

JEAN MONNET UNIVERSITY SAINT-ÉTIENNE

Report Data Mining for Big Data: Project

Authors

Elias Romdan
Md Nur Amin
Nicolas Trotta

MASTER 2 DONNÉES ET SYSTÈMES CONNECTÉS

MASTER 2 MACHINE LEARNING AND DATA MINING

February 1, 2021



Contents

1	Introduction	3
2	Unsupervised Visual Analytics using t-SNE	3
3	Data Analysis	4
3.1	Analyses on the coded weighted edges file	4
3.1.1	Dataset description	4
3.1.2	Communities that used the most of the time an application to tweet	4
3.1.3	Top 10 applications by their community size	6
3.1.4	Users that tweeted the most on each application	7
3.1.5	coded weighted edges as a bipartite graph	9
3.2	Analyses on the the sim matrix file	11
3.2.1	Community analysis	11
4	Data Engineering	17
4.1	Data Description	17
4.2	Feature Encoding and Transformation	17
4.3	Feature Extraction	17
5	Supervised Learning Methodology	17
5.1	Data separation	18
5.2	Classification Algorithms	18
5.3	Bayesian Hyperparameter Optimization	18
5.4	Evaluation Metrics	18
6	Result Analysis for Classification	18
7	Conclusion	19
8	Reproducibility of results	20

List of Figures

1	t-SNE visualization of data in 2D and 3D space	3
2	The most populated communities in the graph	5
3	Users by their tweeting frequency on applications	7
4	Structure composed of a tuple of 3 pieces of information	9
5	Nodes Separation	9
6	Classic Graph Display	10
7	Bipartite graph with on the left the 767 users and on the right the 215 applications	10
8	Global view of the relations without intervention on the data	12
9	Cleaning the graph to remove parasitic edges and nodes	14
10	Display of the cleaned graph	14
11	Displaying the community graph	15
12	Minimal display without the nodes to see the network distribution	16
13	Exploded view of the network from NetworkX Viewer	16
14	PCA and feature importance for feature extraction	17

1 Introduction

Even the best things have their flip side, in the case of messages on the internet, it's the spammers. The role of our 3-member team will be on this project to first detect, with the help of a dataset made available to us, possible spammers with an acceptable prediction rate. In the initial phase, we try to understand the data more before diving into the modeling part. In section 2, we visualize the legitimate and malicious users in a low dimensional space to have an initial intuition about the dataset. In section 3, we use the provided dataset in the graph part to make analysis on group of communities, similarities between users, the most used applications, potential spammers and other computations on social networks. In section 4, we engineer the data where it includes the description of the dataset, feature encoding and transformation. In section 5, we discuss about the methodology for the classification task that includes a brief description of the classification models, it's optimization and the metrics for evaluation part. Finally in section 6, we analyze and compare the performance of different classification models and choose the best performing one.

2 Unsupervised Visual Analytics using t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a non-linear and unsupervised algorithm that can be used to visualize high dimensional data in a low dimensional space. It provides with an idea of arrangement of data in a high dimensional space as well as it's complexity[1]. Probability of similarity between a pair of points in high-dimensional space is calculated and then probability of similarity between a pair of points in high-dimensional space is mapped in the corresponding low dimensional space. Afterwards, it aims at minimizing the sum of Kullback-Leibler divergence of the data points by using gradient descent algorithm for a better representation using the gradient descent in a lower-dimensional space.

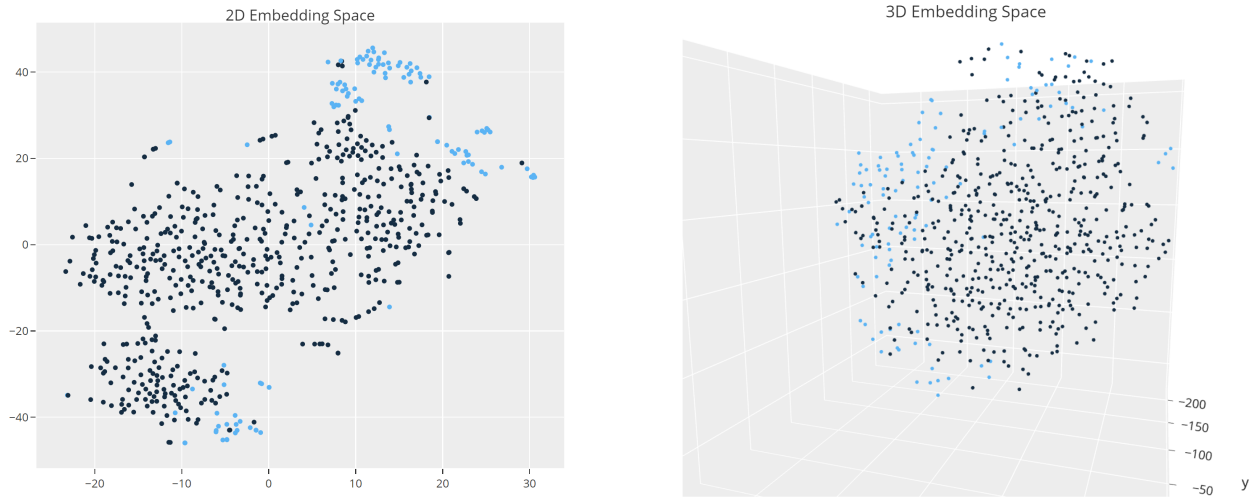


Figure 1: t-SNE visualization of data in 2D and 3D space

3 Data Analysis

3.1 Analyses on the coded weighted edges file

3.1.1 Dataset description

In this section, we are going to show the results of our analysis on the CSV file `coded_weighted_edges`. The idea is to represent this CSV file as a graph, and then to extract different useful informations from the graph. For these analyses, we have not used the CSV file `weighted_app_user_edges` since in some cases we are going to display nodes with labels and those labels must be kept the shortest possible to make the graph clear. Since we focus this study on applications, the CSV file `apps_ids` will be used to make the link between the real name and the coded identifier of an application. The Jupyter notebook `coded_edges_graph.ipynb` contain the Python 3 code to reproduce the results of this study. Let's display the structure of the CSV file that we are going to study:

```
df_coded_edges = pd.read_csv(path + 'app_based_similarity/app_user_interaction_graph/coded_weighted_edges.csv')
print(df_coded_edges)
```

	user_id	app_id	weight
0	1	23	9
1	1	33	391
2	362	200	192
3	362	23	8
4	488	176	61
...
2017	679	23	192
2018	679	118	8
2019	721	2	200
2020	632	200	198
2021	632	201	2

[2022 rows x 3 columns]

For instance, row 0 means the user with coded identifier 1 used the application with coded identifier 23 to do 9 tweets. Starting from this data frame we are going to build different graphs and compute several things like, communities that used the most of the time an application to tweet, top 10 applications by their community size, users that tweeted the most on each application and how many tweet these users have made on each application.

3.1.2 Communities that used the most of the time an application to tweet

```
graph_user_app = nx.Graph()
count_users = 0
# Get the first row of the dataframe as the max weight between a user and an application
data_max = df_coded_edges.iloc[0]
for i in range(1, len(df_coded_edges)):
    # Get the row that will be treated
    data_actual = df_coded_edges.iloc[i]
    # Test if the user is the same as the max row user and if the weight is higher than the max row weight
    if data_actual[0] == data_max[0] and data_actual[2] > data_max[2]:
        # Update the max row by the actual row
        data_max = data_actual
    # Test if the user is different from the max row user
    elif data_actual[0] != data_max[0]:
        # Store the max data content into a graph by drawing an edge between the user and the application
        graph_user_app.add_edge("U" + str(data_max[0]), "A" + str(data_max[1]))
        # Increase the number of user nodes by 1
        count_users += 1
        # Update the max row to treat the next user
        data_max = data_actual
# Store the max data content of the last user into a graph
graph_user_app.add_edge("U" + str(data_max[0]), "A" + str(data_max[1]))
# Include the last user into the counter
count_users += 1
print("Number of user nodes:", count_users)
print("Number of application nodes:", len(graph_user_app) - count_users)
```

Number of user nodes: 767
Number of application nodes: 99

One user can only belong to one community, based on this theory we are going to filter our CSV file

to only keep the application with the maximum tweet for each user. In a graph language, when one user has several edges related to different applications, we only keep the most weighted edge.

