UNIVERSITÀ DI PISA

An analysis of the Twitter dataset

Data Mining Report

Acciaro, Gennaro Daniele
g.acciaro@studenti.unipi.it
635009

Carfì, Giacomo
g.carfi1@studenti.unipi.it
520951

Peluso, Christian
c.peluso5@studenti.unipi.it
641346

January 8, 2023

Contents

1 Introduction

The purpose of this report is to describe and analyze the dataset concerning members of Twitter. The data provided are essentially regarding the subscribers of this social network and the tweets written by them; the dataset contains the real behavior of the users, indeed the challenge is to handle and extrapolate as many as possible details from the tweets, in order to better classify each type of user.

The processes for achieving our goal are grouped into 4 tasks:

1. **Data Understanding**: which consists of unpacking the dataset and realizing the kind of data we are dealing with, capturing if there are semantic or syntactic errors, duplicates, outliers, or missing values, and realize how to deal with them. Keeping this in mind we will try to illustrate the distribution of data and figure out how to process the data in the following sections;

2. **Data Preparation**: here we will try to use the knowledge extracted in order to front off the highlighted problems, filling the missing values gaps and create new indices that can sum up the characteristics of each user;

3. **Clustering**: in this section, we will focus on the most important indices eliminating the less representative. On these features we will use different clustering algorithms in order to detect the types of users and describe common behavior above them;

4. **Predictive Analysis**: we will focus on the binary classification of the bot label, in order to be able to perform this task we will make use of the indicator delined in the first two sections;

5. **Time Series Analysis**: considering only users who used Twitter in 2019, we will generate a timeseries regarding each user's use of the social. After that we analyze these timeseries through clustering and classification techniques, using shapelets.

2 Data Understanding

Bearing in mind the dimension of the files, we decided to take advantage of some libraries, mainly Pandas 1.3.5 for the elasticity and efficiency that it offers to manage heavy data and Numpy 1.21.6 for the data manipulation that we can exploit. For illustration purposes, we adopted MatPlotLib 3.2.2 for the basic plotting functions and SeaBorn 0.11.2 for the customisation capabilities.
In Section 2.1, we faced the data printing of some rows for each column in order to understand what is the content of the cells and what their type should be. Then we checked if there were missing values or <NA>. Finally, in Section 2.2, we assigned column-compatible types in order to properly plot the distributions and perform the true data comprehension and project planning.

2.1 Data Semantics

The whole dataset is divided into two different comma-separated-values files which are: users.csv and the file named tweets.csv. The former represents users registered in the social network and contains a total of **11'508** rows while the latter contains **13'664'696** tweets. In Table 1 we provide information about the type of the features:

Data Type	Tweets Features	Users Features
Nominal	id, user_id	id, lang
Numeric	retweet_count, reply_count, favorite_count, num_hashtags, num_url, num_mentions	statuses_count
Interval	created_at	created_at
String	text	name
Binary		bot

Table 1: Attributes type for tweets and users.

Let us now look at more detailed information regarding the features. For users we have:

1. *id* is an incremental value to uniquely identify users, there are 11'508 integer values, the majority of them are 9 and 10 digits, this indicates that an order of magnitude of 10^8 users have been registered earlier.
2. *name* is the nickname of a user, there are 11'507 values of type object and only 11'361 unique entries, so there is 1 nan value and some duplicated names.
3. *statuses_count* is the count of the tweets made by the user at the moment of data crawling. In this phase, we can notice that only real users have nan values for this feature.
4. *lang* trivially indicates the user's language selected, there are no missing values and they are of type object; but as we stated there are meaningless and redundant values.
5. *created_at* is the timestamp at which the profile was created, the values are spread between 2012 and 2020 so we can suppose that all the users have a coherent date with the Twitter creation. As we can see most of the users have been created between 2017 and 2019.
6. *bot* is a binary variable that indicates if a user is a bot or a genuine one.

For tweets we have instead:

1. *id* is an incremental value in order to uniquely identify the user, the data frame is composed by *13'664'696*, but we have only *11'672'136* unique IDs, therefore we have some duplicates, missing values, and other incoherent values.
2. *user_id* it refers to the *id* column of the user_df. There are *222'286* unique values but the **most of these values are not numbers**, so we believe that this column will be useful to carry out a deep cleaning. There are *217'283* nan, it's not possible to identify the users that originated these tweets without text comprehension, so we will drop them.

3. *retweet_count* is an integer number that indicates how many times a specific tweet has been retweeted. This column can indicate the network of people of a user, there are *647'878* nan.

4. *reply_count* is an integer that indicates how many replies a specific tweet has received, the number of nan we found is *647'878*.

5. *favorite_count* attribute indicates the number of likes received by a tweet; this feature can be useful to point out the relevancy of a user in the platform, this column has 647'542 nan values.

6. *num_hashtags* is a numeric attribute that indicates the number of hashtags written in the tweet. There are *1'057'524* nan values, these are almost 10% of the dataset, so it is better to see fill them once selected the one corresponding to a user ID present in the User data frame.

7. *num_urls* is a column showing the number of links that are present in a tweet, can only take integer values; there are *648'623* nan.

8. *num_mentions* is a numeric attribute, it can only assume integer values and it means the number of accounts tagged in the tweet; there are *854'165* nan.

9. *created_at* tells when the tweet was written, it is an object attribute that must be converted into date-time type; there are no nan.

10. *text* represents the text of the tweet, it is a string attribute. We found that are *537'721* null values.

2.2 Data Visualization

The aim of this section is to provide some visual insights about the data we are dealing with. As we can see from the graphs in Figure 1, there are different entities with large values in the numeric type attributes. These are clearly outliers that will be fixed in the Section 3.
We can also see from the graphs regarding the date of creation of tweets in Figure 2a that there are years relative to date that are before the creation of Twitter, such as 1953, and years in the future, such as 2040.

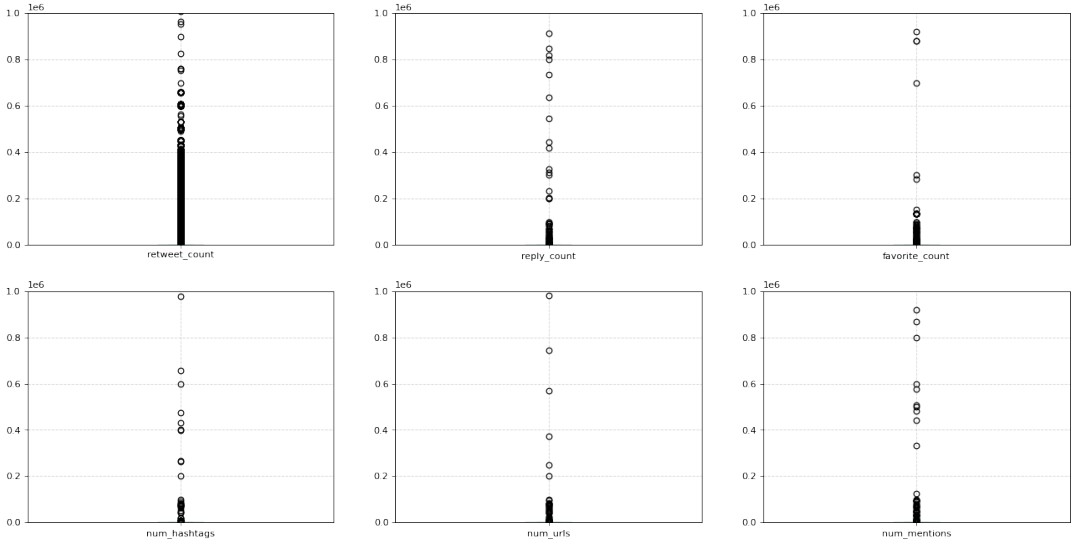
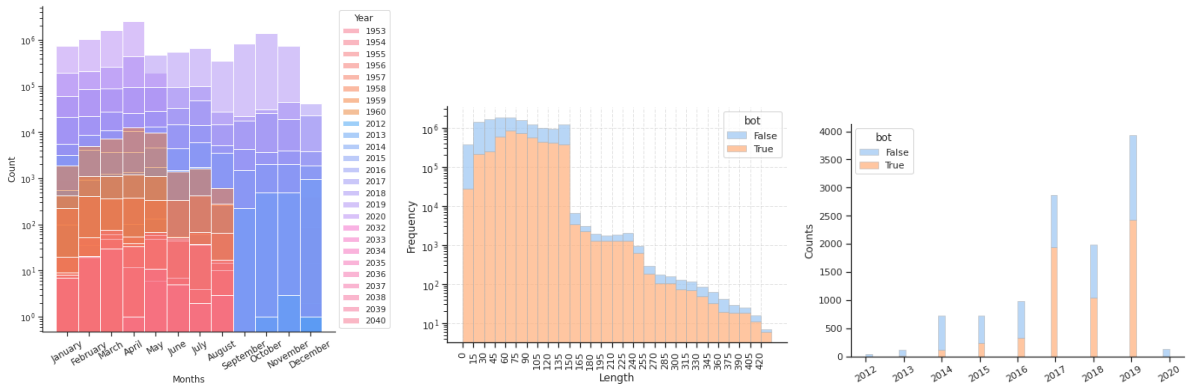


Figure 1: Box plot of numeric type features in the tweets dataframe.

From the graph regarding the length of tweets in Figure 2b, we can see that most tweets are 0 to 150 characters in length. There are some very long tweets that reach 420 characters but this is clearly an anomaly since the maximum length of characters that can be written in a tweet is 140 till November 2017 and 280 thereafter.

Finally, as we can see in the Figure 2c most of the bots were created in 2017 and 2019, we can notice from this plot that the number of bot present on Twitter are growing after 2016, maybe this is due to the fact that the Transformer technology has been introduced by Google Brain in 2017.



(a) Distribution of the tweets creation. (b) Length of the tweets. (c) Subscription date of the users.

Figure 2: These graphs are indicating some features like tweets creation date, length of the tweets and subscription of the users. The last two takes into account the bot label.

We can also see in Figure 3 that bots are present in greater numbers in our dataset with **53.1%** against **46.9%** real users, but that they contribute only **35.5%** of the total tweets.

(a) Percentage of registered users with distinction between bots and non-bots.

(b) Percentage of tweets written by real users and by bots.

Figure 3: Pie charts showing the percentages of bots and real users in the dataset.

The work done up to this point can be viewed in the notebook called *DM_0_TASK1_Understanding.ipynb* contained in the files within the project folder.

3 Data Preparation

In this section we have identified and filled missing values and outliers, in order to be able to re-plot the distributions, create indices and in general be able for the next clustering and classification steps.

3.1 Outliers Management

First of all we converted each column to the correct type, this operation when forced creates *nan* values (e.g. for the strings that obviously cannot be transformed into integers). After that, we removed tweets that have a *user_id* that does not exists in the user dataframe and deleted from the tweets those records that had the *text* field empty since we consider these record meaningless. Clearly, we also removed duplicates in both dataframes. We found that the tweet dataframe contains **1'952'099** duplicates corresponding to about 14% of the entire dataframe while the users dataframe does not contain any duplicate. We would like to enhance the fact that these duplicates are equal in everything, so they are not famous quotes or similar.

Then we continued the analysis performing the outliers selection, this step has been done printing the degree of magnitude of the values for each quantile, from the following Table 2 is clear the level in which the column's value explodes in each quantile (e.g. in #retweet we can see that after 99.9999% of data we can find 1.8^{10} that has an evident difference with the precedent one).

