.drop(columns=['Source', 'Target', 'Class'])

y = df['Class']

x_train, x_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

Similar to training of the other models, we drop the nodes and classes from the network dataset. We assign the class as the target class and use `train_test_split()` method to get our training and testing sets, please refer to section 3.4.1 for an explanation of the method's working.

```
if os.path.exists(xgb_model_path):
    with open(xgb_model_path, 'rb') as f:
        xgb_model = pickle.load(f)
    print('Pre-trained model loaded from disk.')
```

First, we check if the XGBoost model already exists locally from a previous training. This model is loaded if it is. Otherwise, we move on to the training part.

```
else:
    dtrain = xgb.DMatrix(x_train, label=y_train)
    dtest = xgb.DMatrix(x_test, label=y_test)
    params = {
        'objective': 'binary:logistic',
        'max_depth': 6,
        'learning_rate': 0.1,
        'n_estimators': 100,
        'eval_metric': 'auc'
        'tree_method': 'gpu_hist',
        'predictor': 'gpu_predictor',
    }
    xgb_model = xgb.train(params, dtrain,
num_boost_round=100)
    with open(xgb_model_path, 'wb') as f:
        pickle.dump(xgb_model, f)
    print('New model trained and saved to file.')
```

First step before training involves converting our train and test sets to the DMatrix format. This format is specifically optimized for the XGBoost model and can boost the training process astronomically. The next is to declare the parameters for our model. These include setting the objective of the trained model to a binary classification task, the maximum depth for the trees of the model is set to 6 to avoid underfitting or overfitting, learning rate to 0.1 for a slow but highly accurate learning process, boosting rounds of count 100 for a thorough training, evaluation metric set to AUC (area under the curve) which is generally used for binary classification as the score is based on the model's ability to predict the classes correctly. We also add additional parameters `gpu_hist` and `gpu_predictor` as the tree method and predictor of our model, which enables our model to take full advantage of GPU acceleration to perform the training. Finally, the training is initiated and the trained model is saved to a pickle file for future use.

Now that we have the model ready, we can start generating SHAP values. Before that, a quick explanation of SHAP values follows. SHapley Additive exPlanations values uses an ML model, the dataset it was trained on, and the model's predictions to determine how much each feature contributes to the correct or incorrect prediction of each data sample's class for all instances in the dataset. This helps us see how the ML model makes predictions and also gauge the importance of each feature in the network that determines whether the source and target node are connected or disjoint.

```
if os.path.exists(shap_values_path):
    print("Loading previously saved SHAP values...")
    with open(shap_values_path, 'rb') as file:
        shap_values = pickle.load(file)

    explainer = shap.Explainer(xgb_model, x_train)
else:
    print("Calculating new SHAP values...")
    explainer = shap.Explainer(xgb_model, x_train)
    shap_values = explainer(x_test)
    with open(shap_values_path, 'wb') as file:
        pickle.dump(shap_values, file)
    print(f"SHAP values stored at {shap_values_path}")
```

Similar to the previous step we check if the SHAP values exist in the predefined path. If it does, we simply load them and initialize the Explainer framework which is what calculates the SHAP values for the dataset. On the other hand, if the SHAP values do not already exist, then we compute them using the `explainer()` object provided by the initialized Explainer framework. Once computed, we save them in a pickle file using the path we setup previously.

The SHAP values are now generated and ready to be used gain insights such as Feature Influence in Class prediction, Feature Importance, and Feature Dependence using Force Plots, Summary Plot, and Dependence Plots.

```
short_column_names = {
    'Page_Rank_Source': 'PR_Src',
    'Page_Rank_Target': 'PR_Tgt',
    'Shortest_Path': 'S_Path',
    'Source_Followers': 'S_Frs',
    'Source_Following': 'S_Fgs',
    'Target_Followers': 'T_Frs',
    'Target_Following': 'T_Fgs',
    'Mutual_Followers': 'M_Frs',
    'Mutual_Following': 'M_Fgs'
}
```

The columns containing our network features are given short form names to make the force plots more interpretable.

```
def create_force_plot(explainer, shap_values, x_test, idx):
    rounded_shap_values = np.round(shap_values[idx].values,
     2)
    rounded_feature_values =
     np.round(x_test.iloc[idx].values, 2)
    shap.force_plot(
        explainer.expected_value,
        rounded_shap_values,
```

```
        rounded_feature_values,
        feature_names=[short_column_names.get(col, col) for
          col in x_test.columns],
        matplotlib=True,
        show=False
    )
    plt.grid(False)
    plt.gca().set_axisbelow(True)
    plt.show()
```

Function to create force plot for one instance from the dataset. The function takes in the explainer object, generated SHAP values, test set with features and an index. Both the SHAP values and the test features are rounded to 2 decimals for readability. We then use the `force_plot()` method from the shap library to generate the force plot for a data instance by supplying the average model prediction which acts as a baseline, the rounded SHAP and Feature values, and abbreviated feature names for the features from the dataset. Then the grid is disabled for the plot, the axis for the plot is set and the plot is shown.

```
def show_custom_legend(short_column_names):
    fig, ax = plt.subplots(figsize=(12, 1))
    ax.axis('off')
    legend_text = [f"{short_name} = {full_name}" for
     short_name, full_name in short_column_names.items()]
    ax.text(0.99, 0.5, "\n".join(legend_text), ha='right',
     fontsize=10, fontweight='bold')
    plt.subplots_adjust(right=1.0)
    plt.show()
```

This function sets the legend commonly for all our plots and is added at the top before the plots. For each column name, the abbreviation from the `short_column_names` dictionary is used in the legend to indicate the name substitutions that have been made in the force plots.

```
def generate_force_plots_for_random_samples(explainer,
shap_values, x_test, short_column_names, num_samples=5):
```

```
    sample_indices = random.sample(range(len(x_test)),
     num_samples)

    show_custom_legend(short_column_names)

    for idx in sample_indices:

        create_force_plot(explainer, shap_values, x_test,
          idx)
```

Finally we supply the explainer, shap values, test data, abbreviated column names, & the no. of force plots we want to generate which in this case is 5, to the `generate_force_plots_for_random_samples()` function. This function prints the legend first and then chooses 5 random samples from our test set and generates force plots for these samples using our `create_force_plot()` function.

```
generate_force_plots_for_random_samples(explainer,
shap_values, x_test, short_column_names, num_samples=5)
```

At last, our custom function to generate the force plots is called.

```
plt.grid(False)

plt.title('Feature Importance', fontsize=16)

shap.summary_plot(shap_values, x_test, plot_type="bar",
show=True)
```

We disable the grid here as well for a better looking plot. The appropriate title for the plot is set. For each feature in the dataset, the average SHAP value is computed and displayed in the order of highest to lowest. This allows to see which features are the most influential in the dataset.

```
features = ['Page_Rank_Source', 'Page_Rank_Target',
'Shortest_Path', 'Source_Followers', 'Source_Following',
'Target_Followers', 'Target_Following', 'Mutual_Followers',
'Mutual_Following']
```

The features to be used in the dependence plots are declared in a list.

```
fig, axs = plt.subplots(nrows=5, ncols=2, figsize=(16, 24))

plt.subplots_adjust(hspace=0.8, wspace=0.6)
```

```
fig.suptitle('SHAP Dependence Plots - Source_Following vs
Other Features', fontsize=18, fontweight='bold')
```

A grid of subplots that's 5 rows and 2 columns in size is initialized, properties for the subplots are configured.

```
for idx, feature in enumerate(features):
    row_idx = idx // 2
    col_idx = idx % 2
    shap.dependence_plot(feature, shap_values.values,
     x_test, interaction_index="Source_Following",
     ax=axs[row_idx, col_idx], show=False)
fig.delaxes(axs[4, 1])
plt.tight_layout(rect=[0, 0, 1, 1], pad=3.0)
plt.show()
```

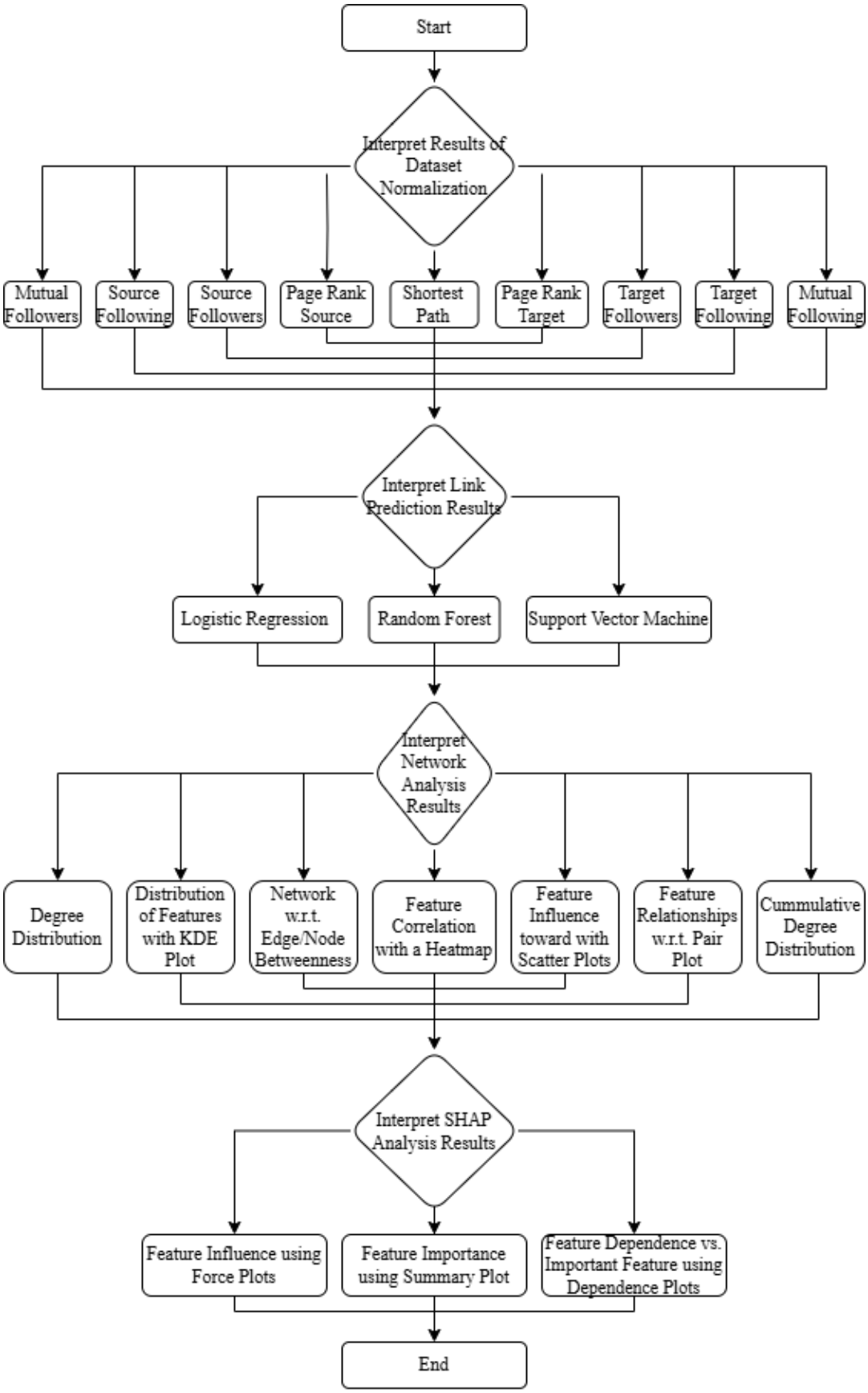
At last, we loop through the features from our list, compute the position for each of our dependence plots, and pass in each feature, shap values, test set, `Source_Following` as the feature to compare against and the computed position to the `dependence_plot()` method from the shap library which generates the dependence plots for each feature compared against `Source_Following` feature. The `delaxes(axs[4, 1])` call removes the last subplot as we only have 9 features so it would have been an empty plot. Spaces between the subplots are also configured and the grid containing all the subplots is shown.

4. Results

In this section, all the results attained after Link Prediction and the detailed analysis of our dataset using various plots is given. Each of the results and the interpretation of them are explored, providing many insights into the GitHub Social Network. The results can be divided into four sections:

i. Dataset Normalization
ii. Link Prediction
iii. GitHub Network Analysis
iv. SHAP Analysis

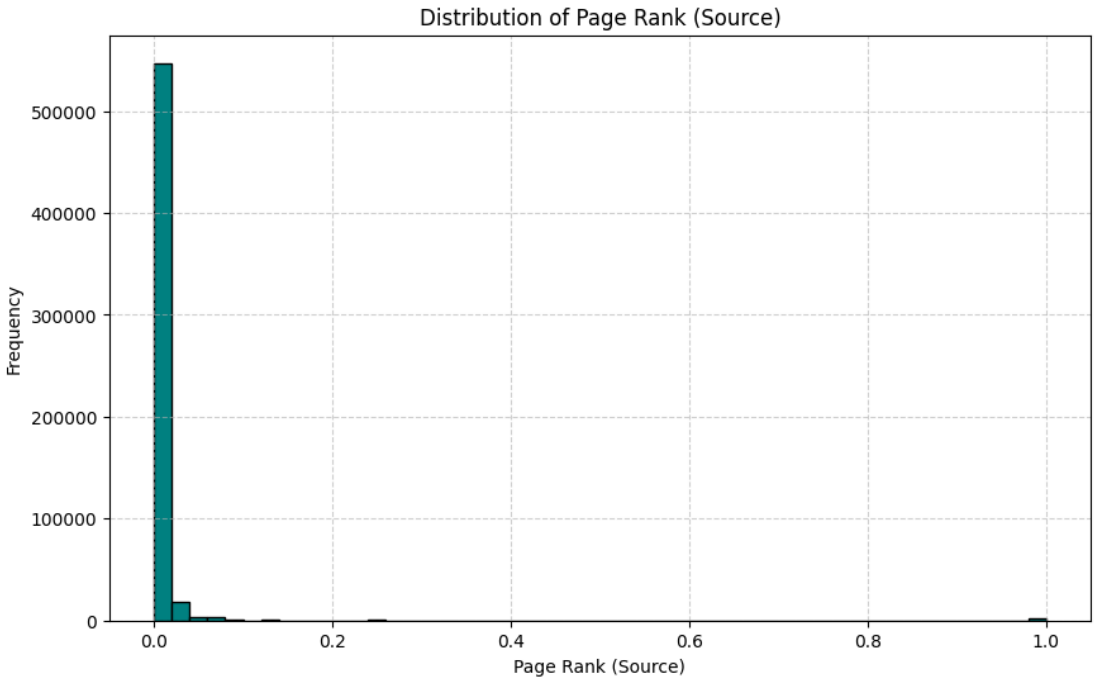
Find the flowchart that visualizes this section:


