Feature extraction and ML for bot and activity classification of Twitter users

Aswestopoulos Georgios - 108
Bektsis Evangelos - 113
Kaliakatsos Charilaos - 119
Parousidou Vasiliki - Chrysovalanto - 106

This article aims to provide the required steps to identify, discover, and extract features from Twitter data that refer to either users or tweets, and use them to train machine learning algorithms in order to detect if a Twitter user is a human or a bot.

Getting access and using the Twitter API

In order to perform an analysis on Twitter data, two things are necessary.

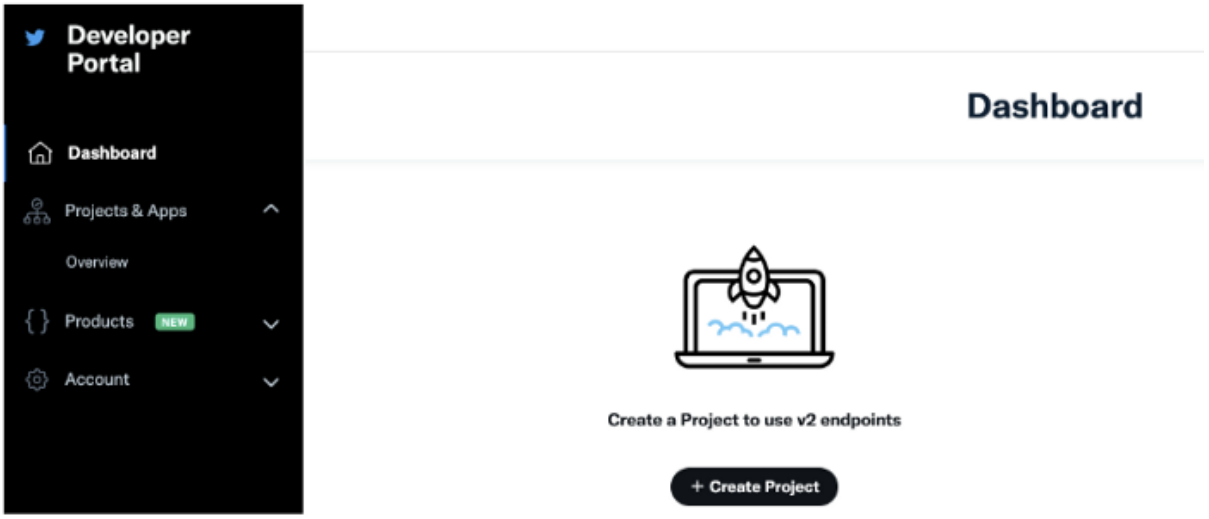
1. Having a Twitter account
2. A way to collect the data (Twitter API with tweepy)

It is essential to have a Twitter account, as a developer-accessible account is required in order to collect the data. Twitter provides a variety of access options, but the best and most comprehensive in terms of the capabilities, is academic access.

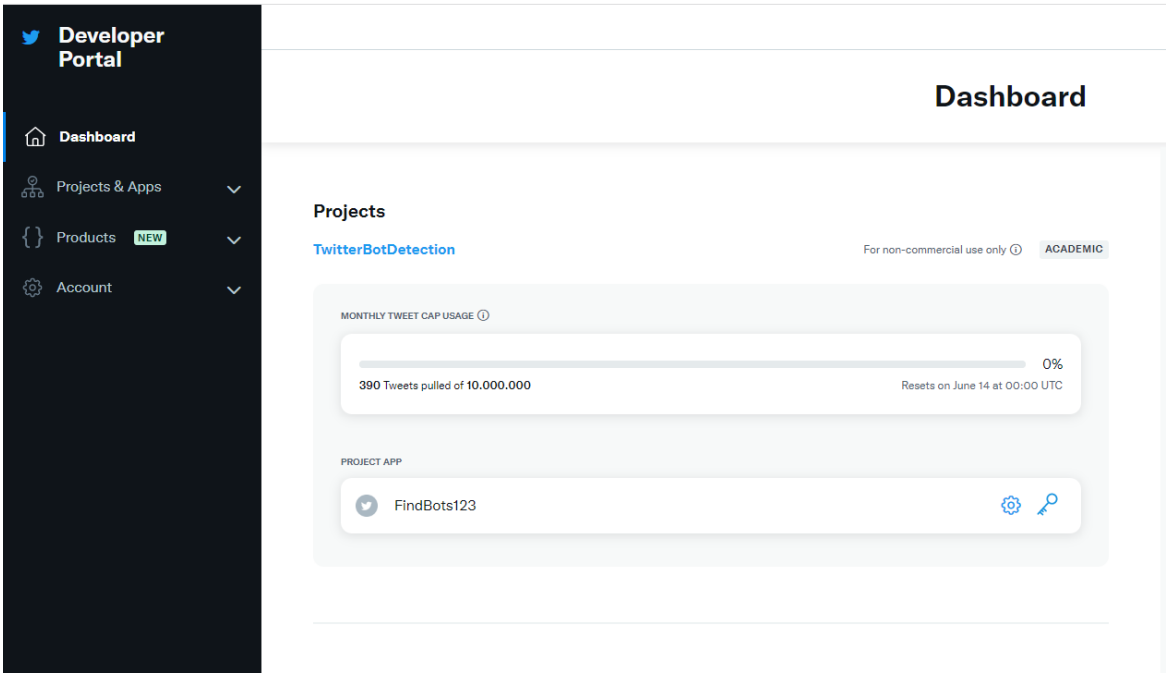
Twitter API

The Twitter API enables unique and advanced access to Twitter. It utilizes Twitter's core components, including Tweets, Direct Messages, Spaces, Lists, and users.

- After setting up our developer account, we are greeted with the following screen:



- When our academic (or the account you selected) access is approved, we can keep track of our data usage via this screen:



Tweepy

Tweepy is an open-source Python library that provides easy access to the Twitter API. It provides an interface to access the API from within a Python application.
In order to start using tweepy, we need to install it by using the following command:

```
pip install tweepy
```

After doing that, we need to complete the authorization to get access to the Twitter API using Python.

```
import tweepy # tweepy module to interact with Twitter
import pandas as pd # Pandas library to create dataframes
from tweepy import OAuthHandler # Used for authentication
from tweepy import Cursor # Used to perform pagination
import json
```