Quantile	#retweet	#reply	#favorite	#hashtags	#urls	#mentions
0.750000	1.0e+00	0.0e+00	0.0e+00	0.00e+00	0.00e+00	1.00e+00
0.900000	2.6e+01	0.0e+00	1.0e+00	1.00e+00	1.00e+00	1.00e+00
0.950000	4.5e+02	0.0e+00	2.0e+00	1.00e+00	1.00e+00	2.00e+00
0.970000	1.5e+03	0.0e+00	3.0e+00	2.00e+00	1.00e+00	2.00e+00
0.990000	7.4e+03	0.0e+00	7.0e+00	3.00e+00	1.00e+00	3.00e+00
0.995000	1.7e+04	0.0e+00	1.2e+01	4.00e+00	1.00e+00	4.00e+00
0.999000	1.1e+05	0.0e+00	4.6e+01	7.00e+00	2.00e+00	7.00e+00
0.999900	3.9e+05	2.3e+01	3.2e+03	1.10e+01	3.00e+00	1.10e+01
0.999990	3.3e+06	7.9e+03	2.2e+04	2.61e+03	3.54e+03	5.78e+03
0.999999	1.8e+10	1.3e+37	8.2e+09	5.13e+11	4.90e+10	9.99e+09

Table 2: We show the maximum values that we have considering a specific percentile.

Thus effectively we can delete the data that exceeds a certain threshold to perform the removal of outliers. Values detected as outliers will be replaced with nan and then filled by the mean of the user computed on that counter. In Table 3 we show the chosen threshold.

Feature	Percentile
#retweet, #url, #mentions	99.9900%
#reply, #favorite, #hashtags	99.9990%

Table 3: Here we can see the percentiles choosed for compute the cut.

3.2 Data Cleaning

We proceed our tasks checking whether there are syntactic or semantic errors for each of the feature.

- **Language** Regarding the *lang* field, we normalized all values to lowercase in order to merge values such as "*en-gb*" and "*en-GB*". Then we removed some errors such as the nonexistent values like "*select language...*" and "*xx-lc*". We changed them into "*en*" language after an analysis of their tweets.
- **ID** There were missing IDs in tweets dataframe, we decided to replace them by creating new IDs using the `.fillna('ffill')` method of Pandas.
- **Created At** The incoherent dates of the tweets have been ordered thanks the relationship with their IDs. Assuming that the IDs are more reliable than the dates, we know that them are part of an incremental field managed automatically by database software, in this way each tweet is assigned to an unique ID. This gives us a peculiar property, a smaller ID will correspond to an older tweet and the bigger ones to the latest. If the initial assumption holds for at least the central values, we can represent each day with an ID chosen with the median, avoiding noise like modified or deleted tweets, this concept is noticeable in 6b on the linear trend from the 2016 on.
  Finally, we can substitute the date of wrong tweets inside a range of coherent timestamp starting from the **subscription of the user** that wrote the mentioned tweet and the **closer representative ID**, this one will give us the day of representation and this will be the new date of the tweet.
- **Hashtags\URLs** To fill missing values on *num_hashtags* and *num_urls* we decided to check with regular expression on *text* attribute of the tweet if hashtags or URLs are present and, if they are present, we count them.
- **Name\Statuses Count** The null value in the name field have been filled with the **UNKNOWN** string, and because during the data understanding phase we saw that only real users have nan values in the *statuses_count* feature, to clear this feature we calculated the mean for actual users only and substituted it for the nan values.
- **Text** As we seen in 6c there are several tweets exceeding the maximum cap of characters permitted. Analyzing the users of these tweets, we recognized that the text was saved with a bad encoding. So we proceeded to encode the text with `.encode("ISO-8859-1")` and we decode it with `decode("utf-8")` back to our format, we confirmed the theory. Indeed the tweets were from languages that make use of alphabets that the $.csv$ file doesn't support, once the encoding has ended the correct format permitted to express the text in less characters and reduced the overall length of these foreign tweets.

3.3 Indicators

In order to properly develop the clustering and classification part, we extended the users dataframe with the following indicators calculated with respect to the single user. All these indicators have a peculiar meaning and later they will compared using the correlation, similar features will be dropped to avoid the curse of dimensionality. As we have created several indicators concerning the average, maximum or minimum of a feature, we have decided to state below sometimes the topic that represents the indicator and not the exact name so as to avoid repetition.

1. **number_of_tweets —** It wants to indicate the amount of tweets published in the overall story of the account, so it doesn't take into account if a user has signed up before or after another.

2. **tweets_2019**, **tweets_2020 —** these years are two of the ones with the major number of tweets, that's why these indicators want to highlight the number of tweets that the user posted in this range of time.

3. **likes sum, max, mean, ratio —** This marker wants to report the amount of likes received in relation to tweets written, indeed there is the total count, maximum, minimum and ratio of favourites per tweets.

4. **time_delta_sec —** This feature is calculated on the average of seconds between a tweet and another, trying to highlight the frequency of posting of the user.

5. **length_tweets —** Here we describe the length of the tweets written by the users, naturally it has been calculated on the average of the length of the tweets but it also takes in consideration the maximum characters written for a tweet.

6. **number_of_special_chars —** Assuming that the users can make use of special character for creating smiles or others kind of visual effects, this indicator wants to emphasize the special characters in the tweets written.

7. **hashtags —** The number of hashtags permits us to understand how many account are pointed by the user, this can be useful when we try to carry out the dimension of the network of the account or the amount of spam that an user can do.

8. **URLs —** Similar to the number of hashtags for the spamming purposes, but relating the pages linked by the user, it's more easy that a spammer account make use of links to point out phishing pages or advertisement rather than a common user.

9. **mention —** The mention in Twitter plays an important role, this is equal of the old fashion tags and can be used in both tweets or comments. An average user uses this rather than hashtags or URLs, so we aspect to see a major number of hashtags in common users.

10. **retweets —** Since Twitter uses the letters *RT* to indicate a tweet that have been retweeted, we can use a regex to find them and count the number of retweet made by each user. The retweets are a copy of a tweet with some comments attached, we think that the normal users make use of this practice to make a point or to express a personal opinion.

11. **consecutive_days —** According to our researches we knew that there are bots that tweet after a command or a question, this kind of bots, like the Telegram ones, offer a service to the community so we think that they can have a nice streak of consecutive active days. Indeed this indicator wants to count the number of consecutive days in which a user have tweeted.

12. **duplicate_tweets —** In this custom indicator we count how many times each tweet text appears, in search for duplicated tweets. We associate this counter with all the users who made that tweet. This indicator can be useful to understand if a tweet is a quote and the importance of the quote among the community.

13. **tweets_per_h -** Here there is the count in average, maximum and minimum of the tweets published in 1 hour.

14. **entropy —** We decided to calculate entropy on the likes received in a user's tweets, the number of hashtags in a user's tweets, the date of creation of the tweets and the length of a user's tweets. Favourites and creation entropy can be really representative, we are speaking of the uncertainty for each user to write another tweet or to receive a like, essentially these features are trying to sum up the precedent indicators.

3.4 Results of data preparation

As described previously the work done aims to get coherent and summarized data to perform clustering and classification algorithms.
Keeping this in mind we have performed some plots to better understand the work done so far and to check if there is still more to smooth. So the aim of this section is to compare the differences with the Section 2.2.
The first graph we will analyse is the matrix of box plots (Fig. 4) the difference with the graph shown above is evident, all the scales of values are revolutionized taking into account dense and reasonable ranges, as the *num_mentions* or *num_urls* features that have gone from 17 to only 2 digits.

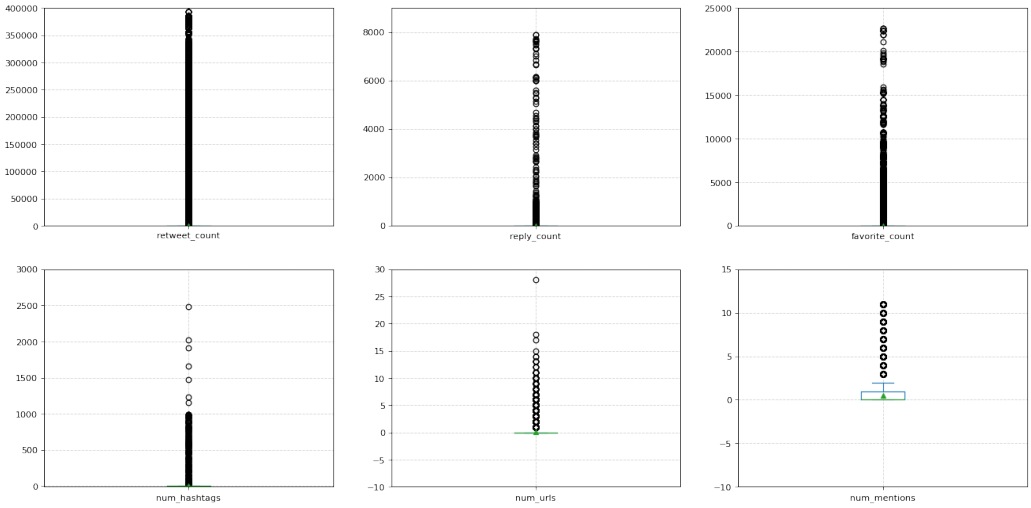


Figure 4: This plot is comparable with the Fig. 1, the differences are clear but we tried every way to keep the maximum reasonable values avoiding to disrupt sensible information out of the boxes.

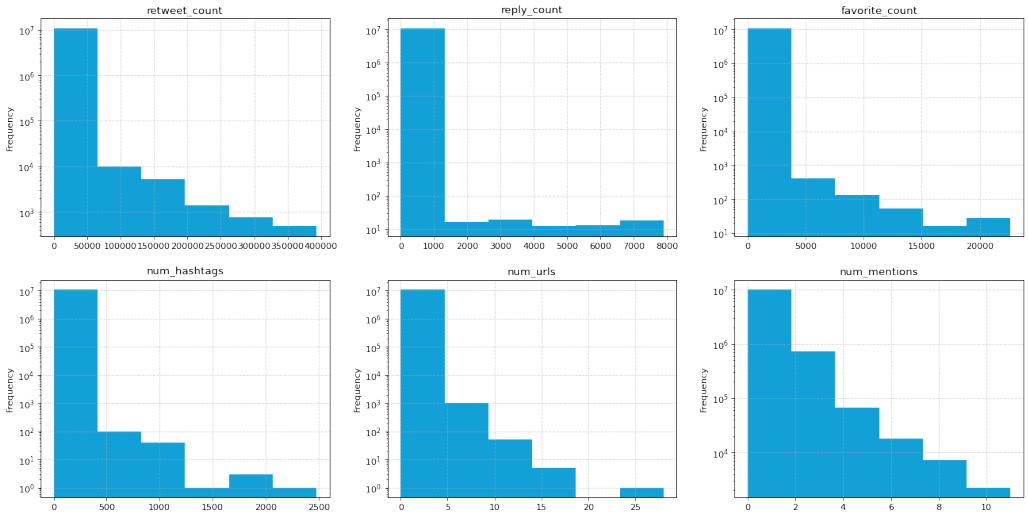
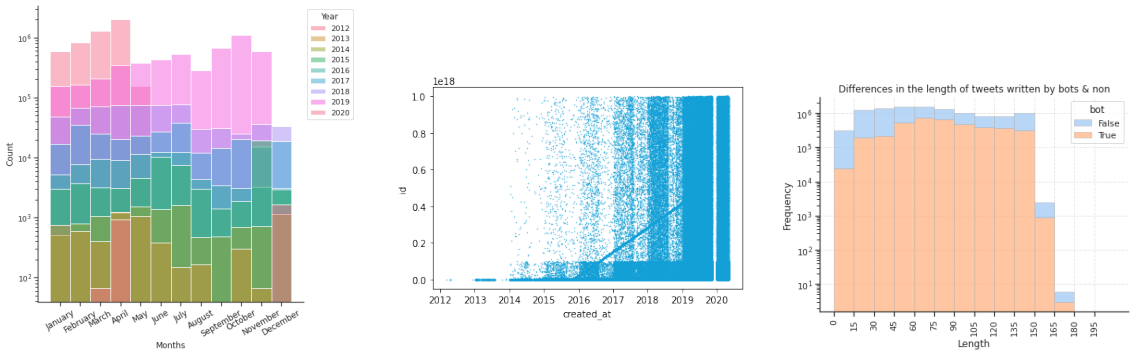


Figure 5: Distribution of numerical features on the tweets dataset after anomaly removal.