Since the data frame is already sorted by user identifier, when we enter the elif bloc, we are sure that data_max contain the row with the highest weight of the previous user, so we proceed directly by adding a new edge between the user coded identifier and the application coded identifier to the graph. A user may have the same coded identifier as an application, so to make the difference between them, we put "U" before the label of a user node and "A" before the label of an application node. When adding a new node to the graph, we do not need to verify its existence since add_edge function already do it, and if it is the case, it pinpoints automatically on the existing one before drawing the edge. Even after the for loop ends, we need to repeat the process of adding an edge one more time to save the row of the last user. We keep track of the number of users in the graph since we are going to use it for the next computations. We use the library NetworkX Viewer to have a clear visualisation on the graph and on the group of communities like showed by figure 2:

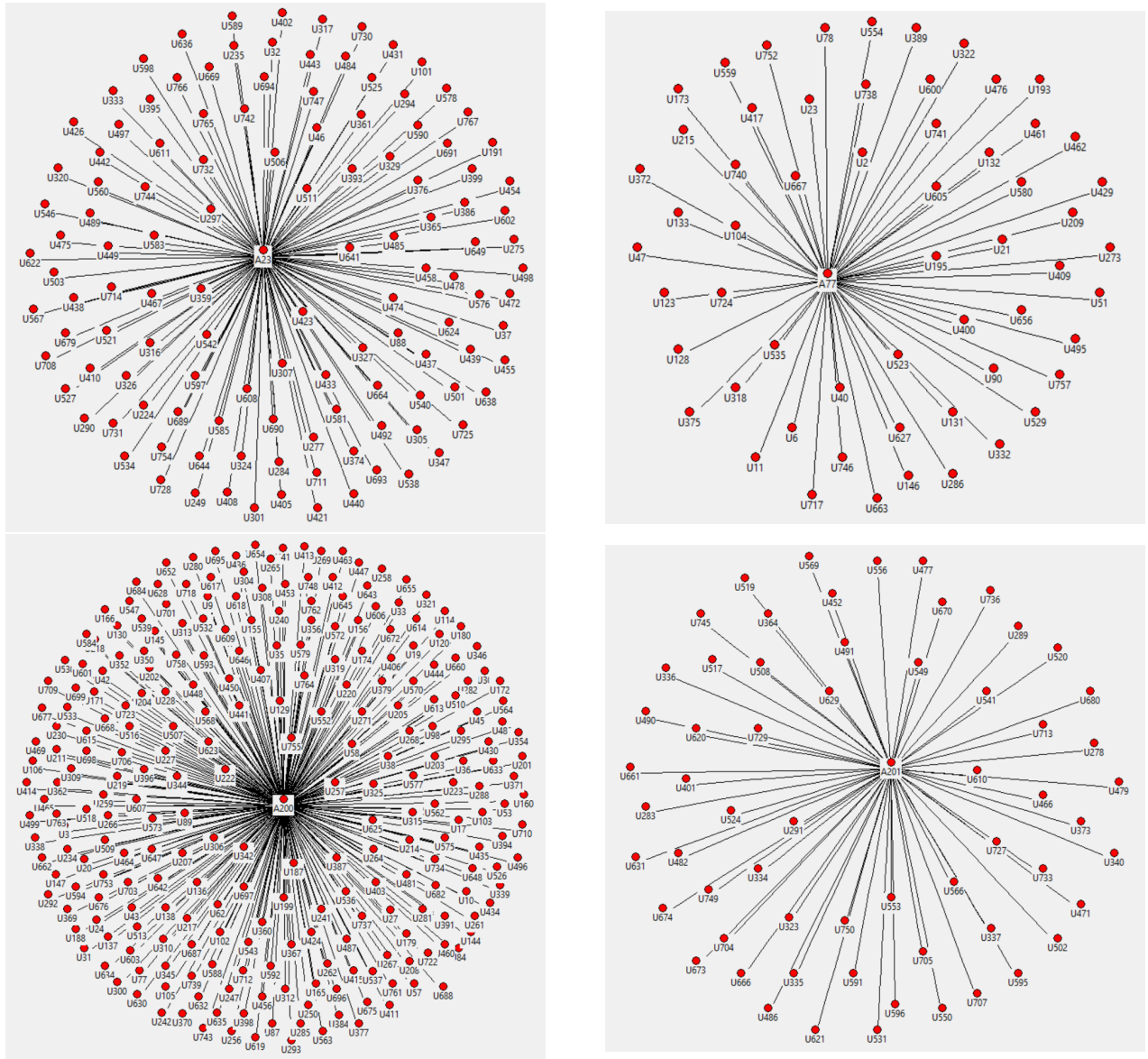


Figure 2: The most populated communities in the graph

The resulted graph is composed on several subgraphs that are not linked to each other and that

constitute the concept of communities. We suppose from figure 2 that the application with the coded identifier 200, has the biggest community, let's try to check this in a programmatic way.

3.1.3 Top 10 applications by their community size

Our objective now is to find the 10 biggest communities in the graph, that means the top 10 applications where each user has the most used to tweet, and in a graph language, the 10 nodes with the highest degree. We import the CSV file apps_ids to use it in the detection of the real name behind a coded application identifier.

```
df_app_ids = pd.read_csv(path + 'app_based_similarity/apps_ids.csv')
print(df_app_ids)
```

```
   app_id  app_desc
0      1  <a href="http://tweepsmap.com" rel="nofollow">...
1      2  <a href="https://www.lithium.com" rel="nofollow...
2      3  <a href="http://coschedule.com" rel="nofollow"...
3      4  <a href="https://stardust.co" rel="nofollow">S...
4      5  <a href="http://tapbots.com/software/tweetbot/...
..      ...
210    211 <a href="http://vd1.co/lop" rel="nofollow">\u0...
211    212 <a href="https://mobile.twitter.com" rel="nofo...
212    213 <a href="https://prod2.sprinklr.com" rel="nofo...
213    214 <a href="http://www.crowdfireapp.com" rel="nof...
214    215 <a href="http://giphy.com" rel="nofollow">Giph...
```

[215 rows x 2 columns]

Each coded application identifier corresponds to a link tag that target to the application website with its name as a value. After reading the CSV file content as a data frame, we proceed on doing some operations on the resulted graph.

```
# Fill and sort a list based on the degree number of each node in a descending order
most_used_app = sorted([(degree, py.int64(node[1:])) for (node, degree) in graph_user_app.degree()], reverse=True)
# Pick only the first 10 rows of the list and put them on a dataframe
df_top_app = pd.DataFrame(most_used_app[:10])
# Add a head to the new dataframe
df_top_app.columns = ['nb_users', 'app_id']
# Merge the new dataframe with the apps ids dataframe based on the app_id column and keep only the intersection
df_top_app_name = pd.merge(df_app_ids, df_top_app, how='inner', on='app_id')
# Compute the utilisation rate of each application and add it to the merged dataframe
df_top_app_name['rate'] = [round(users / count_users * 100, 2) for users in df_top_app_name['nb_users']]
# Loop on the merged dataframe
for i in range(0, len(df_top_app_name)):
    # Replace the content of app_desc column by the value of its tag link <a>
    df_top_app_name.at[i, 'app_desc'] = re.sub('<.+?>', '', df_top_app_name.iloc[i][1])
# Sort and display the merged dataframe based on the rate
df_top_app_name.sort_values('rate', ascending=False)
```

	app_id	app_desc	nb_users	rate
8	200	Twitter for iPhone	261	34.03
0	23	Twitter Web Client	122	15.91
9	201	TweetDeck	61	7.95
4	77	Twitter for Android	58	7.56
2	33	Done For You Traffic	29	3.78
7	176	Hootsuite	26	3.39
1	32	dlvr.it	19	2.48
3	40	IFTTT	18	2.35
6	154	Buffer	10	1.30
5	102	Google	9	1.17

The idea is to loop on the degree of each node, then to add to a list the node label with its degree. We use node[1:] to delete the first character of the node label, since in the previous section we added "U" and "A" to make the difference between nodes, in this part we do not need anymore this distinction. Also, we convert the node type from an object to int64 to make the merge possible in the next code lines. The list is created by the pair (degree, node), the idea behind putting the degree in the left is to make the reverse order list sorting based on the degree. Then, we create a new data frame with the top 10 pairs in the list, which will be merged with the data frame of the CSV file apps_ids based on the app_id column

and using the inner join (we only keep common values in the app_id column). We compute the usage rate of each application (app_id) based on its degree node (nb_users) divided by the number of user nodes in the graph (count_users). The last step to have a clean data frame, is to loop on it and to replace the app_desc column by the value of the link tag which is the real name of the application using a regex. Finally, we sort the data frame based on the rate and display it. The resulted data frame shows the 10 applications that have the biggest communities. For instance, we can confirm that the coded identifier 200 has the biggest community, which correspond to the application Twitter for iPhone with 261 users, which represent 34,03% of the total number of users in this dataset.