Then, we can begin developing the functions that utilize the tweepy library and extract the necessary data.
At this point, three functions are created. The first function authenticates the user, the second one calls the API, and the third collects data in dataframe format.

```
#using my credentials with academic access.
cons_key = 'XXXX'
cons_secret = 'XXXXX'
acc_token = 'XXXXX'
```

```
acc_secret = 'XXXXX'
bearer_token = 'XXXXXX'

# (1). Athentication Function
def get_twitter_auth(): #the authentification to Twitter

    try:
        consumer_key = cons_key
        consumer_secret = cons_secret
        access_token = acc_token
        access_secret = acc_secret

    except KeyError:
        sys.stderr.write("Twitter Environment Variable not Set\n")
        sys.exit(1)

    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_secret)

    return auth #Function to access the authentication API
def get_twitter_client():   #the client to access the authentification
API

    auth = get_twitter_auth()
    client = tweepy.API(auth, wait_on_rate_limit=True)
    return client
#Function creating final dataframe
def get_tweets_from_user(ids, page_limit=2, count_tweet=200):

    #ids: the twitter username of a user (company, etc.)
    #page_limit: the total number of pages (max=16)
    #count_tweet: maximum number to be retrieved from a page

    client = get_twitter_client()
    all_tweets = []

    for page in Cursor(client.user_timeline,
                user_id=ids,

count=count_tweet,tweet_mode='extended').pages(page_limit):
        for tweet in page:
            parsed_tweet = {}
            parsed_tweet['source'] = tweet.source #get the source of the
tweet (e.g. iphone, android)
            parsed_tweet['all_info'] = tweet._json #get all the rest of
the data
```

```
            all_tweets.append(parsed_tweet)
            if len(all_tweets) >200: #Limit the tweets to 200
                break
    # Create dataframe
    df = pd.DataFrame(all_tweets)
    # Revome duplicates if there are any
    df2 = pd.json_normalize(df["all_info"]) #Unpack the data in the
all_info column
    df2 = df2[["created_at", "id_str", 'full_text',
"in_reply_to_user_id_str", "retweet_count",  "favorite_count",
"favorited", "retweeted",  "entities.hashtags", 'entities.symbols',
'entities.user_mentions', 'entities.urls', 'user.id',
'user.id_str','retweeted_status.created_at', 'retweeted_status.id',
'retweeted_status.id_str']]
    df3 = pd.concat([df['source'],df2],axis=1) #concat all data in a
final dataframe
    return df3
```

Following the definition of our functions, we simply execute them for the number of user ids at our disposal. Because there is a limit on the number of API calls that can be made, the collection takes approximately 1 hour and 40 minutes.

```
from tqdm import tqdm
ids = pd.read_csv('gilani-2017.tsv',header=None,delim_whitespace=True)
#read data
ids.drop_duplicates(subset=[0],inplace=True,ignore_index=True)
second_dataset = pd.read_csv('cresci-rtbust-2019.tsv',header=None,
delim_whitespace=True)
second_dataset.drop_duplicates(subset=[0],inplace=True,ignore_index=True
) #drop duplicate ids
ids = pd.concat([ids,second_dataset], axis=0,ignore_index=True) # concat
all ids in one file
column_names =  ["source","created_at", "id_str", 'text',
"in_reply_to_user_id_str", "retweet_count",  "favorite_count",
"favorited", "retweeted",  "entities.hashtags", 'entities.symbols',
'entities.user_mentions', 'entities.urls', 'user.id',
'user.id_str','retweeted_status.created_at', 'retweeted_status.id',
'retweeted_status.id_str']
df_final = pd.DataFrame(columns = column_names) #create a dataframe to
concat all the collected tweets
for i in tqdm(range(len(ids[0]))):
    try:
        tweets = get_tweets_from_user(ids[0][i]) #iterate to get all the
tweets
```

```
    except Exception as e: #to avoid possible errors from missing
account and info
        #print(e)
        continue
    df_final = pd.concat([tweets,df_final])
    df_final.reset_index(inplace=True, drop=True)
df_final.to_csv('Final_Data_v3.csv') #save all the data
```

We also need to collect user data from known users and we just repeat the same steps as before just using a list that contains our ids.

```
#real users
#repeat same steps as above, just with different ids this time
import pandas as pd
from tqdm import tqdm
column_names =  ["source","created_at", "id_str", 'text',
"in_reply_to_user_id_str", "retweet_count",  "favorite_count",
"favorited", "retweeted",  "entities.hashtags", 'entities.symbols',
'entities.user_mentions', 'entities.urls', 'user.id',
'user.id_str','retweeted_status.created_at', 'retweeted_status.id',
'retweeted_status.id_str']
df_final = pd.DataFrame(columns = column_names)
user_ids = [234343780, 48008938, 249130209, 1348351605654106118,
86391789, 70340615, 289527193, 1005766052607938560, 62213337,
1495855082734297095, 1976335622, 1066275282653536256]
for i in tqdm(range(len(user_ids))):
    try:
        tweets = get_tweets_from_user(user_ids[i])
    except Exception as e:
        continue
    df_final = pd.concat([tweets,df_final])
    df_final.reset_index(inplace=True, drop=True)
df_final.to_csv('Real_users.csv') #Save to Real_users.csv
```

Finally, we need to collect and save the user information and for that reason we use the lookup_users method that returns all the required data.

```
#import and auth again
import tweepy
from tweepy import OAuthHandler

CONSUMER_KEY = cons_key
CONSUMER_SECRET = cons_secret
ACCESS_KEY = acc_token
ACCESS_SECRET = acc_secret
```

```
auth = OAuthHandler(CONSUMER_KEY,CONSUMER_SECRET)
api = tweepy.API(auth)
auth.set_access_token(ACCESS_KEY, ACCESS_SECRET)

#search
api = tweepy.API(auth)
test = api.lookup_users(user_id=user_ids) #take all the required user
info using the lookup_users method

json_str = json.loads(json.dumps(test[0]._json)) #unpack the data
df = pd.DataFrame([json_str], columns=json_str.keys()) #create a
dataframe
for i in range(len(test)-1):
    json_str = json.loads(json.dumps(test[i+1]._json))
    df1 = pd.DataFrame([json_str], columns=json_str.keys())
    df = pd.concat([df1,df]) #concat
df.reset_index(inplace=True,drop=True)
df.to_csv('user_data.csv') #save to csv
```

Some of the 3,392 users used to collect the data were deleted or inactive, so there was no data to collect. Finally, exactly 476,413 tweets were collected, and data collection is now complete. We can now move on to feature extraction, which is the primary focus of this tutorial.

Feature Extraction

When training machine learning algorithms, the process of extracting features is essential to getting the best results possible. We leverage a plethora of features that can be classified into five categories: Temporal, Content, Network - Social Neighborhood, Sentiment and User, which are further analyzed.

First, let's import some libraries we need and load the Tweets we will extract the features from:

```
import pandas as pd
import emoji
import numpy as np
import math
import statistics
from scipy.stats import skew,kurtosis
import scipy
import string
from datetime import datetime, timedelta
from collections import Counter
from scipy import stats
```

```
import json
import re
import ast
import time
```

```
#Read file that contains all tweets
tweets = pd.read_csv(r"Final_Data_v3_1.csv")
```

Also, we remove the users that haven't posted more than one tweet since we cannot extract much information.

```
#Group tweets according to user id and remove users with less than 2
tweets
tweets = tweets.groupby('user.id_str').filter(lambda x : len(x)>1)
tweets.reset_index(inplace = True, drop = True)
```

Temporal

First, we expect that temporal features, which refer to user activity, may reveal significant information about the user. More specifically, some examples of temporal features include how often a user posts, tweets, retweets or mentions someone. These features can be calculated by measuring the time between two consecutive posts, tweets, retweets and mentions, the number of tweets per hour or weekday and some statistics about them, such as min, max, median, mean, standard deviation, skewness, kurtosis, entropy and quantiles.

