Data Mining Report

Pasquale Esposito, 649153
Innocenzo Fulginiti, 594051
Luca Moroni, 635966

a.y. 2022/2023

1 Data Understanding

In this section we describe the data given for the project, and the operations needed to handle that data and to clean it.
We had to handle two datasets: the first one is the **users.csv**, which contains data relative to some Twitter's users and the second one is the **tweets.csv**, that contains data relative to a lot of tweets scraped through Twitter APIs.
Now we expose the steps needed to understand the data presented in the two stated datasets.

1.1 Dataset users.csv analysis

The users' data is composed by the following features:

- **user_id** (Int): user's identifier;
- **name** (String): the name of the user;
- **lang** (String): the language of the profile;
- **bot** (Int): an indicator that tells if the user is a bot or not;
- **created_at** (Timestamp): the timestamp of the creation of the profile;
- **statuses_count** (Int): the number of tweets done by the user at the moment of the scraping.

Using the Pandas method `info()` we can see the inferred type of the features and the number of *null* elements as shown in Figure 1. First of all we saw that the number of different user IDs was exactly the number of the samples, so we did not have duplicated user IDs.

1.1.1 Name

The `name` feature presents a single *null* element (Figure 2) and that element was substituted with a default value `"No Name"`.

```
Int64Index: 11508 entries, 2353593986 to 933183398
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   name            11507 non-null  object
 1   lang            11508 non-null  object
 2   bot             11508 non-null  int64
 3   created_at      11508 non-null  object
 4   statuses_count  11109 non-null  float64
```

Figure 1: For each feature is shown the type and the number of *non-null* elements.

id	name	lang	bot	created_at	statuses_count
2166124159	NaN	en	0	2018-11-02 06:39:14	6566.0

Figure 2: User whose name is *null*

1.1.2 Lang

For the feature `lang` we saw that there were no *null* elements, but with the method `unique()` we noticed several inconsistencies and wrong values. We chose to replace all the languages with only their prefixes (e.g. `"en-GB"` to `"en"`), then we substituted the wrong values `"Select Language..."` to `"en"` since the text contained in that field was in English, so we assumed that the language of the registration form was set in English, and `"xx"` again with the value `"en"` because we saw that this user did not post any tweets, so we decided to assign the most frequent language (Figure 3) to that field.

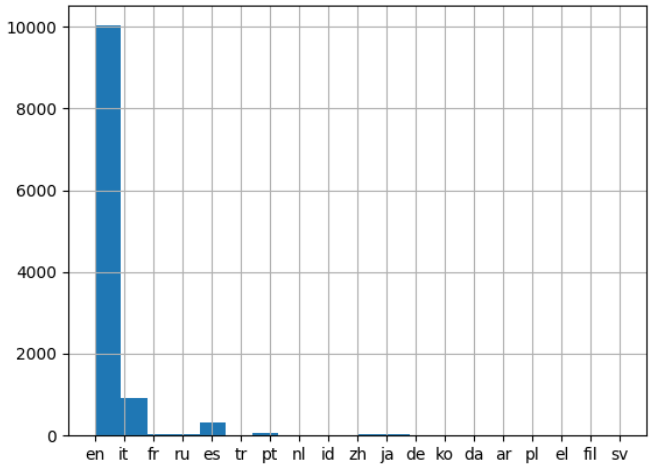


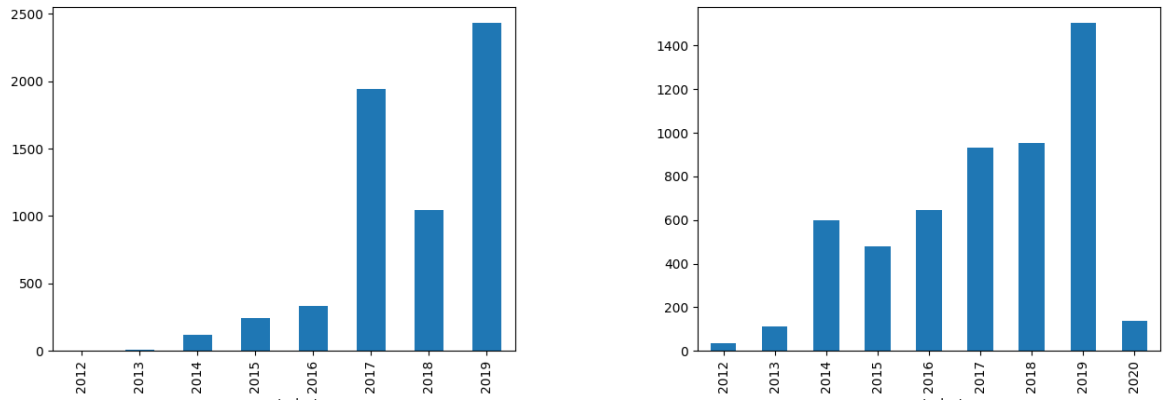
Figure 3: Histogram for the feature **lang** after cleaning

1.1.3 Bot

The `bot` feature did not have *null* values and all of them were semantically correct. Furthermore, we almost have a similar distribution for bot and non-bot users (6116 bots and 5392 normal users).

1.1.4 Created_at

About the feature `created_at`, we checked the syntax of the samples and we did not find any wrong data format. Subsequently, we checked if the oldest data was antecedent to the creation of Twitter and if the most recent one was in the future, and they were not. In Figure 4, we plotted the distribution of the number of new users, bot and non-bot, created for each year. We can see



(a) Bot users

(b) Non-bot users

Figure 4: Distribution of the number of users created for each year

that the histogram for bot users has peaks in more recent years than the histogram for non-bot users.

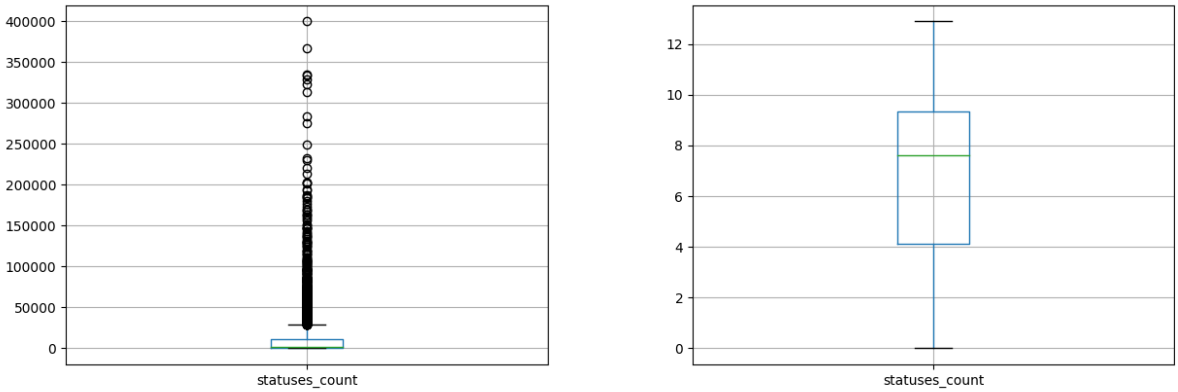
1.1.5 Statuses_count

About the `statuses_count` field, first of all we noticed that it was actually an integer value and not a float so we cast it into integer (also because there were no decimal values in that column). Next, we decided to do two separate analyses, one for normal users and the other for the bot users; we noticed that, in both cases, the distribution was a power law.
For the normal users, once we plotted the data in log scale, we obtained no outliers anymore, as you can see in Figure 5. Furthermore, there were 399 users who had a *null* value for this field and we substituted them with the median relative to normal users.
About the bot users (Figure 6), instead, even after we plotted the data in log scale we kept having some outliers and we decided to remove them fixing the two quantiles to the 20% and 80%. After that, we calculated the median using only the values that are not outliers and we substituted the values of the outliers with such median. Unlike the normal users, we had no *null* values to fix.

1.2 Dataset tweets.csv analysis

The tweets' data is composed by the following features:

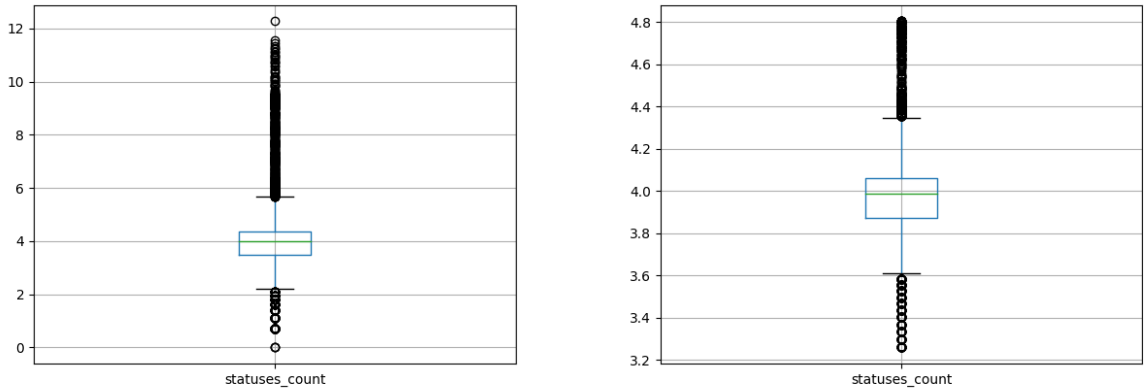
- **id** (Int): tweet's identifier;
- **user_id** (Int): identifier of the user who wrote the tweet;
- **retweet_count** (Int): number of retweets of the tweet;
- **reply_count** (Int): number of replies of the tweet;



(a) Original data

(b) Log scaled data

Figure 5: Boxplots for the field **statuses_count** for normal users



(a) Before cleaning

(b) After cleaning

Figure 6: Boxplots for the field **statuses_count** for bot users in log scale

- **favorite_count** (Int): number of likes received by the tweet;
- **num_hashtags** (Int): number of hashtags used in the text of the tweet;
- **num_urls** (Int): number of URLs in the text of the tweet;
- **num_mentions** (Int): number of mentions in the text of the tweet;
- **created_at** (Timestamp): when the tweet was created;
- **text** (String): the text of the tweet.