We also provide the plot of the distributions of the numerical type feature values, for clarity in Figure 5.

Next we show in Figure 6a how the work done in correcting the dates effect the tweets, falling within an ordinary range. In (Fig. 6b), as we can see lots of the tweets are disappeared from the scatter plot, this is a clear mark of the noise present in the tweet dataset with deleted or modified tweets. Instead in Figure 6c we can appreciate how the correct decoding of the text has reduced the space occupied in the length of the tweets.



(a) Distribution of tweet creation.

(b) Scatter plot of the IDs over the time.

(c) Length of the tweets

Figure 6: Graphs showing the effect of the preparation phase.

In the end there is the correlation matrix, we used it to choose the best representative indices (Fig. 7), the ones that snap a different picture of the users. As we can see different indices have a high positive correlation and some even negative, stated that we choose a few indices that were most representative of the uniqueness of the meaning.

The work done for data preprocessing and outliers management can be viewed in the notebook called *DM_0_TASK1_Preparation.ipynb* contained in the files within the project folder.

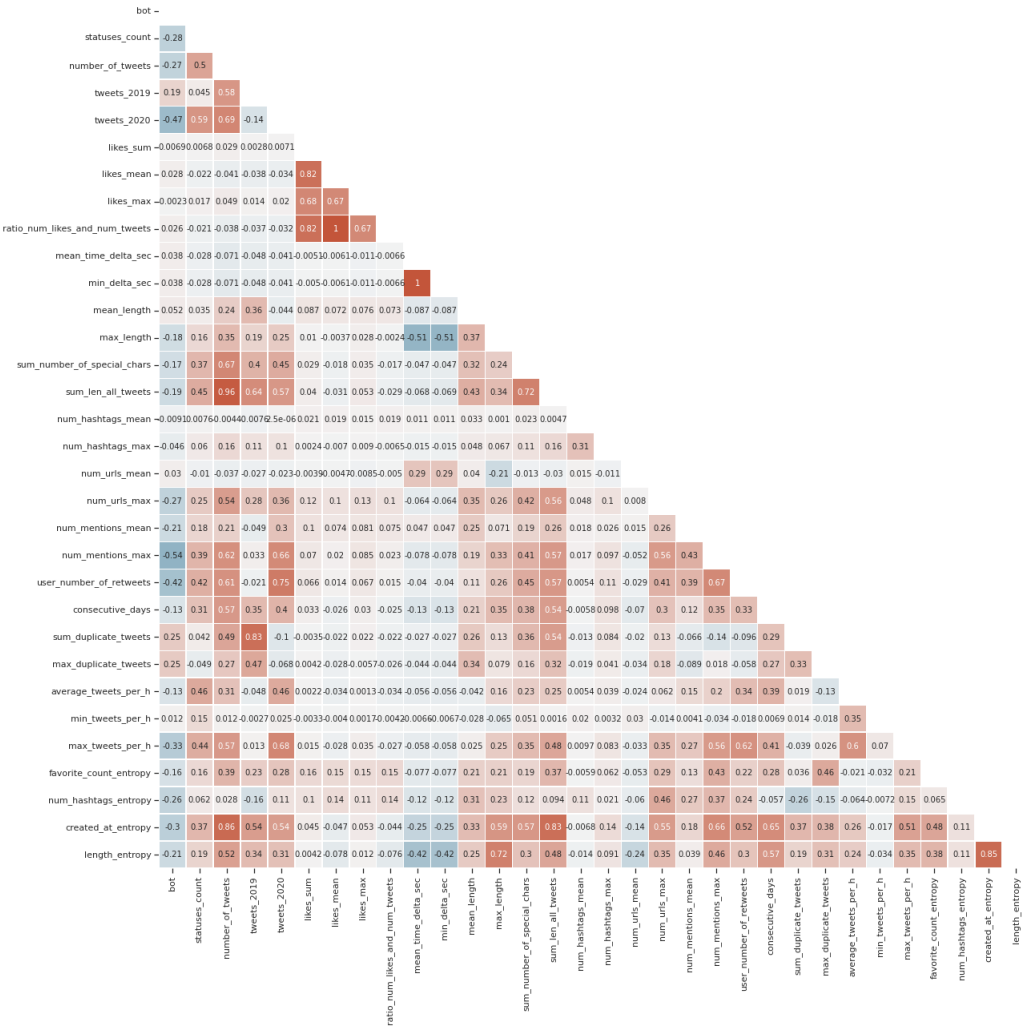


Figure 7: Correlation triangular matrix on whole the indices representing the users.

4 Clustering

The purpose of this section is to use different clustering techniques with the aim of grouping users who have similar habits in the number of tweets they have posted, the number of likes they have received and similar.

In Section 4.1 we will discuss how the data were pre-processed before clustering. In Section 4.2 we will discuss the use of K-Means. In Section 4.3, we will discuss DBSCAN. In section 4.4 we will talk about Hierarchical Clustering and finally we will provide in Section 4.5 further experiments done using X-Means.

4.1 Preprocessing

Since clustering algorithms like K-Means perform poorly when many features are present in the dataset, this is the problem known as "Curse of Dimensionality". We are taken into consideration only few attributes. In particular, clustering was performed **only** on: "*likes_sum*", "*number_of_tweets*", "*user_number_of_retweets*", "*time_delta_sec*" features.

We thought that these attributes give us general hints on the usage of Twitter by his members, but most importantly, as we saw on the correlation matrix, these indicators are not related to each other expressing as much as they can in different aspects. We decided to show the distribution of the values of these features in Figure 8 so that these distributions can be seen to be meaningful and non-uniform.

We then proceed to manipulate the data before clustering. In order to extract meaningful information about groups of users representing a certain behavior, we thought it was necessary that they must have written a minimum number of tweets equal to 10. Therefore, we eliminated, just for this phase, all users who did not meet this parameter. The total number of users considered now is **11'085** users (out of **11'508**, we removed 3.81% of users).

Bearing in mind the magnitude of the values we are dealing with, we have decided to use a logarithmic transformation of values to normalize them. Furthermore, we used the MinMaXScaler in order to have all data within the range [0, 1].

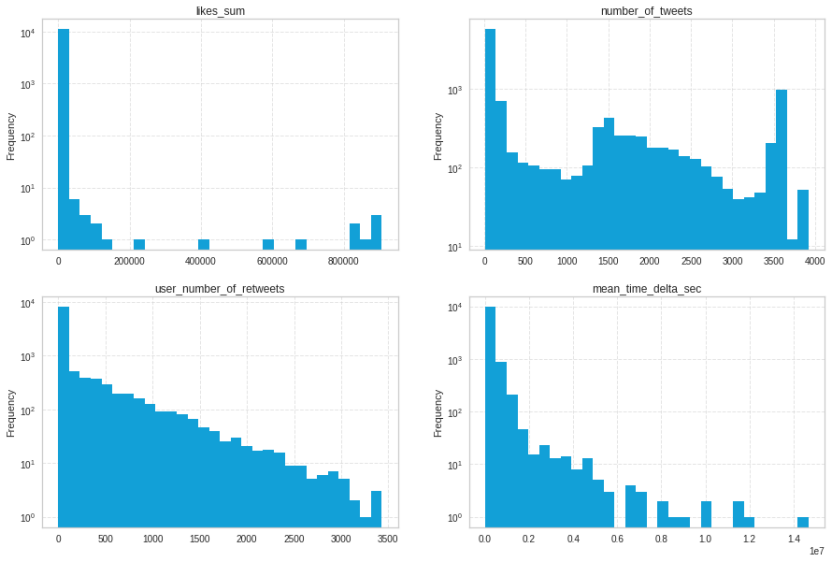


Figure 8: Distribution of chosen features for clustering.

4.2 K-Means

The first algorithm we decided to road test is the classic K-Means, an unsupervised learning method for clustering. This algorithm minimizes the sum of the distances between data points and their corresponding centroid. The first step of K-Means is to select randomly centroid points and assigning each point of the data to the closest centroid. Then a new centroid is computed as the mean point of the cluster, and again each point of the data is assigned to the closest centroid. The procedure is repeated unitl convergence in order to minimize the sum of the squared error (SSE). The most important parameter is k, the number of clusters. To be able to select it analytically, we ran the K-Means algorithm several times with k varying between 2 and 15, the metric used to comprehend the results is the SSE plus an external library (*kneed*[1]) which provides an easy way to perform the Knee method. The knee method is called this way because its purpose is to find the "knee point", where the curve visibly bends, specifically from high slope to low slope. So the knee represents

[1] https://pypi.org/project/kneed/

the point at which adding further clusters fails to add more detail. The results of this procedure are shown in the plot in Figure 9 where we have a knee point for **k=6** and for this k we obtain an SSE value of 278,582.

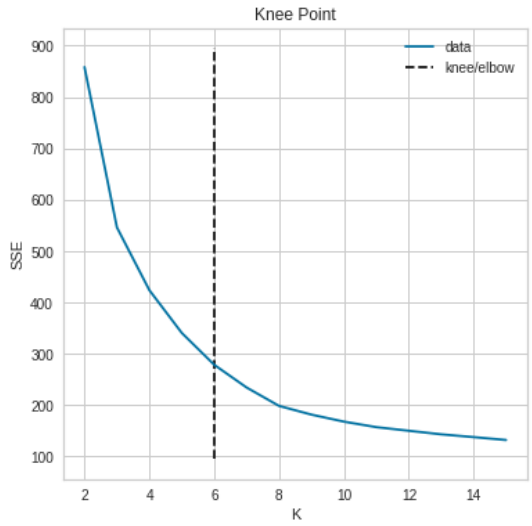


Figure 9: Knee Method for K-Means.

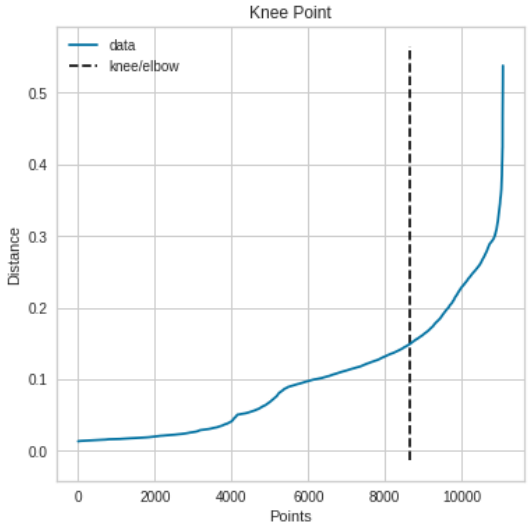


Figure 10: Knee Method for DBSCAN.

Now on, we will discuss the results that emerged by applying this algorithm. The graphs 11 and 12 differentiate the various clusters according to the features mentioned initially. Following we will provide some helpful considerations per each cluster.