3.1.4 Users that tweeted the most on each application

In this section, we are going to focus on the study of the user side. We want to find which users have tweeted the most on each application to detect potential spammers. For this propose, we are going to build a new graph different from the previous one, where we apply a filter on the weight.

```
graph_user_app_weighted = nx.Graph()
# Compute the average weight of the dataframe
avg_weight = df_coded_edges['weight'].sum() / len(df_coded_edges)
for i in range(0, len(df_coded_edges)):
    # Get the row that will be treated
    data_row = df_coded_edges.iloc[i]
    # Test if the weight of the row is higher or equal to the average weight
    if data_row[2] >= avg_weight:
        # Draw an edge between the user and the application and put a weight on it
        graph_user_app_weighted.add_edge("U" + str(data_row[0]), "A" + str(data_row[1]), weight=data_row[2])
print("Average weight:", avg_weight)
```

Average weight: 104.23392680514343

We assume that a potential spammer is someone that tweet more often than the average number of tweets, that means a user with an edge weight higher than the average of weights. After verifying this condition, we draw a weighted edge between the user and the application, we keep information about the weight of each edge to use it in the next computations. The resulted graph is showed by the figure 3:

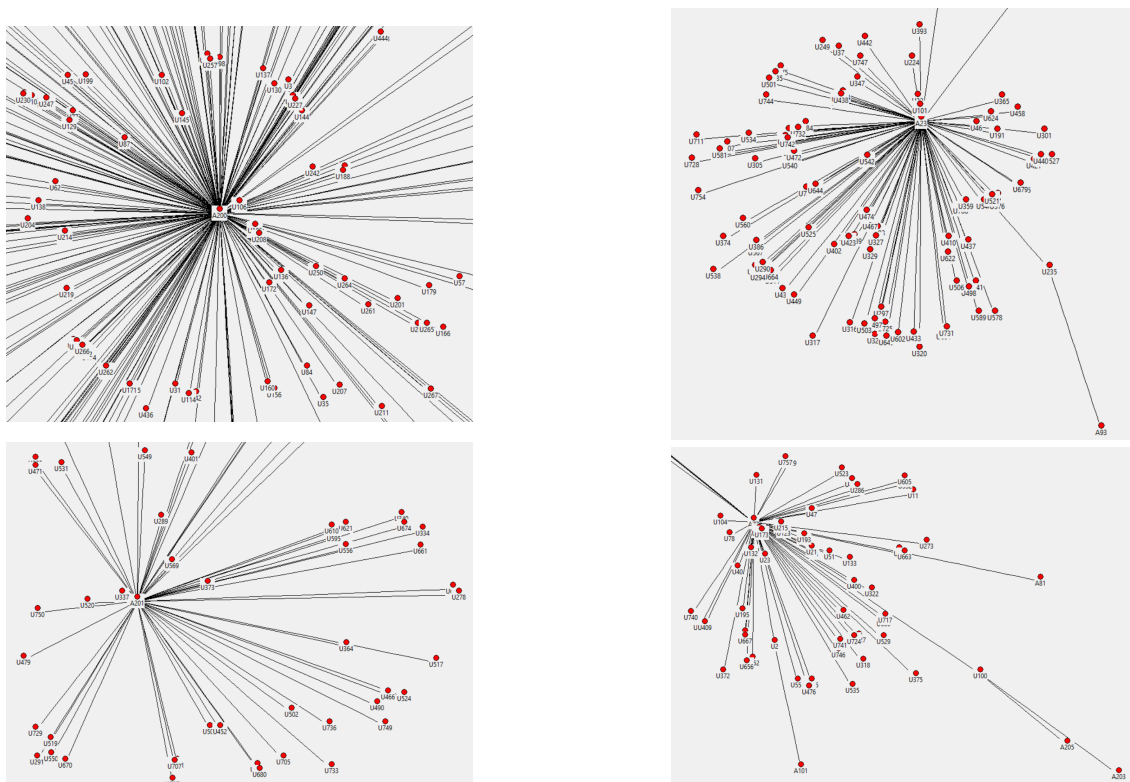


Figure 3: Users by their tweeting frequency on applications

The structure of the subgraphs in figure 3 is close to the one in the previous subgraphs of figure 2. Except that subgraphs can be related between them, since one user node can be linked to more than one application node, also edges now have different lengths. The greater an edge weight will be, the closer the user node will be to the center of a subgraph. Which means potential spammers nodes are generally found near an application node.

```
# Create a dataframe to handle usage statistics on applications
df_app_users = pd.DataFrame(columns=['app_id', 'user_id', 'weight'])
# Loop on the nodes of the graph
for node in graph_user_app_weighted.nodes():
    # Test if the node is an application
    if node[1:] == "A":
        edges = []
        # Loop on the edges of the actual node
        for edge in graph_user_app_weighted.edges(node):
            # Add in the list the user node and the weight of the edge related to the application node
            edges.append((graph_user_app_weighted.get_edge_data(edge[0], edge[1])['weight'], edge[1]))
        # Sort edges by the highest weight and take at maximum the top 5 ones
        edges = sorted(edges, reverse=True)[:5]
        users = ''
        weights = ''
        # Loop on the selected edges
        for edge in edges:
            # Merge the users together
            users = users + ' | ' + edge[1][1:]
            # Merge the weights together
            weights = weights + ' | ' + str(edge[0])
        users = users + ' | '
        weights = weights + ' | '
        # Add to the dataframe a row that contain informations on app id, users ids and their weights
        df_app_users = df_app_users.append({'app_id':py.int64(node[1:]), 'user_id':users, 'weight':weights}, ignore_index=True)
# Remove the limit on the number of displayed rows of a dataframe
pd.set_option("display.max_rows", None, "display.max_columns", None)
# Sort and display the dataframe based on the app id
df_app_users.sort_values('app_id')
```

We create a data frame to store usage statistics for each application, which means the users that tweeted the most on each application. We loop through the nodes of the lastly build graph and we take only the application nodes, that we can distinct them by the letter "A". Next, we build a list of edges with the pair (weight, user node) that are related to the application node. This list is sorted based on the weight and we take, at most, the top 5 edges from the list. The picked edges represent the users that tweeted the most on this application. We separate the users from the weights, we add the application node, and we put all these informations in a new row of the created data frame. Since our data frame has many rows, the limitation must be removed to visualise all the data.

Out[123]:

	app_id	user_id	weight
92	2	721 637	200 196
83	3	751	181
48	6	111	537
36	8	80 119 81	400 400 333
60	9	253 154 162	400 400 365
21	16	29	419
31	17	66 68 63 70 65	400 399 399 395 344
20	18	28	402
79	19	419	166
33	22	69 112 116	492 463 191
5	23	224 46 101 32 37	587 400 400 399 345
81	26	378 671	155 106

If we take row 33 as an example, we can say that user with coded identifier 69 had done 492 tweets, user 112 had done 463 tweets and user 116 had done 192 tweets. They used the application with the coded identifier 22 to do these tweets. From this we can identify users that have the most tweeted on each application, and how many times they had done this.

3.1.5 coded weighted edges as a bipartite graph

In addition to the analysis of the coded_weighted_edges.csv file performed above, we will start from the same file but we will treat it as a bipartite graph. This short chapter will briefly describe the work done in the notebook entitled 'BigData_CodedWeightedEdgesGraph-V2.ipynb'.