We started by deleting the duplicated rows (**i.e.** the rows that contained the same values for every field) that there were in the table.

1.2.1 Id

We started cleaning data since we noticed there were some alphanumeric IDs. Those records did not contain many *null* values so we decided not to drop them. In Figure 7 is shown the number

and the ratio of the *null* values with alphanumeric indexes.
Next, we saw there were some records with the same ID and we realized that almost all the fields were *null* (some fields had even a percentage higher than 80%, **i.e.** `num_hashtag`), as shown in Figure 8, so we dropped them all. Finally, we obtained a more consistent dataset with less *null* values to handle (Figure 9).

```
Number of NaN tweets with non-numerical ID:
 user_id            28912
retweet_count      94160
reply_count       151602
favorite_count    151163
num_hashtags      252578
num_urls          151486
num_mentions      203979
created_at             0
text              132770
dtype: int64
Total number of tweets with non-numerical ID:  433860
```

(a) Number of *null* elements per field

```
Ratio of the NaN elements per column:
 user_id           0.066639
retweet_count     0.217029
reply_count       0.349426
favorite_count    0.348414
num_hashtags      0.582165
num_urls          0.349159
num_mentions      0.470149
created_at        0.000000
text              0.306020
dtype: float64
Ratio of the NaN elements: 0.299
```

(b) Ratio of *null* elements per field

Figure 7: *Null* values for the tweets' fields with non-numerical indexes

```
Number of NaN tweets in the non unique ID:
 user_id            3847
retweet_count     25452
reply_count       38934
favorite_count    38805
num_hashtags      52393
num_urls          38820
num_mentions      47294
created_at            0
text              37501
dtype: int64
Total number of tweets in the non unique ID:  58060
```

(a) Number of *null* elements per field

```
Ratio of the NaN elements per column:
 user_id           0.066259
retweet_count     0.438374
reply_count       0.670582
favorite_count    0.668360
num_hashtags      0.902394
num_urls          0.668619
num_mentions      0.814571
created_at        0.000000
text              0.645901
dtype: float64
Ratio of the NaN elements:  0.542
```

(b) Ratio of *null* elements per field

Figure 8: *Null* values for the tweets' fields with duplicated indexes

```
Number of NaN elements per field:
 user_id            213429
retweet_count      411682
reply_count        608939
favorite_count     608736
num_hashtags      1005115
num_urls           609791
num_mentions       806855
created_at              0
text               492530
dtype: int64
Total number of tweets:  11654033
```

(a) Number of *null* elements per field

```
Ratio of the NaN elements per field:
 user_id           0.018314
retweet_count     0.035325
reply_count       0.052251
favorite_count    0.052234
num_hashtags      0.086246
num_urls          0.052324
num_mentions      0.069234
created_at        0.000000
text              0.042263
dtype: float64
Mean of the ratio of the NaN elements:  0.045
```

(b) Ratio of *null* elements per field

Figure 9: *Null* values for the tweets' fields

1.2.2 User_id

The `user_id` feature was exploited to join the two tables, since our aim was to compute a profiling of the users. What we did was to delete all the tweets that were posted by users whose `user_id` was not contained in the `users.csv` dataset.

1.2.3 Numerical features

We now report the study we did on the numerical features. We did the same operations for all the numerical features, since they all had more or less the same distribution (**i.e.** positive power law). For each feature, first of all we only took the numeric fields; secondly, we cast them into integers. Next, we set all the alphanumeric, negative and infinite elements to *null* and we rounded the float numbers. Then we did the following reasoning: in our opinion, a field like the `num_urls`, for example, cannot ever be higher than an upper bound u, so we set to *null* all the fields that were higher than u (this is not the case for all the features). Subsequently, we scaled the not *null* values using the function `np.log()`. Later, for the outliers detection, we first tried to remove the outliers with the boxplot but then we found out that applying this method we would have set a huge number of fields to zero, so we chose a different approach: we set a different quantile and all the values over that threshold were recognized as outliers and set to *null*. That threshold was chosen in order to give us an acceptable and non-sparse distribution, without removing too many elements. Finally, we set all the *null* fields we obtained from all this analysis to the median of the non-outlier values.
We now show all the parameters we set for each numeric feature:

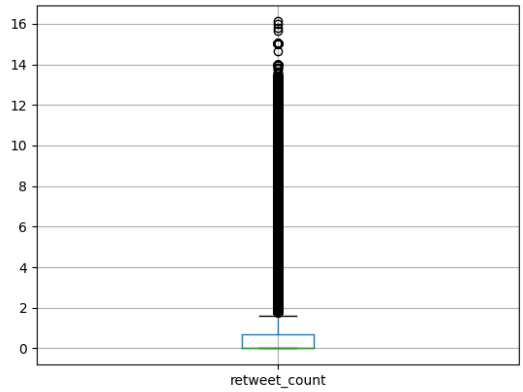
- `retweet_count`: the chosen quantile is 0.995 (Figure 10);
- `reply_count`: the chosen quantile is 0.99995 and the upper bound is 1e6 (Figure 11);
- `favourite_count`: the chosen quantile is 0.999 (Figure 12);
- `num_hashtags`: the chosen quantile is 0.9999 and the upper bound is 50 (Figure 13);
- `num_urls`: the chosen quantile is 0.999995 and the upper bound is 30 (Figure 14);
- `num_mentions`: the chosen quantile is 0.99995 and the upper bound is 50 (Figure 15).

1.2.4 Created_at

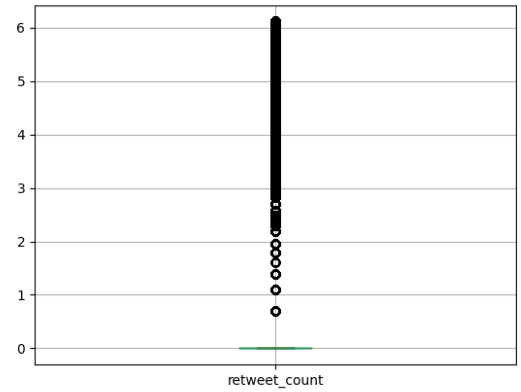
In the `created_at` feature there are no *null* items and moreover, all the dates have the correct format. We noticed that some of them were not semantically correct, so we chose to clean up this feature. We did a deep control, for each tweet we saw if the `created_at` value of the tweets was less than the `created_at` of the user who posted such tweet, performing a join of the tables, and we substituted those dates with a default one. We also chose an upper bound "2022-01-01 00:00:00" after which we set a default date, "1800-01-01 00:00:00".

1.2.5 Text

In the `text` feature, there were some *null* elements that we substituted with the empty string `""`.