Start
Interpret Results of Dataset Normalization
Mutual Followers
Source Following
Source Followers
Page Rank Source
Shortest Path
Page Rank Target
Target Followers
Target Following
Mutual Following
Interpret Link Prediction Results
Logistic Regression
Random Forest
Support Vector Machine
Interpret Network Analysis Results
Degree Distribution
Distribution of Features with KDE Plot
Network w.r.t. Edge/Node Betweenness
Feature Correlation with a Heatmap
Feature Influence toward with Scatter Plots
Feature Relationships w.r.t. Pair Plot
Cummulative Degree Distribution
Interpret SHAP Analysis Results
Feature Influence using Force Plots
Feature Importance using Summary Plot
Feature Dependence vs. Important Feature using Dependence Plots
End

4.1. Dataset Normalization

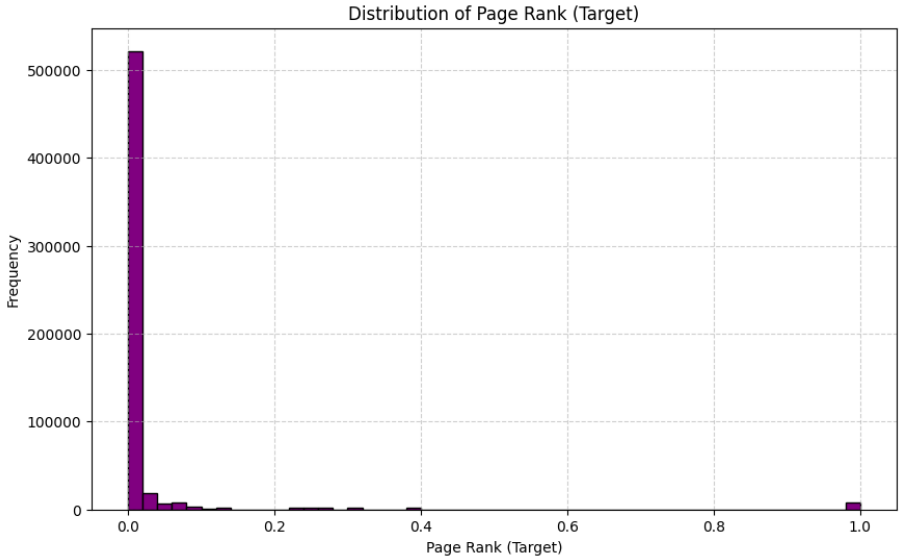
The normalization step involved using Min-Max Normalization to normalize the features computed for our dataset such as the PageRank values, Shortest Path lengths, and followship features. These normalized features were also visualized using histograms created with the help of Matplotlib's Pyplot. These plots are given below, along with their interpretations, i.e., what these features and their values mean for the GitHub Network.

4.1.1. Normalized Page Rank Source Column



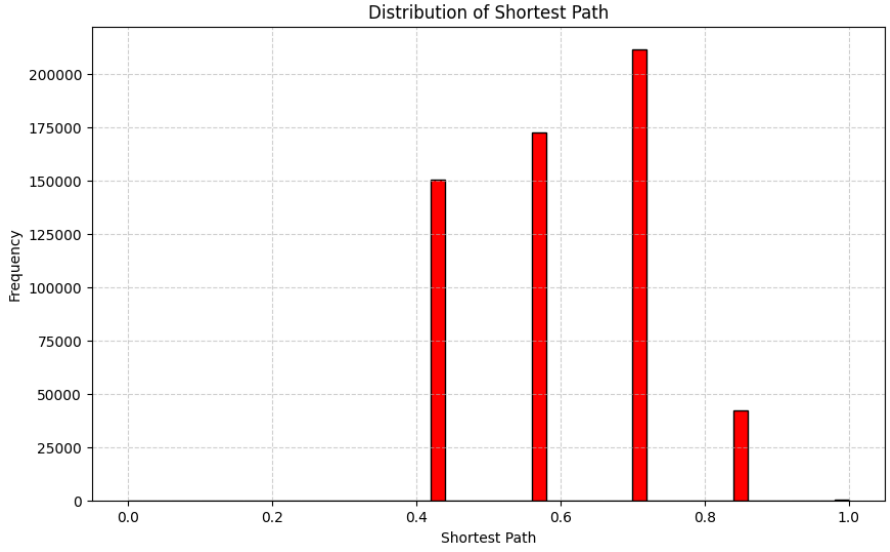
Interpretation: We see that almost all values in the column are skewed towards the left, at around 0.025 and lesser than that, this indicates that most source nodes in the network are relatively less important or have fewer incoming links as per PageRank. Only around 20000 samples seem to lie in the range 0.025 to 0.05 which shows that a small percentage, around 4% of the samples have a higher importance. Remaining values on the left side include around 5000 samples in total lying in the range 0.05 to 0.2 and around 1000 samples at 0.25, similarly an even smaller number of samples around 2% have a much higher importance. There also appears to be a very small number of samples, around 1500 with close to a perfect value of 1, indicating less than 1% of the source nodes have a large number of connections or followers.\

4.1.2. Normalized Page Rank Target Column



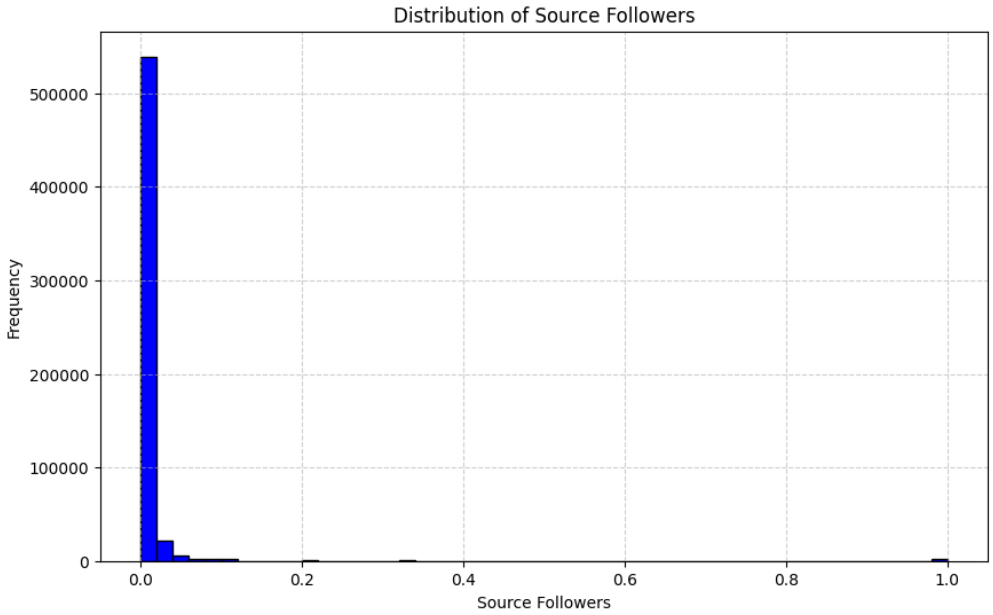
Interpretation: Similar to previous column, almost all values are skewed towards the left, at around 0.025 and lesser than that, this indicates that most target nodes in the network are relatively less important or have fewer incoming links as per PageRank. Only around 25000 samples seem to lie in the range 0.025 to 0.05 which shows that a small percentage, around 5% of the samples have a higher importance. Remaining values on the left half of the plot include around 20000 samples in total lying in the range 0.05 to 0.2 and around 5000 samples in the range 0.2 and 0.4. Therefore, 5% of the samples seem to have a relatively higher importance compared to the previous subset. There also appears to be a very small number of samples, around 10000 with close to a perfect value of 1, indicating around 2% of the target nodes have a large number of connections or followers.

4.1.3. Normalized Shortest Path Column



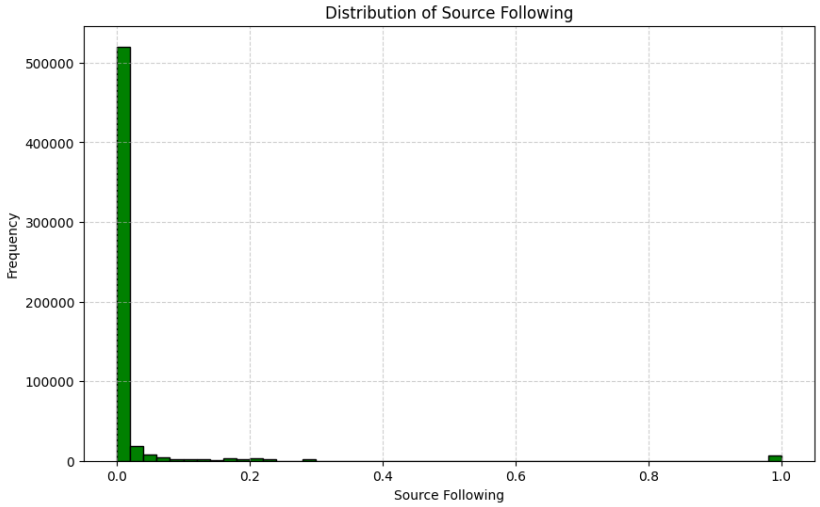
Interpretation: The distribution of the column values into five specific ranges, namely (0.41, 0.425), (0.57, 0.585), (0.7, 0.725), (0.825, 0.84), and (0.9, 1.0) indicates that the shortest paths are a fixed set of values. The most amount of nodes at around 250000 have a value of (0.7, 0.725) as the shortest path, with (0.57, 0.585) as the next highest frequency at around 174000, and (0.41, 0.425) in the middle at around 150000, (0.825, 0.84) in the second last at around 40000, and a very small amount at (0.9, 1). This kind of distribution indicates that most nodes are closely connected in the network, with relatively short paths and only around 30% of the nodes are sparsely connected and a very small number that are connected through very long path connecting them. This demonstrates the behavior of real world social networks accurately, as only a few popular people are followed by everyone while most people only have a small set of friends or acquaintances.

4.1.4. Normalized Source Followers Column



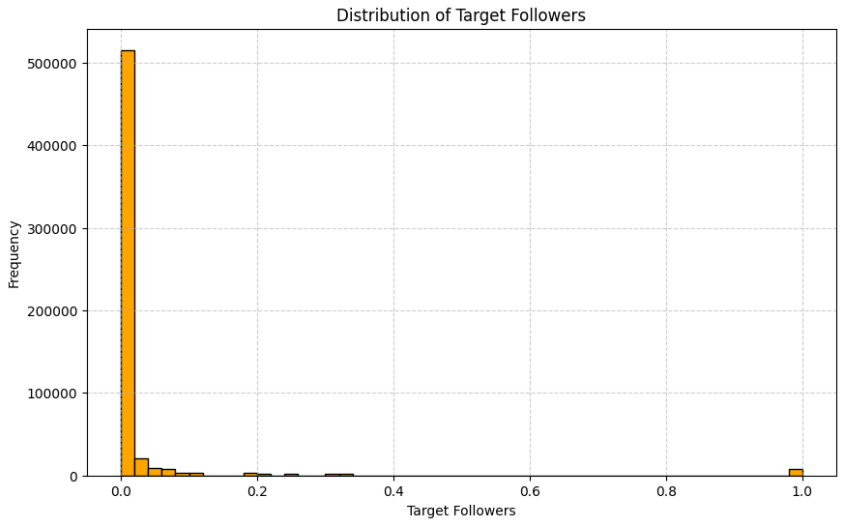
Interpretation: A similar skew towards the left of the graph is observed here as well. Around 7% of data appears after the huge first bar but still the count is low, and an extremely marginal amount of nodes are very close to 1 at the right end. So what this means is that, almost 90% of the developers have quite few or almost no followers. This kind of behavior is typical in real-world social networks where a few developers are quite influential, essentially acting as the hubs of the network connecting everyone as almost everyone follow these developers. These developers could be organizations such as Meta, Amazon or influential people such as Tom Payne, Asim Aslam and so on. Power-Law distribution, a phenomena in real-world social networks where popular people keep becoming more popular while the average stay average, can be seen here.

4.1.5. Normalized Source Following Column



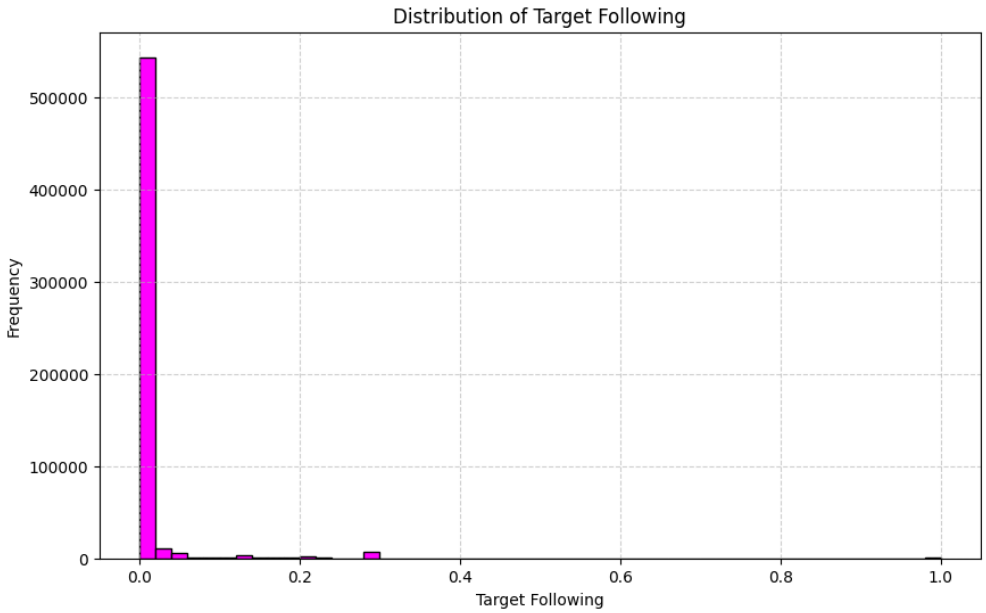
Interpretation: We see a leftward skew of the values with around 525000 nodes having a very low count in and around 0 to 0.1. The remaining 25000 or so nodes in left side have slightly higher following counts but they are still in the left of the graph suggesting their following is still low. However, around 10000 nodes are at the very end of the plot, meaning their following counts are quite high. So what does this distribution of data mean? It means most developers are casual folk who only follow their friends, colleagues or people with similar interests, and some developers follow people at a slightly higher rate i.e., they are not so selective. On the other hand, a very small subset of developers follow an extremely large number of other developers as their following count is as high as it can be, suggesting that these developers are extremely active in the community and they're enthusiastic to follow everybody in a number of different fields. On the other hand, these nodes with high following counts could also be bots deployed by organizations or even malicious bots that want to steal data, keeping track of the activities of a large group of related developers.