As this notebook was chronologically the first one, it uses another data architecture to store the parsed data.

```
1 # Creation of the class for receiving csv data 'coded_weighted_edges.csv'
2 # Renaming of app nodes due to a confusion in the library to compute degrees
3 class CodedWeightedEdges:
4     def __init__(self, user_id, app_id, weight):
5         self.user_id = user_id
6         self.app_id = str(app_id) + '_'
7         self.weight = weight

1 # Creation and filling of the 'CodedWeightedEdges' structure list.
2 CodedWeightedEdgesList = []
3 for line in df.itertuples():
4     listTmp = []
5     for i in range(1, 4):
6         listTmp.append(line[i])
7
8     CodedWeightedEdgesList.append( CodedWeightedEdges(listTmp[0], listTmp[1], listTmp[2]) )
```

Figure 4: Structure composed of a tuple of 3 pieces of information

As this is a two-part graph, we will have to differentiate between the list of users and the list of applications.

```
1 # Creation of the 2 lists of nodes for user and app
2 nodeUserList = []
3 nodeAppList = []
4 for struct in CodedWeightedEdgesList:
5     if struct.user_id not in nodeUserList:
6         nodeUserList.append(struct.user_id)
7     for struct in CodedWeightedEdgesList:
8         if struct.app_id not in nodeAppList:
9             nodeAppList.append(struct.app_id)
10
11 nodeUserList.sort()
12 nodeAppList.sort()
13 print('UserList :')
14 print(nodeUserList)
15 print('\nAppList :')
16 print(nodeAppList)
```

Figure 5: Nodes Separation

The traditional graph looks more or less the same as the analysis part, except for the cleaning part. If this one has the advantage to differentiate at a glance the isolated nodes from the rest of the herd, it remains crystalline compared to the most abundant clusters as soon as the number of nodes becomes too important.

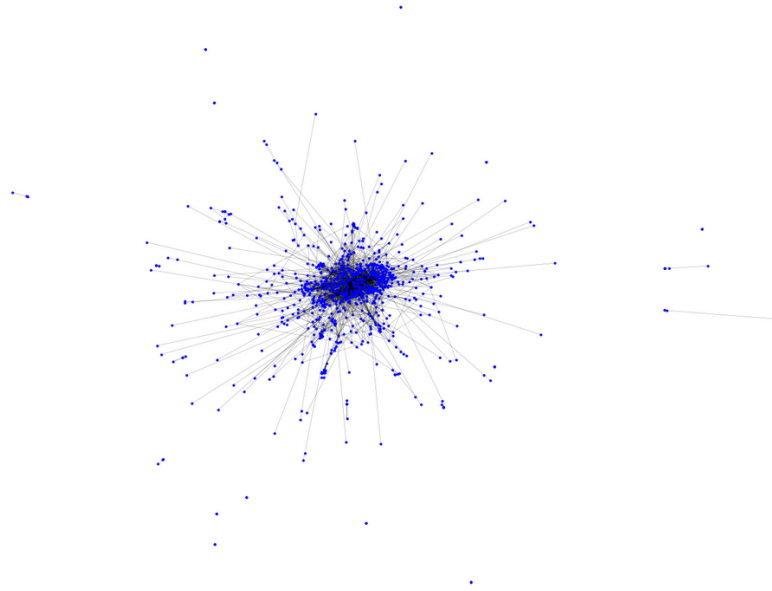


Figure 6: Classic Graph Display

Although it is basically the same graph as the one presented earlier in the analysis and in the previous chapter, the new representation gives a clearer idea of the architecture of the graph. A dozen or so applications alone hardly phagocyte almost all users. The few diehards who use certain applications without success are now drowned in the mass and become almost invisible.

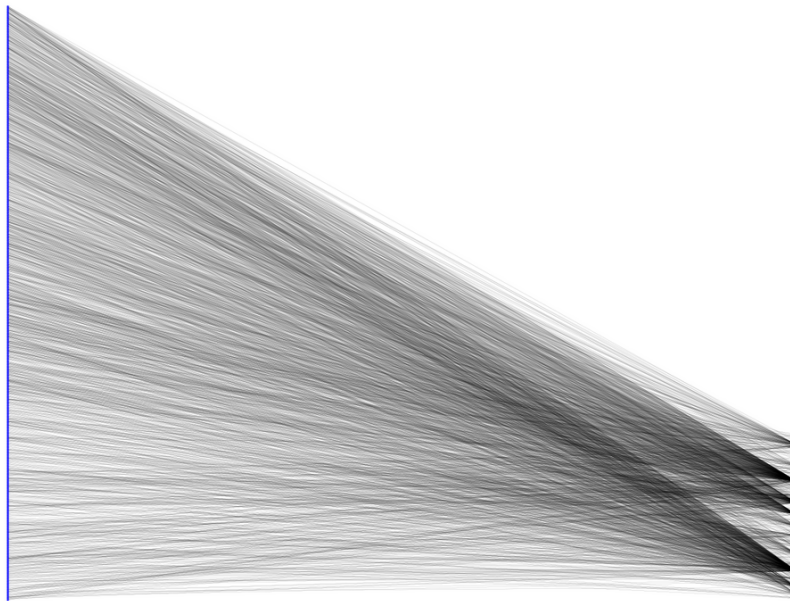


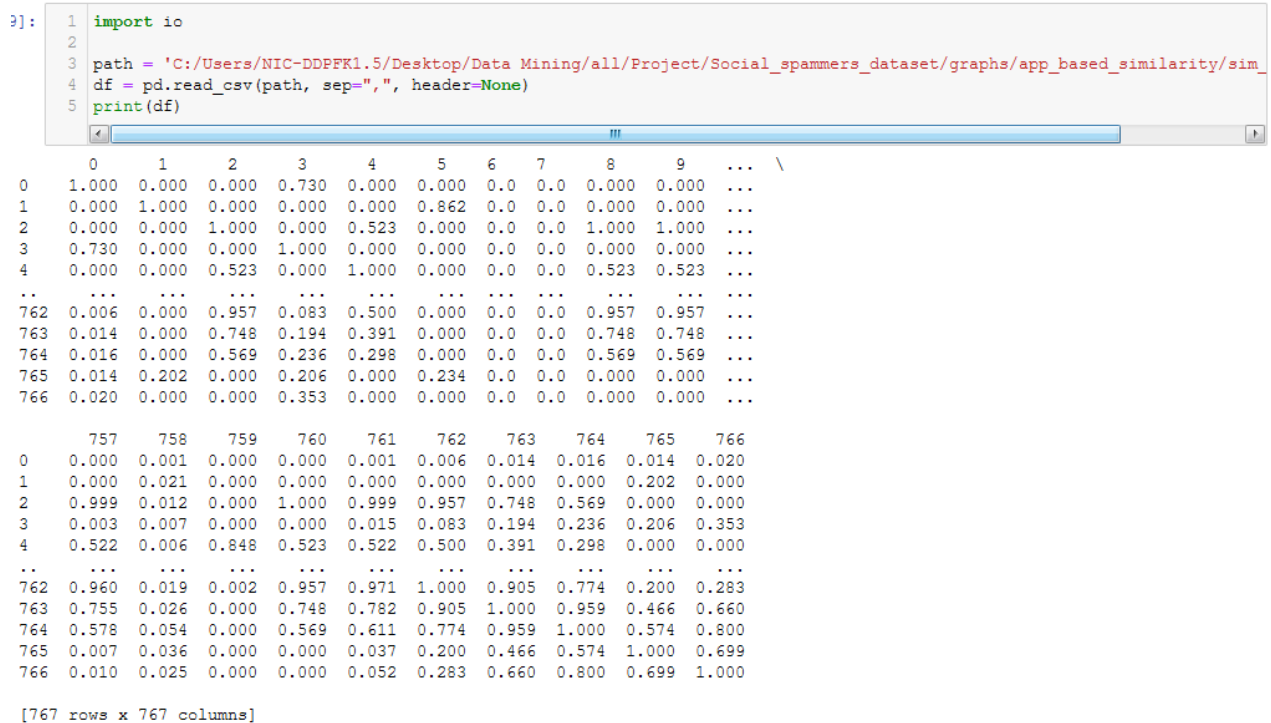
Figure 7: Bipartite graph with on the left the 767 users and on the right the 215 applications