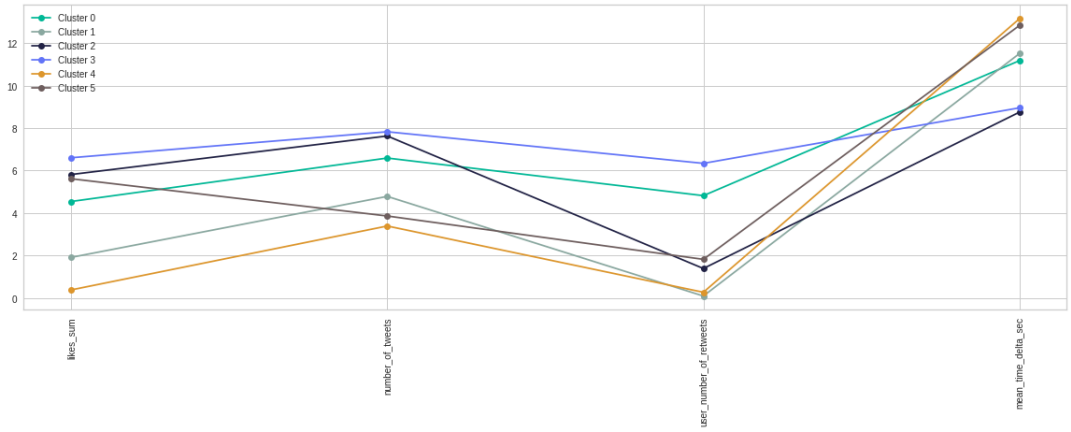


Figure 11: Distance of each cluster among the chosen attributes.

Black cluster This cluster contains users who tweeted infrequently and received a low amount of likes and retweets. We can notice that here there are 674 genuine users and 1'479 bots, so we can consider that here there are users that do not use much Twitter and bots malfunctioning or deprecated.

Brown cluster The users gathered in this cluster have a tweet frequency slightly higher than the ones in the black cluster, but they have lots of likes per few tweets and in general few retweets, this may indicate that these are celebrities or useful bots that writes original contents without interacting so much with the community, but any way being appreciated. The theory is also confirmed by the low number of points present here, only 529, indeed users like this are rare.

Cyan cluster This is the biggest cluster among all, in fact, we can see an average behavior both in the number of published tweets and likes received, remembering that *likes_sum* is the sum of total likes received, we can immediately infer that the average likes per tweet would drop compared with this value. Here we find 1'140 non-bots and 2'511 bots.

Yellow cluster Here users posted a higher number of tweets than average, as well as the number of likes and retweets. Since they have a higher posting frequency we can assume that these are daily users that make a good use of the social network, which is confirmed by the presence of predominantly genuine users 91% out of 976.

Blue and Grey clusters Finally, these 2 clusters are really close in all the indices but *number_of_retweets*, this clue says that blue users interact way less than the grey one with the twitter community. Evidence of this is the number of bots and real users in the clusters: the blue one is mainly composed of bots (1292 on 1384), while the grey cluster consists of genuine users (2'218 on 2'375).

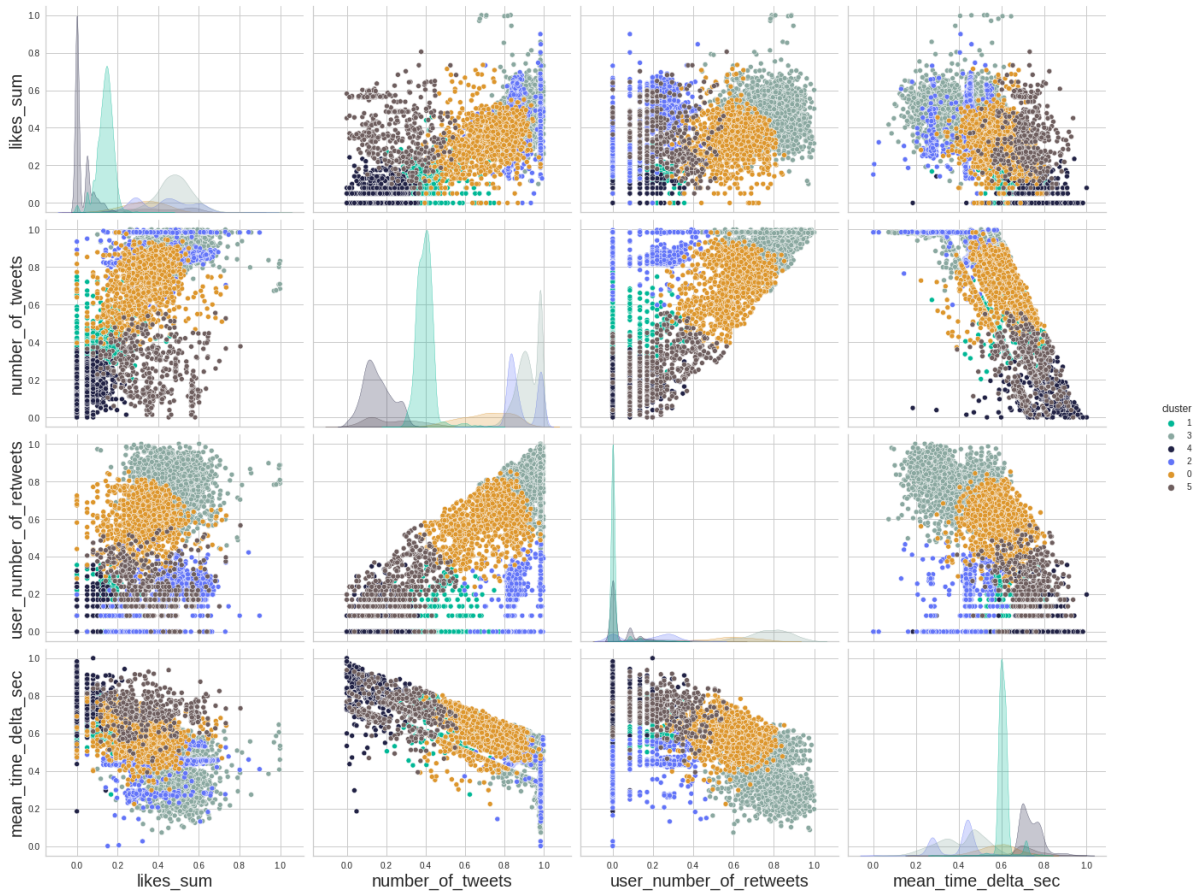


Figure 12: This is a matrix of scatter plots in which the points belonging each cluster is plotted, each row or column is a feature and in the diagonal there is the frequency distribution.

In these bi-dimensional graphs the clusters are well scattered, as we can confirm by the Figure 11. We notice that for some features like *user_number_of_retweets* and *likes_sum*, at least three clusters are overlapped. Later in the discussion of the analysis we will deal with this kind of problem. The silhouette score reached is **0.525**, remembering that this metric ranges from 1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters.

4.3 DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm that is used to identify dense regions in a dataset. It works by identifying points in the dataset that are closely packed together and labeling them as a cluster. It also identifies points that are not part of any cluster, which are called noise points. The algorithm requires two parameters:

- **Epsilon**: the radius of the ball around the points to gather neighbors, or the maximum distance between two input data to be considered in the same cluster;
- **Minimum samples**: the minimum value of input data needed to create a cluster.

To find the best value for the epsilon parameter we fixed the number of minimum points to be considered on a cluster, then we performed the *K-NearestNeighbor* algorithm to compute the average distance of every data point to its *k*-nearest neighbor, this method is useful to find a suitable distance to incorporate close points starting from a central one, finally the best epsilon is given again using the Knee method. We can see the results in Figure 10, where the optimal value found is **0,1493**.
To find the best value for the minimum samples parameter, we performed an ordered search for the number of points that would return the highest silhouette score, the range of numbers were between 8 and 298. The number of points that maximises the silhouette score is **288** and the resulting score achieved is **0,5407**. After the model selection, we refit the DBSCAN and the resulting clusters were only 4, one of those found by the K-Means was considered total noise, or almost. Indeed, DBSCAN identified 1127 points as noise, some consideration will follow.

Cluster	**Points**
0	5783
1	422
2	2918
3	826

As we well know this algorithm merges dense points between them, as the theory says it is clear that scattered points will not be labeled as core points nor border points, so they will be considered noise. A further experiment was therefore to analyse the dataset without considering the points identified by DBSCAN as noise, so first we compute again the DBSCAN without considering the noise points and then the K-Means; as we expected the latter found that the knee-point is in correspondence of 5 clusters.

Comparing Figure 13 with Figure 12, we can see that the **Brown** cluster and the bottom part of the **Yellow** cluster are the areas most affected, as we have stated before this zones are in correspondence of rare users, maybe celebrities or really appreciated bots, it's easy to understand why the DBSCAN has decided to label this points as "out of the dense areas", being these users infrequent.

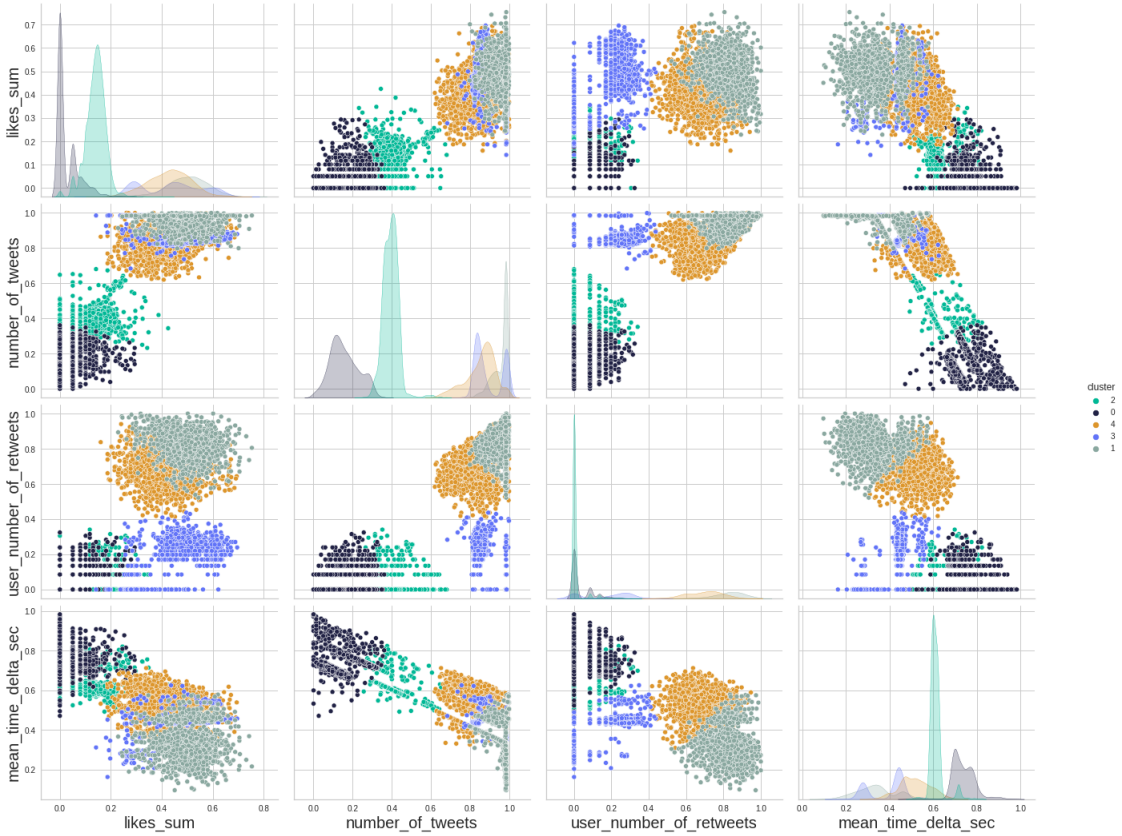
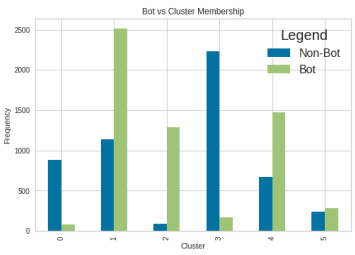
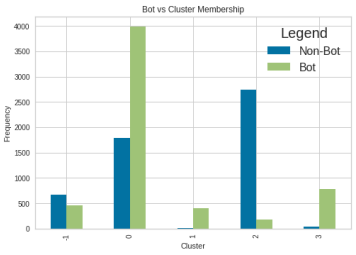


Figure 13: K-Means clusters after removing noisy points with DBSCAN.