4.1.6. Normalized Target Followers Column



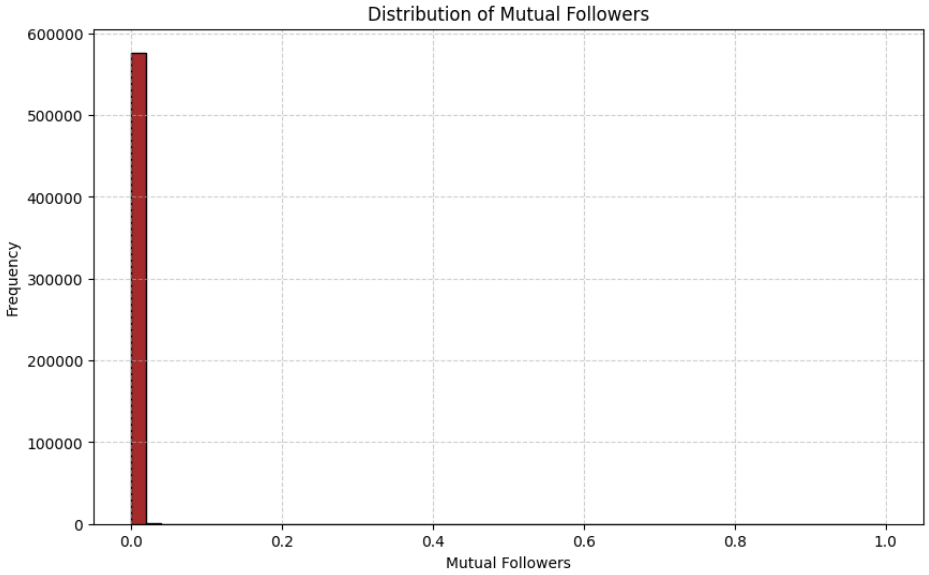
Interpretation: The same behavior from Source Followers is prevalent in Target Followers as well. We see almost 90% of the developers have low number of followers while a very small number of developers have all the followers. Thus, target nodes i.e., developers in the social network also exhibit the same behavior that is indicative of the power-law distribution.

4.1.7. Normalized Target Following Column



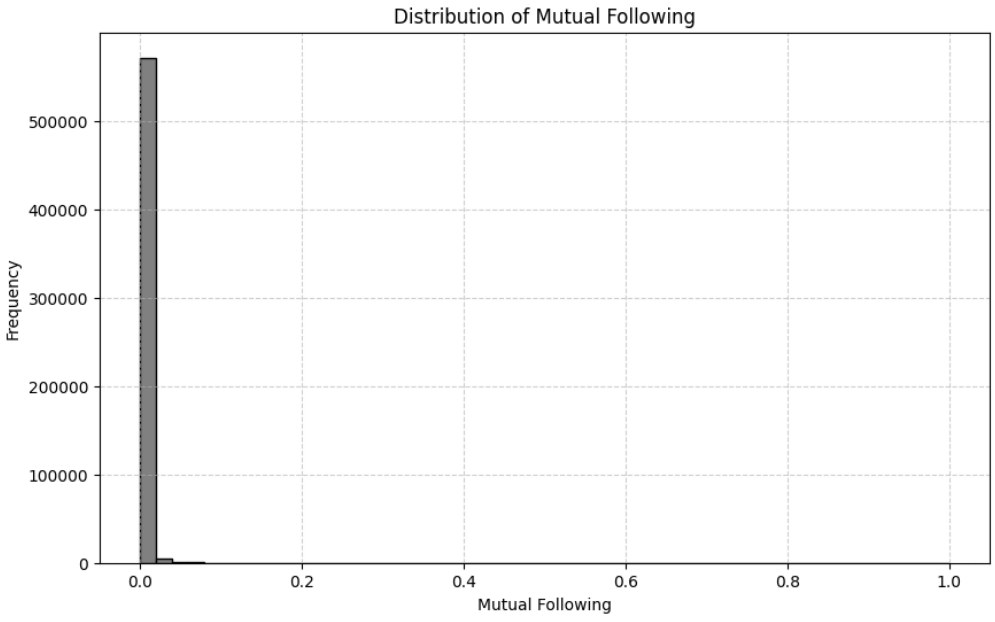
Interpretation: Similar to the Source following column, the target nodes or developers also mostly follow a small number of people in their own immediate circles. The rest of the developers have slightly higher following counts with dev count peaking around 10000 developers at around the range (0.275, 0.3). It seems that a small subset of developers, maybe in a particular field, have the habit of following more people in their own industry. There's also an extremely small number of devs around 2.5% of the network who like to follow almost everyone. As explained before these are most likely bots as Humans would not be able to accumulate that kind of following count. In the real world, 1000s of bots crawl around GitHub Repositories hoping to find secrets such as API keys, important passwords, and so on that owners of these bots, likely malicious people can exploit for monetary gains obtained by forgery, DoS (Denial of Service) attack, etc.

4.1.8. Normalized Mutual Followers Column



Interpretation: An overwhelming majority of the developers have mutual follower count that is very close to 0, while an iota of developers have a slightly higher count starting at around 0.03. This indicates that most of the developers when compare pair-wise do not share a large number of mutual followers, meaning that our network of GitHub Developers is quite sparse or not densely interconnected. The very small number of developers, around 1% with a higher mutual follower count could be because these devs are part of the same field or team or organization who are followed by the other developers who belong to the same group. However, as evident from the plot, this kind of relationship is highly rare.

4.1.9. Normalized Mutual Following Column



Interpretation: We see almost the same trend as in Mutual Followers here as well with 97% of the developers having a very small mutual following count. However, a small but noticeable difference is that, here around 3% of the developers have a higher number of following count, meaning some developers follow the same people in the network. This is explainable due to the nature of a social network where only a few people are the most popular, thus it makes sense for some group of developers to follow these other popular developers. This kind of effect didn't apply to mutual followers because most developers are average and unknown, making the majority of mutual follower count close to 0, while on the other hand, only a small number of developers are extremely popular, therefore increasing the mutual following count for a very small number of developers. Thus, mutual followship features reflect the nature of real world social networks.

4.2. Link Prediction

The next step in the project involved the normalized GitHub Network Dataset to train three Machine Learning models, Logistic Regression, Random Forest, and Support Vector Machine. These models were then tested on a test set derived from the dataset, results of which were shown in a classification report and was visualized using plots such as Confusion Matrix, Receiver Operating Characteristic Curve, & Precision-Recall Curve. This section provides these results and the interpretations of these results, i.e., how well did our ML models do in the Link Prediction task for our GitHub Network.

4.2.1. Logistic Regression Classifier

The first model that was trained on the GitHub Network Dataset was the Logistic Regression Classifier model.

4.2.1.1. Training Results

```
Time taken for Logistic Regression training: 3.96 seconds

Optimal parameters identified: {'penalty': 'l1', 'C': 1.0}

Highest accuracy score: 0.8675909042358398
```

Interpretation: The model was trained in around 4 seconds. This high speed was thanks to the high speed computed of an Nvidia GPU which this model was able to take advantage of due to it being the cuML version of the Logistic Regression model which was designed to make use of the CUDA API. After trying multiple combinations of training parameters, a penalty of l1 and a regularization strength of 1 was found to be the most optimal. The highest accuracy obtained when tested against a test set is around 87%, meaning our model predicts an existing link or a missing

link between a source and target node correctly 87% of the time. This is a great score as per industry standards for the Logistic Regression model.

4.2.1.2. Classification Report

	precision	recall	f1-score	support
0	0.83	0.92	0.87	86926
1	0.91	0.81	0.86	86476
accuracy			0.87	173402
macro avg	0.87	0.87	0.87	173402
weighted avg	0.87	0.87	0.87	173402

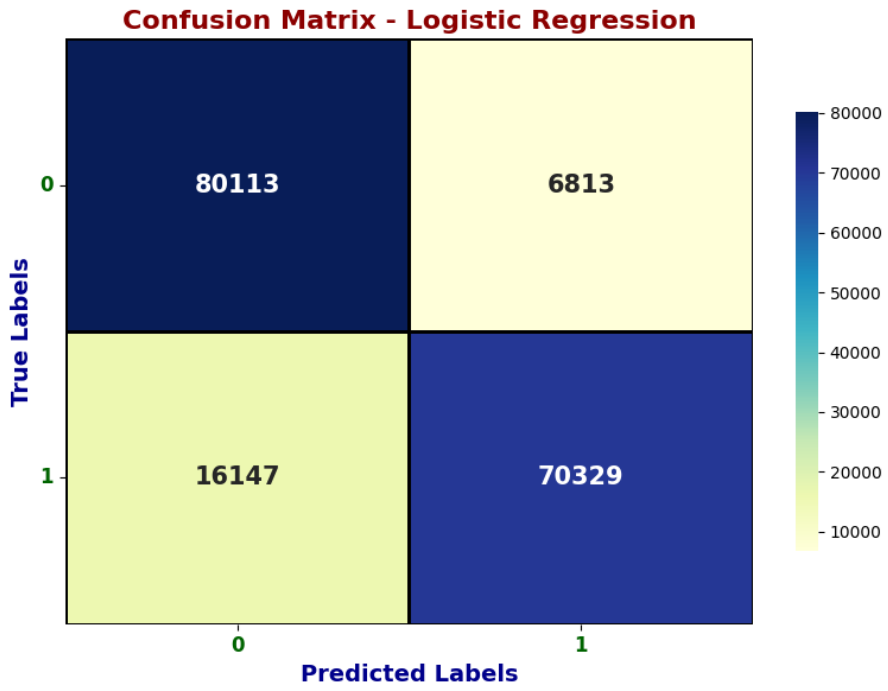
Interpretation:

Class – 0: The Precision is 0.83, meaning out of all model predictions for class 0, 83% of them were correct predictions. The Recall is 0.92, meaning out of all actual class 0 instances that exist, the model was able to identify 92% of them correctly. The F1 score if 0.87, which is the mean of the Precision and Recall scores, providing an overall accuracy score. Support indicates the number of class 0 instances in the test set, which in our case is 86926, which forms a decent baseline size for our testing.

Class – 1: The Precision is 0.91, meaning out of all model predictions for class 1, 91% of them were correct predictions. The Recall is 0.81, meaning out of all actual class 1 instances that exist, the model was able to identify 81% of them correctly. The F1 score is 0.86, which is slightly lower than that of the F1 score for Class 0 but the difference is negligible. Support indicates the number of class 1 instances in the test set, which here is 86476, which again is a good enough size for testing purposes.

Overall: Accuracy of the model across the classes is found to be 0.87, meaning the class predictions are correct 87% of the time which is very good odds. Macro average provides the unweighted metrics of the classes, treating each classes individually and equally, and computing the average of the scores, which here is 87% for all three metrics. Lastly, the weighted average accounts for the number of instances that each class has in the test set and adjusts the scores accordingly, which again is found to be 87% which is understandable considering that the test set is fairly balanced so weighing the scores does not make much difference, therefore, the macro and weighted averages are the same.

4.2.1.3. Confusion Matrix



Interpretation: The true labels and the predicted labels by the model are compared in this plot , with the hues in each of the cell highlighting the sample size found for True Negatives (TN), False Positives (FP), False Negatives (FN), and True Positives (TP).

For the **True Negatives**, the value given is 80,113, which means that the model correctly predicted that many class 0 instances as TN indicates the number of cases where both the actual class from the test set and the predicted class were both 0.

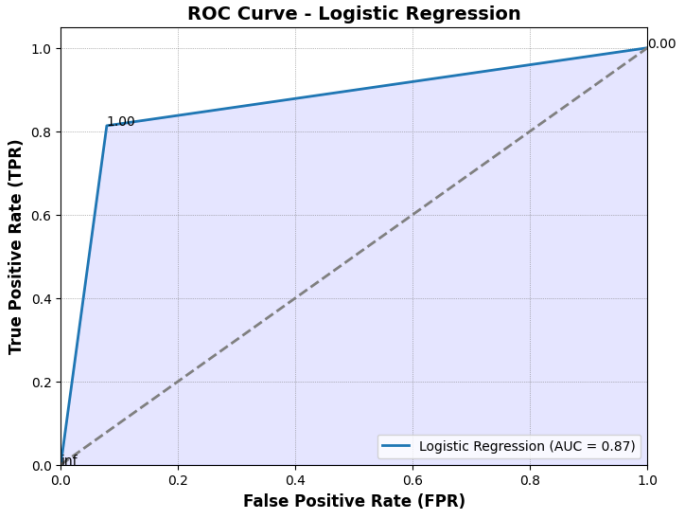
For the **False Positives**, the value found is 6,813, which means that the model incorrectly predicted that many class 1 instances as FP indicates the number of cases where the actual class from the test set was class 0 while the predicted class was class 1. These can even be seen as false alarms, where the model thought there was a link between the source and target nodes but it turns out, there was no link.

For the **False Negatives**, the value is 16,147, which means that the model incorrectly predicted that many class 0 instances as FN indicates the number of cases where the actual class from the test set was class 1 while the predicted class was class 0. These can be thought of as missed cases where the model thought there was no link between the source and target nodes but it turns out that there was a link.

For the **True Positives**, the value seen is 70,329, which means the model correctly predicted that many class 1 instances as TP indicates the number of cases where both the actual class from the test set and the predicted class were both 1.

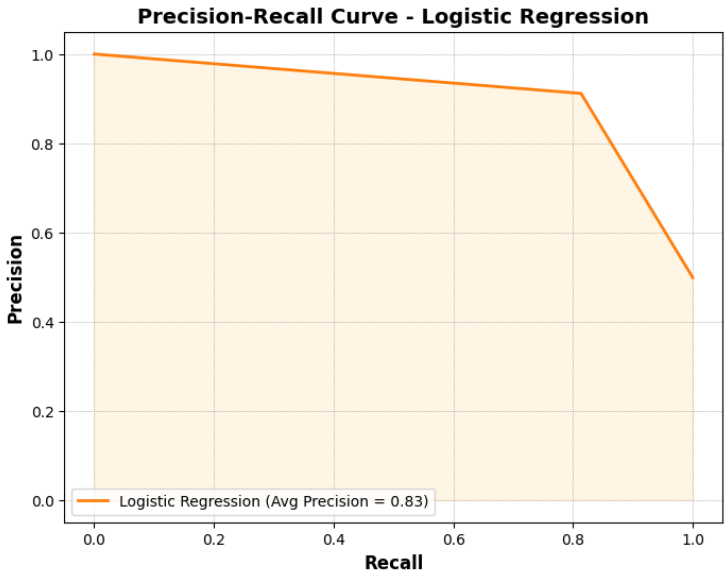
Overall, the model's performance is quite balanced, with around 70,329 True Positives and 80,113 True Negatives.