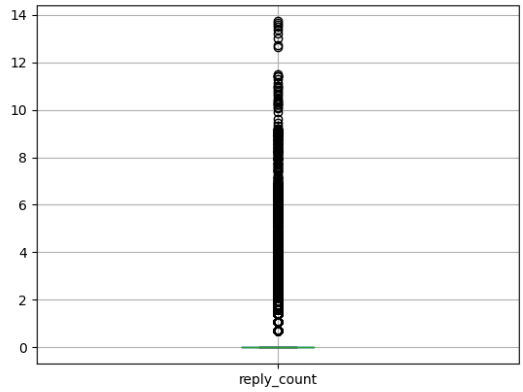


(a) Boxplot before the outlier removal

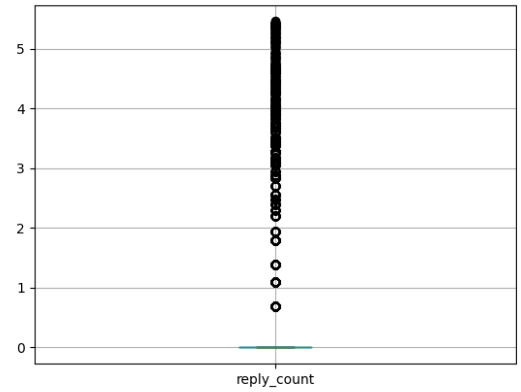


(b) Boxplot after the outlier removal

Figure 10: Outliers of the feature **retweet_count** in log scale

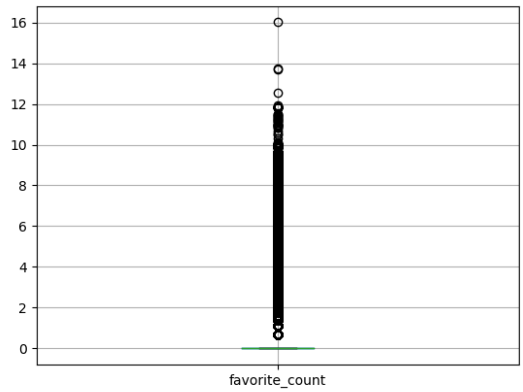


(a) Boxplot before the outlier removal

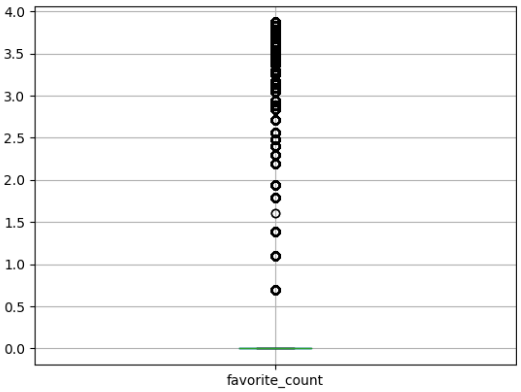


(b) Boxplot after the outlier removal

Figure 11: Outliers of the feature **reply_count** in log scale

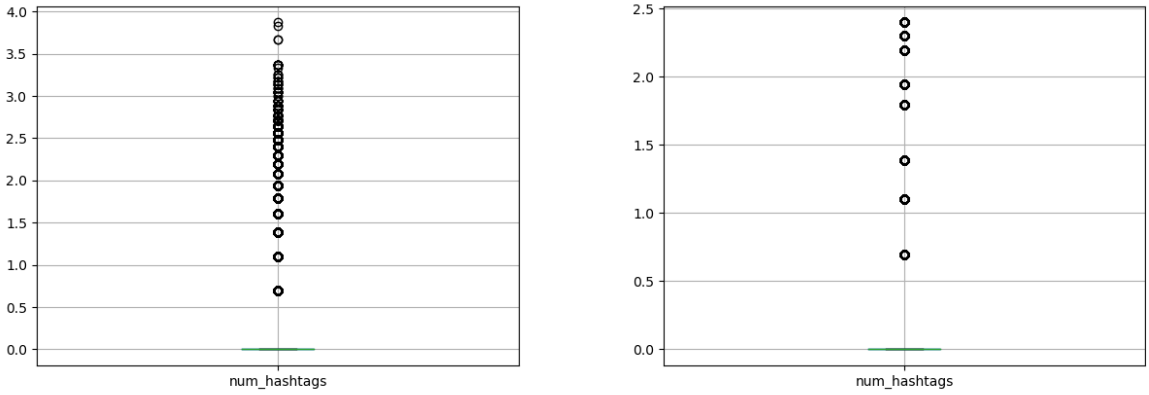


(a) Boxplot before the outlier removal



(b) Boxplot after the outlier removal

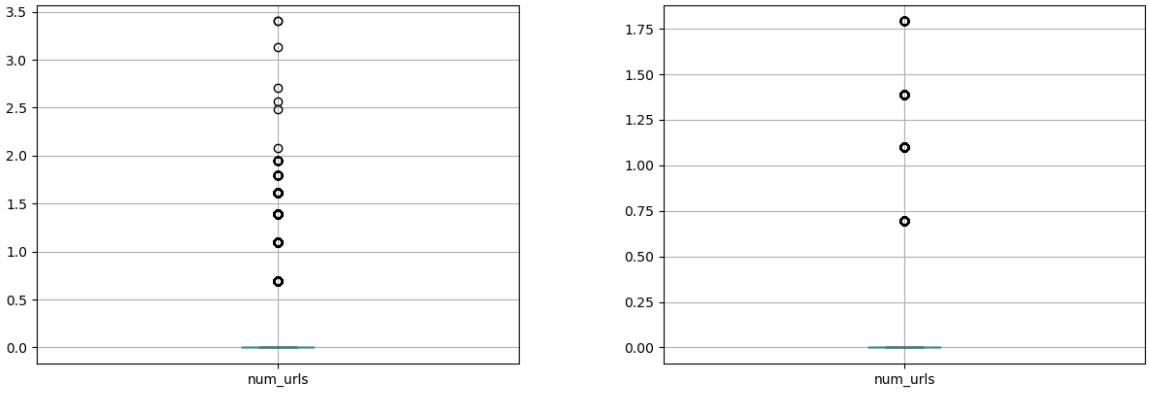
Figure 12: Outliers of the feature **favourite_count** in log scale



(a) Boxplot before the outlier removal

(b) Boxplot after the outlier removal

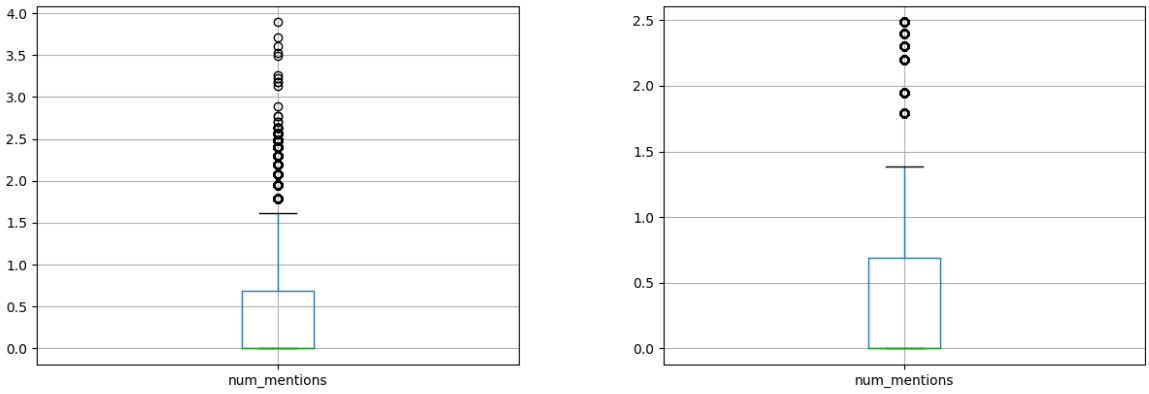
Figure 13: Outliers of the feature **num_hashtags** in log scale



(a) Boxplot before the outlier removal

(b) Boxplot after the outlier removal

Figure 14: Outliers of the feature **num_urls** in log scale



(a) Boxplot before the outlier removal

(b) Boxplot after the outlier removal

Figure 15: Outliers of the feature **num_mentions** in log scale

1.2.6 Numerical Correlation

We saw that there were no notable correlations among the numerical features (Figure 16).

	retweet_count	reply_count	favorite_count	num_hashtags	num_urls	num_mentions
retweet_count	1.000000	0.000764	-0.026854	0.081767	0.005785	0.156503
reply_count	0.000764	1.000000	0.009391	-0.002671	-0.005381	0.008867
favorite_count	-0.026854	0.009391	1.000000	-0.004837	-0.074517	-0.062508
num_hashtags	0.081767	-0.002671	-0.004837	1.000000	0.099902	0.084343
num_urls	0.005785	-0.005381	-0.074517	0.099902	1.000000	-0.052074
num_mentions	0.156503	0.008867	-0.062508	0.084343	-0.052074	1.000000

Figure 16: Correlation matrix of the numerical features of **tweets.csv** dataset

2 Data Preparation

In this section we describe the new features we selected to define a user profile. We added the following features:

- **avg_length**: average length of the tweets of a user;
- **avg_special_chars**: average number of special characters of the tweets of a user, that is the characters not in [A-Z a-z 0-9 _];
- **urls_ratio**: ratio between the number of URLs (in the tweets) and the number of tweets of a user;
- **mentions_ratio**: ratio between the number of mentions and the number of tweets of a user;
- **hashtags_ratio**: ratio between the number of hashtags (in the tweets) and the number of tweets of a user;
- **reply_count_mean**: mean of the replies of the tweets for a user;
- **reply_count_std**: standard deviation of the replies of the tweets for a user;
- **reply_count_entropy**: entropy of the replies of the tweets for a user;
- **favorite_count_mean**: mean of the number of likes received by the tweets of a user;
- **favorite_count_std**: standard deviation of the number of likes received by the tweets of a user;
- **favorite_count_entropy**: entropy of the number of likes received by the tweets of a user;
- **retweet_count_mean**: mean of the number of retweets received by tweets of a user;
- **retweet_count_std**: standard deviation of the number of retweets received by tweets of a user;
- **retweet_count_entropy**: entropy of the number of retweets received by tweets of a user.

2.1 Data Cleaning

Now we describe the cleaning of the generated indicators for user profiling that we performed in the notebook **DataUnderstandingProfiling.ipynb**.

We did an outlier detection for all the features listed above one by one. If a feature had a distribution that was a power law, first we put it in logarithmic scale, then if there were outliers we removed them. On the other hand, if the distribution of a feature was not a power law, we removed the outliers without putting the distribution on logarithmic scale. To remove the outliers, we applied the same approach we used to clean the numerical features in the **tweets.csv** dataset, that is, first we tried to remove them with the boxplot, but when many values were removed, we chose to eliminate the outliers setting two different quantiles, one as a lower threshold and one as an upper threshold, used to consider a value as an outlier.

After the data cleaning phase, we looked for correlated variables. We chose to consider two variables correlated, if their correlation value was greater than 0.8 and we found that `retweet_count_entropy` and `favorite_count_entropy` had a correlation value equal to 0.89, so we decided to remove the variable `retweet_count_entropy`.

3 Clustering Analysis

In this section we expose our results about clustering. We decided to apply four clustering methods to the user profile, defined in the previous section. As preprocessing, before applying any clustering algorithm, we chose to apply the log scale to the features that have a skewed distribution, and a normalization to the entire dataset using a standard scaler.

3.1 K-means

For the K-means clustering method we needed to choose the K parameter; we tried some K values and chose the proper one with respect to the plot of **SSE** (Sum of Squared Errors), **Separation** (Davies-Bouldin score) and **Silhouette** (Figure 17). We used the implementation of the *scikit-learn*[2] library, fixing `n_init`=10 and `max_iter`=100. Given those curves we chose a K that gave us the minimum value of Separation, a high value of Silhouette and possibly a low value of SSE, so we chose to set $K = 3$. In Figure 18 is shown the PCA of K-means.