The statistics for each feature are calculated using the following method:

```
#Method that extracts statistics from a vector
#More specifically, it calculates MIN, MAX, MEAN, MEDIAN, Standard
Deviation, skewness, kurtosis, entropy and quantiles 0.25 and 0.75

def calc_vector_stats(vectors, column_names):

      df_final = pd.DataFrame(columns = column_names)

      for key in vectors:
      a = vectors[key]
      MIN = min(a)
      MAX = max(a)
      MEAN = statistics.mean(a)
      MEDIAN = statistics.median(a)
      STD = np.std(a)
      SKEW = skew(a)
      KYRT = kurtosis(a)
```

```
        pd_series = pd.Series(a)
        counts = pd_series.value_counts()
        ENTROPY = scipy.stats.entropy(counts)
        q1 = np.percentile(a, 25)
        q3 = p = np.percentile(a, 75) #Q3
        x = [key, MIN, MAX, MEAN, MEDIAN, STD, SKEW, KYRT, ENTROPY, q1,
q3]
        df_final.loc[-1] = x
        df_final.index +=1
        df_final = df_final.sort_index()

        return df_final
```

Some other useful methods for adding new features in a dataframe are also introduced:

```
def call_both(column_name, previous_df):
        vector = function_column_adder(column_name)
        a = merge_dataframes(vector,column_name,previous_df)
        return a

def function_column_adder(column_name):
        users  = tweets['user.id_str'].value_counts()
        vectors = {}
        for ID in users.index:
        new_df = tweets[tweets['user.id_str'] == ID]
        new_df2 = new_df[column_name]
        a = new_df2.values.tolist()
        vectors.update({ID:a})
        return vectors

def merge_dataframes (vectors,column_name,previous_df):
        column_names =  ["User_id", "Min_"+column_name,
"Max_"+column_name,"Mean_"+column_name, "Median_"+column_name,
"Std_"+column_name, "Skew_"+column_name, "Kyrt_"+column_name,
"Entropy_"+column_name, "Q1_"+column_name, "Q3_"+column_name]
        to_be_added = calc_vector_stats(vectors,column_names)
        return pd.merge(previous_df, to_be_added , on='User_id')
```

Next, we need a method that calculates the time between consecutive posts for each user separately. This is why we add the following method that splits the tweets of a given dataframe based on the user that posted them and after that it creates a dictionary where each key represents a user and the value is a vector containing the various durations.

```
#Method that calculates time between consecutive tweets created by each
user
```

```
def get_time_between_tweets(tweets):
      gaplist = {}
      users  = tweets['user.id_str'].value_counts()
      vectors_text_length = {}
      for ID in users.index:
      gap = []
      new_df = tweets[tweets['user.id_str'] == ID]
      new_df.reset_index(inplace = True, drop = True)
      #print(ID)
      for i in range(0, len(new_df) - 1):
            #Find the time that the current tweet was created
            first = datetime.strptime(new_df['created_at'][i],'%a %b %d
%H:%M:%S +0000 %Y')
            #Find the time that the next consecutive tweet was created
            second = datetime.strptime(new_df['created_at'][i + 1],'%a
%b %d %H:%M:%S +0000 %Y')
            #Store the difference
            gap.append((first - second).seconds)
      gaplist.update({ID:gap})
      return gaplist
```

Also, we need two methods that calculate the number of tweets each hour of a day and each weekday for each user.

```
#Method that calculates the number of tweets each hour of day

def tweets_per_hour(tweets):
      users  = tweets['user.id_str'].value_counts()
      vectors = {}
      for ID in users.index:
      a = np.zeros(24)
      new_df = tweets[tweets['user.id_str']== ID]
      new_df2 = new_df['Hour'].value_counts()
      for times,value in zip(new_df2.index,new_df2):
            a[times] = value
      vectors.update({ID:a})
      return vectors
```

```
#Method that calculates the number of tweets each weekday

def tweets_per_weekday(tweets):
      testdf  = tweets['user.id_str'].value_counts()
      vectors = {}
```

```
    for ID in testdf.index:
    a = np.zeros(7)
    new_df = tweets[tweets['user.id_str']== ID]
    new_df2 = new_df['Weekday'].value_counts()
    for times,value in zip(new_df2.index,new_df2):
          a[times] = value
    vectors.update({ID:a})
    return vectors
```

Now, we are ready to calculate the required features that were described above using the auxiliary methods that we implemented. We also calculate the seconds needed to extract each feature, which is presented in the following section.

```
start = time.time()
#Time between two consecutive posts (both tweets and retweets) and
statistics
times = get_time_between_tweets(tweets)
features = merge_dataframes(times, "time", features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Time between two consecutive retweets and statistics
re_tweets = tweets[~tweets['retweeted_status.id_str'].isnull()]
re_tweets.reset_index(inplace = True,drop = True)
re_tweets = re_tweets.groupby('user.id_str').filter(lambda x : len(x)>1)
re_tweets.reset_index(inplace = True,drop = True)

times = get_time_between_tweets(re_tweets)
features = merge_dataframes(times, "time_rt", features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Time between two consecutive tweets and statistics
normal_tweets = tweets[tweets['retweeted_status.id_str'].isnull()]
normal_tweets.reset_index(inplace = True,drop = True)
normal_tweets = normal_tweets.groupby('user.id_str').filter(lambda x :
len(x)>1)
normal_tweets.reset_index(inplace = True,drop = True)

times = get_time_between_tweets(normal_tweets)
features = merge_dataframes(times, "time_normal", features)
end = time.time()
```

```
print(end - start)
```

```
start = time.time()
#Time between two consecutive mentions and statistics
new_df = tweets[tweets['entities.user_mentions'].map(lambda d: len(d)) >
2]
new_df.reset_index(inplace= True,drop = True)
new_df = new_df.groupby('user.id_str').filter(lambda x : len(x)>1)
times = get_time_between_tweets(new_df)
features = merge_dataframes(times, "time_mentions", features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Tweets per Hour

#Convert 'created_at' feature to datetime
tweets['Datetime'] = pd.to_datetime(tweets['created_at'], format= "%a %b
%d %H:%M:%S %z %Y")
tweets['Hour'] = tweets['Datetime'].apply(lambda x: x.hour)
hour = tweets_per_hour(tweets)
features = merge_dataframes(hour,'hour', features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Tweets per Weekday

tweets['Weekday'] = tweets['Datetime'].apply(lambda x: x.weekday())
day = tweets_per_weekday(tweets)
features = merge_dataframes(day,'day',features)
end = time.time()
print(end - start)
```

Content

Another important category of features is content and language features. More specifically, it has been proven in research papers that the length of a tweet or the number of mentions plays an important role in categorizing a user as a bot. Part-of-speech would also be helpful but we try to build a language agnostic model and this feature is difficult to be retrieved independently of language. Thus, the features we will extract are the number of words, characters, punctuation marks, URLs, mentions and hashtags in each tweet and their statistics.