4.2.1.4. Receiver Operating Characteristic Curve



Interpretation: This curve gives us the performance of the model across two thresholds, which are True Positive Rate which can be defined as the ratio of actual class 1 instances correctly predicted by the model, computed by TP / (TP + FN), and False Positive Rate which can be defined as the ratio of actual class 0 instances incorrectly classified as class 1 by the model, computed by FPR = FP / (FP + TN). The Diagonal line in the plot can be interpreted as a baseline that shows the performance of a random model with the area under the curve (AUC) equaling 0.5. If our model's AUC is closer to this baseline, then we can conclude that our model performs very poorly as in with a AUC of 50%, its predictions may as well be random. However, our model's AUC score is 0.87 or 87%, meaning it has great ability to distinguish between our dataset's classes.

4.2.1.5. Precision-Recall Curve



Interpretation: The Precision-Recall Curve evaluates how the Precision and Recall of our model are related to each other. Precision and Recall as we know are the ratios of correct predictions of actual class 1s and actual class 1s that were correctly predicted. When a plot is created with these two and the area under the curve is measured, the value is 0.83, which indicates that our model identifies positive instances correctly 83% of the time.

4.2.2. Random Forest Classifier

The next model that was trained on the GitHub Network Dataset was the Random Forest Classifier model.

4.2.2.1. Training Results

```
Time taken to train Random Forest model: 37.44 seconds

Best parameters found: {'n_estimators': 50, 'max_depth': 24,
'min_samples_split': 121, 'min_samples_leaf': 20}

Best accuracy score: 0.8813854455947876
```

Interpretation: The model was trained in around 38 seconds. The speed of training is again quite fast thanks to the cuML version of the Random Forest model making use of the CUDA API. Here, we train with different variations of parameters for the model and the best parameters found are as follows. The number of decision trees for the forest in the model, best for this use-case is found to be 50 (from 50, 100, 150) which could be because when the conditions are right, lower number of trees significantly reduce computation time while also increasing the model's performance. The best depth found from the set of depths {9, 12, 15, 24} was 24, meaning that the trees in the model have that many layers and this could be due to the fact that deeper trees are able to capture more details compared to smaller trees. The optimal value for the minimum number of samples each node must have before splitting further into more is found to be 121 and this could've been chosen because it allows the model to avoid splitting often which could lead to poor performance. The minimum number of samples each leaf should have is fixed at 20, which seems to be just good enough for each leaf to capture the required information without any noise. Finally, the best accuracy that was obtained from testing is found to be around 88%, which means the RF model succeeds in predicting a link or no link 88% of the time. This score is extremely good for a binary classification problem obtained by a Random Forest Model as per industry standards.

4.2.2.2. Classification Report

	precision	recall	f1-score	support
0	0.86	0.91	0.88	86926
1	0.90	0.86	0.88	86476
accuracy			0.88	173402
macro avg	0.88	0.88	0.88	173402
weighted avg	0.88	0.88	0.88	173402

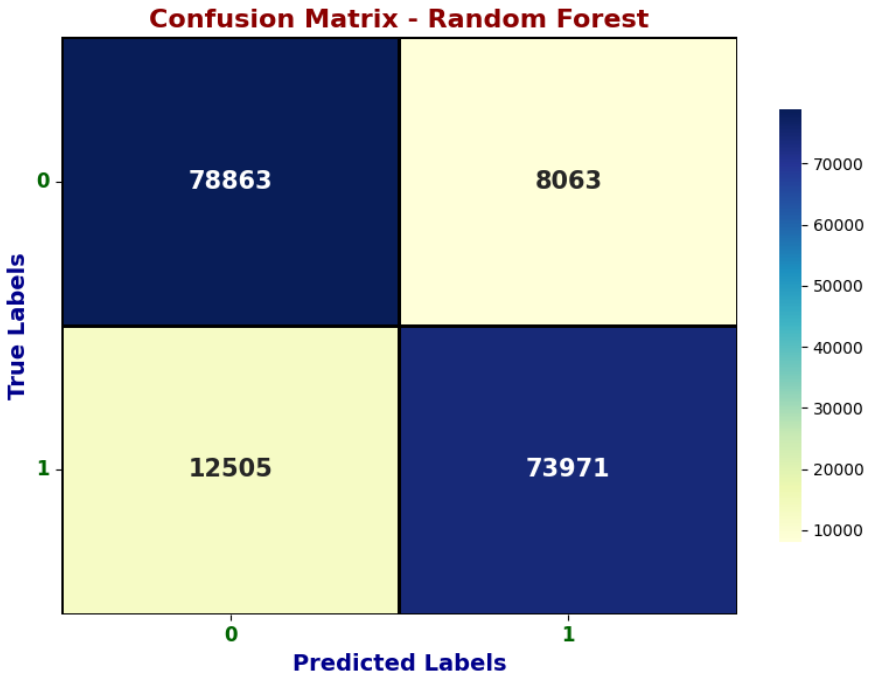
Interpretation:

Class – 0: The Precision is 0.86, meaning out of all model predictions for class 0, 86% of them were correct predictions. The Recall is 0.91, meaning out of all actual class 0 instances that exist, the model was able to identify 91% of them correctly. The F1 score of 0.88, which is the mean of the Precision and Recall scores, providing an overall accuracy score. Support indicates the number of class 0 instances in the test set, which in our case is 86926, which forms a decent baseline size for our testing.

Class – 1: The Precision is 0.90, meaning out of all model predictions for class 1, 90% of them were correct predictions. The Recall is 0.86, meaning out of all actual class 1 instances that exist, the model was able to identify 86% of them correctly. The F1 score is 0.88 is the same as the F1 score of class 1, meaning our model performs equally well for both classes. Support indicates the number of class 1 instances in the test set, which here is 86476, which again is a good enough size for testing purposes.

Overall: Accuracy of the model across the classes is found to be 0.88, meaning the class predictions are correct 88% of the time which is very good odds. Macro average provides the unweighted metrics of the classes, treating each classes individually and equally, and computing the average of the scores, which here is 88% for all three metrics. Lastly, the weighted average accounts for the number of instances that each class has in the test set and adjusts the scores accordingly, which again is found to be 88% which is understandable considering that the test set is fairly balanced so weighing the scores does not make much difference, therefore, the macro and weighted averages are the same.

4.2.2.3. Confusion Matrix



Interpretation: The true labels and the predicted labels by the model are compared in this plot , with the hues in each of the cell highlighting the sample size found for True Negatives (TN), False Positives (FP), False Negatives (FN), and True Positives (TP).

For the **True Negatives**, the value given is 78,863, which means that the model correctly predicted that many class 0 instances as TN indicates the number of cases where both the actual class from the test set and the predicted class were both 0.

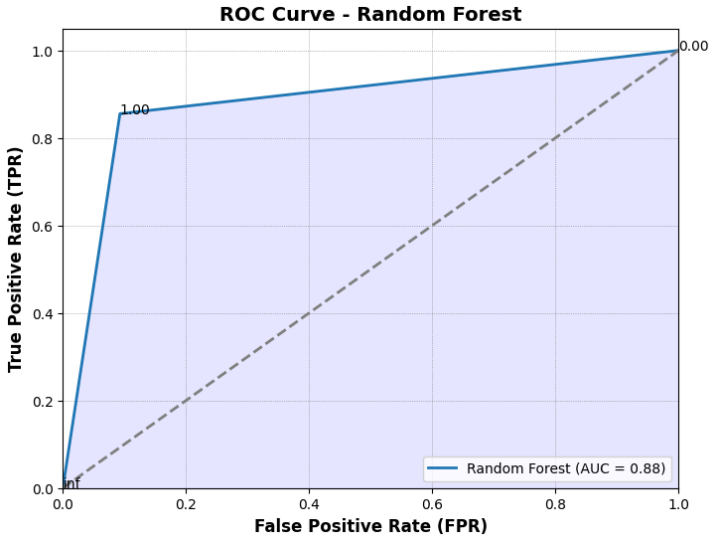
For the **False Positives**, the value found is 8,063, which means that the model incorrectly predicted that many class 1 instances as FP indicates the number of cases where the actual class from the test set was class 0 while the predicted class was class 1. These can even be seen as false alarms, where the model thought there was a link between the source and target nodes but it turns out, there was no link.

For the **False Negatives**, the value is 12,505, which means that the model incorrectly predicted that many class 0 instances as FN indicates the number of cases where the actual class from the test set was class 1 while the predicted class was class 0. These can be thought of as missed cases where the model thought there was no link between the source and target nodes but it turns out that there was a link.

For the **True Positives**, the value seen is 73,971, which means the model correctly predicted that many class 1 instances as TP indicates the number of cases where both the actual class from the test set and the predicted class were both 1.

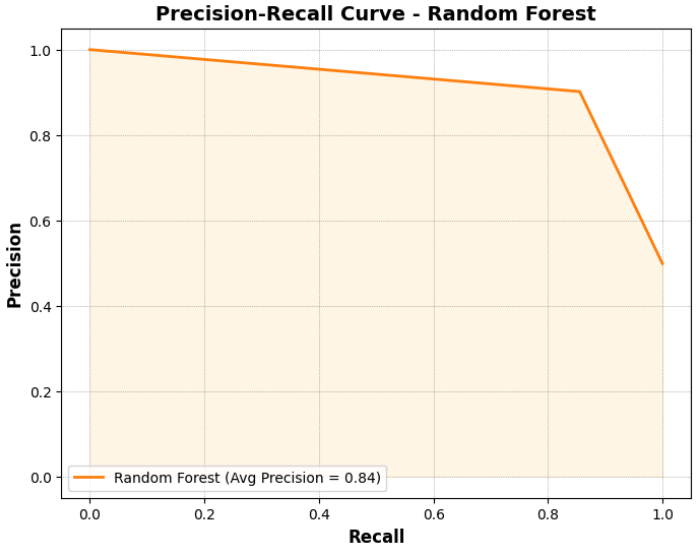
Overall, the model's performance is quite balanced, with around 73,971 True Positives and 78,863 True Negatives.

4.2.2.4. Receiver Operating Characteristic Curve



Interpretation: This curve gives us the performance of the model across two thresholds, which are True Positive Rate which can be defined as the ratio of actual class 1 instances correctly predicted by the model, computed by TP / (TP + FN), and False Positive Rate which can be defined as the ratio of actual class 0 instances incorrectly classified as class 1 by the model, computed by FPR = FP / (FP + TN). The Diagonal line in the plot can be interpreted as a baseline that shows the performance of a random model with the area under the curve (AUC) equaling 0.5. If our model's AUC is closer to this baseline, then we can conclude that our model performs very poorly as in with a AUC of 50%, its predictions may as well be random. However, our model's AUC score is 0.88 or 88%, meaning it has great ability to distinguish between our dataset's classes.

4.2.2.5. Precision-Recall Curve



Interpretation: The Precision-Recall Curve evaluates how the Precision and Recall of our model are related to each other. Precision and Recall as we know are the ratios of correct predictions of actual class 1s and actual class 1s that were correctly predicted. When a plot is created with these two and the area under the curve is measured, the value is 0.84, which indicates that our model identifies positive instances correctly 84% of the time.

4.2.3. Support Vector Machine Classifier

The third model that was trained on the GitHub Network Dataset was the Support Vector Machine Classifier.

4.2.3.1. Training Results

```
Time taken to train SVM model: 152.67 seconds

Best parameters: {'kernel': 'linear', 'C': 1}

Best accuracy score: 0.8578909039497375
```

Interpretation: This model was trained in around 158 seconds, which is quite fast for the SVM model. This is possible, thanks again to cuML and CUDA. The model here is also trained using several parameters and their combinations so that the best model can be found. The kernel in an SVM is a mathematical function that determines how the training data should be used, organized and classified. Here, linear kernel seems to work the best, which could be because our task is binary classification which in and of itself is linearly separable. The regularization strength that was found to be the best was the highest in the range of values {0.001, 0.01, 0.1, 1}, i.e., 1. This is a moderate score and it would appear that it balances between overfitting and underfitting perfectly. In the end, the accuracy that was the best to be achieved by the SVM model with the aforementioned parameters is around 86%. Thus, the model predicts a link or a missing link between source and target nodes in the GitHub network 86% of the time. The score for this model is quite good as well when checked against industry standards.

4.2.3.2. Classification Report

	precision	recall	f1-score	support
0	0.82	0.91	0.87	86926
1	0.90	0.80	0.85	86476
accuracy			0.86	173402
macro avg	0.86	0.86	0.86	173402

```
weighted avg       0.86      0.86      0.86    173402
```

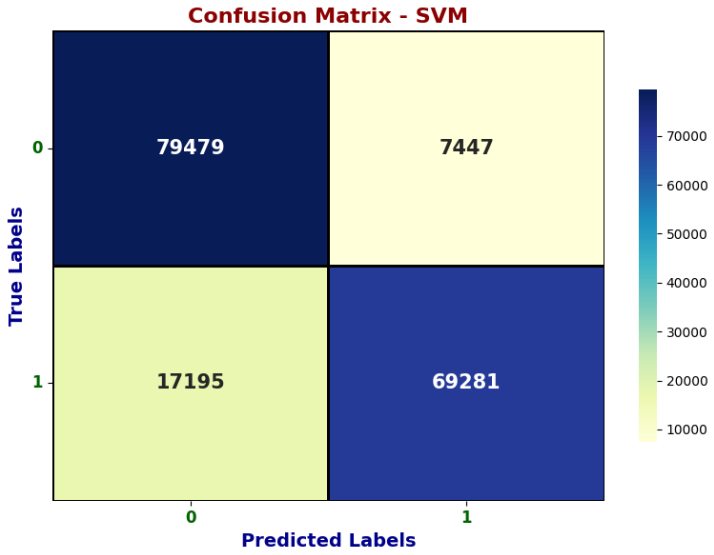
Interpretation:

Class – 0: The Precision is 0.82, meaning out of all model predictions for class 0, 82% of them were correct predictions. The Recall is 0.91, meaning out of all actual class 0 instances that exist, the model was able to identify 91% of them correctly. The F1 score of 0.87, which is the mean of the Precision and Recall scores, providing an overall accuracy score. Support indicates the number of class 0 instances in the test set, which in our case is 86926, which forms a decent baseline size for our testing.

Class – 1: The Precision is 0.90, meaning out of all model predictions for class 1, 90% of them were correct predictions. The Recall is 0.80, meaning out of all actual class 1 instances that exist, the model was able to identify 80% of them correctly. The F1 score is 0.85, which is slightly lower than the F1 score for Class 0 but the difference is negligible. Support indicates the number of class 1 instances in the test set, which here is 86476, which again is a good enough size for testing purposes.

Overall: Accuracy of the model across the classes is found to be 0.86, meaning the class predictions are correct 86% of the time which is very good odds. Macro average provides the unweighted metrics of the classes, treating each classes individually and equally, and computing the average of the scores, which here is 86% for all three metrics. Lastly, the weighted average accounts for the number of instances that each class has in the test set and adjusts the scores accordingly, which again is found to be 86% which is understandable considering that the test set is fairly balanced so weighing the scores does not make much difference, therefore, the macro and weighted averages are the same.