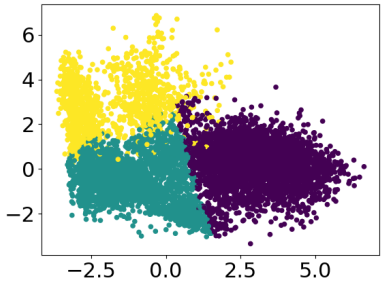
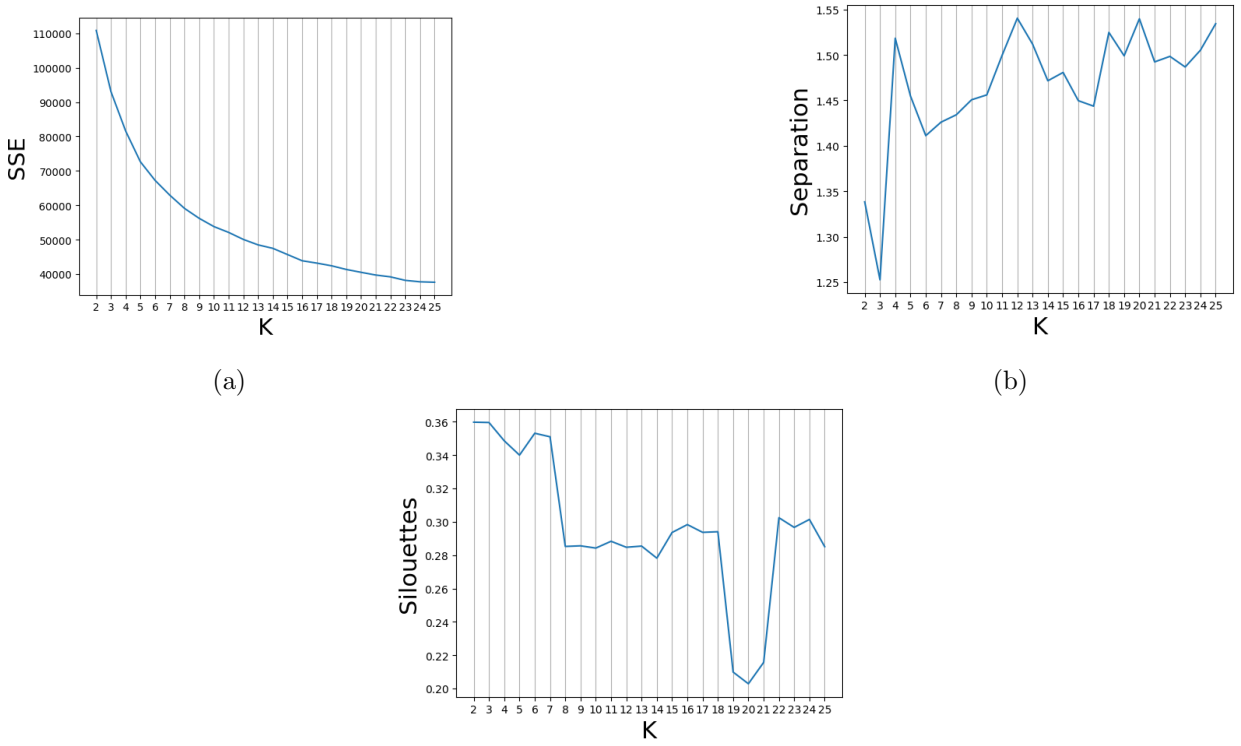


Figure 18: Representation of K-means' clusters using PCA reduced dimensions



(a) (b) (c)

Figure 17: Curve of the SSE varying K (a), curve of the Separation varying K (b), Curve of the Silhouette varying K (c), for K-means clustering model

3.1.1 Cluster Analysis

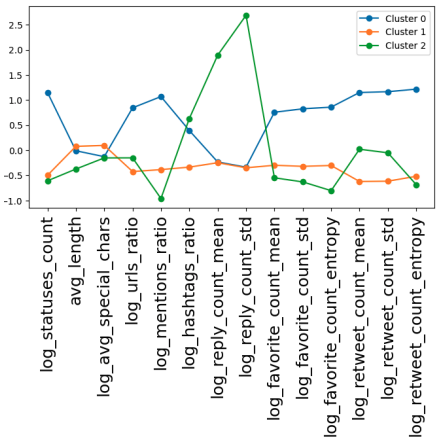
In Figure 19 we can see the parallel and radar plots of the clusters' centers found by the algorithm. In Figure 20 we can see the plots of the bot distribution for each cluster. We can notice that in the first cluster there is a majority of normal users, while the second and third ones have more bot users. This leads us to think that those clusters geometrically represent the differentiation between bot and non-bot users.

3.1.2 Features distributions

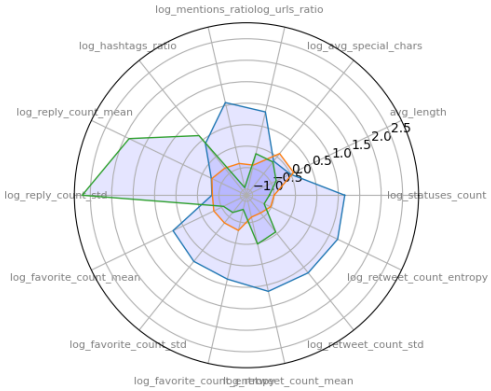
In Figure 21a is shown the total distribution of the features of our dataset, while in the other figures we can compare the distribution of them for each cluster. We can see that in cluster 0 (the one with more non-bots), in Figure 21b, the features are less skewed with respect to the distributions of the cluster 1 (the one with more bots), in Figure 21c. Moreover, in the cluster 0 the features are more normal distributed, even with respect to the distributions of all dataset.

3.2 Hierarchical clustering

The hierarchical clustering is composed by two phases: the first phase consists of analyzing the dendrograms (Figure 22) obtained using different linkage algorithms (single, average, complete, ward) and two different metrics (euclidean and cosine) to decide the best number of clusters to look for with the `AgglomerativeClustering()` function, which determines the beginning of the second phase. During the second phase, instead, we try to look for the best combination of the linkage and metrics, using also the Silhouette and Davies-Bouldin scores. Given the dendrograms



(a) Parallel plot of the clusters' centers



(b) Radar plot of the clusters' centers

Figure 19: K-means analysis

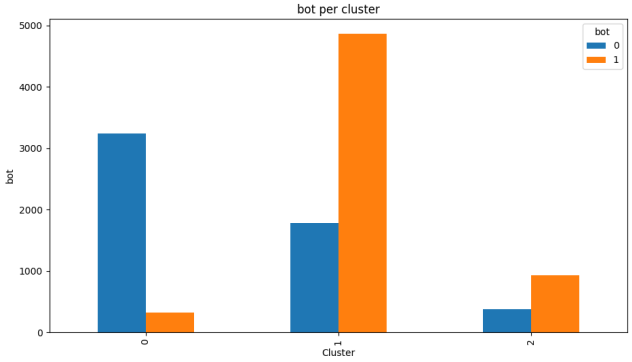


Figure 20: Bot distribution for each cluster, for K-means

and the scores of Silhouette and Separation, we chose the **Euclidean** as metric and **Ward** as aggregator, choosing 3 clusters. In Figure 23 is shown its PCA.

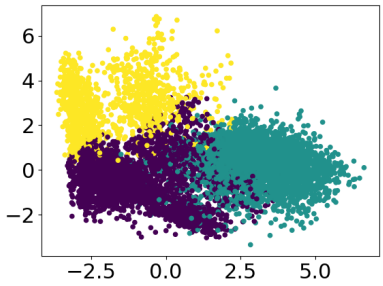
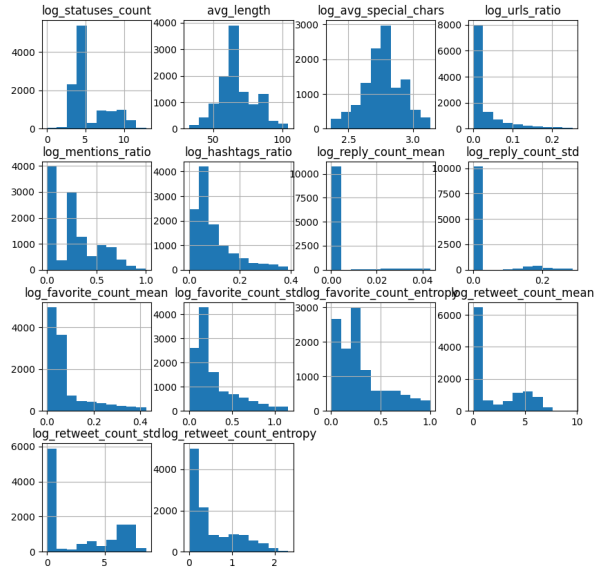


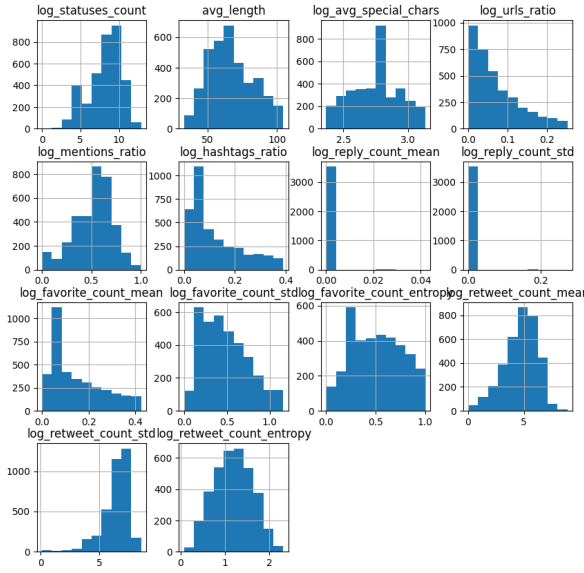
Figure 23: Representation of hierarchical' clusters using PCA reduced dimensions

3.2.1 Cluster Analysis

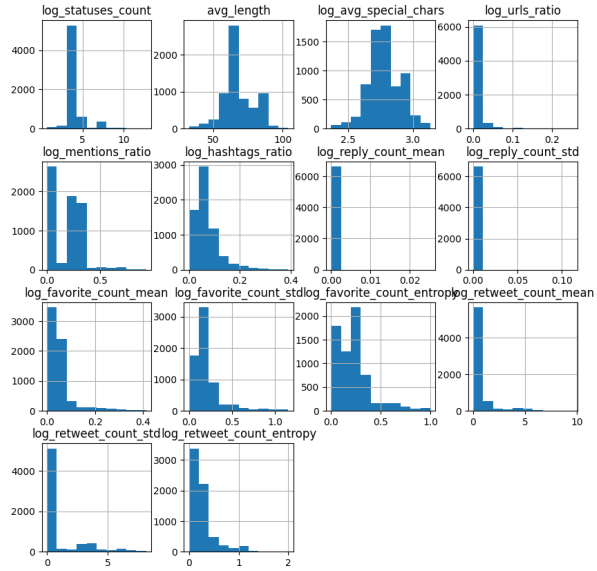
First of all we can see in Figure 24 the parallel and radar plot of the clusters' centers found by the algorithm.