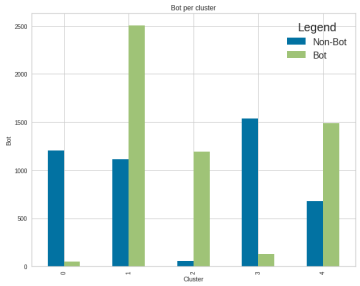
Finally, we show the distribution of real users and bots by cluster in Figure 14, noticing how indeed certain clusters are mostly composed of real users with few bots or vice versa. So the clustering algorithms well characterised the behaviour of bots and real users. In this figure, we first show the result of K-Means, then DBSCAN and finally the result of applying K-Means without the noise points.



(a) K-Means



(b) DBSCAN



(c) K-Means no noise points

Figure 14: From a first analysis we could say that at least 3 clusters can properly distinguish the bot class, this clue reassures us that through the selected features it's possible to separate the behavior of users.

4.4 Hierarchical Clustering

This is a clustering approach that aims to build a hierarchy of clusters. The result can be visualized by a plot called *dendrogram*. This algorithm doesn't require a fixed number of clusters, but it's based on a linkage function, that determines the distance metric to compute between clusters. There are several different linkage functions that can be used, between all we used: **Ward**, **Complete** and **Average**. Which linkage function is used can have a significant impact on the shape of the resulting dendrogram and the final clustering result.

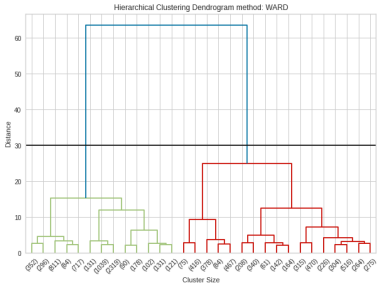


Figure 15: Ward

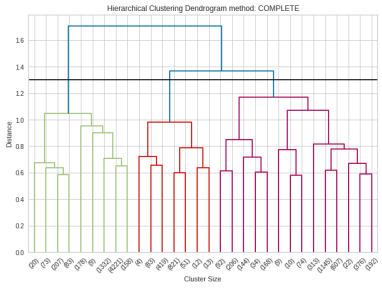


Figure 16: Complete

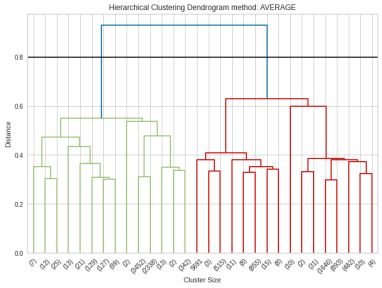


Figure 17: Average

Ward Method This is a special kind of linkage applied to Hierarchical Clustering, indeed it refers to an objective function that maximizes the investigator's purposes. In our case we opted for the Ward's minimum variance method with an euclidean distance to decide which clusters divide. Using this method we identified 2 clusters, using *30* as a threshold. The silhouette score is **0,63**.

Average Method The distance between clusters is the average distance between all pairs of points. The Average Method was used in order to group the data points into clusters based on their similarity and minimize the average distance between points in different clusters. Using this method we identified *2* clusters. The silhouette score is **0,62**.

Complete Method The Complete Linkage method was used in order to maximize the distance between the clusters and produce well-separated, well-defined clusters of different sizes. Using this method we identified *3* clusters. The silhouette score is **0,59**.

We can see the results of the 3 clustering methods in Figures 15, 16, 17 where we have also specified the cut represented by an horizontal black line. All methods were used on normalized data, as in previous algorithms.
We continue the analysis of the linkage function that performed better, the **ward** method. We can see in Figure 18 how clusters are distributed. From this plot we can see that the users who belong to the **Brown** cluster are users who have a low to high number of likes in their tweets and they are users who post less consecutively. The points belonging to the **Grey**, on the other hand, contains users who have a high average

number of tweets and post tweets consecutively. The number of likes received for these tweets is medium to very high.

From this initial analysis, it would appear that the **Brown** cluster is dominated by bots and/or people who have been using socials for a short time and in a rush, this could be due to the fact that users find the social network non-enjoyable or they can't use it properly. While the **Grey** cluster, would seem to be human-dominated, because they tweet more frequently and have more social interactions (e.g., more likes and more retweets) so they provide more human-like behaviors. This theory is confirmed if we consider the number of bots per cluster as in Figure 19a.

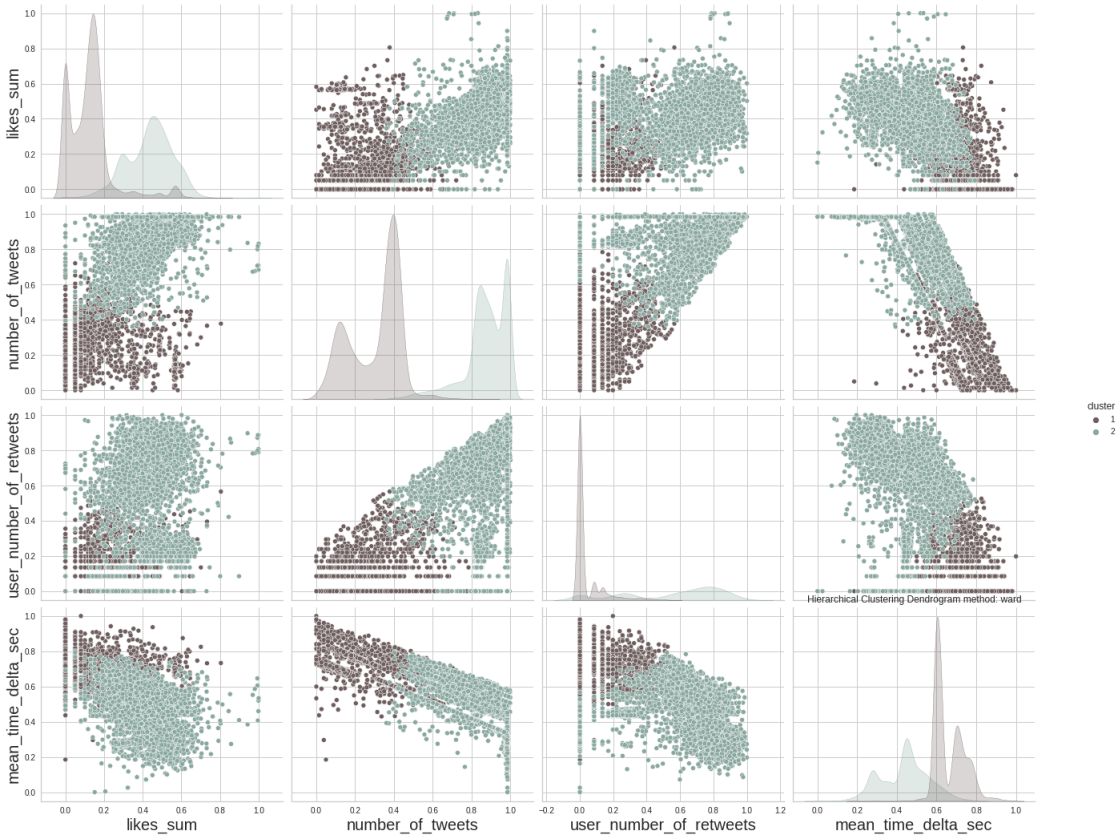
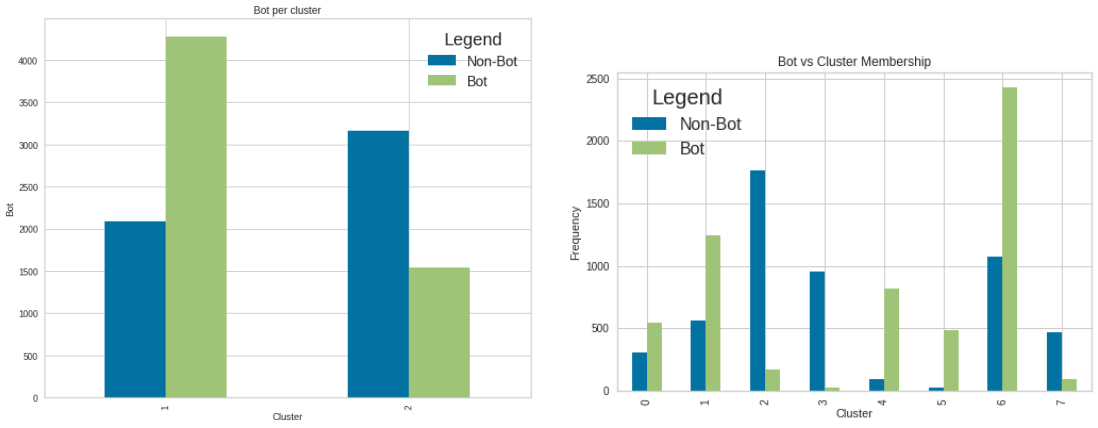


Figure 18: Hierarchical clustering results with Ward as linkage function.



(a) Hierarchical Clustering

(b) X-Means

Figure 19: Distribution of real users and bots per cluster for Hierarchical Clustering and X-Means.

4.5 Extra: X-Means

X-Means is an extension of the K-means algorithm. X-Means is an iterative algorithm that partitions the data set into a predefined number of clusters. However, unlike K-means, which requires the user to specify the

number of clusters beforehand, X-Means determines the optimal number of clusters automatically.

To implement this clustering algorithm, we used the (*pyclustering*[2]) library. We set a maximum number of clusters to be found equal to 8 and than we run the model selection, as a result we obtained 8 clusters according to the best silhouette score **0.658**.

In Figure 20 we provide the classic scatter plot for clusters. We can see that by having more clusters the graph is more fragmented. The **blue**, **cyan**, **grey** and **black** clusters are more or less in the same areas founded by the K-Means. The noise marked by the DBSCAN became the new **red** cluster separating the most moderated users of old the **brown** cluster. The second new cluster, the **white** one, is situated in areas indicating addicted users, this is clear for the little time separating a tweet with another and the number of tweets, indicating seniority in the platform.

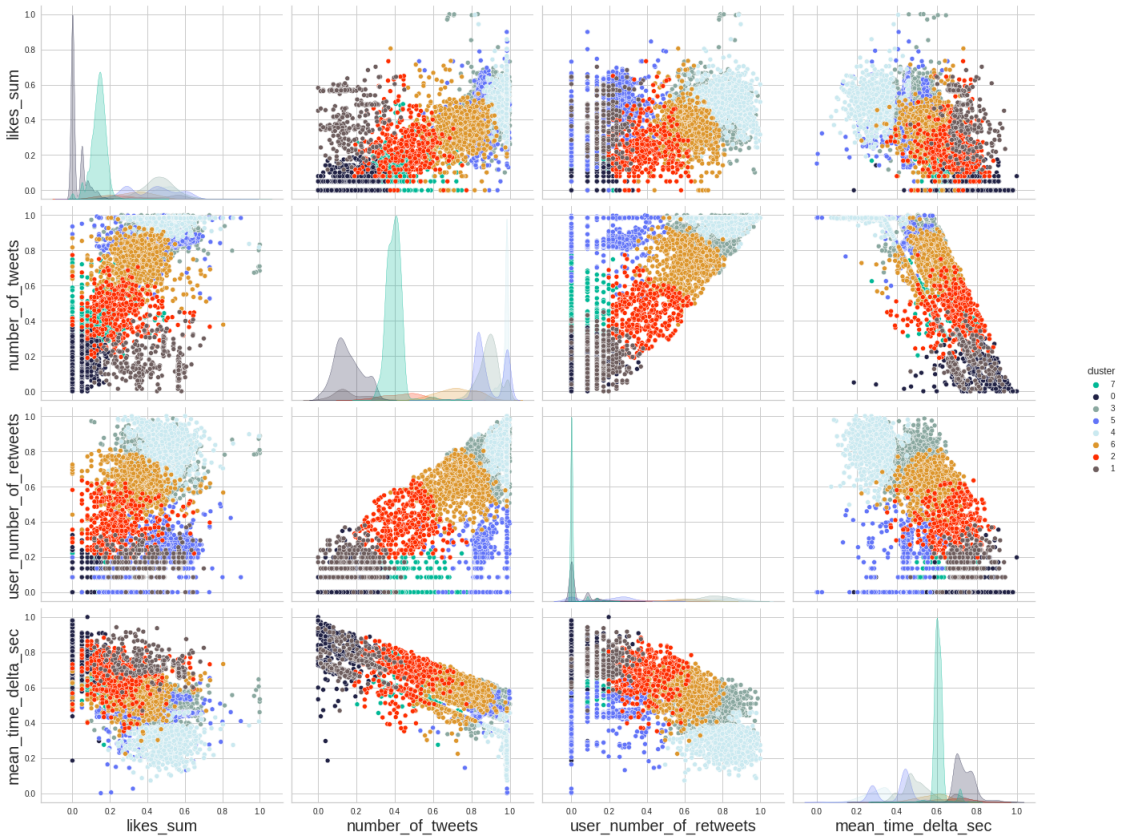


Figure 20: X-Means results.