4.2.3.3. Confusion Matrix



Interpretation: The true labels and the predicted labels by the model are compared in this plot , with the hues in each of the cell highlighting the sample size found for True Negatives (TN), False Positives (FP), False Negatives (FN), and True Positives (TP).

For the **True Negatives**, the value given is 79,479, which means that the model correctly predicted that many class 0 instances as TN indicates the number of cases where both the actual class from the test set and the predicted class were both 0.

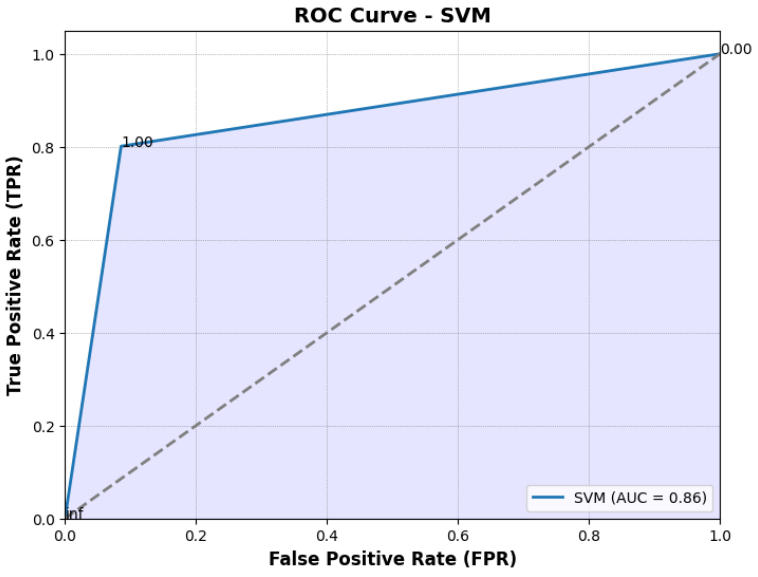
For the **False Positives**, the value found is 7,447, which means that the model incorrectly predicted that many class 1 instances as FP indicates the number of cases where the actual class from the test set was class 0 while the predicted class was class 1. These can even be seen as false alarms, where the model thought there was a link between the source and target nodes but it turns out, there was no link.

For the **False Negatives**, the value is 17,195, which means that the model incorrectly predicted that many class 0 instances as FN indicates the number of cases where the actual class from the test set was class 1 while the predicted class was class 0. These can be thought of as missed cases where the model thought there was no link between the source and target nodes but it turns out that there was a link.

For the **True Positives**, the value seen is 69,281, which means the model correctly predicted that many class 1 instances as TP indicates the number of cases where both the actual class from the test set and the predicted class were both 1.

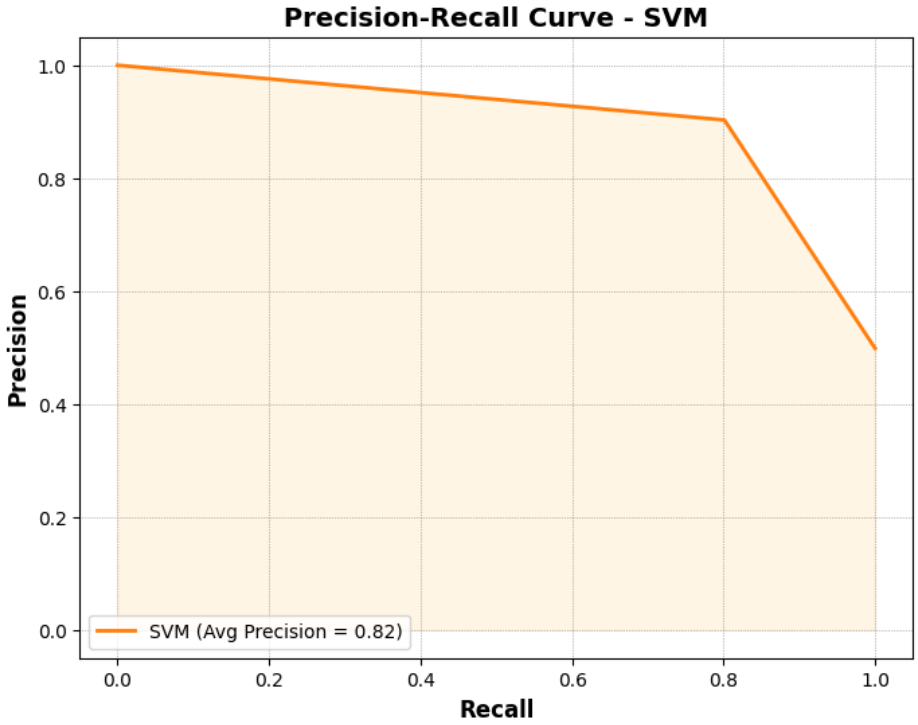
Overall, the model's performance is quite balanced, with around 69,281 True Positives and 79,479 True Negatives.

4.2.3.4. Receiver Operating Characteristic Curve



Interpretation: This curve gives us the performance of the model across two thresholds, which are True Positive Rate which can be defined as the ratio of actual class 1 instances correctly predicted by the model, computed by TP / (TP + FN), and False Positive Rate which can be defined as the ratio of actual class 0 instances incorrectly classified as class 1 by the model, computed by FPR = FP / (FP + TN). The Diagonal line in the plot can be interpreted as a baseline that shows the performance of a random model with the area under the curve (AUC) equaling 0.5. If our model's AUC is closer to this baseline, then we can conclude that our model performs very poorly as in with a AUC of 50%, its predictions may as well be random. However, our model's AUC score is 0.86 or 86%, meaning it has great ability to distinguish between our dataset's classes.

4.2.3.5. Precision-Recall Curve

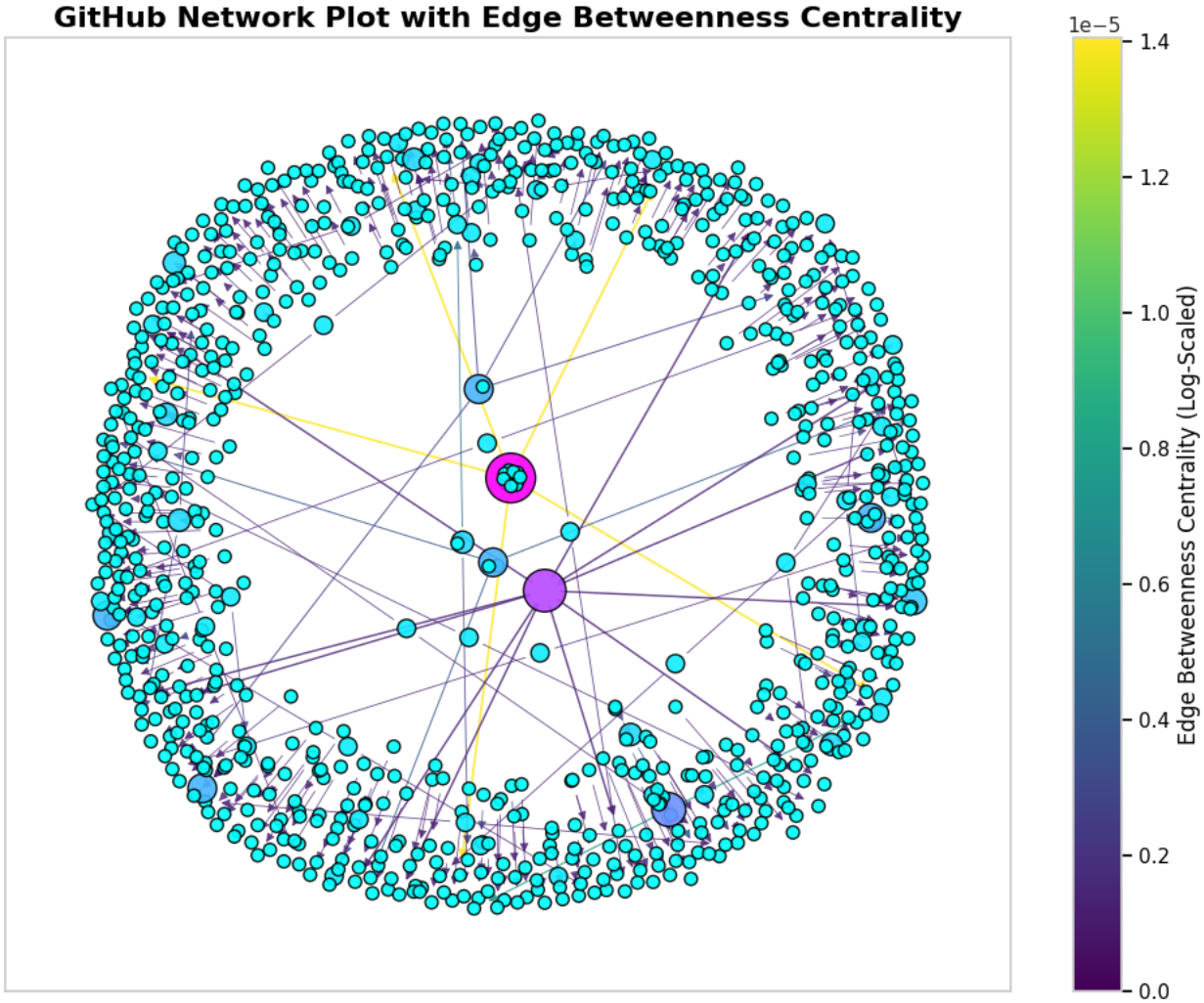


Interpretation: The Precision-Recall Curve evaluates how the Precision and Recall of our model are related to each other. Precision and Recall as we know are the ratios of correct predictions of actual class 1s and actual class 1s that were correctly predicted. When a plot is created with these two and the area under the curve is measured, the value is 0.82, which indicates that our model identifies positive instances correctly 82% of the time.

4.3. GitHub Network Analysis

The GitHub network was analyzed in many ways, using a number of visualizations such as the network itself with respect to Edge and Node Betweenness, Feature Correlation, Influence, Distribution, & Relationships using Heatmap, Scatter, Kernel Density Estimate, & Pair plots, and Degree of nodes such as Degree Distribution and Cumulative Degree Distribution plots. The results and interpretations of these visualizations are included in this section, i.e., the interesting insights that were gained from analyzing the GitHub network.

4.3.1. Network Plot with respect to Edge Betweenness Centrality

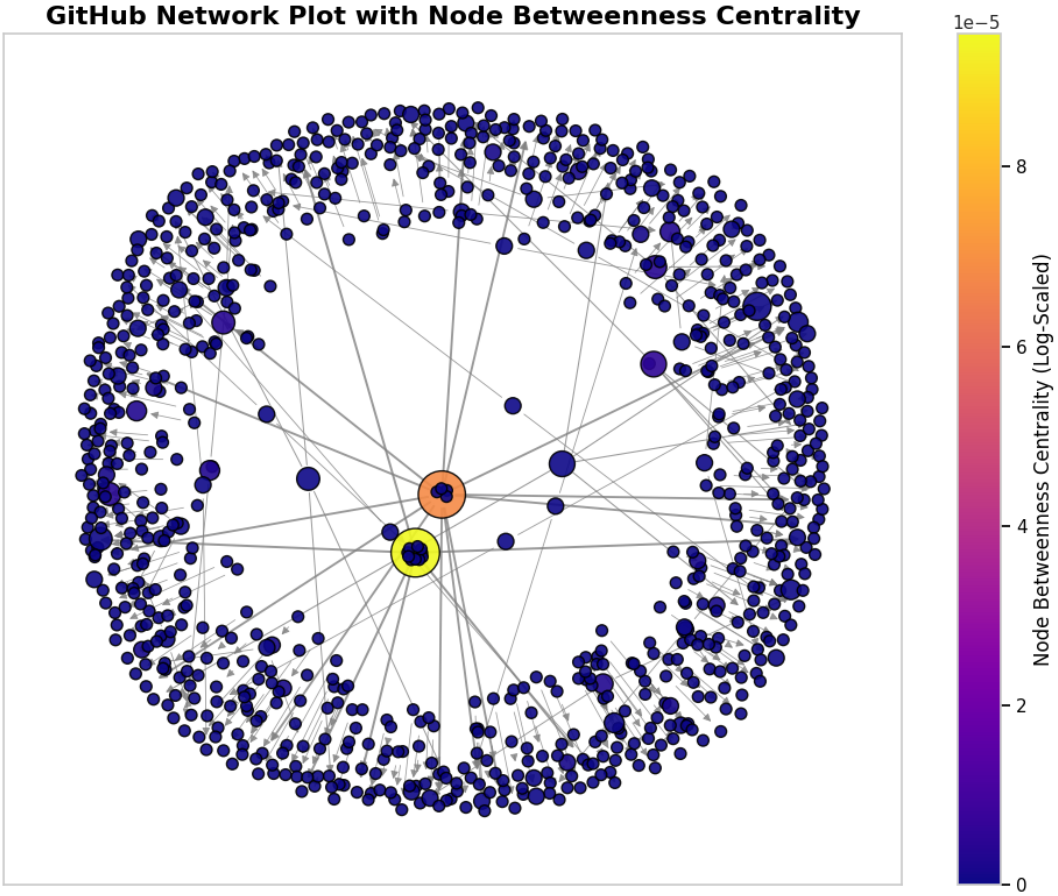


Note: Node size & color are proportional to node degree (number of connections)

Interpretation: The network plot visualizes a small subset of the network, i.e., around 500 nodes to get a look into the properties of the GitHub network. We see that the nodes in the center of the

network are bigger than the others and their colors are stark in comparison with the outer nodes which as per the code that was written means that these nodes have a higher degree than other nodes as the node sizes are dependent on them. These nodes can be seen to connect various parts of the network and the lighter colors of their edges denotes that their edge betweenness centrality is also high. This could mean that some developers in the GitHub network are prominent and followed by a large number of people from different communities. These developers essentially act as hubs for these sub-communities of developers to connect and removing them could fragment or disconnect different parts of the network, thus highlighting their importance. On the other hand, most nodes are found on the outer edge of the network, meaning that these nodes are less connected or sparse as our layout is spring layout which has the property that if a node has more connections, the more it will be pulled to the center. This means that these edge nodes are developers who only follow or are followed by a small group of people or they're not very active in the community. This kind of structure where a small set of people are surrounded by a large group of people is known as a hub-and-spoke structure and it is a typical structure in real life social networks.