The methods used are presented below:

```
start = time.time()
#Number of words in a Tweet and statistics
tweets["text_length_words"] = tweets["text"].map(lambda x:len(x.split("
")))
features = call_both("text_length_words", df_emoji_stats)
end = time.time()
print(end - start)
```

```
start = time.time()
#Number of characters in a Tweet and statistics
tweets["text_length_characters"] = tweets["text"].map(lambda x:len(x))
features = call_both("text_length_characters", features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Number of punctuation marks in a Tweet and statistics
count = lambda l1,l2: sum([1 for x in l1 if x in l2])
tweets["text_puncts"] = tweets["text"].map(lambda x
:count(x,set(string.punctuation)))
features = call_both("text_puncts", features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Number of URLs in a Tweet and statistics
tweets['url_number'] = tweets['entities.urls'].apply(lambda x
:len(ast.literal_eval(x)) if (len(x)>2)  else 0 )
features = call_both('url_number', features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Number of mentions in a Tweet and statistics
tweets['mention_number'] = tweets['entities.user_mentions'].apply(lambda
x :len(ast.literal_eval(x)) if (len(x)>2)  else 0 )
features = call_both('mention_number', features)
end = time.time()
print(end - start)
```

```
start = time.time()
#Number of hashtags in a Tweet and statistics
#new_df = tweets[tweets['entities.hashtags'].map(lambda d: len(d)) > 0]
tweets['hashtag_number'] = tweets['entities.hashtags'].apply(lambda x
:len(ast.literal_eval(x)) if (len(x)>2)  else 0 )
features = call_both('hashtag_number', features)
end = time.time()
print(end - start)
```

Network - Social Neighborhood

To create the network of each user, we start by applying `ast.literal_eval` in the `user_mentions` column so we can parse each row as json afterwards.

```
tweets['entities.user_mentions'] =
tweets['entities.user_mentions'].apply(lambda x :ast.literal_eval(x))
```

The second step is to remove all the users that haven't mentioned anyone ever.

```
mask = tweets[tweets['entities.user_mentions'].str.len() > 0]
mask.reset_index(inplace = True, drop = True)
```

Next, we just use the code that we implemented in the first tutorial to create a graph in neo4j. However, this time we create 2 node types (User , Tweet) and 1 relationship (Mention).

We follow up the creation of the graph with the queries we need. We want to get how many neighbors each user has and how many times he has been mentioned or mentions someone. We start by setting each user with 0 mentions and 0 neighbors. In this way, we fill the empty users (we dropped each user that didn't have a single mention. By doing this, we cut down the number of calculations for each user). After that, we run the query and count our results while passing them into the dataframe for each user.

```
#Store the number of times each user mentions someone or is mentioned by
someone
#Store the number of neighbors of each user

features["mentions"] = 0
features["neighbors"] = 0
i = -1
for user_id in features["User_id"]:
      i = i + 1
      a = graph.run(f'MATCH (n1:User)-[r]-(n2:User) WHERE
n1.`user.id_str` = {user_id} RETURN count(r) as relationships,
n2.`user.id_str` as neighbor')
      df = pd.DataFrame(a)
```

```
        try:
        features["mentions"].iloc[i] = sum(df[0])
        features["neighbors"].iloc[i] = len(df)
        except:
        pass
```

The second query follows the same logic as the first one. We want as a feature the DegreeOut of each user. As we did before, we create a column filled with zeros. Then we run the query and change it for each user.
And just like this we got three new features!

```
from neo4j import GraphDatabase
import networkx as nx

driver = GraphDatabase.driver('bolt://localhost:7687', auth=("neo4j",
"0000"))

#Store the out degree of each user
features["degreeOut"] = 0
i=-1
for user_id in features["User_id"]:
        i += 1
        query =f"MATCH (u:User) WHERE u.`user.id_str` = {user_id} RETURN
size((u)-->()) AS degreeOut"
        results = driver.session().run(query)
        df = pd.DataFrame(results)
        try:
        features["degreeOut"].iloc[i] = df[0]
        except:
        pass
```

Sentiment

Sentiment analysis is also useful to describe emotions and extract useful information so that we can analyze the profile of the user that posted the tweet. Some features that can be extracted from the tweets and are sentiment related are “happiness”, “arousal”, “valence” and “dominance” scores as well as “polarity”. However, the libraries that calculate these scores are not language agnostic and are focused only on English. We could first use a translator to translate the tweets in English and then calculate these scores but it would be really time consuming and we decided to omit it.

This is why we chose to extract the number of emoticons used in each tweet and the statistics that result from the constructed vectors. Also, we calculate the ratio of tweets that contain emoticons.

```
#Method that extracts emojis from a string

def extract_emojis(s):
  return ''.join(c for c in s if c in emoji.UNICODE_EMOJI['en'])
```

```
start = time.time()
#Number of emoticons in each tweet and statistics
tweets["emoji"] = tweets["text"].map(lambda x:extract_emojis(x))
tweets["number_of_emojis"] = tweets["emoji"].map(lambda x:len(x))
column_names =  ["User_id", "Min_emo", "Max_emo","Mean_emo",
"Median_emo", "Std_emo", "Skew_emo", "Kyrt_emo", "Entropy_emo",
"Q1_emo", "Q3_emo"]

testdf  = tweets['user.id_str'].value_counts()
vectors_emoji = {}
ratio = {}
for ID in testdf.index:
      new_df = tweets[tweets['user.id_str']== ID]
      new_df2 = new_df['number_of_emojis']
      a = new_df2.values.tolist()
      ratio.update({ID:((len(a)-a.count(0))/len(a))})
      end1 = time.time()
      vectors_emoji.update({ID:a})
df_emoji_stats = calc_vector_stats(vectors_emoji,column_names)
end = time.time()
print(end - start)
print(end1 - start)
```

```
ration = pd.DataFrame.from_dict(ratio, orient='index')
ration.reset_index(inplace = True)
ration.rename(columns = {'index':'User_id', 0:'Emoji_Ratio'}, inplace =
True)
df_emoji_stats = pd.merge(df_emoji_stats, ration , on='User_id')
```

User

Finally, we utilize features that are extracted from user metadata since they provide useful information about the user's identity. More specifically, the connectivity of a user such as their friends, followers or number of mentions can indicate if a user is a bot or not.

First, we load the metadata of the users that will be used in this project:

```
users1 = pd.read_json(r"gilani-2017_tweets.json")

user_features1 = pd.DataFrame(users1['user'].values.tolist(),
```

```
                                index=users1.index)
user_entities1 =
pd.DataFrame(user_features1["entities"].values.tolist(),
                                index=user_features1.index)
user_url1 =  pd.DataFrame(user_entities1["url"].values.tolist(),
                          index=user_entities1.index)
merged1 = pd.concat([user_features1,user_url1], axis=1)
```

```
users2 = pd.read_json(r"cresci-rtbust-2019_tweets.json")

user_features2 = pd.DataFrame(users2['user'].values.tolist(),
                                index=users2.index)
user_entities2 =
pd.DataFrame(user_features2["entities"].values.tolist(),
                                index=user_features2.index)
user_url2 =  pd.DataFrame(user_entities2["url"].values.tolist(),
                          index=user_entities2.index)

merged2 = pd.concat([user_features2,user_url2], axis=1)
merged = pd.concat([merged1, merged2], axis = 0)
merged.reset_index(inplace = True, drop = True)
```

Now we can analyze the screen name and user name of the user by calculating the number of their characters and words used.

```
start = time.time()
#Number of characters in user name
name_char_len = []

for i in range(len(merged)):
      name_char_len.append(len(merged['name'][i]))

merged['name_char_len'] = pd.DataFrame(name_char_len)
end = time.time()
print(end - start)

start = time.time()
#Number of characters in screen name
screen_name_char_len = []

for i in range(len(merged)):
      screen_name_char_len.append(len(merged['screen_name'][i]))

merged['screen_name_char_len'] = pd.DataFrame(screen_name_char_len)
end = time.time()
```

```
print(end - start)

start = time.time()
#Number of words in user name
name_words_num = []

for i in range(len(merged)):
      name_words_num.append(len(merged['name'][i].split()))

merged['name_words_num'] = pd.DataFrame(name_words_num)
end = time.time()
print(end - start)
```