3.2 Analyses on the the sim matrix file

This chapter will describe the work done in the notebook called 'BigData_SimMatrix_V2.ipynb'. We will analyze the data contained in the text file 'sim_matrix.txt' in the 'graph' folder of the dataset provided for the project, more precisely in the part entitled 'app_based_similarity', mainly focused on the points of convergence that can be detected between different types of entities. In the 'sim_matrix.txt' file, 2 major informations are stored, the identifier of each user and the similarity he shares with another user. Each index of the matrix represents a user, both horizontally and vertically, also at the diagonal of the matrix, the similarity will always be at its highest value because each one is compared to itself. The size of this square matrix is equal to the number of users, i.e. 767, so we can deduce that we will find 766 relevant measures for each user, i.e. a measure of similarity with respect to each user other than himself. This similarity is a measure between 0 and 1, but it is not binary, it includes a gradation up to 3 digits after the decimal point. We will remind you if necessary that a similarity of 0 will indicate a total dissimilarity 2 Internet users and a similarity of 1 will mean a perfect similarity. Initially started on Google Colab, this part had to migrate to Jupyter Notebook largely because of a limitation in colab's computing power that resulted in sometimes indecent compilation times on a large graph, such as the one we will see. Once the matrix contained in the TXT file has been extracted, here is what we get:

Parsing of the matrix from the TXT File

```
3]: 1 import io
    2
    3 path = 'C:/Users/NIC-DDPFFK1.5/Desktop/Data Mining/all/Project/Social_spammers_dataset/graphs/app_based_similarity/sim_
    4 df = pd.read_csv(path, sep=" ", header=None)
    5 print(df)
```



	0	1	2	3	4	5	6	7	8	9	...
0	1.000	0.000	0.000	0.730	0.000	0.000	0.0	0.0	0.000	0.000	...
1	0.000	1.000	0.000	0.000	0.000	0.862	0.0	0.0	0.000	0.000	...
2	0.000	0.000	1.000	0.000	0.523	0.000	0.0	0.0	1.000	1.000	...
3	0.730	0.000	0.000	1.000	0.000	0.000	0.0	0.0	0.000	0.000	...
4	0.000	0.000	0.523	0.000	1.000	0.000	0.0	0.0	0.523	0.523	...
...
762	0.006	0.000	0.957	0.083	0.500	0.000	0.0	0.0	0.957	0.957	...
763	0.014	0.000	0.748	0.194	0.391	0.000	0.0	0.0	0.748	0.748	...
764	0.016	0.000	0.569	0.236	0.298	0.000	0.0	0.0	0.569	0.569	...
765	0.014	0.202	0.000	0.206	0.000	0.234	0.0	0.0	0.000	0.000	...
766	0.020	0.000	0.000	0.353	0.000	0.000	0.0	0.0	0.000	0.000	...
...
757	0.000	0.001	0.000	0.000	0.001	0.006	0.014	0.016	0.014	0.020	...
758	0.000	0.021	0.000	0.000	0.000	0.000	0.000	0.000	0.202	0.000	...
759	0.999	0.012	0.000	1.000	0.999	0.957	0.748	0.569	0.000	0.000	...
760	0.003	0.007	0.000	0.000	0.015	0.083	0.194	0.236	0.206	0.353	...
761	0.522	0.006	0.848	0.523	0.522	0.500	0.391	0.298	0.000	0.000	...
762	0.960	0.019	0.002	0.957	0.971	1.000	0.905	0.774	0.200	0.283	...
763	0.755	0.026	0.000	0.748	0.782	0.905	1.000	0.959	0.466	0.660	...
764	0.578	0.054	0.000	0.569	0.611	0.774	0.959	1.000	0.574	0.800	...
765	0.007	0.036	0.000	0.000	0.037	0.200	0.466	0.574	1.000	0.699	...
766	0.010	0.025	0.000	0.000	0.052	0.283	0.660	0.800	0.699	1.000	...

[767 rows x 767 columns]

3.2.1 Community analysis

The first point we will study is the architecture of the graph structure and the way the different users are linked together. This network allows not only to visualize who is linked to whom but also offers an overview of the groups formed by individuals, which agglomerate in successive clusters of varying sizes, knowing that a major group at a given scale may be only a sub-part of a much larger whole. In order to be able to exploit the file we just parsed earlier, we will use the Numpy library to transform our parsed object into a matrix and then we will use the NetworkX library to transform our new matrix into a graph.

```

1 # Creation of the graph
2 import numpy as np
3
4 MatrixArray = np.array(df)
5 G = nx.from_numpy_matrix(np.matrix(MatrixArray))

```

One can see by the numbers what we observed visually before, community n°1 dispenses with an overwhelming number of members compared to the 4 others who are next to it.

```

: 1 # Details of the main communities
2
3 connectedComponentList = list(nx.connected_components(G))
4 counter = 0
5 for component in connectedComponentList:
6     print('Community ' + str(counter) + ' -> ' + str(len(component)) + ' members')
7     counter = counter + 1

```

```

Community 0 -> 30 members
Community 1 -> 544 members
Community 2 -> 8 members
Community 3 -> 25 members
Community 4 -> 16 members

```

• On a global scale

Our first glance at the set of relationships allows us to see that around a galaxy of an immense community, a myriad of small groups orbit at a respectable distance. The scale hides them from us, this belt of minor clusters is sometimes composed only of isolated individuals, in perfect similarity only with themselves. The major cluster in the center knows how to hide its game from it and is actually composed, as we will see later, of a myriad of clusters of various sizes. This first overview gives us the first piece of information that clustering is the norm and isolation the exception. As this trick will be repeated throughout the rest of the notebook, we can point out that to improve the display of the NetworkX library's default graph, which is not very readable, we will add 2 elements related to the parameters. First of all, I would point out that a tip found on a forum brings a quality of life to the developer who will be able to externalize graphic parameters such as the color or the size of the nodes, or the size of the labels font and the thickness of the edges line. The other useful function 'figure()', not much emphasized in the documentation of the library, will allow to change the size of the graph output file from tiny at first, from unreadable to clear.

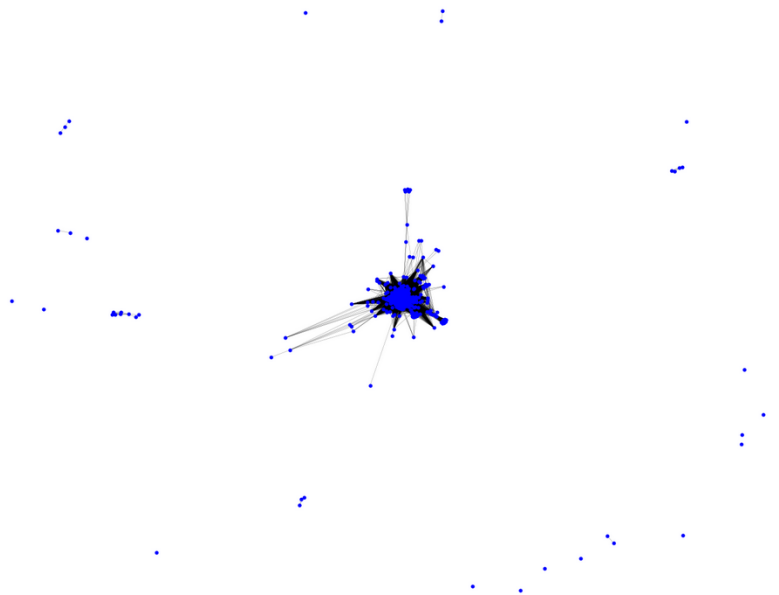


Figure 8: Global view of the relations without intervention on the data

As it is, the graph is hardly usable as it is saturated with parasitic data. Visually, it is unreadable because the concentration of certain clusters masks what is really hidden in the center. Each user is

potentially equally important at the beginning, but it will now be necessary to separate individuals according to the number of relationships they have with other members and the similarity that unites them.

The second criterion that we will take into account when cleaning up the graph will be the similarity itself, which is at the center of the concerns in the file we are using as a basis. Through the numerous tests carried out, it has been determined that a good balance is reached if we set the similarity threshold at 0.8, so all the edges that bear a weight lower than this value are cut clean. What will result in the source code by replacing the value of the node that has the misfortune of being lower than the threshold value is simply downgraded to 0, allowing not to be considered in the various operations and calculations.

The NetworkX library function `'get_edge_attributes()'` will retrieve the list of weights for each edge, then the function `'remove_edges_from()'` will give us the possibility to remove the edges that do not reach the required weight threshold. It is not so much the users, who are represented by each of the nodes in the graph, that we are trying to get rid of, but groups of such a small size that they are of little importance to us. Thus by testing different values we observed that a minimum threshold of 5 members was the limit for a group to be considered valid.