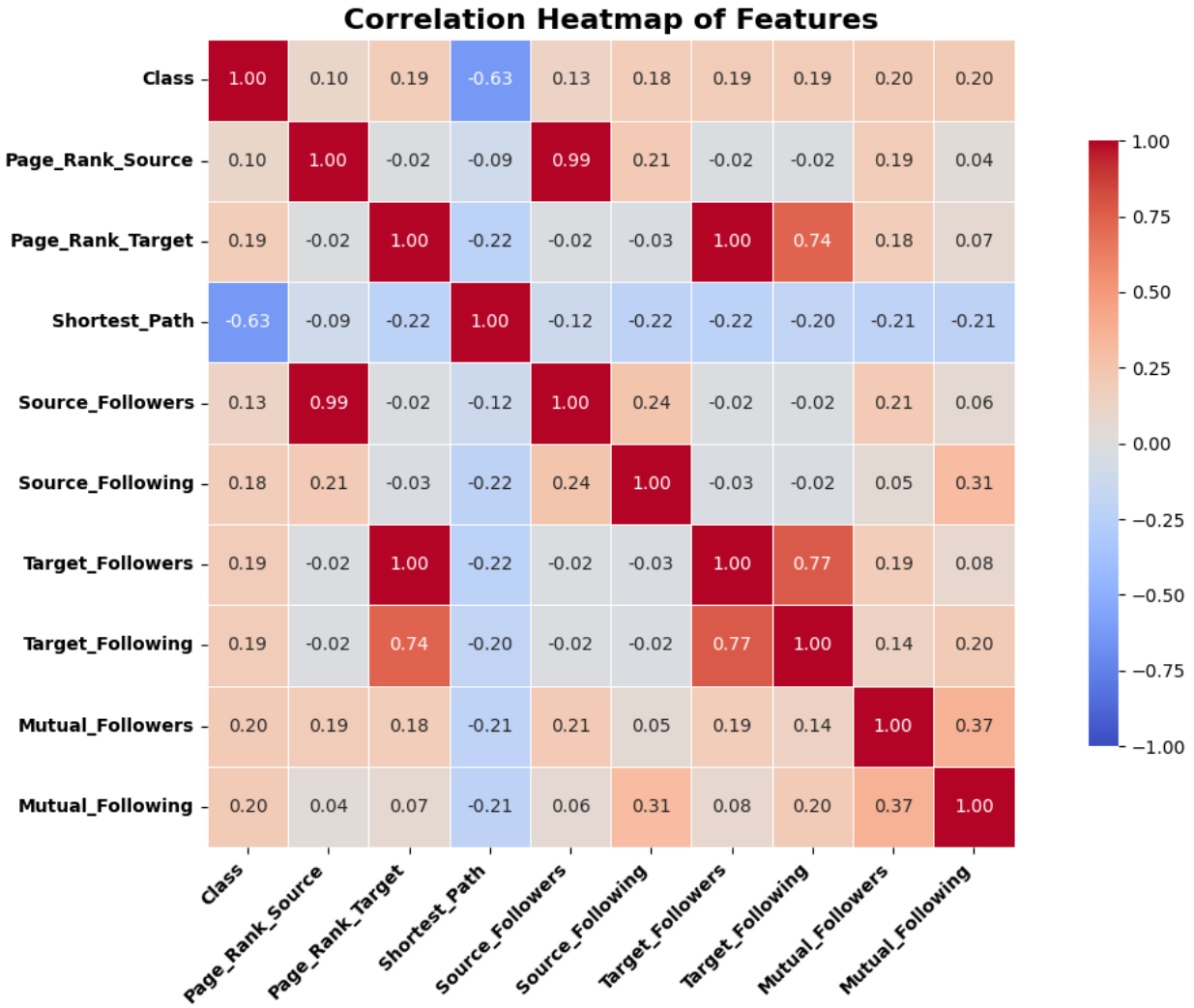
4.3.2. Network Plot with respect to Node Betweenness Centrality



Note: Node size is proportional to node degree (number of connections)

Interpretation: A similar pattern could be observed with this network plot as well. The key difference however is that the previous plot stressed the importance of nodes with respect to the number of connections but here the focus is on nodes that facilitate the connection of highest number of nodes in the network. These nodes in the center could be organizations that are responsible for the spread of knowledge, setting industry trends, new standards for existing technologies and so on, thus being able to influence the entire network across multiple regions. These nodes are again proof of the centralized nature of the GitHub network, which while it has its advantages, is also prone to risks because losing these hub nodes could lead to the network being isolated into their own communities. On the other hand, they're high connectivity could also lead to more collaboration and cross-community knowledge-sharing which is crucial in the tech industry. Thus, the network plot provides good insights in the cohesive and robust nature of the GitHub developer network.

4.3.3. Feature Correlation with a Heatmap



Interpretation: The heatmap shows a correlation matrix which contains the relationship between the features of the GitHub Network. If the correlation value corresponding to the two features is positive, then it means both features increase together, if it is negative, then the features have an inverse relationship, and if it is neutral, then the features have no relationship between each other. Let's look at some interesting relationships that are more notable than the others.

For the **Class** and **Shortest Path**, we see that the correlation is -0.63. This indicates that when shortest path decreases, class value increases. This makes sense because a class of 0 indicates a missing link and a class of 1 indicates an existing link and if the path between two nodes is shorter, then the odds of them being connected are higher. This notable negative correlations shows the importance of Shortest Path for the Link Prediction task, and how it is possible to determine if two developers could be connected by analyzing the shortest distance at which they exist from each other in the network.

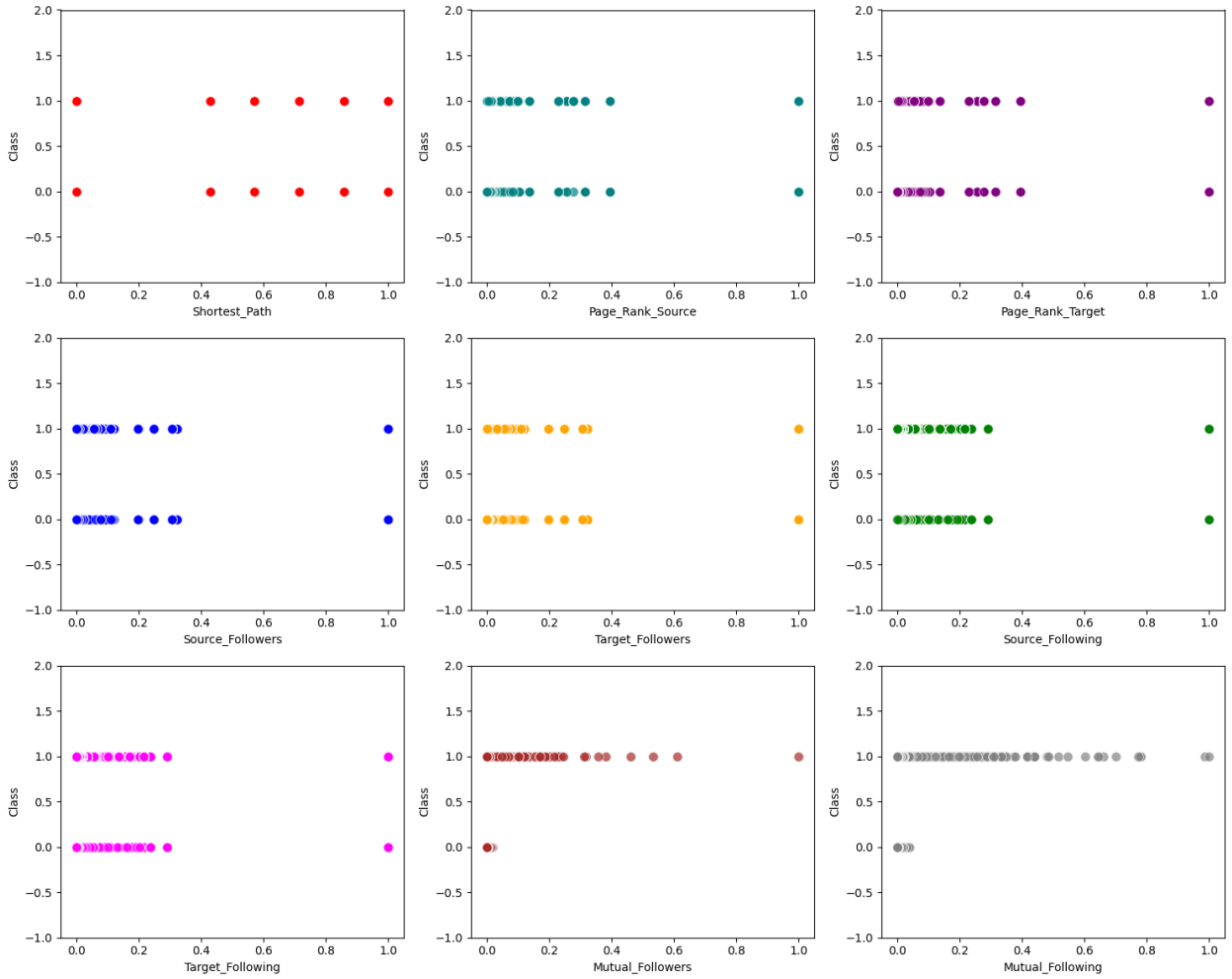
For the pairs (**Source_Followers, Page_Rank_Source)** and **(Target_Followers, Page_Rank_Target)**, the correlation is almost perfect for the former and an actual perfect positive value of 1 for the latter. This is also interpretable because the way the PageRank algorithm works is that nodes with more incoming connections are given a higher PageRank value, thus higher the follower count of a source or target node, more the PageRank value that the node receives by the algorithm. This kind of correlation between PageRank values and Follower counts indicates that popular developers are central to the GitHub network and they play a significant role in the connectivity of the network.

For the **Target_Following** & **Target_Followers**, the correlation is a high 0.77, which could be because on average developers have the same following and follower count, that is they tend to follow each other and that could be the reason for this correlation but of course this is not true for all developers and that is why the correlation ends at 0.77.

For the **Target_Following** & **Page_Rank_Target**, the correlation is a positive 0.74. This correlation is actually surprising as the Page Rank algorithm does not depend on outgoing links at all and yet we see a positive correlation. However, we already know that the Target_Following and Target_Followers correlate positively and we know Page_Rank_Target and Target_Followers correlate positively as well. Thus, this indirect connection could explain this unusual behavior.

For the **Mutual_Followers** & **Mutual_Followers**, the correlation while low is still mildly noticeable with a positive value of 0.37. This indicates that there are communities and sub-groups within the GitHub network who all follow each other due to their similar fields or interests.

4.3.4. Feature Influence with Scatter Plots



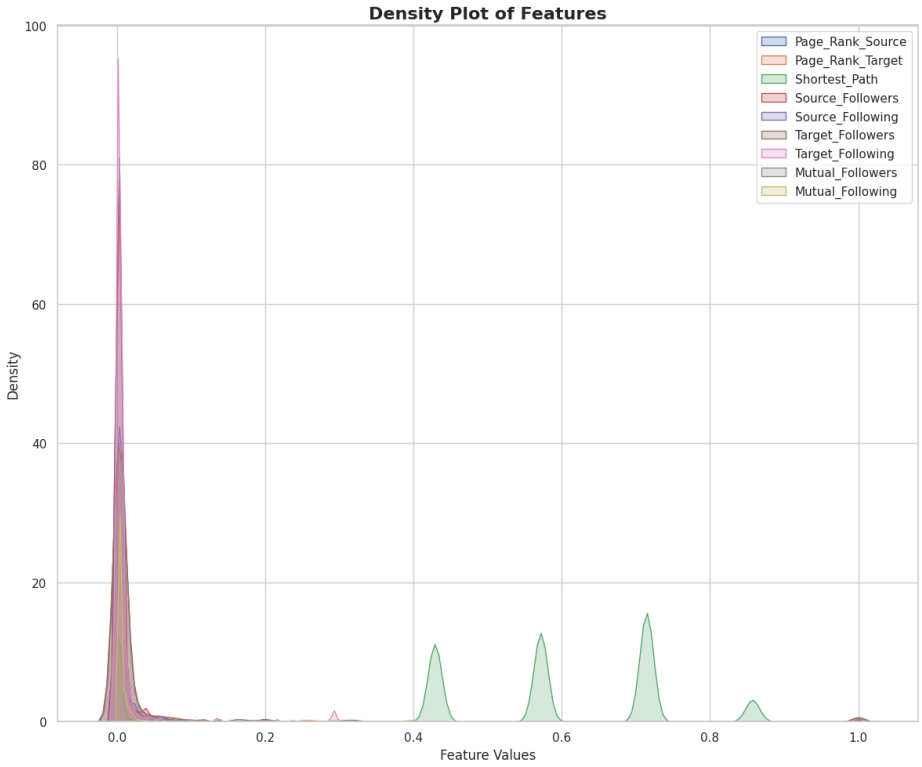
Interpretation: The scatter plots show the influence that each feature has on the Class prediction resulting in 0 or 1. For each value of feature, we see the number of samples that are clustered at the classes 0 and 1. In each scatter plot, the class which has more samples clustered at any given plot point is influenced more by that particular value of that feature. Let's look at the plots overall and some noticeable differences.

For the features **Shortest_Path**, **Page_Rank_Source**, **Page_Rank_Target**, **Source_Followers**, **Target_Followers**, **Source_Following**, & **Target_Following**, the scatter plots indicate that there is no striking difference between the class results that are being influenced by any values for these features because the number of samples clustered around classes 0 and 1 for all values that exist for these features seem to be more or less the same. This makes sense because for the Link Prediction task, the number of missing links that were generated were the exact same number as the existing links and are also part of the same network. So, these missing links have the same kind of properties as the existing links for the most part.

When **Mutual_Followers** & **Mutual_Following** columns are considered however, things change dramatically and fast. We see that for almost all values of Mutual_Following and

Mutual_Followers, a large number of samples are clustered around class 1, while only a small number of samples are clustered around class 0, only around the range 0 to 0.075. This indicates that Mutual followship features correlate strongly with Class 1, meaning that shared links between the nodes can be strong proof for the existence of a direct link between the nodes. This makes sense because in real life, there are communities and sub-communities that exist within large social networks and the existing links of the GitHub Network reflect this behavior as well. On the other hand, the missing links were generated by choosing two nodes at random, so there are no communities that exist among the missing links.