Next, we calculate the number of days the account is active. This is important not only as a feature but it will be also used in calculating additional features.

```
merged['Datetime'] = pd.to_datetime(merged['created_at'], format= "%a %b
%d %H:%M:%S %z %Y" )

start = time.time()
#Calculate how many days have passed after the creation of the account
from datetime import date
dates =[]
#s1 = now.strftime("%m/%d/%Y, %H:%M:%S")
# mm/dd/YY H:M:S format
#print("s1:", s1)
for i in range(len(merged)):
      year = merged['Datetime'][i].year
      month = merged['Datetime'][i].month
      day = merged['Datetime'][i].day
      d0 = date(year, month, day)
      d1 = date(2022, 5, 14)
      delta = d1 - d0
      dates.append(delta.days)

# Account Age
merged['Date_Delta'] = pd.DataFrame(dates)
end = time.time()
print(end - start)
```

Now we can count the number of user's followers, friends and favorites divided by the number of days the user is active.

```
start = time.time()
#Number of followers per day
```

```
merged['Followers_Per_Day'] = merged['followers_count'] /
merged['Date_Delta']
end = time.time()
print(end - start)

start = time.time()
#Number of friends per day
merged['Friends_Per_Day'] = merged['friends_count'] /
merged['Date_Delta']
end = time.time()
print(end - start)

#Number of favourites per day
start = time.time()
merged['Favourites_Per_Day'] = merged['favourites_count'] /
merged['Date_Delta']
end = time.time()
print(end - start)
```

We also extract the user's time zone and calculate the ratio between friends and followers.

```
start = time.time()
#User's timezone
merged["timezone"] = merged["Datetime"].map(lambda x : x.strftime("%z"))
end = time.time()
print(end - start)

start = time.time()
#Followers-Friends Ratio
merged["Followers_Friends_Ratio"] =
merged["followers_count"]/merged["friends_count"]
end = time.time()
print(end - start)
```

Moreover, users can add an image and a description of themselves. We think that a useful feature would be to check whether the user uses the default profile picture that Twitter provides. Another one would be to check if their description is unique among the users.

```
start = time.time()
#If uses default image
merged["Uses_default_profile_image_url"] =
np.where(merged["profile_image_url"] ==
"http://abs.twimg.com/sticky/default_profile_images/default_profile_norm
al.png",True,False)
end = time.time()
```

```
print(end - start)

start = time.time()
#If uses unique description
merged["not_using_unique_description"] =
merged['description'].duplicated(keep=False)
end = time.time()
print(end - start)
```

Evaluation of Time

It is crucial to evaluate the time needed to extract each feature so as to produce near real time results. The tables below show the average time needed in seconds to extract each feature for 2103 users and overall 476413 tweets in a machine with processor AMD Ryzen 7 5800H and 16,0 GB RAM.

Temporal Features

Feature	**Average Time (sec)**
Time between two consecutive tweets	11.42632
Time between two consecutive retweets	9.93065
Time between two consecutive posts	16.75213
Time between two consecutive mentions	12.63801
Tweets per Hour	9.69937
Tweets per Weekday	7.14085

Content Features

Feature	**Average Time (sec)**
Number of words in a tweet	6.01968
Number of characters in a tweet	5.74534
Number of punctuation marks in a tweet	7.88424
Number of mentions in a tweet	14.54806
Number of URLs in a tweet	7.74850
Number of hashtags in a tweet	8.46630

Network - Social Neighborhood

Feature	**Average Time (sec)**

Number of neighbors (nodes)	140.17072
Number of edges (overall mentions)	140.17072
Out-strength distribution	111.93256

Sentiment Features

Feature	**Average Time (sec)**
Number of emoticons in a tweet	31.42851
Ratio of tweets that contain emoticons	27.17612

User Features

Feature	**Average Time (sec)**
Number of characters in user name	0.01200
Number of characters in screen name	0.01000
Number of words in user name	0.01200
Number of days the account is active	0.05801
Number of followers per active day	0.00099
Number of friends per active day	0.00100
Number of favorites per active day	0.00099
Time Zone	0.01900
Followers/Friends Ratio	0.00100
If uses default picture	0.00100
If uses unique description	0.00100

Development of Supervised ML Models

In this section, we will develop some supervised machine learning models to classify a user as a human or bot, using the features we extracted earlier. More specifically, the performance of some well known ML algorithms will be put into test via the Pycaret library, which automates workflows and allows the user to test their dataset in a lot of different ML algorithms. In this manner, pycaret can be described as a wrapper of all the corresponding libraries and as a useful tool for speeding up the test process.

To begin with the preparation of the data, it was necessary that the features extracted from the Twitter api be concatenated into a single table together with the bot_not dataframe. This dataframe encapsulates the id of the user and their real known status, meaning whether or not they are real human beings or some kind of bots. The analogous piece of code can be seen below:

```
#preprocess the data before feeding it to pycaret
df = pd.read_csv("final_features.csv")
gilani = pd.read_csv("gilani-2017.tsv", header = None, sep="\t")
creski = pd.read_csv("cresci-rtbust-2019.tsv", header = None, sep="\t")
df.drop(columns = {"Unnamed: 0.1",'Unnamed: 0'}, inplace =True)
bot_not = pd.concat([gilani,creski])
bot_not.reset_index(drop= True, inplace=True)
bot_not.rename(columns={0:'id',1:'status'},inplace =True)
df = df.merge(bot_not, left_on='User_id', right_on='id')
df.drop(columns='id',inplace=True)
df.drop(columns = "degree", inplace = True)
df.drop(columns = "degreeIn", inplace = True)
df1=df.copy()#for later use since pycaret changes the original df
```

The data is ready to be fed to the algorithm, and the `compare_models` function of the pycaret is called to do just that and to return the best model, which in our case was the gradient boosting classifier. Some indicative results coming from it as well as the code used, are presented below.

```
# compare machine learning algorithms on the sonar classification
dataset
from pandas import read_csv
from pycaret.classification import setup
from pycaret.classification import compare_models
# load the dataset
# set column names as the column number
n_cols = df.shape[1]
df.columns = [str(i) for i in range(n_cols)]
# setup the dataset
grid = setup(data=df, target=df.columns[-1], html=False, silent=True,
verbose=False)
# evaluate models and compare models
best = compare_models()
# report the best model
print(best)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
gbc	Gradient Boosting Classifier	0.7843	0.8185	0.8857	0.7933	0.8363	0.5227	0.5319	0.708
lightgbm	Light Gradient Boosting Machine	0.7802	0.8071	0.8813	0.7911	0.8329	0.5141	0.5232	0.244
rf	Random Forest Classifier	0.7645	0.7939	0.8747	0.7753	0.8216	0.4785	0.4879	0.156
et	Extra Trees Classifier	0.7570	0.7967	0.8912	0.7595	0.8199	0.4534	0.4687	0.103
ada	Ada Boost Classifier	0.7509	0.7840	0.8418	0.7770	0.8073	0.4559	0.4606	0.196
lda	Linear Discriminant Analysis	0.7434	0.7729	0.8451	0.7663	0.8032	0.4364	0.4425	0.035
ridge	Ridge Classifier	0.7427	0.0000	0.8473	0.7646	0.8033	0.4338	0.4402	0.018
dt	Decision Tree Classifier	0.6805	0.6595	0.7484	0.7410	0.7441	0.3185	0.3193	0.041
qda	Quadratic Discriminant Analysis	0.6300	0.5997	0.8923	0.6471	0.7482	0.1048	0.1444	0.029
nb	Naive Bayes	0.6232	0.5083	0.9681	0.6275	0.7614	0.0307	0.0595	0.018
dummy	Dummy Classifier	0.6212	0.5000	1.0000	0.6212	0.7663	0.0000	0.0000	0.015
knn	K Neighbors Classifier	0.5884	0.5669	0.7604	0.6427	0.6964	0.0705	0.0728	0.251
svm	SVM - Linear Kernel	0.4265	0.0000	0.2000	0.1238	0.1529	0.0000	0.0000	0.021
lr	Logistic Regression	0.4026	0.5100	0.1000	0.0619	0.0765	0.0000	0.0000	0.922