4.6 Conclusions

Thanks to the K-Means model we began to identify clusters and to interpretate data, understanding users behaviors and experimenting configurations to continue the analysis. We understood the importance of point density and how DBSCAN is able to interpret this to detect common behaviour from rarer behaviour. Through the dendrogram of the hierarchical we understood how close are certain clusters and the importance of choosing one cut instead of another. Finally with the X-Means the number of clusters found is higher and achieves the best score considering the silhouette, we would like to see a correspondence with the precedent methods, but we understood the impossibility of doing so given the profound differences that separate these methodologies.

The work done for clustering can be viewed in the notebook called *DM_0_TASK2.ipynb* contained in the files within the project folder.

5 Predictive Analysis

In this section, we will explore various classification techniques and how we applied them, recall that our purpose is to correctly classify social users between bot and real user. In section 5.1 we will provide information on how we preprocessed the data and what classification algorithms we used, also specifying the parameters

[2]https://pyclustering.github.io/docs/0.8.2/html/index.html

used and the method by which they were obtained. In section 5.4 the results will be shown and the conclusions will be carried out.

5.1 Analysis

Our analysis, in order to train the classifiers, began by standardizing the data through Sklearn's *StandardScaler*. We chose this scaler because it is less sensitive to the presence of outliers. After that we divided the dataset into train and test sets, obtaining **10'357** elements in the train **1151** elements in the test set.
All models were implemented using the scikit-learn library, so they share the usage API: the `.fit()` is used to train the models, while the `.predict()` method to make predictions about the test set. From these predictions, we then calculated some metrics such as **accuracy**, **recall**, and **F1 score**.
In order to find the best values for the hyperparameters, we used two classical methodologies: **random search** and **grid search**. In both cases, we applied k-fold cross-validation in order to use the whole training set, given the little number of users present in the data.
In the initial stages of this task we were achieving significantly lower accuracy, so we first performed experiments using feature selection algorithms such as PCA, ISOMAP or RFE. Unfortunately, however we did not increase our results as expected. At that point, we took a few steps back in the project and revised the indicators, creating new ones (e.g. *length_entropy* or *favorite_count_entropy*).

Decision Trees Decision trees are a kind of ML algorithm that uses a tree-like model to make predictions. They are easy to interpret and understand and can handle both categorical and numerical data. We implemented this model using the *DecisionTreeClassifier* class. Table 4 shows the values used in the Random Search. We chose this model for its interpretability, in fact once the training phase is over, the resulting model is transparent and readable, especially at low depths.

Parameter	Values	Winner
max_depth	`[10,12,14,16,18,20,22,None]`	14
max_features	`list(range(2, 323, 2))`	26
min_samples_split	`list(range(10, 101, 5))`	50
min_samples_leaf	`list(range(10, 101, 5))`	100
criterion	`["entropy", "gini"]`	gini
class_weight	`['balanced', None, {0: 0.3, 1: 0.7}]`	`{0: 0.3, 1: 0.7}`

Table 4: Hyperparameters of Decision Tree, some values are presented with Python Code.

Random Forest Random forests are an ensemble method that combines multiple decision trees to make predictions. They reduce overfitting by training each tree on a bootstrapped sample of the training data and using a random subset of features. The output of a random forest is the mode of the classes for classification or the mean prediction for regression. We implemented this model using the *RandomForestClassifier* class. Table 5 contains the values used in the Random Search.

Parameter	Values	Winner
max_features	`list(range(2, 33, 2))`	28
min_samples_split	`list(range(50, 100, 5))`	70
min_samples_leaf	`list(range(50, 100, 5))`	30
bootstrap	`[True,False]`	True
criterion	`["entropy", "gini"]`	gini
class_weight	`['balanced', None, {0: 0.3, 1: 0.7}]`	None
n_estimators	`list(range(20, 101, 10))`	70

Table 5: Hyperparameters of Random Forest, some values are presented with Python Code.

Naive Bayes Classifier The Naive Bayes classifier is a simple and fast probabilistic classifier that makes strong independence assumptions between features. In particular, it relies on the Bayes Theorem. We chose

this model to have a baseline from which to start since it is a relatively simple algorithm. We implemented this model using the *GaussianNB* class. Table 6 shows the values used in the Grid Search.

Parameter	Values	Winner
var_smoothing	`np.logspace(0,-9, num=250))`	`1.64765951e-09`

Table 6: Hyperparameters of Naive Bayes Classifier, values are presented with Python Code.

Neural Networks Neural networks (NNs) are a type of machine learning algorithm modeled after the structure and function of the brain. They are composed of multiple layers of interconnected "neurons". They are highly flexible and can learn to perform a task by analyzing training data and adjusting their own internal parameters. NNs require a large amount of labeled data and can be computationally expensive to train, but they can achieve very high accuracy on many tasks. We chose this model because of their power in predictions and for our experience with them. We implemented this model using the *MLPClassifier* class. Table 7 shows the values used in the Random Search.

Parameter	Values	Winner
hidden_layer_sizes	[64, 64, 64], [32, 32, 32], [16, 16, 16], [64, 32], [64, 32, 16]	[32, 32, 32]
learning_rate_init	`[0.1, 0.01, 0.001]`	0.01
learning_rate	`["constant", "invscaling", "adaptive"]`	adaptive
solver	`["lbfgs", "sgd", "adam"]`	adam
alpha	`[0.1, 0.01, 0.001]`	0.1
activation	`["logistic", "tanh", "relu"]`	tanh

Table 7: Hyperparameters of Neural Network, some values are presented with Python Code.

Support Vector Machines Support Vector Machines (SVM) are a type of supervised ML algorithm used for classification or regression tasks. They find the hyperplane that maximally separates or predicts the value of the two classes or continuous values, respectively. SVMs are effective in high-dimensional spaces and can handle linear and non-linear data, but are computationally expensive to train and sensitive to hyperparameter choice. We choose SVM for its elasticity in many classification contexts. We implemented this model using the *SVC* class. Table 8 shows the values used in the Random Search.

Parameter	Values	Winner
kernel	`["linear", "rbf", "sigmoid"]`	rbf
gamma	`[0.001,0.0001]`	0.001
C	`[5,10,15,20]`	20

Table 8: Hyperparameters of SVM.

K-Nearest Neighbors K-Nearest Neighbors is a simple algorithm used for classification or regression tasks. It predicts the outcome for a new case based on the similarity to the "k" in most similar cases. We tried this model because it is one of the simplest models, and despite that it can perform well in certain contexts. We implemented this model using the *KNeighborsClassifier* class. Table 9 shows the values used in the Random Search.

Parameter	Values	Winner
n_neighbors	`[range(5,101,5)]`	15
weights	`['uniform', 'distance']`	distance
algorithm	`['auto', 'ball_tree', 'kd_tree', 'brute']`	auto
leaf_size	`list(range(10, 60, 5))`	45
p	`[1,2]`	2
metric	`['minkowski', 'cityblock']`	minkowski

Table 9: Hyperparameters of KNN, some values are presented with Python Code.

KNN with Principal Component Analysis In order to handle the Curse of Dimensionality, we perform the KNN a second time, but in this case, we applied the PCA algorithm. We repeated the random search of KNN with the same parameters described in Table 9 several times by changing the number of $n\_components$ used as input for PCA. In particular, we tried these values: $[4, 8, 12, 16, 20]$.

Gradient Boosting Classifier After all these models we experimented using ensamble strategies. The first method is the Gradient Boosting that is a general approach in which multiple models are trained sequentially, with each model trying to correct the mistakes of the previous model. The term "boosting" comes from the idea that the models are "boosting" the overall performance of the ensemble by focusing on the mistakes of the previous models. We implemented this model using the *GradientBoostingClassifier* class. Table 10 shows the values used in the Random Search.

Parameter	Values	Winner
loss	`["deviance", "exponential"]`	'deviance'
n_estimators	`[10, 25, 50]`	25
learning_rate	`[0.5, 1.0, 1.5]`	0.5
min_samples_split	`list(range(50, 100, 5))`	95
min_samples_leaf	`list(range(50, 100, 5))`	85
max_depth	`[10,12,14,16,18,20,22,None]`	18
max_features	`list(range(2, 33, 2))`	28

Table 10: Hyperparameters of Gradient Boosting, some values are presented with Python Code.

AdaBoost Classifier Next, we used AdaBoost in combination with Decision Tree and Random Forest. This technique is a special Boosting technique that tries to correct the errors of the previous model by assigning higher weights to the examples that were misclassified. We implemented this model using the *AdaBoostClassifier* class. Table 11 shows the values used in the Random Search.

Parameter	Values	Winner
n_estimators	`[25,50]`	50
learning_rate	`[0.001, 0.01, 0.1]`	0.001

Table 11: Hyperparameters of AdaBoost

Bagging Classifier After Boosting we tried the Bagging technique, that is another ensamble method. The main difference between Bagging and Boosting is that Bagging involves training models in parallel on different random subsets of the training data, while Boosting involves training models sequentially, with each model attempting to correct the mistakes of the previous model. We implemented this model using the *BaggingClassifier* class. Table 12 shows the values used in the Random Search.

Parameter	Values	Winner
n_estimators	`list(range(20, 101, 10))`	80
max_features	`list(range(2, 33, 2))`	22
bootstrap	`[True, False]`	True

Table 12: Hyperparameters of Bagging Classifier, some values are presented with Python Code.

Ensemble averaging At the end, we applied a combination of all the methods using a mean strategy over their prediction. This combination is a well-known technique called "Ensemble Averaging" and we chose it because uses the diversity of all models to reduce the variance of predictions keeping the same bias, therefore avoiding overfitting.

5.2 Methodology

We have seen the use of various machine learning models in order to learn how to classify Twitter users by distinguishing between **real users** and **bots**. In Table 13 the results are shown. Recall that the cross-validation score indicated in the table was calculated by performing a k-fold cross validation, using **5** folds ($k = 5$) taking the training set into consideration. Accuracy and F1 Score were finally calculated taking the test set into consideration.
It is important to note that other experiments were performed trying a feature selection of attributes using various methods including PCA. While these methods certainly use fewer features and thus in large datasets may be useful, these experiments did not lead to an increase in accuracy and are therefore not reported.