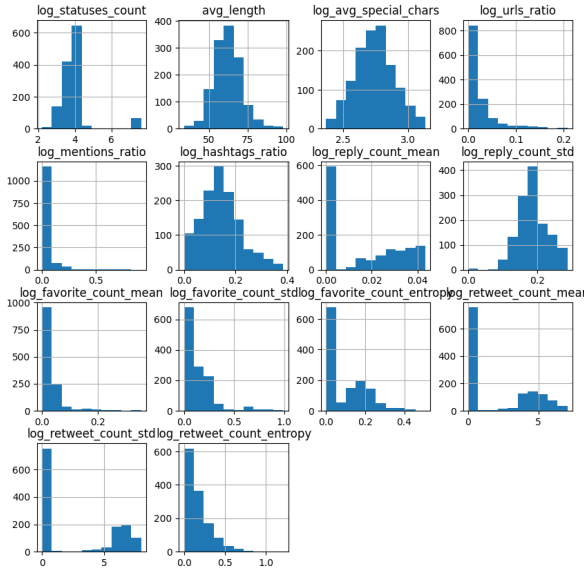
(a) Histograms of all dataset



(b) Histograms of cluster 0

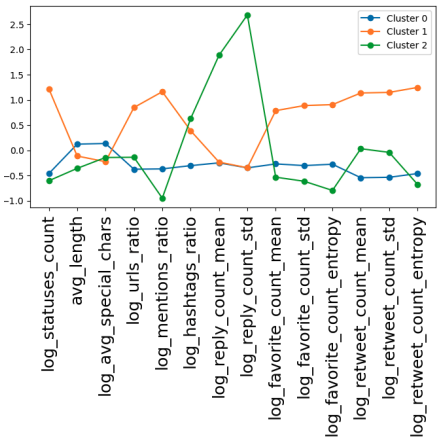


(c) Histograms of cluster 1

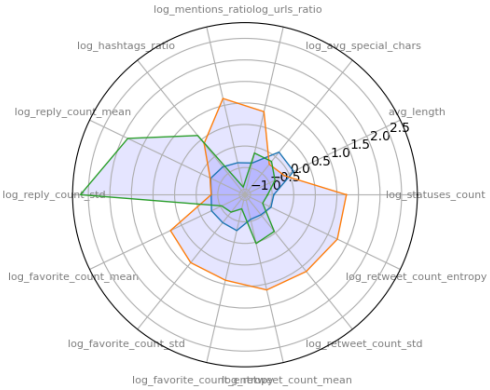


(d) Histograms of cluster 2

Figure 21: Histograms of the features of the overall dataset and of the three clusters, K-means

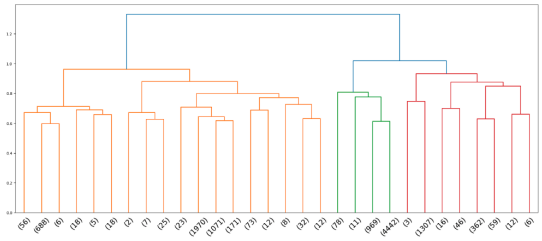


(a) Parallel plot of the clusters' centers

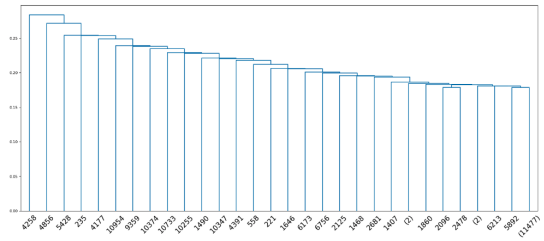


(b) Radar plot of the centers' clusters

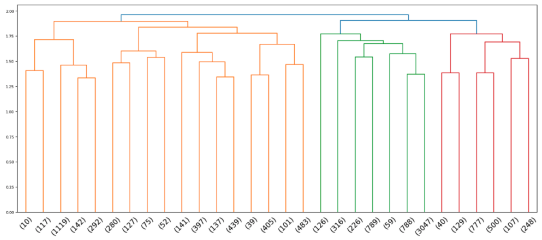
Figure 24: Hierarchical clustering analysis



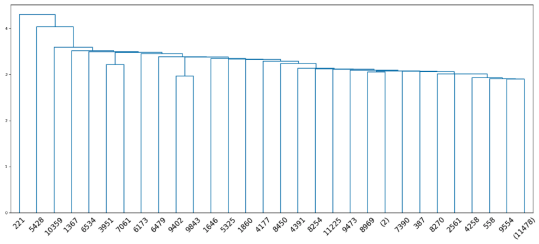
(a) Cosine - Average



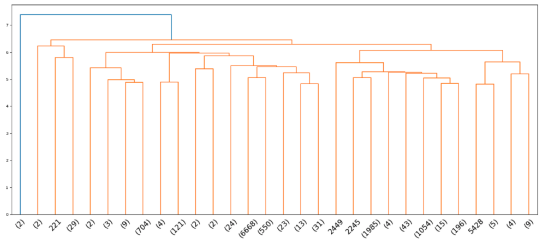
(b) Cosine - Single



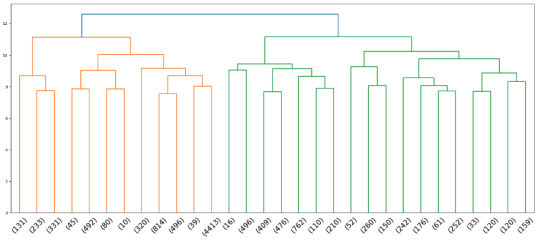
(c) Cosine - Complete



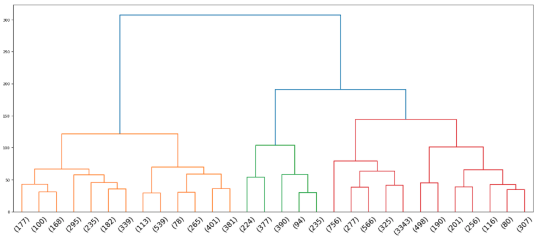
(d) Euclidean - Single



(e) Euclidean - Average



(f) Euclidean - Complete



(g) Euclidean - Ward

Figure 22: Dendrograms of hierarchical clustering

Now we can see the plots of the distribution of the bot users inside the clusters (Figure 25). We can notice that, for the bot distribution, there is the second and the third cluster that have majority of bot; on the other hand, the first has a majority of non-bot users. This leads us to think that those clusters geometrically represent the differentiation between bot and non-bot users.

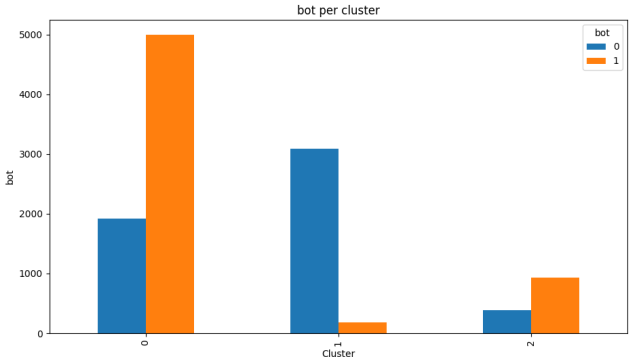


Figure 25: Bot distribution given clusters, for hierarchical clustering

3.3 DBSCAN

The DBSCAN clustering is based on two fundamental parameters: **eps** and **MinPts**. In order to find the best values for both parameters, we plotted many charts showing the sorted distance of the K-th item (tried $K \in [1, 15]$), then with the Elbow method we tried to find the best eps (fixed K), giving us the pair of values to use in DBSCAN. Unfortunately, we did not find a good pair of parameters out of our multiple tries, the best that we found was $eps = 2.1$ and $MinPts = 11$, obtaining 2 clusters, the first one with the majority of samples and the second one containing the noise data points. In Figure 26 is shown the PCA of DBSCAN.

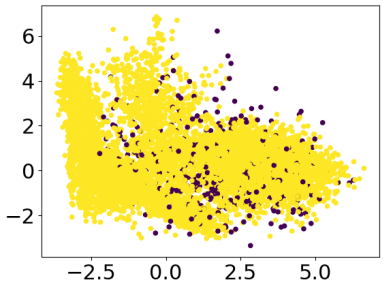


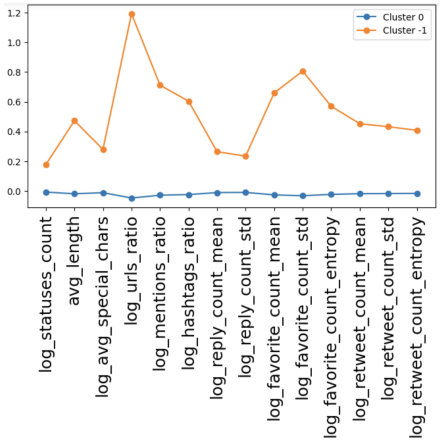
Figure 26: Representation of DBSCAN' clusters using PCA reduced dimensions

3.3.1 Cluster Analysis

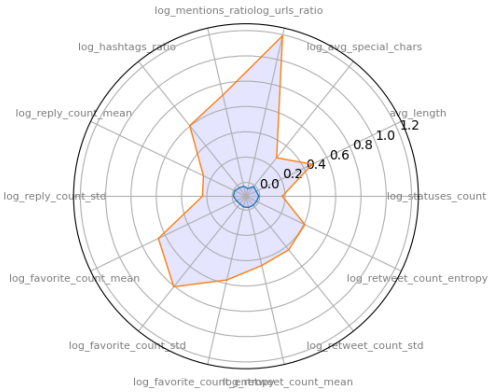
In Figure 27 we can see the parallel and radar plot of the center' clusters found by the algorithm. Now we can see the plots of the distribution of the bot feature inside the clusters in Figure 28. We can notice that, for the bot distribution, there is high entropy in each cluster. This leads us to think that those clusters don't geometrically represent the differentiation between bot and non-bot users. Moreover, in the radar and parallel plots, in Figure 27, we can see that the noise cluster (-1) has very high value for each feature. So the DBSCAN clustering model finds a big cluster with zero center values and noises all around.