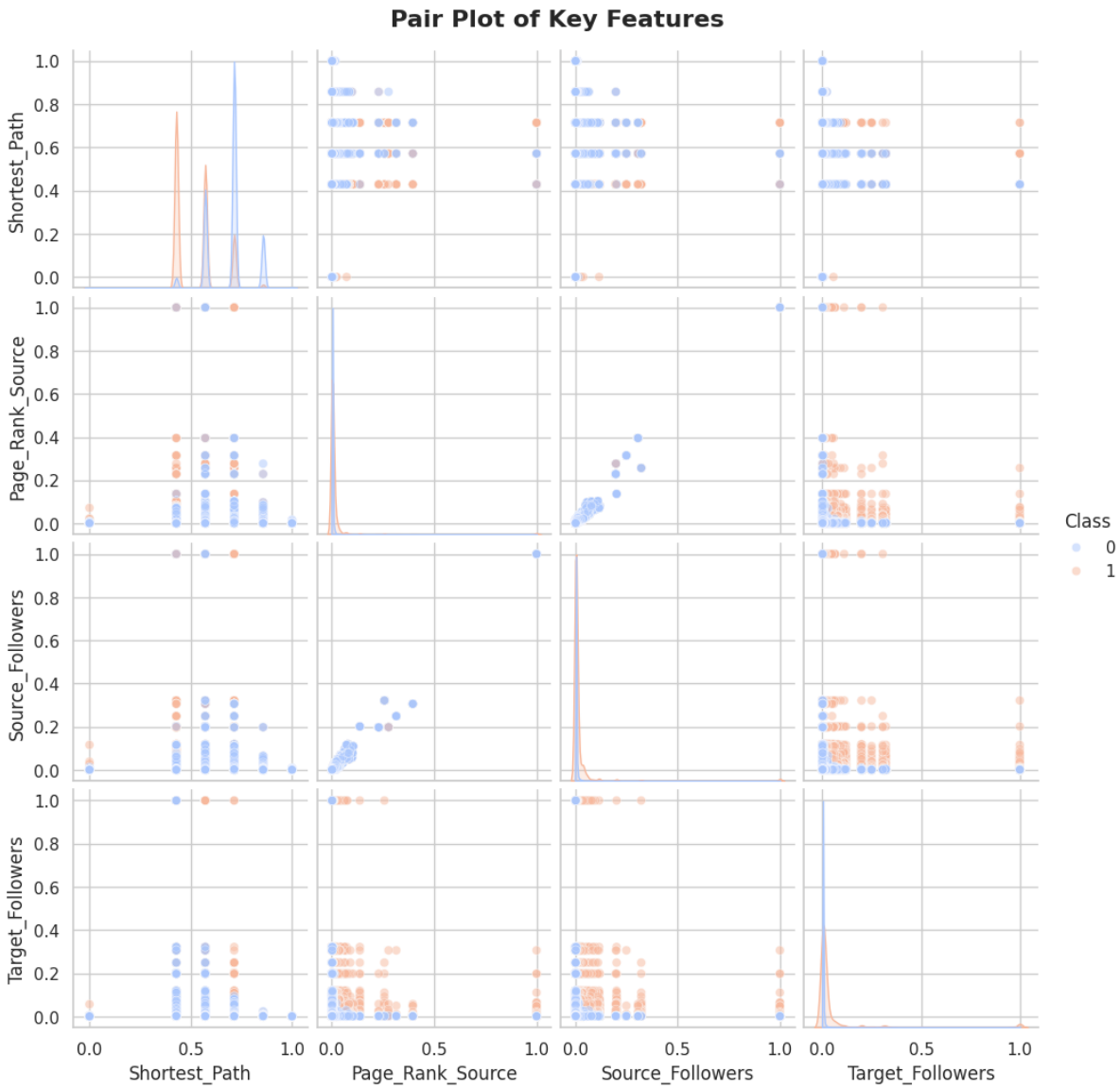
4.3.5. Feature Distribution with Kernel Density Estimate Plot



Interpretation: The KDE plot shows us how the features in the GitHub Network dataset are distributed across the dataset. Let's look at some key observations that can be gleaned from the plot. Almost all the features except Shortest Path have values that are peaked and skewed towards the left hand side of the plot with only a small tinge that is skewed to the right situated at the very end. This suggests that most nodes in the network are average with average to low feature values while a small amount of nodes have extremely high values. This kind of pattern shows that the developers for the most part are connected sparsely, meaning they have limited interaction or influence in the social network. The small contingent of developers with high feature values, i.e., the outliers are influential and well-connected. This pattern is typical in real social networks. One distinction however are the shortest path values which has 4 distinct peaks, meaning that the values

correspond to specific path lengths that exist between nodes in the network suggesting that certain path lengths are common in the GitHub network indicating the existence of specific structural patterns that could possibly be communities or regions with common characteristics.

4.3.6. Feature Relationships with Pair Plots



Interpretation: The pair plot visualizes how the four features – Shortest_Path, Page_Rank_Source, Source_Followers, Target_Followers are related to each other, with respect to classes 0 and 1 which are colored blue and orange respectively. Let's look at some key observations that were made from this plot.

For the plots located on the diagonal from top-left to bottom-right, they show only the individual features so these can be ignored as they do not provide any insights into feature relationships.

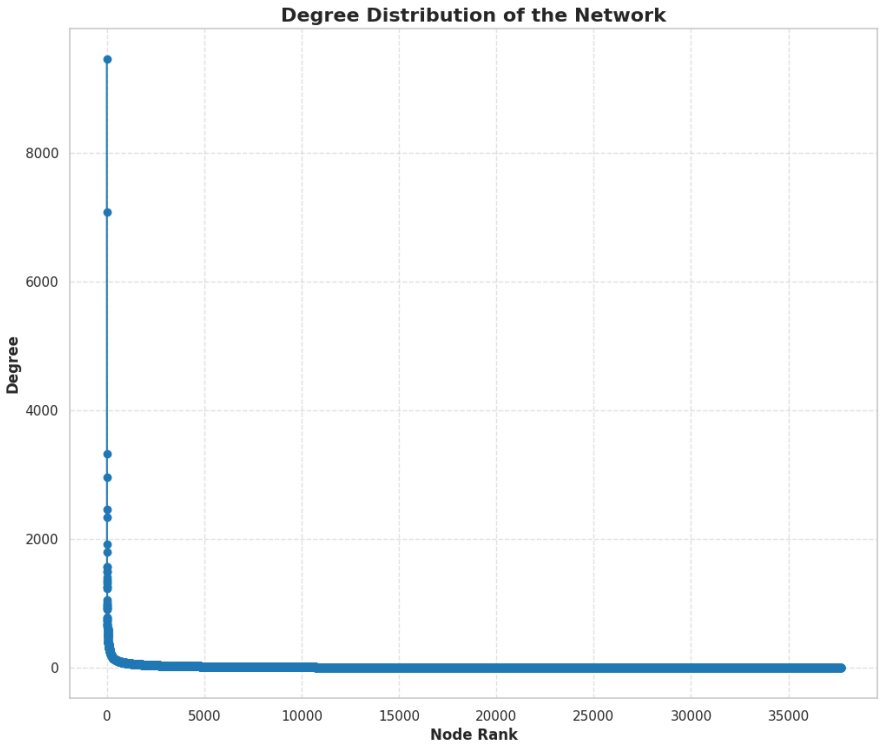
When **Shortest_Path** is compared with other features, one interesting thing we see is that the shortest path values are higher for class 0 when page rank source is lower and lower for class 1 when page rank values of source nodes is higher. This inverse relationship is interesting and suggests that nodes with existing links (Class 1) are closely connected compared to Class 0 nodes.

When **Page_Rank_Source** is compared with other features, we see an interesting pattern with regards to the feature Source_Followers. The source nodes that have high Page Rank values are also seen to have more followers. However, the classes are not very separated here, they are clustered on top of each other, and this indicates that both classes have the similar sort of relationship as far as Page_Rank_Source and Source_Followers are concerned.

When **Source_Followers** is compared with other features, an interesting pattern that can be observed is when it is compared with Target_Followers, clustering of points can be seen at the lower quadrant of the plot and higher counts are seen a lot more for Class 1 nodes. This indicates that Class 1 nodes both source and target have more followers in general.

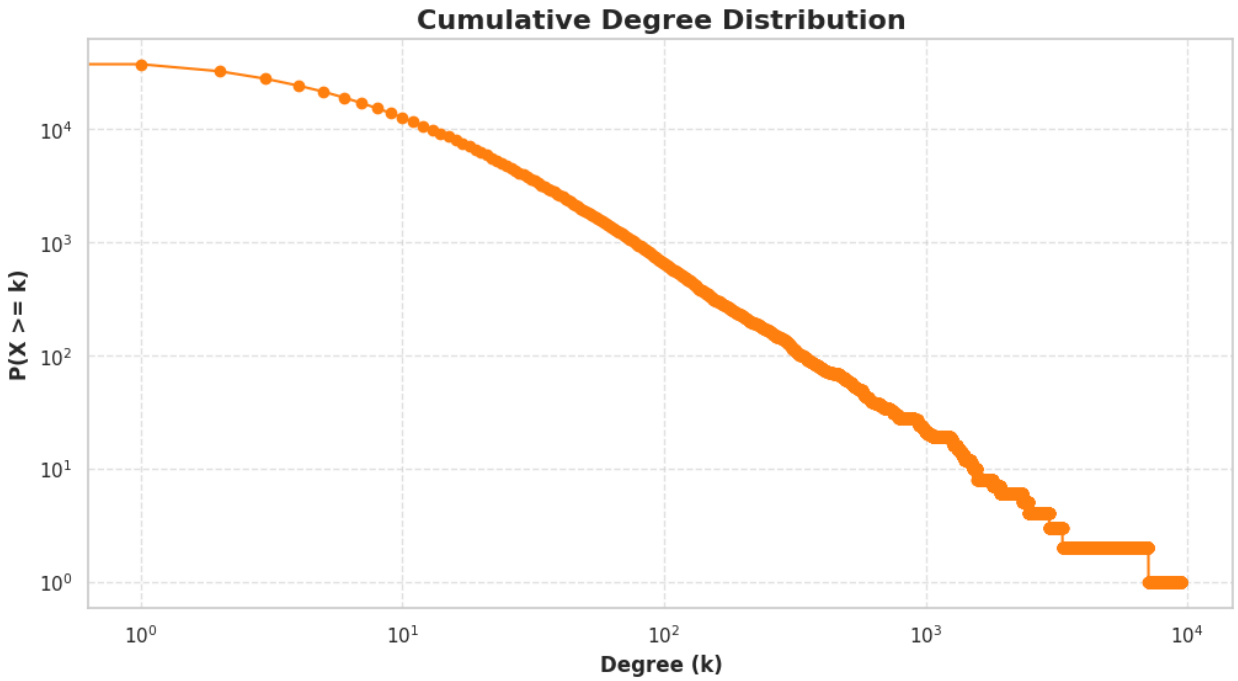
When **Target_Followers** is compared with other features, we see that it generates a curious result, especially with Page_Rank_Source, where most points are present at the start and lower end of the plot for both features, with higher values only corresponding to Class 1, suggesting that popular developers more likely to belong to Class 1. This makes sense because missing links (class 0) do not have any influential developers as these links are just randomly generated.

4.3.7. Degree Distribution



Interpretation: The degree distribution plots provides the rankings of the nodes in the network with respect to their degrees, i.e., the number of connections that they have. We see that the node with the highest number of connections is on the left side, meaning it was ranked the highest and when we move slowly to the right, the number of connections go down and the ranking value increases in number (actual ranking lowers). A steep downward drop can also be seen at the left hand side, this means that only a few small number of nodes have very high degrees. As talked about previously, these are the hubs of the network. Then the curve flattens out almost immediately and follows at the same height for the rest of the plot. This kind of pattern is quite common in real-world networks due to the prevalence of concepts such as Power-Law Distribution.

4.3.8. Cumulative Degree Distribution



Interpretation: This plot is a more detailed approach to inspect the degrees of the nodes in our GitHub network and the way they were distributed with respect to their degrees. Each point in the plot corresponds to the fraction of nodes that have k or more degree and the curve in the plot follows a cumulative nature. As the degree increases, the number of nodes with that degree drops as evident from the downward curve in the plot. We also see some drops at the end of the curve that appear as though they are steps, and these sharp drops show that as the degrees reach the highest that the GitHub network has, only a small number of nodes have them. This plot yields similar insights compared to the previous one but this is another way to visualize and interpret then degrees of the nodes as this uses a log scale for better interpretability and shows the nodes with lower degrees at the start compared to the ones with higher degrees in the previous plot. This plot proves the existence of hubs or centralized nodes in the GitHub network.

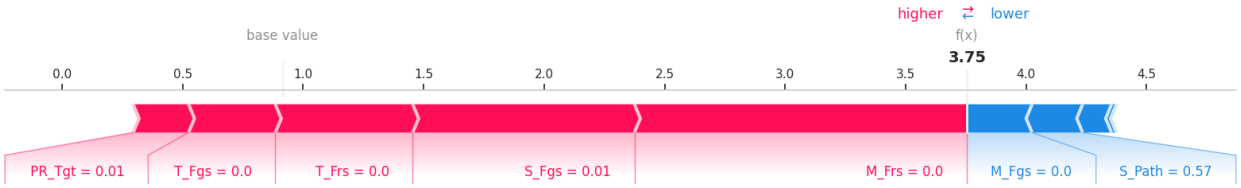
4.4. SHAP Analysis

The last form of analysis on the GitHub dataset was done using SHapley Additive exPlanations or SHAP. These values are computed using an ML model to perform predictions and assigning a fraction of a whole to each of the features that contributed to the class prediction with the actual number determined by how much each feature contributed to the prediction. These values once computed where then used to visualize feature influences, importances and dependencies. The results that include these visualizations and the interpretations for them are given in this section.

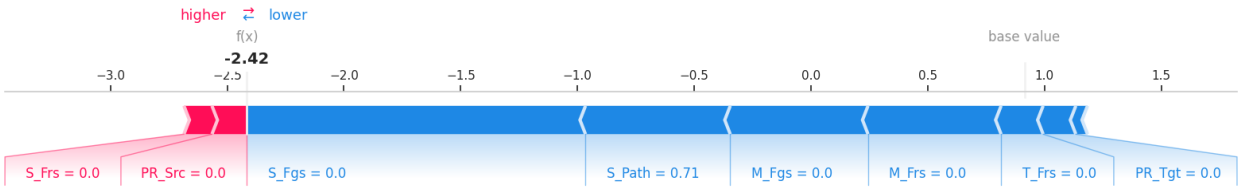
4.4.1. Feature Influence using SHAP Force Plots

Page_Rank_Source = PR_Src
Page_Rank_Target = PR_Tgt
Shortest_Path = S_Path
Source_Followers = S_Frs
Source_Following = S_Fgs
Target_Followers = T_Frs
Target_Following = T_Fgs
Mutual_Followers = M_Frs
Mutual_Following = M_Fgs

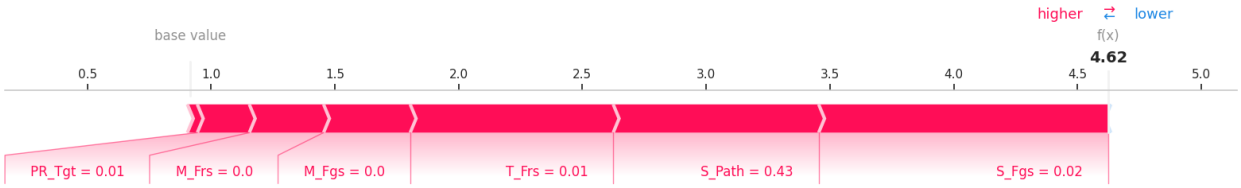
The columns containing the features were found to clutter the plot due to their long lengths so short forms for the feature names were substituted as given above. Force plots show us how much each feature contributes to the final prediction of a class for an instance from the dataset. The base value is the average prediction value across the entire dataset and acts a baseline for comparison. f(x) is the model's predicted value for the data sample represented by the force plot. For each feature, we see the positive (red) or negative (blue) contribution that it makes towards the f(x) value. Let's look at the force plots of five random data samples from our test set to see what we can learn about the features in terms of their contributions to Class 0 or 1 prediction.