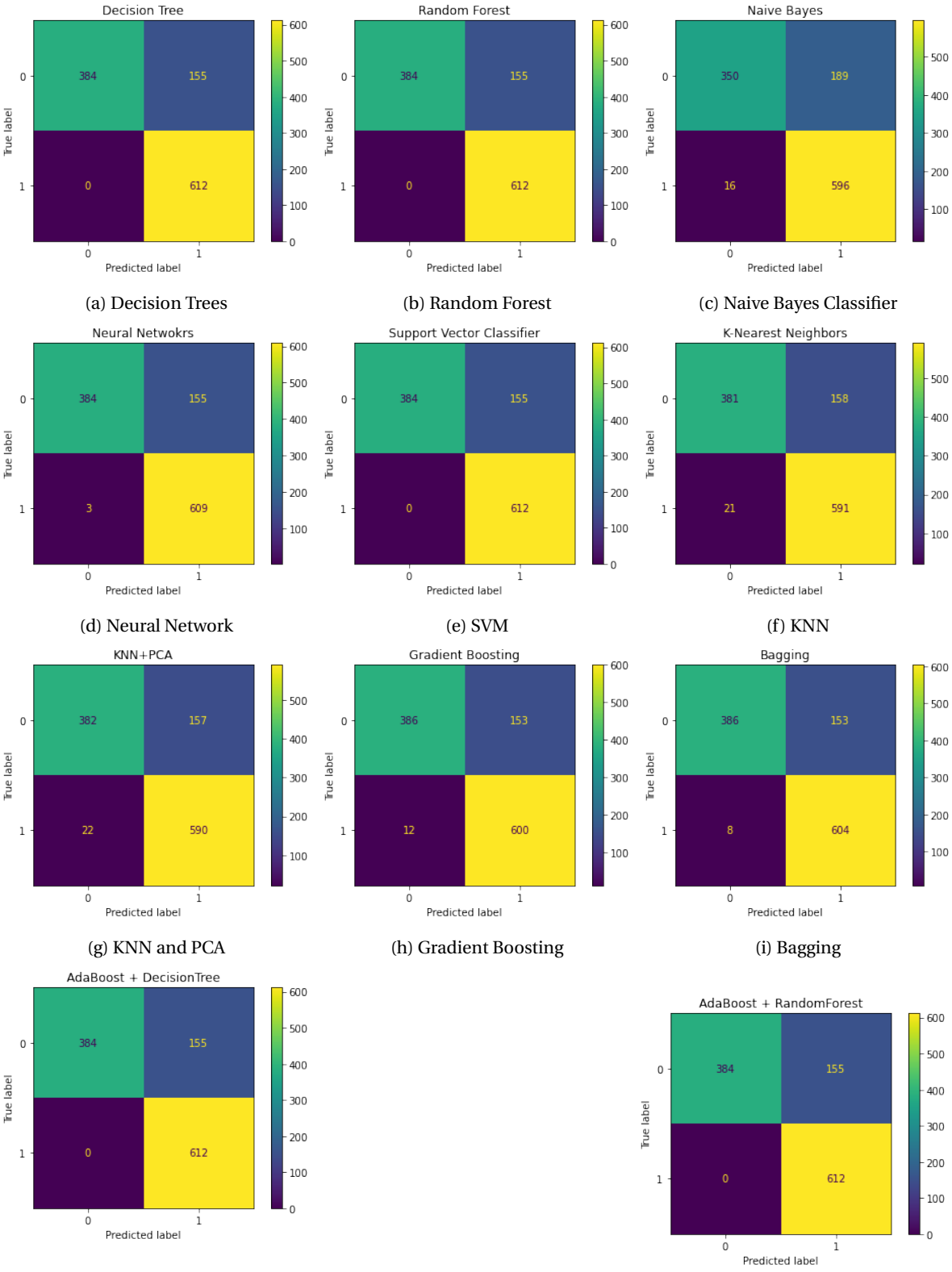
5.3 Results

As can be seen, there is no clear model that wins out over the others as at most we achieve **86.5**% in accuracy with several models. A sign that the errors made by the various models are the same according to this dataset.

Model	Cross-Validation	Accuracy	F1 Score
Decision Tree	0.860	0.865	0.861
Random Forest	0.860	0.865	0.861
Naive Bayes	0.816	0.821	0.815
Neural Networks	0.858	0.862	0.856
Support Vector Classifiers	0.859	0.865	0.861
Gradient Boosting w. Random Forest	0.851	0.856	0.853
Bagging	0.850	0.860	0.856
AdaBoost w. Decision Tree	0.860	0.865	0.861
AdaBoost w. Random Forest	0.860	0.865	0.861
K-NN	0.841	0.844	0.840
K-NN w. PCA	0.840	0.844	0.841
Ensemble averaging	-	0.865	0.861

Table 13: Results Table. For the method called ensamble, we note that there is no score for cross-validation as the average of the predictions made by all other classifiers was taken.

To compare all models we plot in Figure 21 all confusion matrices side by side. We noticed that, considering the classifiers that get the best accuracy level, the classifiers were wrong to predict users who are real in bots for about 155 examples of the test set. While none of these classifiers misidentified bots as real users, thus, we can conclude that these models can, with certainty, find the bots. A future work could be the identification of the features that lead to a certain label prediction, in this way we can do back programming and highlight the indicators that suggests a particular label, enhancing the interpretability.
To better show the results, we report the plot in Figure 22a showing the results obtained from the various methods in comparison. We also performed the calculation of the Receiver Operating Characteristic curve (ROC curve) as shown in Figure 22b. Also taking one of the best-performing models, the decision tree, we plotted the distribution of points on a 2D plane using PCA and assigned the color pink to bots and black to normal users. By comparing the plot with the actual data in Figure 22c and the predicted data in Figure 22c, we can see which points the model got wrong.



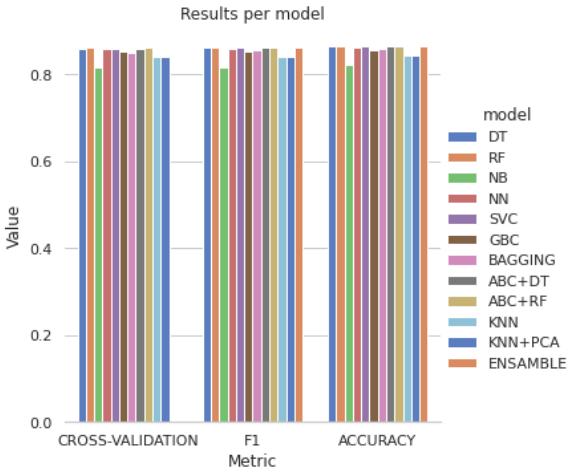
(a) Decision Trees (b) Random Forest (c) Naive Bayes Classifier

(d) Neural Network (e) SVM (f) KNN

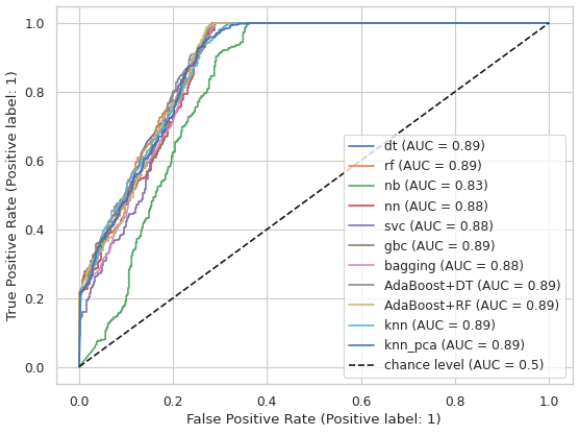
(g) KNN and PCA (h) Gradient Boosting (i) Bagging

(j) AdaBoost + DT (k) AdaBoost + RF

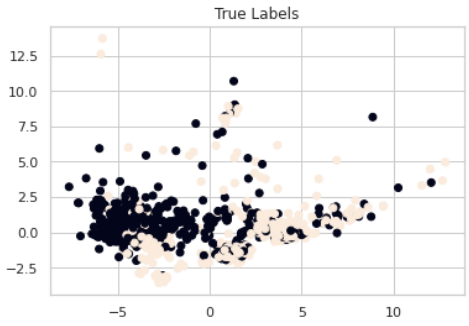
Figure 21: Confusion Matrices



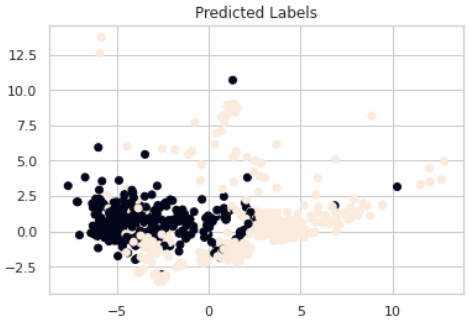
(a) Results



(b) ROC curve



(c) Real data



(d) Prediction

Figure 22: In the graphs **a** and **b**, with DT we refer to the Decision Tree model, with RF to the Random Forest, with NB to the Naive Bayes, with NN to the Neural Networks, with svc to the Support Vector Classifier, with gbc to the Gradient Boosting Classifier, with knn to the K-Nearest Neighbours.

5.4 Conclusions

In our opinion, the accuracy obtained, in all models, is strongly influenced by the presence of dirty data within the dataset. We expected better results for the Neural Network, being really flexible. Probably our expectations were not met by the fact that the architecture was too simple. Looking the confusion matrix we noticed the wrong classification of genuine users in bots, this particular task is best suited for the boosting strategies, indeed we expected that the AdaBoost (with Random Forest) solved this problem instead as we can see in figure 21k it behaved in line with the others.

Given the fact that 155 users are always misclassified, which seems to be the same at this point, it could be that they belong to the cluster identified by DBSCAN as noise and therefore the less dense part. In fact a lack of enough users of that type in the training phase would have led to an undertraining and therefore to their poor classification. Surely the fact that all the models have achieved excellent results means that the task was not very complicated, with more data you could get even more satisfactory results.

The work done for the predictive analysis taski contained in the files within the project folder by the name *DM_0_TASK3.ipynb*

6 Time Series Analysis

In this section we will discuss the thoughts and processes encountered during the timeseries extraction and classification. The timeseries was constructed using the *SuccessScore* for each day of 2019, we define it in the Equation 1.

$$SuccessScore = \frac{AcceptanceScore}{DiffusionScore + 0.1} \tag{1}$$

where:

$$AcceptanceScore = retweet\_count + reply\_count + favorite\_count \tag{2}$$

$$DiffusionScore = num\_hashtag + num\_mentions + num\_urls \quad (3)$$

First of all, in the Sections 6.1 and 6.2 we will tell how we constructed the dataset. In Section 6.3 we will perform the analysis using clustering techniques and in Section 6.4 we will talk about how we performed the classification using shapelets and bringing also an extra method i.e. classification using Recurrent Neural Networks, finally, in Section 6.5 we will bring in our conclusions.

6.1 Setting the stage

Our analysis necessarily requires the construction of the dataset we are going to work on. To do this we wrote a function, `success_score(user, day)`, which calculates the success score given individual user and a specific day in 2019. At this point iterating over the days of 2019, it is possible to extract a timeseries given a user, this is exactly what we do in the function `extract_success_score_timeseries(user)`. Let us now see an example of a timeseries generated by this function; the selected user has the id 576148031.

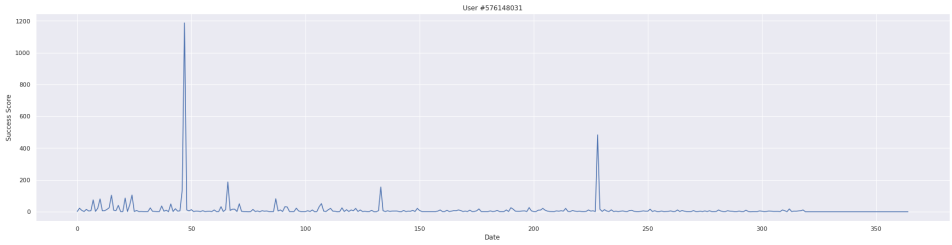


Figure 23: Row timeseries for user 576148031

At this point, in order to get the complete dataset we have to run the function that extracts the time series for each user; this operation takes about 2 hours, so to avoid computing it several times we saved this data frame in a file. It is still possible to create the data frame by setting the boolean flag `generate_time_series` to `True`. If for a user, for a specific day, we have multiple SuccessScore values, we average these values.