```
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=1970, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

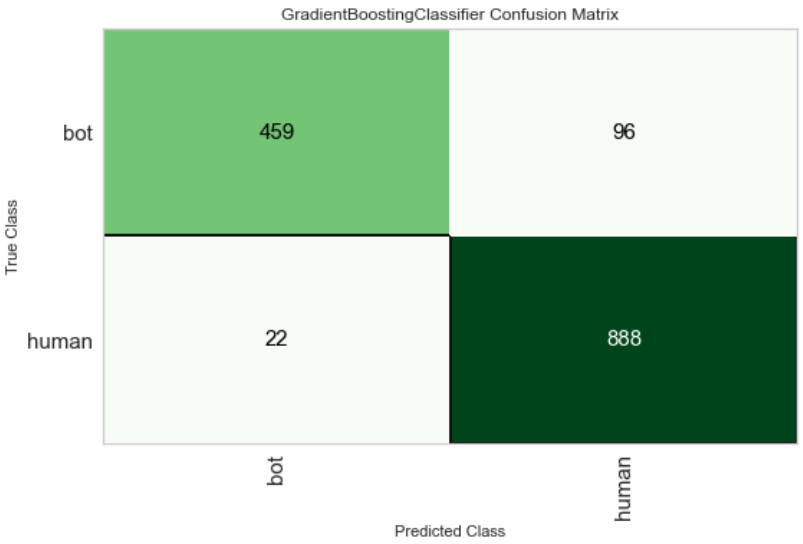
Now that the best model is defined, the `plot_model` function of `pycaret` can be used to plot important information such as the confusion matrix, the learning curves for training and cross validation phases, as well as a table containing information about the most used metrics and their variation across the folds of the cross validation phase.

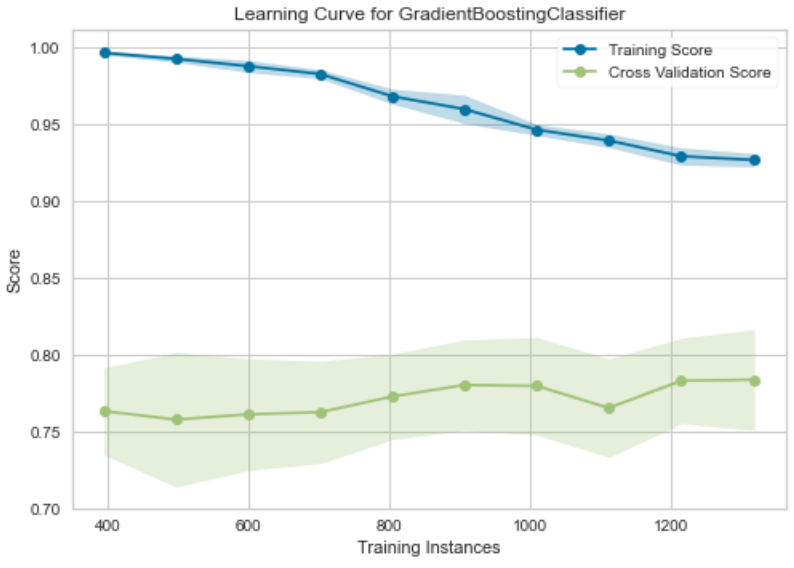
```
from pycaret.classification import *
gbc = create_model('gbc')
plot_model(gbc, plot = 'confusion_matrix', use_train_data = True)

plot_model(gbc, plot = 'learning', use_train_data = True)

plot_model(gbc,plot="error",use_train_data=True)
```

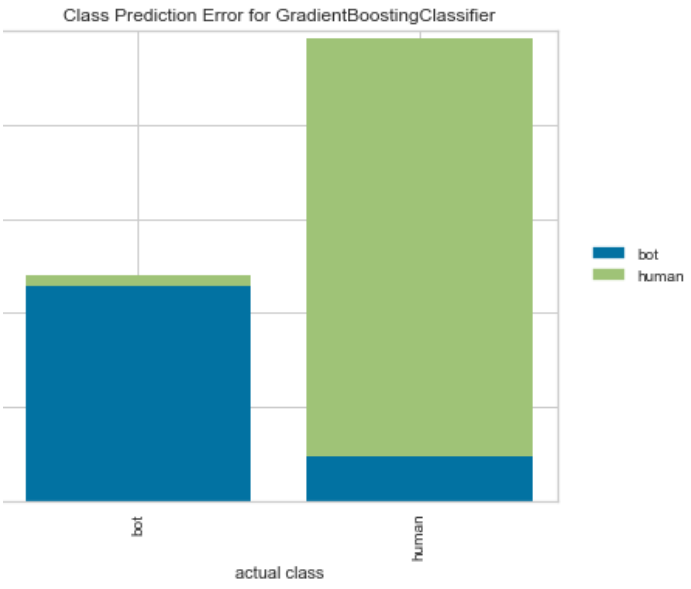
Below, the most significant metrics and the confusion matrix are presented. As for the metrics, it can be noted that all of them boost quite good values, indicating that our model and thus our dataset is sufficient enough to perform well. The same comments can be made for the confusion matrix as well, since the misclassifications for both classes are a magnitude smaller than the right ones.

The class prediction error bar plot shown also below, strengthens even more the facts described above.