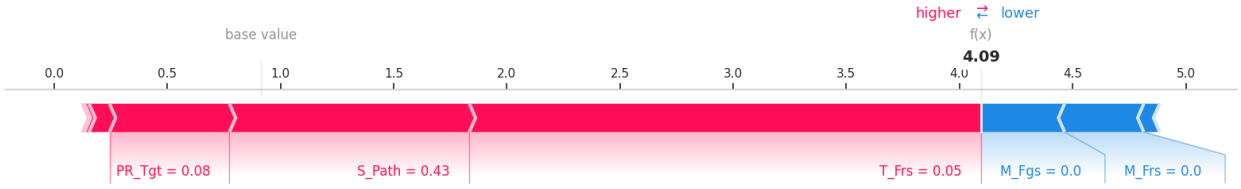
Interpretation: For the first instance, the f(x) value is high, exactly at 3.75. We see that the features Page_Rank_Target, Target_Following, and Target_Followers make slight contributions while Source_Following and Mutual_Followers make the most contributions to the positive prediction. On the other hand, Mutual_Following and Shortest_Path contribute negatively to the prediction, pushing it closer to the base value.



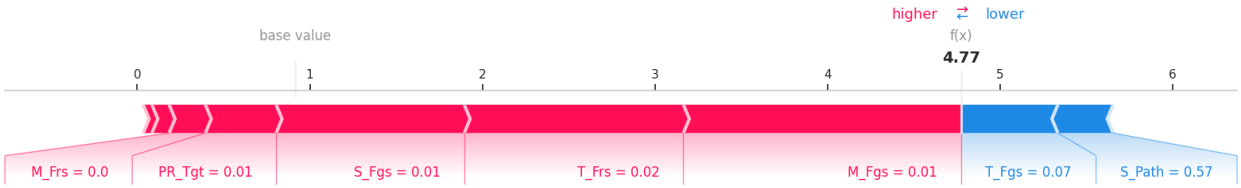
Interpretation: Here, the f(x) value is a negative one, -2.42 to be precise. We see that the features Page_Rank_Target, and Target_Followers make slight contributions, Mutual_followers, Mutual_Following, and Shortest Path make moderate contributions and Source_Following contributes the most to the negative prediction. On the other hand, Source_Followers and Page_Rank_Source although slight, still contribute to a positive prediction, attempting to push it closer to the base value.



Interpretation: This force plot is unique in that the model's prediction is extremely high, sitting at a good 4.62. None of the features seem to contribute negatively against the f(x). The features Page_Rank_Target, Mutual_Followers, Mutual_Following, Target_Followers contribute the lowest, the contributions of Target_Followers and Shortest_Path is moderate and the Source_Following contributes the most towards the model's positive prediction.



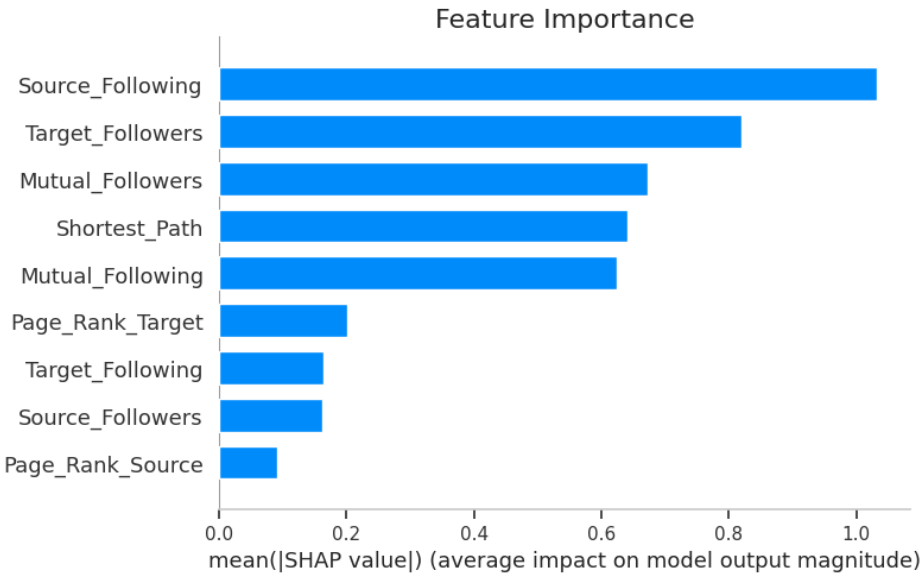
Interpretation: The fourth plot also has a f(x) value that is higher but there is a slight negative contribution by certain features. These features include Mutual_Followers and Mutual_Following both of which contribute a decent amount towards pushing f(x) closer to the base value. However, Page_Rank_Target, Shortest_Path, and Target_Followers each contribute to the positive prediction with the level of contribution corresponding with the order they are listed.



Interpretation: In the last force plot, the prediction is again positive with a f(x) value of 4.77. Mutual_Followers, Page_Rank_Target, Source_Followers, Target_Followers, and Mutual_Followers contribute to the positive prediction while Shortest_Path and Target_Followers contribute to the negative prediction.

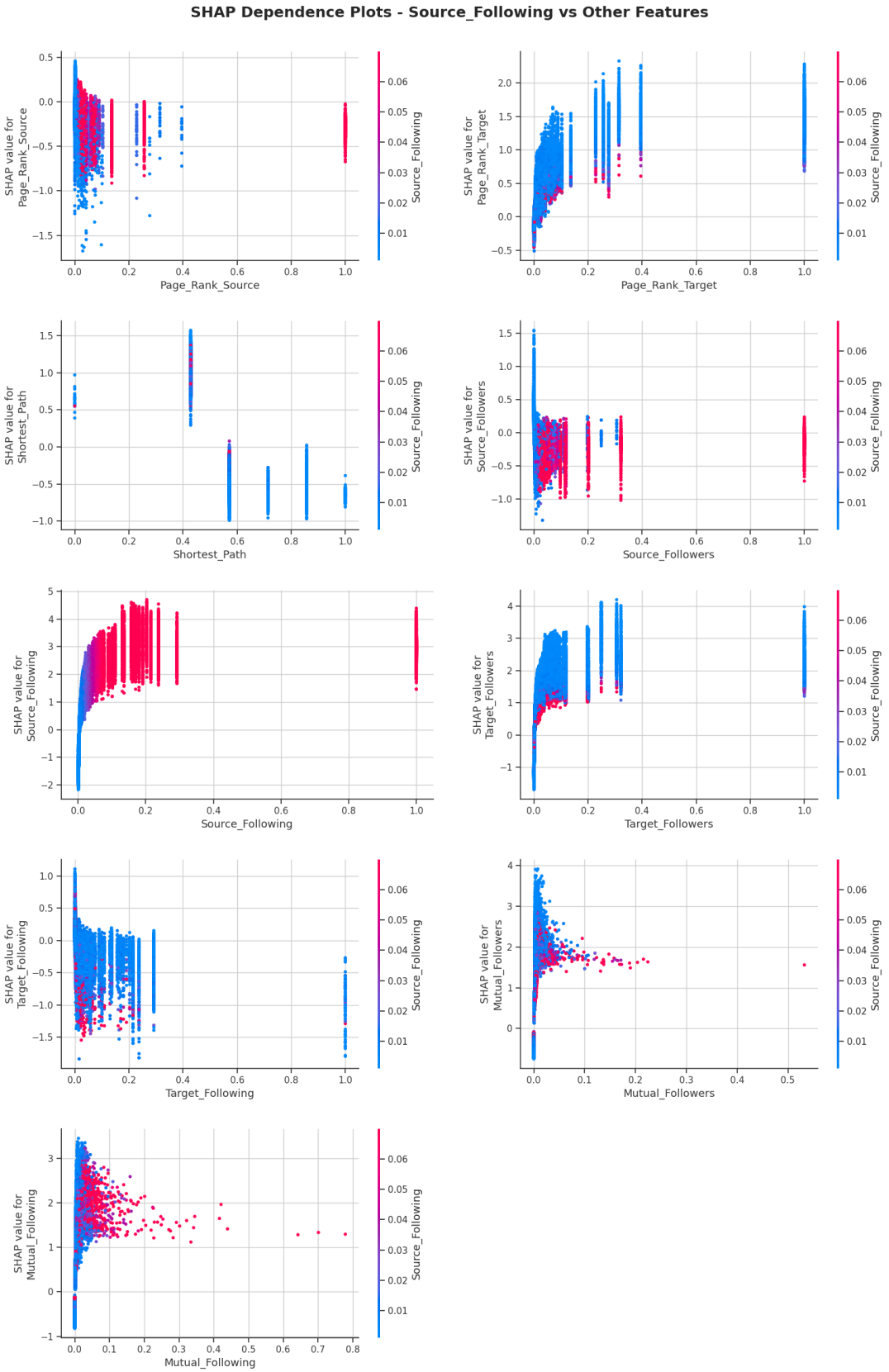
Overall Interpretation: When all the force plots are considered holistically, we can see which features tend to influence model predictions the most and towards which class i.e., positive or negative. First, the Shortest_Path is often seen to have a blue contribution for class 0 predictions and red contributions for class 1 predictions, meaning that a shorter path essentially equates to existence of a link as it indicates that the nodes are closer. Then, Source_Following is frequently seen to be red in class 1 predictions, and this could mean that the source nodes' following count when they increase also increase the likelihood of a link existing. We also have Mutual followships which play a consistent role in nudging predictions, even though they're relatively smaller, meaning that mutual connection between the nodes can determine existing links to a degree. Lastly, Page Rank values of Source and Target nodes appear less impactful that originally imagined but still do contribute to the final prediction. These features show us what values for each of them lead to the existence of a relationship and no relationship at all, between developers in our GitHub Network.

4.4.2. Feature Importance using SHAP Summary Plot



Interpretation: Summary Plot ranks the features in the GitHub Network in terms of their importance to Link Prediction using their average SHAP values with the top features having the highest and the value dropping as we go progressively down. We see that the Source_Following is the most important feature to class prediction and what this means is that on average, the number of users a source node is following has the highest impact on the prediction of a relationship (link). Mutual connections also seem to play an important role in determining the existence of a link which means that shared relationships between nodes in the GitHub network is a good indicator that they are in fact connected. Page_Rank_Source on the other hand seems to have the lowest impact on link prediction which suggests that even though overall importance of a node matters, it is not as important as direct connections.

4.4.3. Feature Dependence using SHAP Dependence Plots



Interpretation: The last set of plots visualize how the feature that was found to be the most important, Source_Following affects other features while predicting the existence of links between GitHub developers. Let's look at some important insights that have been gained through this interesting comparison.

Broadly speaking, Source_Following can be seen to be the most dominant feature across all the plots, with the highest values, which also seem to correlate with high SHAP values. This means that, when a source developer follows a lot of other devs, he/she is more likely to form a link/relationship with the target developer.

Interaction with Page_Rank values: When PageRank values for the source and target nodes are lower, Source_Following is still at a neutral level, which means that PageRank and Source node's follow count do not interact with each other much. However, when PageRank values are around the average range, the SHAP values are seen to increase, so Source_Following does have a better influence on the prediction when the developer has moderate influence in the network.

Interaction with Shortest_Path: When shortest path distance is low, Source_Following's SHAP values are higher, which essentially means that when the developers are located closer in the network, the source developer's follow count is given more importance for link prediction. For longer path distances however, the contribution disappears as its obvious that nodes located farther away from each other are much less likely to connect with each other even if they happen to follow a lot of people.

Interaction with Source & Target Followers: For a moderate number of followers, the SHAP values are relatively high. This goes to show that when follower count is balanced, it actually contributes to link prediction positively. But also, when the count is lower, the SHAP values are also low, and ultimately less important.

Interaction with Mutual Followship features: These features have a positive correlation with Source_Following in that when Mutual Followers or Following increase, Source_Following also does, which means that the existence of shared relationships between two nodes can also mean the existence of a link between them as they are more likely to belong to the same community.

In summary, these dependence plots solidify the idea that direct activity metrics such as Source_Following, Target_Followers, shared relationships such as Mutual Followers, Mutual Following, and proximity metrics such as Shortest Path are in fact the most important and influential predictors of relationships between developers in a GitHub Network.

5. Summary

To summarize the outcome of this project, it successfully predicted future links or relationships between developers in a GitHub Network and its structure was analyzed using Machine Learning models, Network Analysis metrics and visualizations, and lastly SHapley Additive exPlanations. Below are the key findings summarized:

Link Prediction Performance – Out of all the trained and tested ML models, Random Forest Classifier seems to have achieved the highest accuracy of 88.1% using parameters such as 50 trees and a depth of 24. Logistic Regression follows suit with an accuracy of 86.7% which is a simple classifier touted for its efficiency, which was actually evident from the quick training. SVM takes the last spot with a 85.7% accuracy.

Feature Insights – Out of all the features that were used, Source_Following i.e., the number of nodes followed by the source node and Target_Followers i.e., the number of nodes that follow the target were found to be the best and most influential features that can be used for the task of Link Prediction.

Network Metrics – We were able to visualize a subset of the GitHub Network with its most critical pathways and the most influential & centralized nodes highlighted thanks to Edge and Node Betweenness showing us the potential structure of the full GitHub network. The GitHub network is also a sparse one with an extremely low density of 0.001 and transitivity of 0.013 which indicates the network's extremely low clustering tendency.

Visualizations & Interpretations – Correlation visualizations such as the Heatmap and Pair plots showed a strong dependence between features such as Source and Target Followers while some features such as shortest paths were entirely independent. Feature distribution plots such as the Scatter plots, Kernel Density Estimate plot showed the feature data is highly skewed towards lower values with only a small percentage of high values. Degree distribution plots showed that the network has a scale-free nature, closer to that of real-life social networks where only a small group of people have the most amount of connections.

SHAP Analysis – SHAP plots such as Force plots and Summary plots illustrated the influence that individual features can have on model predictions while Dependence plots went on to reveal nuanced relationships between the features, showing us how the most important feature interacts with other features in the network.

Overall, this comprehensive project has been a great endeavor that provided influential insights into a GitHub Network's ecosystem and the features that can influence relationships between developers in the network, demonstrating that Graph-based Data Analysis and Machine Learning when used in unison have the capability to reveal hidden dynamics in real-world Social Networks.