All the manipulations performed are now carried out on the graph constructed earlier and the matrix is for the time being discarded. We will use almost exclusively from there the NetworkX library, this one has an excellent documentation which gave 90% of the tools used but as of course, it misses sometimes essential contextualisation, also some of the operations remained blocking enigmas until finding the solution of their use on forums dedicated to this topic, each of these technical borrowings will be indicated in the comments of the source code.

As its name indicates, the `'connected_components()'` function returns the list of groups of nodes connected to each other, so from this generated list we will search in all the `'detected components'` which are those with a maximum of 5 members and delete the nodes that are part of them.

We first make sure that at the beginning of the dedicated cell in the source code we have a graph split into a multitude of clusters.

Then, to illustrate the numerical evolution of this cleaning, the count of the effective edges connecting the nodes, i.e. the edges that do not yet exist or that do not have a weight of 0, will be displayed at the end of each major cleaning step. In the end, starting from a weighted edge count of 148464, the graph will be reduced to 40746 edges once the weights below 0.8 have been removed and then the graph will only contain 40674 edges once the nodes in a cluster below 6 members have been removed

The result is a much clearer and more analyzable graph than the original graph, which you can compare to the previous chapter. We can now, without even referring to a list of related components, count 5 major groups.

```

1 # We check if the graph is separated into several entities
2 print('Is the graph connect ? ' + str(nx.is_connected(G)))
3
4 print('Original size fo the graph = ' + str(G.size()))
5
6 # We remove the weights from the diagonals of the matrix, i.e. the weight of 1.0 of each node with itself.
7 for i,j,data in G.edges(data=True):
8     if i == j:
9         data['weight']=0
10
11 # We will remove all edges that have a weight less than 0.8
12 # We could have done that in the matrix we extracted from the TXT file but work will be done on the graph
13 # technical solution found on stackoverflow
14 # see https://stackoverflow.com/questions/60174232/removing-links-over-a-specific-weight
15 edgeAttWeightList = nx.get_edge_attributes(G,'weight')
16 G.remove_edges_from((edge for edge, weight in edgeAttWeightList.items() if weight < 0.8))
17 print('Size of the graph after the removal of the weakest weights = ' + str(G.size()))
18
19 # If a component have less than 6 nodes then the node are removed from the graph
20 # Technical solution found on stackoverflow
21 # see https://stackoverflow.com/questions/38308865/how-to-remove-small-components-from-a-graph
22 connectedComponentList = list(nx.connected_components(G))
23 for component in connectedComponentList:
24     if len(component) < 6:
25         for node in component:
26             G.remove_node(node)
27
28 print('Size fo the graph after the removal of smallest community = ' + str(G.size()))
29
30 options = {
31     "node_color": "blue",
32     "node_size": 10,
33     "linewidths": 0,
34     "width": 0.2
35 }
36
37 # Change of the output size
38 plt.figure(3,figsize=(15,10))
39 nx.draw_spring(G, **options)
40 plt.show()

```

```

Is the graph connect ? False
Original size fo the graph = 148464
Size of the graph after the removal of the weakest weights = 40746
Size fo the graph after the removal of smallest community = 40674

```

Figure 9: Cleaning the graph to remove parasitic edges and nodes

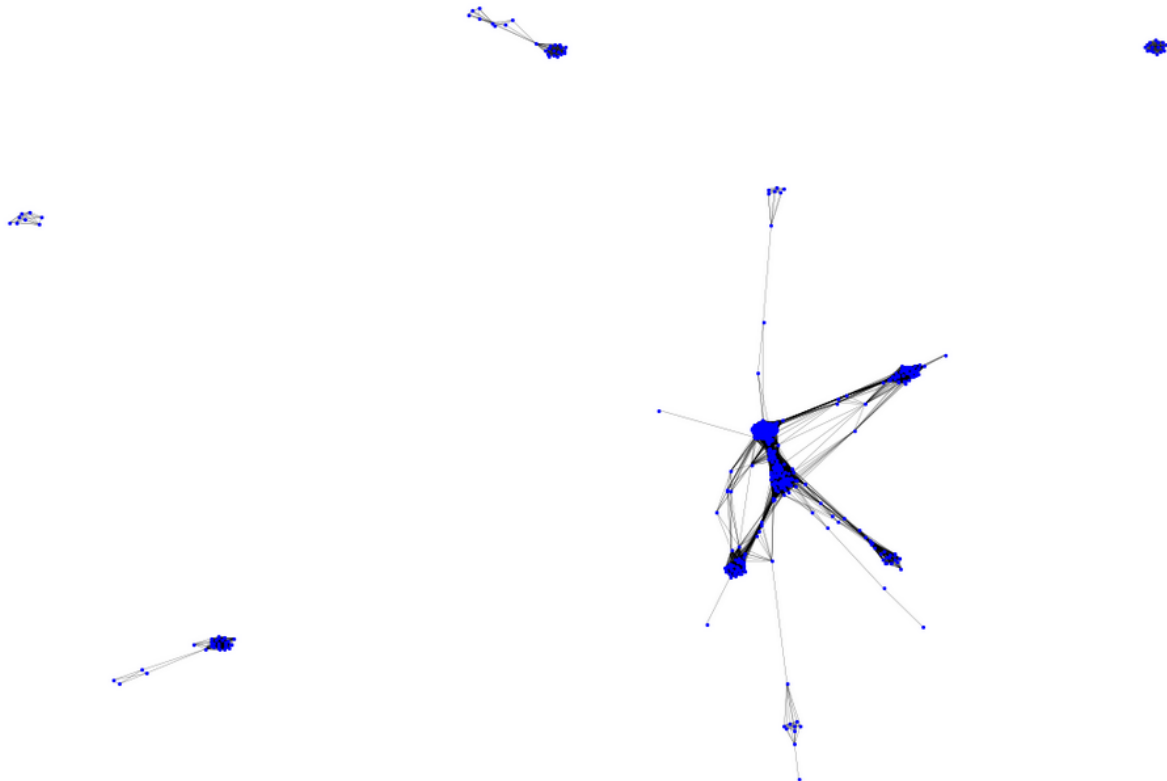


Figure 10: Display of the cleaned graph

- **Community detection**

In order to better illustrate the division of communities, we will use the 'community' module from the 'python-louvain' library, which will allow us to detect communities, thus allowing us to display them first. The library will proceed to 'partitions' based on the nodes and the value of the associated weights, which will add one more parameter compared to the graph displayed above which, apart from eliminating the lowest weights, will mainly take into account the number of nodes in a cluster.