3.4 Extra method: SOM (Self-organizing maps for clustering)

As extra clustering method we chose to use SOM (Self-organizing maps for clustering) and we use the implementation of the *pyclustering*[1] library. SOM is a clustering methodology based on



(a) Parallel plot of the centers' clusters



(b) Radar plot of the centers' clusters

Figure 27: DBSCAN analysis

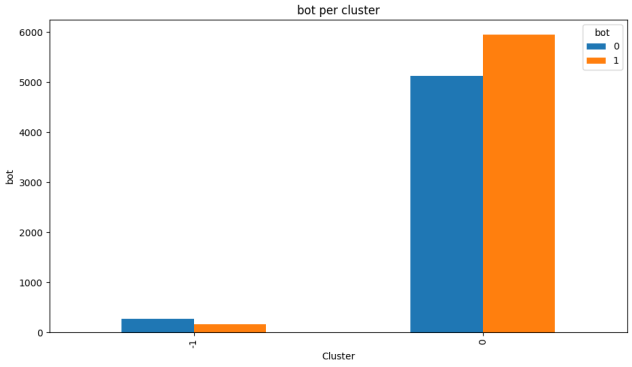
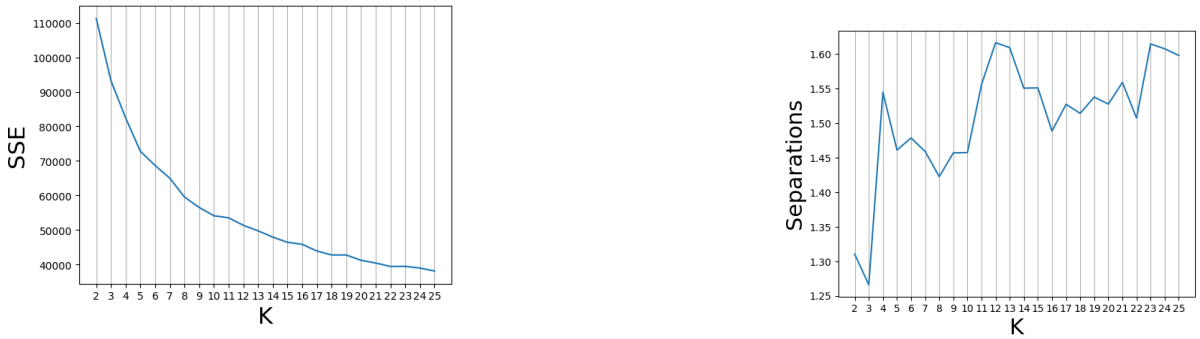


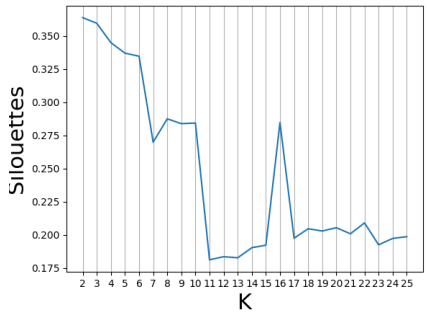
Figure 28: Bot distribution given clusters, for DBSCAN

a 2-D grid whose nodes represent centers of the cluster, and where they are learned through a neural network mantaining a kind of similarity between adjacent nodes. The SOM low-dimension representation takes the nodes of the lattice as centroid of clusters. SOM needs to choose the K parameter, we tried some possible values and chose the proper one with respect to the plot of **SSE** (Sum of Squared Errors), **Separation** (Davies-Bouldin score) and **Silhouette**.



(a)

(b)



(c)

Figure 29: Curve of the SSE varying K (a), curve of the Separation varying K, Curve of the Silhouette varying K (c), for SOM clustering model

Given those curves we chose the K that gave us a minimum of the Separation, a high value of Silhouette and possibly a low value of SSE, so we chose to take $K = 3$. In Figure 30 is shown the PCA of SOM clustering.

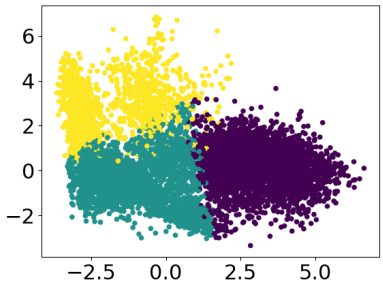
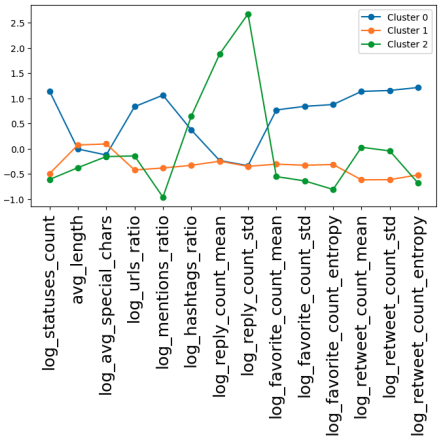


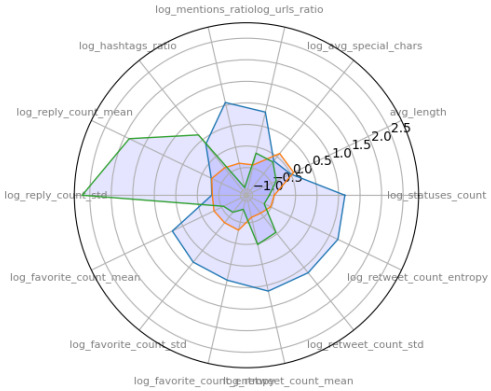
Figure 30: Representation of SOM's clusters using PCA reduced dimensions

3.4.1 Cluster Analysis

In Figure 31 we can see the parallel and radar plot of the clusters' centers found by the algorithm.



(a) Parallel plot of the centers' clusters



(b) Radar plot of the centers' clusters

Figure 31: SOM analysis

Now we can see the plots of the distribution of the bot feature inside the clusters (Figure 32). We can notice that, for the bot distribution, there is the second and third clusters that have the majority of bot users; on the other hand the first one has a majority of non-bot users. This leads us to think that those clusters geometrically represent the differentiation between bot and non-bot users.

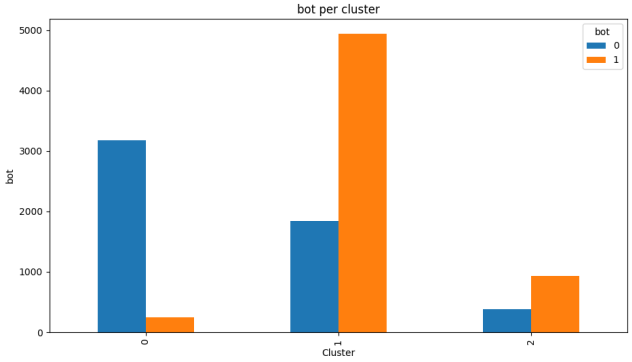


Figure 32: Bot distribution given clusters, for SOM

3.5 Clustering Results Analysis

In the end, the clustering methods gave us good results, except the the DBSCAN method. The other three models gave us comparable results, since they found more or less the same space separation where they retrieved the clusters. SOM, K-means and hierarchical clustering gave us three clusters: one with mainly non-bot users, the other two with a majority of bots and only two of them have a consistent number of samples.

3.5.1 Features analysis

With *interaction features* we refer to the features that represent the interactions among the users, that are all the features except `average_length` and `avg_special_chars`.

Analyzing the features values of the centroids given by K-Means, SOM and hierarchical clustering, we noticed that the two biggest clusters differ on all the values, the cluster with a high number of bots has low interaction features values, except for `reply_count` feature, for which both have low values. The other cluster (the one with few samples) has high values for those features.
The behavior found is coherent with the definition of a bot behavior.

3.5.2 Comparison Table

The following table reports the values of the evaluation metrics of the different methods, the values represent the already stated differences.

Method	SSE	Separation	Silhouette
K-means	93046.70	1.25	0.35
Hierarchical	95689.84	1.29	0.34
DBSCAN	217615.79	3.56	0.22
SOMC	93252.93	1.26	0.35

4 Classification

In this section, we consider the classification problem of predicting if a user is a bot or not. We show all the classifier models we used, and we evaluate their performances.
Before entering in details, we just want to point out an observation we did: we tried to classify the data points once using the categorical feature *language* and once without it, finding out that it does not influence in any way the prediction of an item in a specific class.

To perform the task, which is *supervised*, we decided to consider all the numerical features and the `created_at` feature, transforming it in a timestamp, and splitting the dataset into the training set and test set, equal to the 70% and 30% of the dataset respectively. For each model we performed a grid search to find the best values for some hyperparameters; for that, we used a *k-fold cross validation*, varying the k parameter for each model based on the time needed to train an estimator.

4.1 Bayesian Network

The Naive Bayes classifier tries to estimate the class conditional probability for each item, using the *Bayes* theorem, with the assumption that all the features are conditionally independent. In particular, we used `GaussianNB()`, where the likelihood of the features is assumed to be Gaussian. Before fitting the estimator on the training data, we normalized the data applying the standard scaler.
In 33a we can see the performances of this model; in particular, it reaches an accuracy of 84% on the training set and 83% on the test set.