6.2 Preprocessing

Now that we have obtained the dataset, we have performed some cleaning operations in order to get better results in the clustering and classification tasks. First of all, we removed all users who did not tweet in 2019, specifically we removed 4610 users. After that, we reduced the noise in the timeseries by applying a sliding window using the function `ts.rolling(window=40).mean()`. Figure 24 shows how it works on an example time series. The value of the window size was selected empirically choosing the one that maximizes the silhouette score in the clustering algorithms.

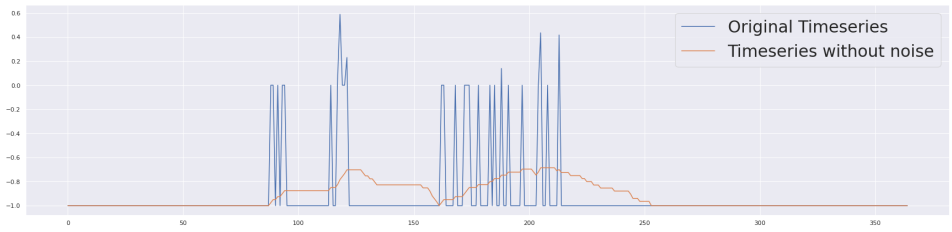


Figure 24: User time series 576148031 without noise

After removing the noise, we normalized the dataset by applying a *MinMaxScaler* through the class `TimeSeriesScalerMinMax(value_range=(0, 1.))`. This ensures that all data are on the same scale, making comparisons between timeseries possible. Figure 25 shows how the scaler works on the timeseries shown above.

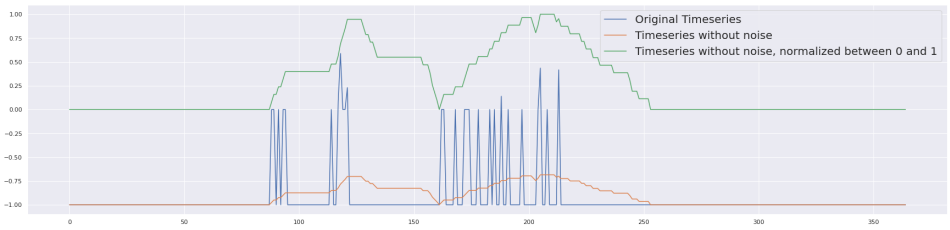


Figure 25: Row time series in orange of the user 576148031 without noise and scaled

6.3 Clustering

Clustering was performed in two different methodologies: Shape-based clustering and Statistical feature-based clustering:

Shape-based clustering Shape-based clustering is a method of clustering data points based on their shape. We used the class `TimeSeriesKMeans`. The number of clusters was found using the same method (and library) used in Task 2, i.e., the $KneeMethod$. The result is thus 5 clusters.
Figure 26 shows what the centroids are for each cluster using this methodology. Note that in light of the fact that we are working with timeseries, therefore the centroids themselves are timeseries. The silhouette score obtained for this method is 0.44.

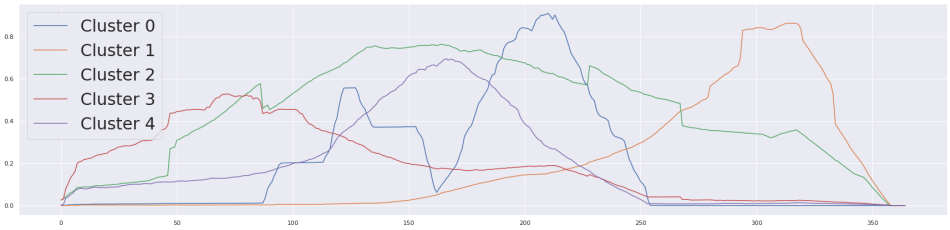


Figure 26: Centroids using shape-based clustering.

The metric used in this methodology is Euclidean so shift of the time series are not taken into consideration.

Statistical feature-based clustering Statistical feature-based clustering is a method of clustering data points based on their statistical features, such as their mean, median, variance, and other summary statistics. So first we proceeded to transform each timeseries into a vector of 13 values where each value corresponds to mean of the timeseries, standard deviation, variance, median, 10th percentile, 25th percentile, 50th percentile, 75th percentile, intertquartile range, covariance, skewness and kurtosis. Then, since we have a dataset of vector in 13 dimensions, we applied a traditional clustering method (Kmeans). The number of clusters was found using the same way as before (the Kneed method), and the result was 4.

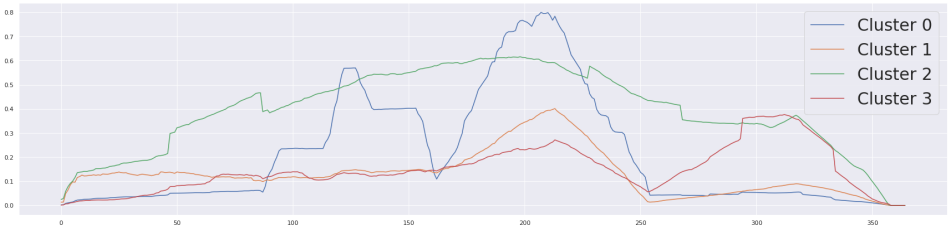


Figure 27: Centroids using Statistical feature-based clustering

Figure 27 shows us the centroids. The silhouette score obtained for this method is 0.46. Finally, we also provide a graph in Figure 28 showing the number of bots and real users per cluster found.

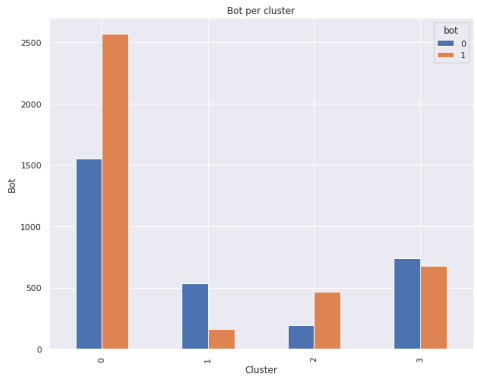


Figure 28: Distribution of bots and actual users per cluster using K-Means on timeseries.

We can see that the two clustering methods behave similarly in that it is possible to find centroids very close to each other. In fact, as is possible in figure 26 the centroid identified by the green color line can also be found in figure 27, as well for the rising summer trend in blue. Furthermore, we notice the presence of users with a fairly high success score consistently, we associate this behavior with users whose actions always have good feedback from the community (e.g., influencers). Another recurring pattern is due to users who have success score spikes only at certain times of the year, we associate this behavior with some specific tweet that went viral randomly (and not because these users have a strong fan base). In general there is a trend in the usage of Twitter during the summer, we do not know if it is due to a particular event or so, but it is present in 3 of 5 clusters.

6.4 Classification

In this section, we will explain the methods we used in classifying the users on the timeseries constructed with the SuccessScore. The aim is to predict a user's label between bots and real users as the previous classification task.

Classification using shapelets Shapelets are short, discriminative subsequences of a time series that are used for time series classification and clustering. So, we used them to classify timeseries. In particular, we used (*tslearn's LearningShapelets*[3]) class. The parameters on which the classification task is based (e.g., the length of the shapelets or the number of epochs) were found by doing model selection using a random search and they are reported in Table 14, where we also provide the chosen value. By "0.1" in *shapelet_length* we mean that 10% of the length of the timeseries was used as the length for shapelets.

The shapelet model uses a TensorFlow neural network under the hood, so *learning_rate, epochs* and *weight_regularizers* are parameters of a neural network model. The accuracy obtained by this methodology is **77.89%** on the test set. Finally, Figure 29 shows which shapelets were found.

Parameter	Values	Winner
shapelet_length	[0.05, 0.1, 0.15, 0.2, 0.25]	0.25
learning_rate	[0.0001, 0.001, 0.01, 0.1]	0.1
epochs	[20, 30, 40, 50, 70]	20
weight_regularizer	[0.0, 0.1, 0.001, 0.0001]	0.0

Table 14: Hyperparameters of the shapelet classification model.

[3]https://pyts.readthedocs.io/en/stable/generated/pyts.classification.LearningShapelets.html

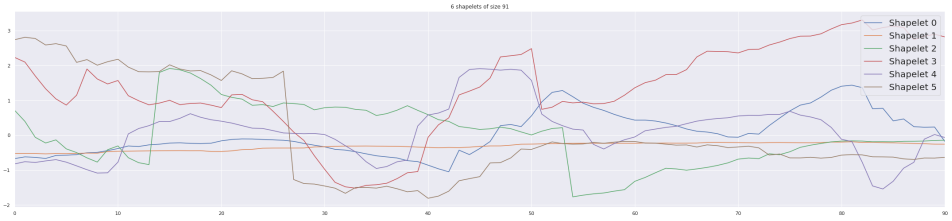


Figure 29: Shapelets found

As an example, in the figure 30 we show a random timeseries on which we show the shapelets that characterize it. The selected shapelets is the normalized version of the blue shapelet in the figure 29.

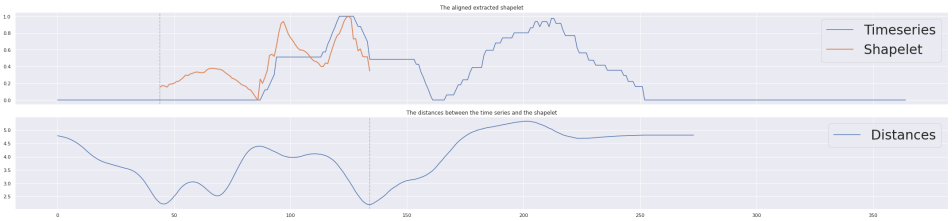


Figure 30: Alignment of a shapelet with a specific timeseries

Classification using GRUs GRUs are a type of recurrent neural network used for modeling sequential data. They function similarly to other types of recurrent layers, such as LSTMs and RNNs, in that they take in a sequence of input data and produce a corresponding sequence of output data. However, GRUs differ in their use of "gates" which control the flow of information within the layer's internal memory. These gates, modeled using sigmoid activation functions, are trained using backpropagation and are beneficial because they are computationally efficient and easy to train.
The selected architecture involves two GRU layers and a final dense layer. The latter uses the sigmoid function as the activation function since we are performing a classification task. The two GRU layers are adjusted through the L2 regularization. The values of the units and the two regularizations were found by model selection performed through the Hyperband class of Tensorflow. The accuracy obtained is **74.6%**.

6.5 Conclusion

In this section, we have seen how to generate, manage and clusterize and classify timeseries. Clustering was carried out in two separate ways: first using the shapes of the timeseries and finding **5** different clusters, and then using feature vectors representing the timeseries, finding **4** different clusters. Two separate methods have been proposed for classification, one based on shapelets and one based on recurrent neural networks. In both cases, an initial model selection phase was performed using a random search to select the best values for the hyperparameters. Although Recurrent Neural Networks are a more flexible and powerful tool than shapelets, they have not shown better results than using shapelets. A better selection of parameter values could lead to better accuracy results. As a future work we plan to make use of Dynamic Time Warping metrics with more powerful machines and experiment the approximation methods. The analysis performed using timeseries can be viewed in the notebook called *DM_0_TASK4.ipynb* contained in the files within the project folder.

7 How to run the experiments

The experiments were performed using Google Colaboratory, an online platform that allows Python code to be executed as notebooks thanks to Jupyter. The notebooks contain the necessary commands to install the libraries used in the experiments. The preprocessed files of the datasets were saved on the Google Drive platform and are downloaded when needed.