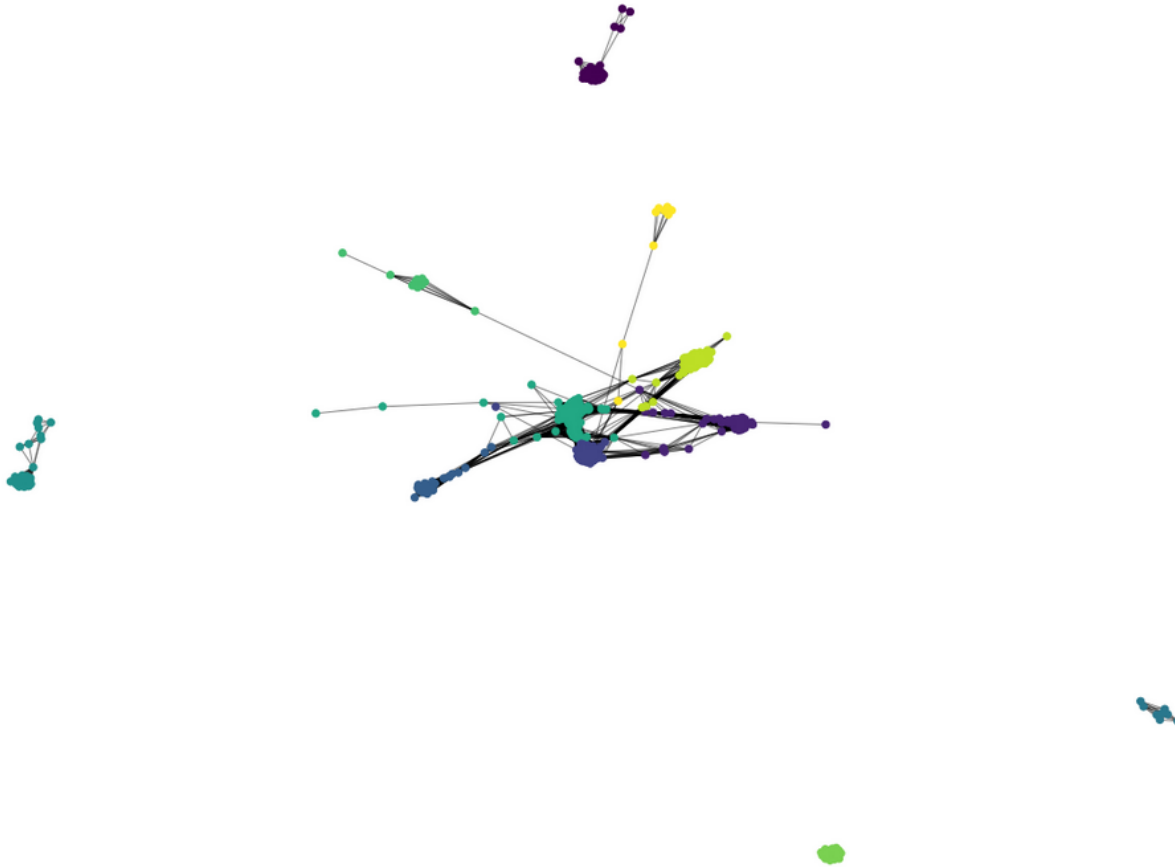


Figure 11: Displaying the community graph

- **To the naked eye**

Even with the help of the previous graphs, the large number of nodes transforms each cluster into a forest of edges from which it is extremely difficult to quantify the number of nodes and links that unite them. In order to measure at a glance the importance of the major node in comparison with the others, this image allows to realize instinctively the overwhelming importance of the major node on the minor nodes.

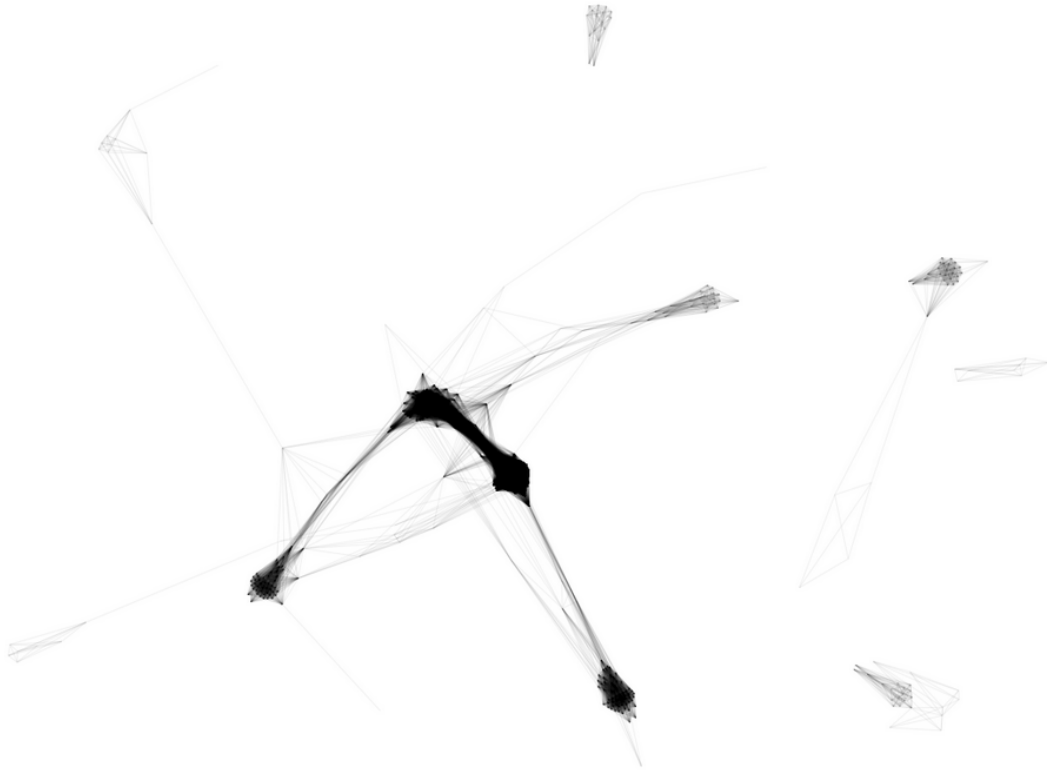


Figure 12: Minimal display without the nodes to see the network distribution

From the dedicated viewer 'Networkx_Viewer' we get a more accurate view of the architecture of the communities with, as we observed earlier, 4 dwarf communities and 1 giant community. To make the reading easier, each one of them has been moved by hand in order to better see how the network is composed.

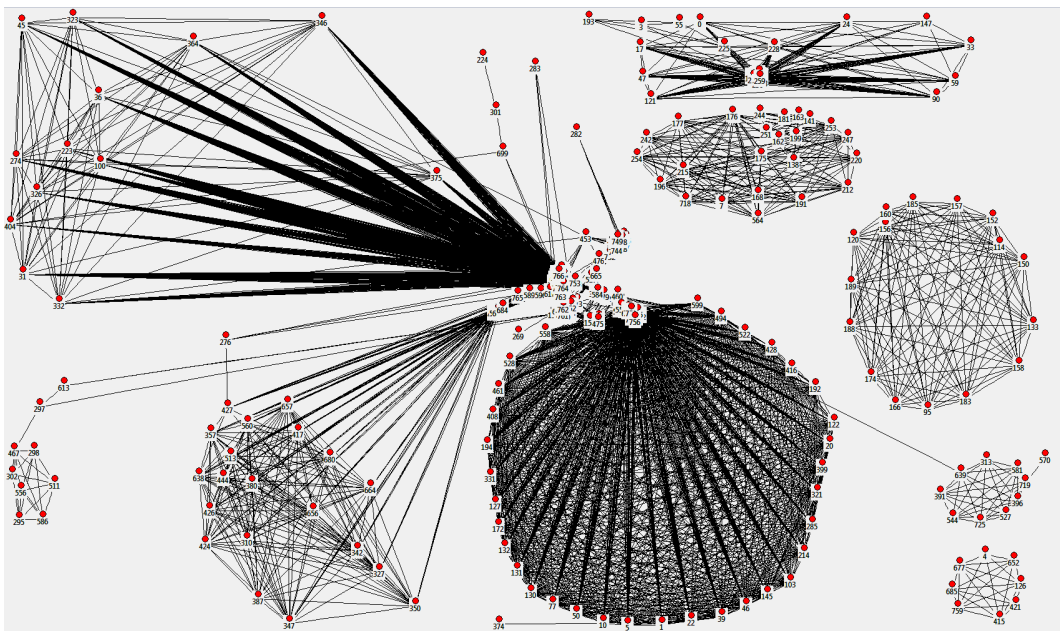


Figure 13: Exploded view of the network from NetworkX Viewer

4 Data Engineering

Data Engineering involves includes all related process such as data collection, pre-processing, feature selection and data transformation. Every single of these process impacts the end results of the experiments. In this section, the data engineering processes are described briefly that carried out in this experiment.

4.1 Data Description

The dataset used for this experiment was crawled from Twitter in November and December 2014 on 767 social spammers and legitimate users. The train dataset consists of 686 samples where the test dataset contains 86 samples. Both train and test dataset have equal 145 features.

4.2 Feature Encoding and Transformation

Feature encoding and transformation is yet another important before deploying the model. To encode categorical features, we have used the one hot encoding method where the categorical variables are encoded as binary vectors. For the numerical feature transformation, power transformation [1] is applied in order to make the data more more Gaussian-like and to handle the heteroscedasticity issues.

4.3 Feature Extraction

In this experiment, we applied two feature extraction method such as Principal component analysis (PCA) and feature importance. PCA is a dimension reduction technique that aims minimizing the number of features at the same time retaining maximum variance among the data points. Feature importance is a method to find out the best features that impacts the most during prediction. From our experiments, we found that feature extraction did not improve the classification performance significantly. Moreover, considering the small size of the data, high dimension of the data was easily handled.