4.2 Support Vector Machine

The Support Vector Machine (SVM) is a classifier that searches for a hyperplane that can linearly separate the data points, possibly in a transformed space, in the non-separable case. Before fitting the SVM on the training set, we normalized the dataset applying a standard scaler. Afterward, we applied a grid search with a 5 folds and the best hyperparameters we obtained were a SVM

classifier with a *linear* `kernel` and the regularization parameter `C` equal to *10*.
In 33b we can see the performances of this model; in particular, it reaches an accuracy of 84% on both training and test set.

4.3 K-nearest neighbors

The K-Nearest Neighbors model is an instance-based classifier, that, for each record, selects the class label based on the majority of its nearest neighbors.
In this case, we preprocessed the data applying a unit norm normalization to help the classifier to perform better. About this model, we performed a grid search with 3 folds and it gave us *ball_tree* as `algorithm` and *15* as the `n_neighbors`, so that these values were the best hyperparameters to choose.
The estimator reached 88% of accuracy on both training and test set and 33c it is shown its confusion matrix.

4.4 Neural Network

We now show the neural network, in particular a multi-layer perceptron implemented with the *scikit-learn* library. Once again, we first normalized the dataset applying a standard scaler; then we applied a grid search to find the best hyperparameters using 3 fold. We obtained a neural network with *two* `hidden layers` containing *16* and *8* `perceptrons`, respectively, *0.01* as the `learning rate` and *0.1* as `momentum`.
The accuracy obtained from this model was 86% on the training set and 84% on the test; in 33d its performances.

4.5 Decision Tree

A Decision Tree model learns some decision rules from the training data, in order to grow a tree that is able to predict the class for our test points. Again, we performed a grid search using 3-folds and the best combination of hyperparameters we found are *0.001* as `ccp_alpha`, *gini* as `criterion`, *1* as `min_sample_leaf`, *2* as `min_sample_split`, *0* as `min_weight_fraction_leaf`.
With the Decision Tree model (and also with the models that will be presented in the next paragraphs) we reached the best accuracy on both sets; in fact, it is of the 90% on the training set and of the 89% on the test set. In 33e it is shown its confusion matrix.

4.6 Ensembles

4.6.1 Random Forest

A Random Forest model is an ensemble model composed by several decision trees and the result is given by averaging the predictions of the single classifier. Also in this case we performed a grid search using 3-folds and we got *0.001* as ccp_alpha, *3* as min_sample_leaf, *4* as min_sample_split, *0.001* as min_weight_fraction_leaf and *120* as n_estimators as the best set of hyperparameters.
As already mentioned before, also the random forest model gets a high accuracy score on both sets; in fact, it gives 90% of accuracy on the training set and 89% on the test set. In 33f it is shown its performances.

4.6.2 Bagging

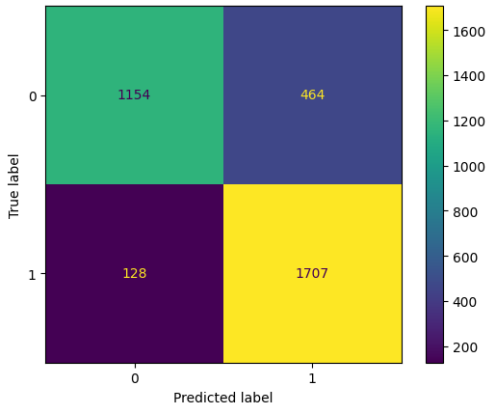
Bagging methods form a class of algorithms that build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (**e.g.** a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.
Again, we performed a grid search using 3-folds on the following hyperparameters: `n_estimators`, which is the number of base estimators in the ensemble, `max_features`, which is the number of features to draw from X to train each base estimator, and `max_samples`, which is the number of samples to draw from X to train each base estimator. The grid search gave us as results *37* as `n_estimators`, *0.9* as `max_samples` and *0.8* as `max_features`.
Like the previous models, also the Bagging ensemble achieved a remarkable result: in fact, it scored 90% of accuracy on the training set and 89% on the test set. The goodness or this result is also shown in its confusion matrix in 33g.

4.6.3 AdaBoost

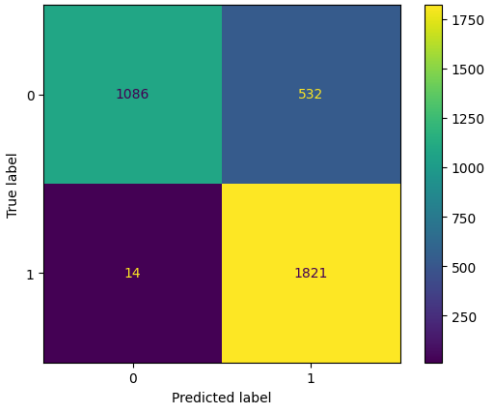
An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. Finally, the grid search was performed with 3-folds and the hyperparameters tuned were: `n_estimators`, which is the maximum number of estimators at which boosting is terminated, `learning_rate`, which is the weight applied to each classifier at each boosting iteration, and the `algorithm` that can be chosen between *SAMME.R* and *SAMME* (the first one typically converges faster than the other). The best set of hyperparameters we obtained was *40* as `n_estimators`, *SAMME.R* as `algorithm` and *0.9* as `learning rate`.
Also this model achieved the upper bound we set with the previous models, reaching 90% of accuracy on the training set and 89% on the test set. In 33h it is present its confusion matrix.

4.7 Classification considerations

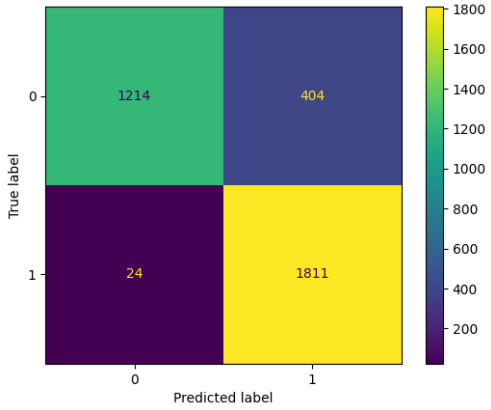
Drawing conclusions, we feel satisfied with the results we achieved; the worst models are the SVM, the neural network and the bayes classifier. Nevertheless, we reached good results with the last models we presented, **i.e.** Decision Tree, Random Forest, Bagging and AdaBoost ensembles, which achieved 89% of accuracy on the test set and which tells us that the problem of classifying if a user is a bot or not is not that trivial.



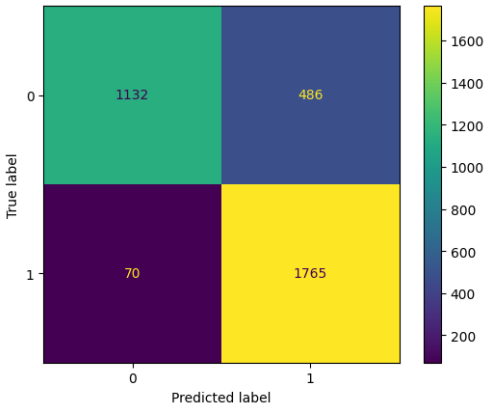
(a) Naive Bayes



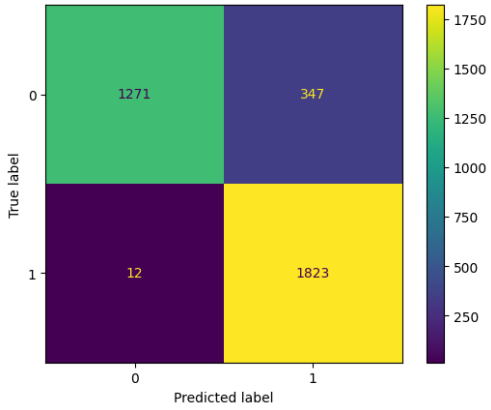
(b) Support Vector Machine



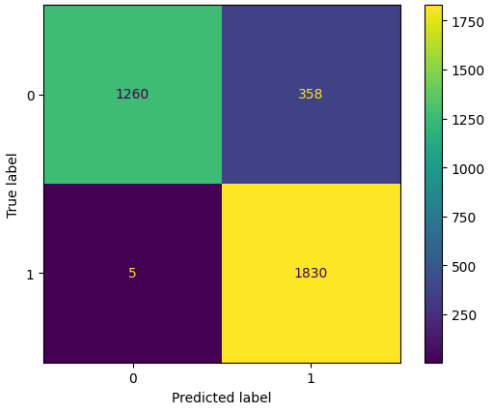
(c) K-nn



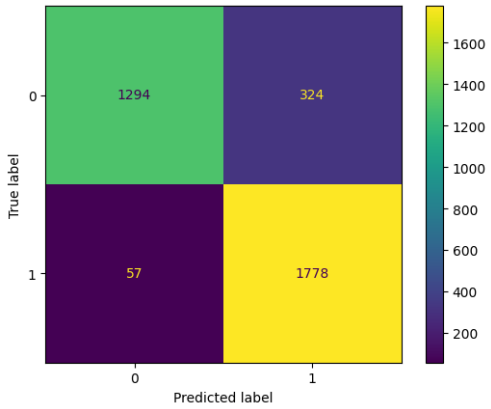
(d) Neural network



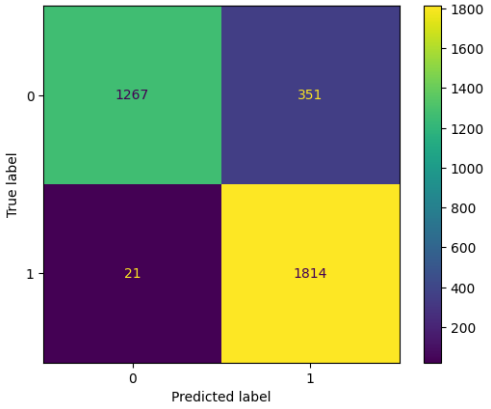
(e) Decision Tree



(f) Random Forest



(g) Bagging



(h) AdaBoost

Figure 33: Confusion matrices of the used classifiers

5 Explainable AI

In this section we are going to describe the part of the project about Explainable AI, discussing the two different approaches used: explanation with post-hoc methods and explanation by training interpretable models.