Fold	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.7619	0.7558	0.9341	0.7456	0.8293	0.4519	0.4844
1	0.8639	0.8609	0.9231	0.8660	0.8936	0.7055	0.7082
2	0.7755	0.7843	0.8791	0.7843	0.8290	0.5054	0.5124
3	0.7755	0.8052	0.8352	0.8085	0.8216	0.5191	0.5196
4	0.7415	0.7955	0.8901	0.7431	0.8100	0.4158	0.4327
5	0.7740	0.8158	0.8352	0.8085	0.8216	0.5134	0.5139
6	0.7808	0.8202	0.8681	0.7980	0.8316	0.5194	0.5232
7	0.8151	0.8430	0.9121	0.8137	0.8601	0.5900	0.5984
8	0.7877	0.8558	0.8901	0.7941	0.8394	0.5292	0.5368
9	0.7671	0.8482	0.8901	0.7714	0.8265	0.4778	0.4892
Mean	0.7843	0.8185	0.8857	0.7933	0.8363	0.5227	0.5319
Std	0.0319	0.0324	0.0315	0.0340	0.0228	0.0754	0.0710







Feature Importance

In order to study the most important features that contribute to the result, we will train a Random Forest classifier. First, we create a class that will help us LabelEncode our columns. As you can see from the name, it does support multi-column label encoding at once.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.pipeline import Pipeline
#Use this class to label encode multiple classes at once
class MultiColumnLabelEncoder:
      def __init__(self,columns = None):
      self.columns = columns # array of column names to encode

      def fit(self,X,y=None):
      return self # not relevant here

      def transform(self,X):
      '''
      Transforms columns of X specified in self.columns using
      LabelEncoder(). If no columns specified, transforms all
      columns in X.
      '''
      output = X.copy()
      if self.columns is not None:
            for col in self.columns:
                  output[col] =
LabelEncoder().fit_transform(output[col])
      else:
            for colname,col in output.iteritems():
                  output[colname] = LabelEncoder().fit_transform(col)
      return output

      def fit_transform(self,X,y=None):
      return self.fit(X,y).transform(X)
```

After we define the class, let's proceed to apply it to our dataframe with the columns we want, and after that, drop any unwanted data from our dataframe.

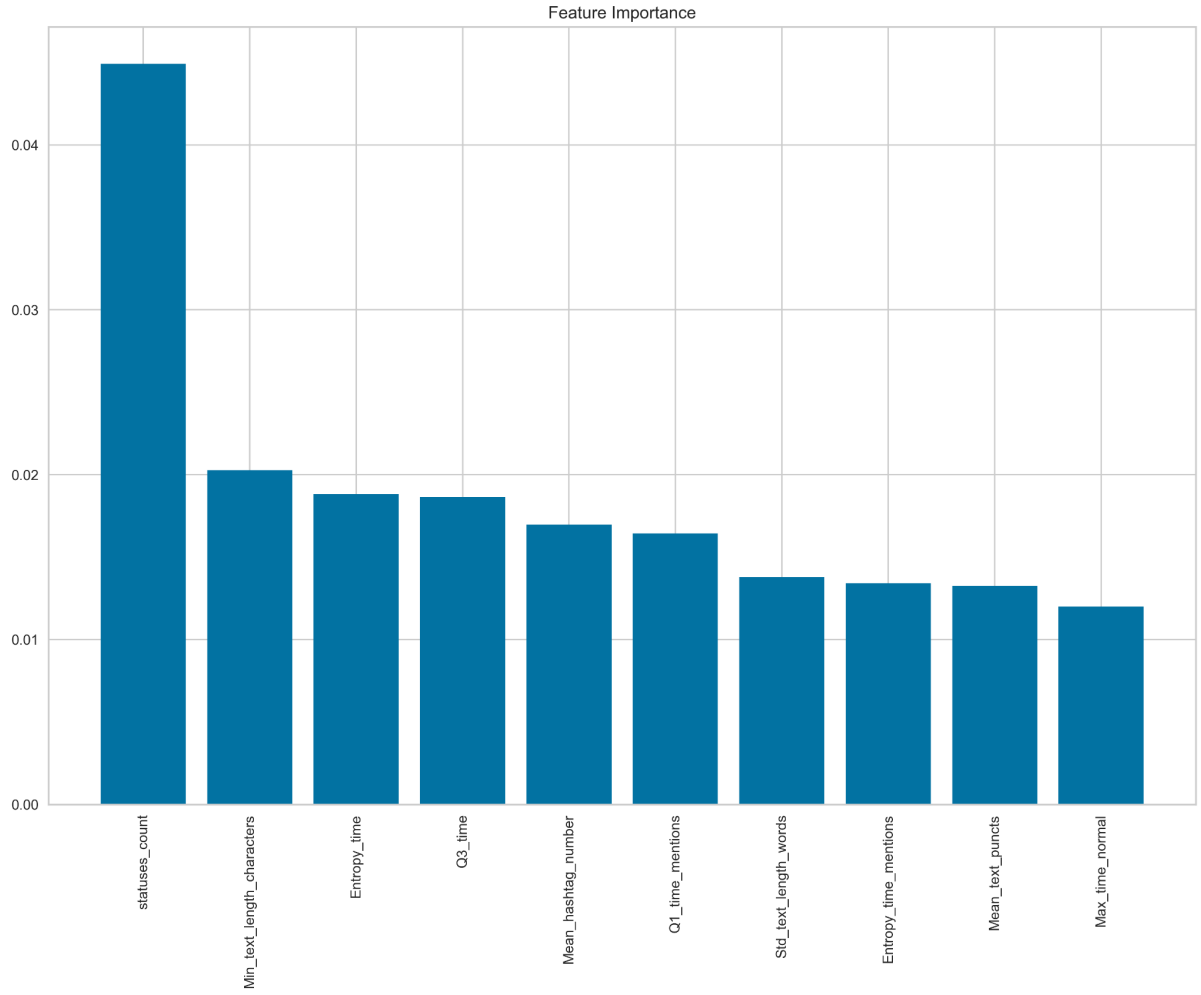
```
df_new = MultiColumnLabelEncoder(columns
=['protected','geo_enabled',"verified",'lang',

"is_translation_enabled","profile_background_tile",

"has_extended_profile","default_profile","following",

"follow_request_sent","translator_type",

"Uses_default_profile_image_url",

"not_using_unique_description",'status']).fit_transform(df1) #label
encode all of the selected columns
df_new.drop(columns=['Datetime'],inplace =True) #drop datetime since its
```

```
not used
import numpy as np
df_new = df_new[~df_new.isin([np.nan, np.inf, -np.inf]).any(1)] #drop
nan or inf values
```

Last but not least, we use the Random Forest Classifier. After being fit, the model provides a `feature_importances_` property that can be accessed to retrieve the relative importance scores for each input feature. So after we get the most important features, we proceed to plot them.

```
from sklearn.model_selection import train_test_split

df = df_new

# Create training and test split

X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, :-1],
df.iloc[:, -1:], test_size = 0.25, random_state=1)

# Training / Test Dataframe

from sklearn.ensemble import RandomForestClassifier

# Train the mode

forest = RandomForestClassifier()
forest.fit(X_train, y_train.values.ravel())

importances = forest.feature_importances_

# Sort the feature importance in descending order

sorted_indices = np.argsort(importances)[::-1]