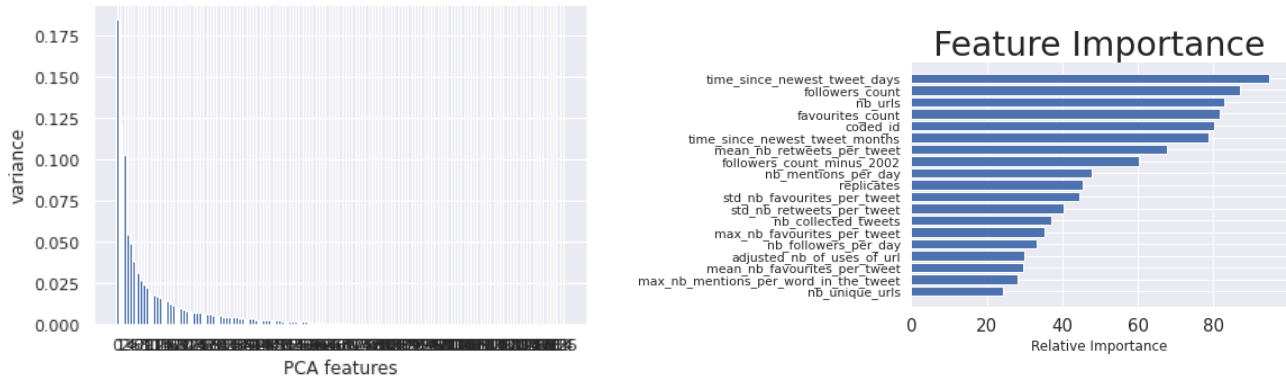


Figure 14: PCA and feature importance for feature extraction

5 Supervised Learning Methodology

In this section, we will discuss about the state-of-the-art methodology used to carry out the experiment. We will also discuss about the optimization of the hyperparameters, selection of correct evaluation metrics as well as brief discussion about the algorithms.

5.1 Data separation

Before moving on to the modeling part, the given train data is splitted into a train and validation set. 80% of the data is used a training set and the rest 20% is used as a validation set. During the separation of the data, stratified random sampling was maintained in order to keep the original ratio of between legitimate and malicious account. Out of 686 samples, 549 samples were used during the train phase and the rest 97 were during validation.

5.2 Classification Algorithms

For the classification task, 3 state-of-the-art algorithms and one traditional machine learning algorithms have been applied. Support Vector Machine is chosen as our baseline model. Among the state-of-the-art, two of them are tree based ensemble boosting algorithms and other one is TabNet that is a deep tabular data learning network. From the t-SNE visualization, we can see that the data ponts are moderately hard to separate. This is the reason we have chosen the tree based ensemble boosting algorithms, as they are capable of capturing the complex relationship of the data as well suited for small to medium size dataset which is the case for our experiment. Moreover, these algorithms have exhibited outperforming results in various data mining competitions and real life application when compared to the traditional models[1]. Deep Learning algorithms have shown to obtain state-of-the-art performance in various machine learning and data mining task. But when it comes to tabular data, deep learning does not preform well. keeping this in mind, we applied TabNet which is developed by the google researcher that has proven show good results for tabular data. Using the SVM, we will set a baseline performance and later use the SOTA algorithms to improve upon the baseline models. We will also discuss about the predictive performance of the algorithms comparing the training and validation dataset.

5.3 Bayesian Hyperparameter Optimization

Hyperparamter aims at finding the optimal set of parameters that minimizes the cost function. This is one of the most important part of the classification task, choosing the right set of parameters may significantly improve the predictive performance of the model. With optimization of hyperparameters, the predictions may be biased or overfitted. For the hyperparamter optimization we have applied a distributed Bayesian optimization approach for the algorithms. The reason behind choosing a Bayesian method is that set of optimal parameters are found in less time with this approach since it reasons about the optimal parameters to estimate based on the past trials.

5.4 Evaluation Metrics

To evaluate the performance of the model, we have use 8 different evaluation metrics. The evaluation metrics are Accuracy, Precision, Recall, True Negative Rate (TNR), True Positive Rate (TPR), False Negative Rate (FNR), False Positive Rate (FPR), F1-score and ROC-AUC. Each evaluation metric has its own way of evaluating the model. However, we will only consider the Accuracy, F1-score, ROC-AUC as these metrics will reflect how good the models are fitted.

6 Result Analysis for Classification

In this section, we will describe about the results obtained as well as comparison of the models in terms of both training and testing set. This will ultimately help us to choose the best model for the unseen data. The results can be found in table bellow.

		TNR	FPR	FNR	TPR	ROC-AUC	Precision	Recall	F1-score	Accuracy
SVM	Train	0.823	0.003	0.005	0.166	0.981	0.978	0.968	0.973	0.991
	Val	0.819	0.007	0.014	0.159	0.954	0.954	0.9167	0.936	0.978
CGB	Train	0.828	0.000	0	0.172	1	1	1	1	1
	Val	0.812	0.007	0	0.174	0.99	0.960	1	0.975	0.992
NGB	Train	0.823	0.005	0.004	0.168	0.986	0.968	0.979	0.973	0.991
	Val	0.811	0.014	0.022	0.152	0.928	0.913	0.875	0.893	0.964
TabNet	Train	0.821	0	0	0.171	1	1	1	1	1
	Val	0.811	0.007	0.014	0.159	0.954	0.954	0.917	0.936	0.978

From the result we can see that, SVM, despite being a traditional model, it has a good performance. It sets a baseline score of 0.981 (ROC-AUC), 0.973 (F1-score) and 0.991 (Accuracy). If we look at the SOTA algorithms, CatBoost has an impressive performance for ROC-AUC, F1-score as well as Accuracy. For NGBoost, it performs similar to to base model in terms of accuracy but worse for F1-score. TabNet performs similar to CatBoost but suffers during validation time and performs close to base model. As CatBoost has the best results in terms of both training and validation set, we choose CatBoost as our best model.

7 Conclusion

In this project, we focused on 2 main tasks: One is to train and test different machine learning models that will try to predict spammer twitter users from a dataset, after that compare those models based on the resulted accuracy. The second task is to effectuate operations on graphs, different graphs can be created from the dataset, and different things can be calculated like diameter, density, community detection and several statistics. The objective is to be familiarized with how data mining is used in an enterprise scale. We found the task moderately easy for the classification task provided the result in the validation set. We are also confident that our best model(CatBoost) will keep up the performance for the test data.

References

- [1] https://www.researchgate.net/publication/344457137_Predicting_the_Return_of_Orders_in_the_E-Tail_Industry_Accompanying_with_Model_Interpretation
- [2] https://github.com/jsexauer/networkx_viewer
- [3] Community Detection
<https://github.com/taynaud/python-louvain>
- [4] stackoverflow.com - Remove small component from a graph
<https://stackoverflow.com/questions/38308865/how-to-remove-small-components-from-a-graph>
- [5] stackoverflow.com - Remove link
<https://stackoverflow.com/questions/60174232/removing-links-over-a-specific-weight>
- [6] stackoverflow.com - Draw directed graph
<https://stackoverflow.com/questions/20133479/how-to-draw-directed-graphs-using-networkx-in-python>
- [7] stackoverflow.com - Community Generator
<https://networkx.org/documentation/stable/reference/algorithms/community.html>

8 Reproducibility of results

In this section, we will discuss about our reproducibility of results setup while our study was conducted, the tools and libraries we used and their versions.

Libraries used in this experiment-

- Python v3.9.1
- Scikit-learn v0.22
- PowerTransformer
- James-Stein Encoder
- t-SNE v0.2.5
- SVM v3.24
- catboost v0.24.4
- ngboost v0.3.7
- tabnet v3.1.0

Hyperparameters of the models-

- SVM(Kernel = 'rbf')
- CatBoost (bootstrap type = 'Bernoulli', l2 lea reg=1,learning rate=0.1442, max depth=4,min child samples=3, n estimators=120,scale pos weight= 5.769578,use best model = False)
- NGBoost(Dist= Bernoulli, learning rate=0.0109620, n estimators= 170, natural gradient= True)
- TabNet (gamma = 1.77913778, lambda sparse = 1.67548407, momentum = 0.258169546, n a = 45.0, n d = 19.0, n independent = 5.0, n shared = 5.0, n steps = 9.0)