5.1 Explanation by post-hoc methods

We described the decision tree, SVM and neural network models using two different post-hoc methods:

- **LIME** (*Local Interpretable Model Agnostic Explanation*): it is a method that performs local explanation, **i.e.** it provides explanations for the prediction relative to each observation of a machine learning model and it is a method that works generating random samples;
- **SHAP** (*SHapley Additive exPlanations*): with this method we perform both local and global explanations. SHAP explains the prediction of an observation by computing the contribution of each feature to the prediction.

For all three classification models (DT, SVM, NN) we used the dataset without the feature `lang`, since we noticed that it is the only non-numerical feature and it was irrelevant for the classification. Moreover, we used LIME to perform local explanation and SHAP to perform both local and global explanation.

5.1.1 Explanation analysis for the decision tree model

We explained the decision tree model, using LIME and SHAP, in the notebook **dt_xai.ipynb**.

Using LIME we noticed that the most important feature is `stauses_count`, all the other features are of very little importance. We calculated the metrics of faithfulness and monotonicity using 100 samples, obtaining not high faithfulness mean (0.59) and a very low monotonicity ratio (2/100).

With SHAP we performed local and global explanation (using 200 elements Figure 34a) and in both cases we observed that the most important feature is `stauses_count`, the same we obtained with LIME. We calculated the metrics of faithfulness and monotonicity using 100 samples, we had a good value for faithfulness (0.92), but a very low monotonicity ratio (1/100).

5.1.2 Explanation analysis for SVM model

The SVM model is explained in the notebook **nn_xai.ipynb**, using LIME and SHAPE methods.

According to the LIME method, the behavior we have with the SVM model is the same as we had with the decision tree model. Indeed, the only important feature is `stauses_count`. We calculated the metrics of faithfulness and monotonicity using 100 samples, obtaining a faithfulness with a mean value of 0.33 and a monotonicity ratio of 67/100.

Explaining the SVM model with SHAP, we saw that also in this case the most important feature is `stauses_count` for both local and global explanation. To perform a global explanation we

used 50 elements of the dataset and from Figure 34b we can see how all the other features are of very little importance with respect to `stauses_count`. We calculated the metrics of faithfulness and monotonicity using 50 samples, obtaining a faithfulness with a mean value of almost 0.6 and a monotonicity ratio of 40/50.

5.1.3 Explanation analysis for the neural network model

The neural network model is explained in the notebook **nn_xai.ipynb**, using LIME and SHAPE methods.

Using LIME, also for the neural network model, the only very important feature is `stauses_count`. We calculated the metrics of faithfulness and monotonicity using 100 samples, obtaining a faithfulness with a mean value of 0.62 and a monotonicity ratio of 0/100.

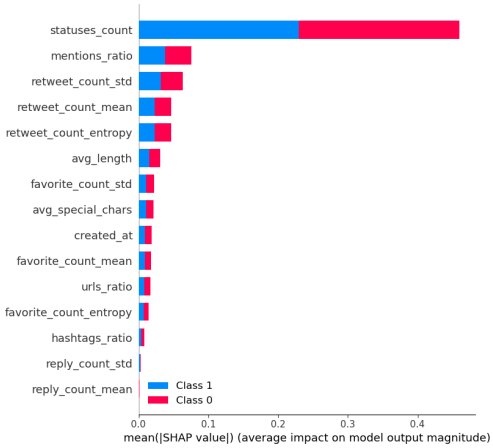
We used SHAP also to perform a global and local explanation for the neural network model. For the global explanation we used 200 samples of the dataset and we saw that the only very relevant feature is `stauses_count` as shown in Figure 34b. Also from the local explanation we noticed that the most important feature is `stauses_count`. We calculated the metrics of faithfulness and monotonicity using 100 samples, obtaining a faithfulness with a mean value of 0.73 and a monotonicity ratio of 0/100.

5.2 Explanation by training interpretable models

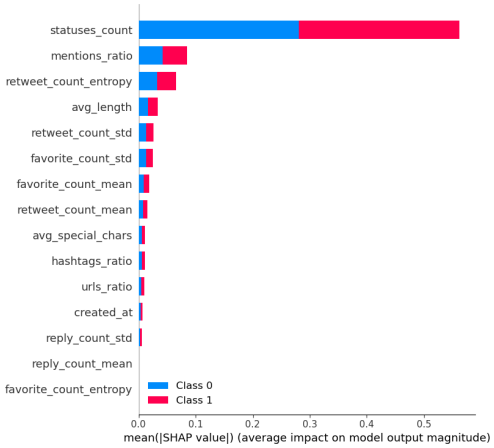
5.2.1 EBM: Explainable Boosting Machine

EBM uses modern machine learning techniques like bagging, gradient boosting, and automatic interaction detection to breathe new life into traditional GAMs (Generalized Additive Models). This makes EBMs as accurate as state-of-the-art techniques like random forests and gradient boosted trees. However, unlike these blackbox models, EBMs produce exact explanations and are editable by domain experts.
In the notebook **EMB.ipynb**, we trained EBM over the dataset obtained during the data preparation phase, without any normalization procedure, we only removed the `lang` and `name` features.

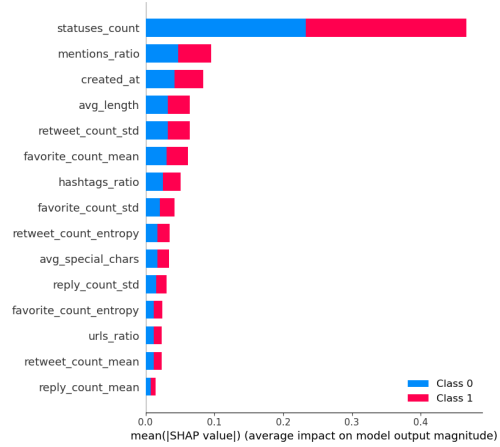
The model can reach very high values in accuracy (more or less 90%) and the most important feature for EBM is `stauses_count`.



(a) Decision tree



(b) Support Vector Machine



(c) Neural network

Figure 34: Shap values of the features given by a global explanation

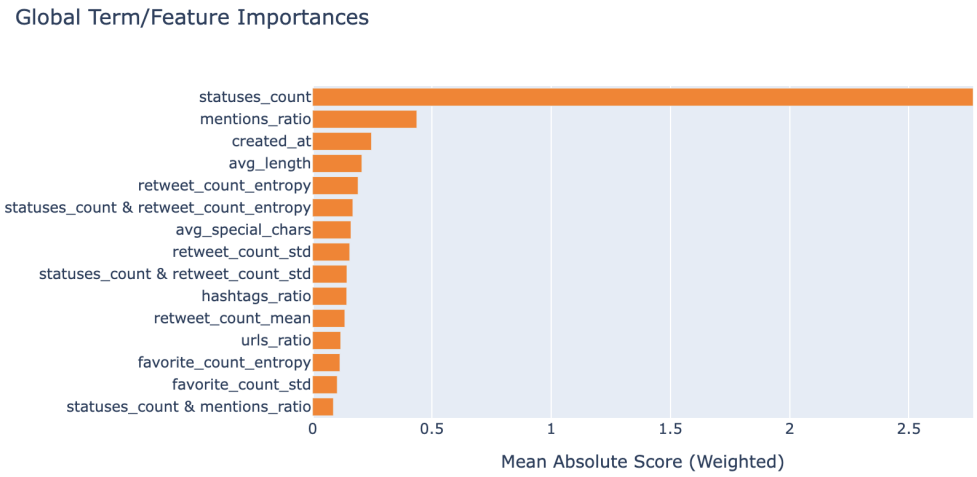


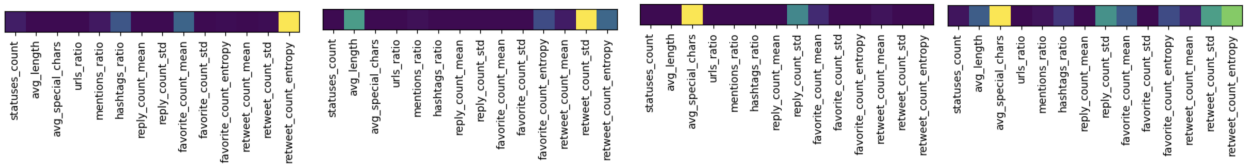
Figure 35: EBM features importance.

5.2.2 TabNet

TabNet is a high-performance and interpretable canonical deep tabular data learning architecture, that uses sequential attention to choose which features to reason from at each decision step, enabling interpretability and more efficient learning as the learning capacity is used for the most

salient features.

In the notebook **TabNet.ipynb**, we trained TabNet over the dataset we created during the data preparation phase, without any normalization procedure, as we did with EBM we removed `lang` and `name` features, but we noticed that at the first execution the model (without grid-search) gave poor results, but removing the `created_at` feature this problem vanished. The model reached an accuracy of 88% in the test set. In Figure 36 we reported the masks (averaged over some samples) of the layers (and the aggregated one). Each of them represents the importance of each feature for the classification of a subset of the training set.
In contrast to the other XAI methods, the most important feature for the classification is not `stauses_count` but the importance is assigned to a different feature for each layer.



(a) Mask 1 (b) Mask 2 (c) Mask 3 (d) Aggregated Mask

Figure 36: TabNet averaged masks

References

[1] Andrei Novikov. Pyclustering: Data mining library. *Journal of Open Source Software*, 4(36):1230, apr 2019.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.