import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
figure(figsize=(12, 10), dpi=300)
plt.title('Feature Importance')
plt.bar(range(10), importances[sorted_indices][0:10], align='center')
plt.xticks(range(10), X_train.columns[sorted_indices][0:10],
rotation=90)
plt.tight_layout()
plt.show()
```

Below we can see the plotted results.



Notice that the most important feature is the `statuses_count` followed by the `min_text_len_characters` in the user's tweets and after these, there are two more based on the time the user posted a tweet.

It is easy to understand why these features stand out from the rest. The more statuses someone posts, the more likely for him to be a bot. After all, humans have lives while bots don't. Moreover, the minimum characters in a tweet make someone less of a bot. Bots usually retweet a lot while humans are more likely to post something unique. The time features on the other hand are a bit tricky in the sense that both humans and bots have time patterns when they use twitter. But humans tend to stray away from those patterns based on situations in their lives, while bots are forced to follow them despite the situation.

Development of Unsupervised ML Models

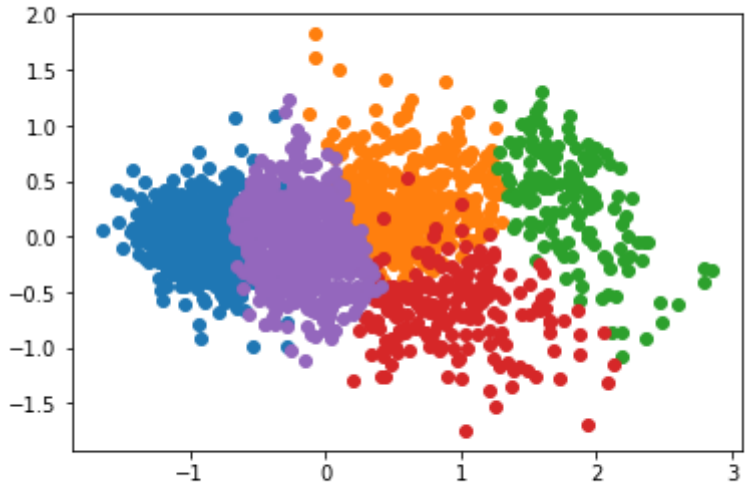
Now, we would like to group the users by their activity into five groups: those who are extremely active, very active, active, slightly active and inactive. In order to do that, we will develop an unsupervised Machine Learning model. The model that we chose is "K-means" since it is appropriate when someone wants to declare the exact number of groups.

First, we select the temporal features since only these features are related to the activity of the users. Next, we scale them using MinMaxScaler and apply Principal Component Analysis in order to reduce the dimensionality to 16. As it is shown in the results the percentage of variance explained by the selected components is 92.46%.

```
import pandas as pd
import numpy as np
from numpy import unique, where
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

#Feature Selection for grouping the users by activity (select only )
X = df_new[["Min_time", "Max_time","Mean_time",
            "Median_time", "Std_time", "Skew_time",
            "Kyrt_time", "Entropy_time", "Q1_time", "Q3_time",
            "Min_time_rt", "Max_time_rt","Mean_time_rt",
            "Median_time_rt", "Std_time_rt", "Skew_time_rt",
            "Kyrt_time_rt", "Entropy_time_rt", "Q1_time_rt",
"Q3_time_rt",
            "Min_time_normal", "Max_time_normal","Mean_time_normal",
            "Median_time_normal", "Std_time_normal", "Skew_time_normal",
            "Kyrt_time_normal", "Entropy_time_normal", "Q1_time_normal",
"Q3_time_normal",
            "Min_time_mentions",
"Max_time_mentions","Mean_time_mentions",
            "Median_time_mentions", "Std_time_mentions",
"Skew_time_mentions",
            "Kyrt_time_mentions", "Entropy_time_mentions",
"Q1_time_mentions", "Q3_time_mentions",
            "Min_hour", "Max_hour","Mean_hour",
            "Median_hour", "Std_hour", "Skew_hour",
            "Kyrt_hour", "Entropy_hour", "Q1_hour", "Q3_hour",
            "Min_day", "Max_day","Mean_day",
            "Median_day", "Std_day", "Skew_day",
            "Kyrt_day", "Entropy_day", "Q1_day", "Q3_day"]].values


#Preprocessing
#Scale the data
scaler = MinMaxScaler()
x = scaler.fit_transform(X)
```

```
#Apply principal component analysis
pca = PCA(n_components=16)
x = pca.fit_transform(x)
#Print the percentage of variance explained by the selected components.
print(sum(pca.explained_variance_ratio_))

#Apply k-means using 5 clusters
kmeans = KMeans(n_clusters = 5, init = 'k-means++', max_iter = 1000,
n_init = 100, random_state = 0)
kmeans.fit(x)
yhat = kmeans.predict(x)
# retrieve unique clusters
clusters = unique(yhat)
# create scatter plot for samples from each cluster
for cluster in clusters:
    # get row indexes for samples with this cluster
    row_ix = where(yhat == cluster)
    # create scatter of these samples
    plt.scatter(x[row_ix, 0], x[row_ix,1])

# show the plot
plt.show()
print(f'Silhouette Score: {silhouette_score(x, yhat,
metric="sqeuclidean")}')
```

```
0.924662916548505
```




```
Silhouette Score: 0.2749279241910388
```

We can detect clearly separable groups in the graph above. However, due to Principal Component Analysis that was applied, the result is not interpretable and we cannot

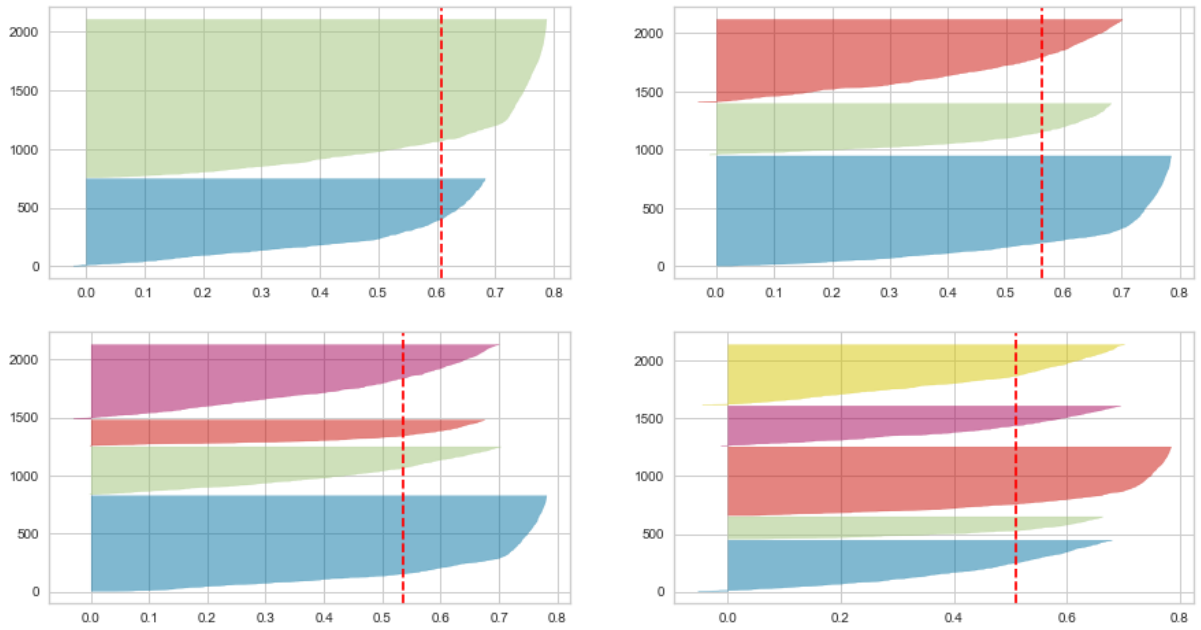
distinguish which group is the most active and which is not. Moreover, the Silhouette Score is low, only 0.2749.

Next, we will repeat the analysis but we will use only two of the temporal features and, more specifically, the mean and median time between consecutive posts.

```
#Select only features that refer to mean and median time between
consecutive posts
column_name = ['time']
X = df_new[["Mean_"+column_name[0], "Median_"+column_name[0]]].values

#Preprocessing - scale the data
scaler = MinMaxScaler()
x = scaler.fit_transform(X)
```

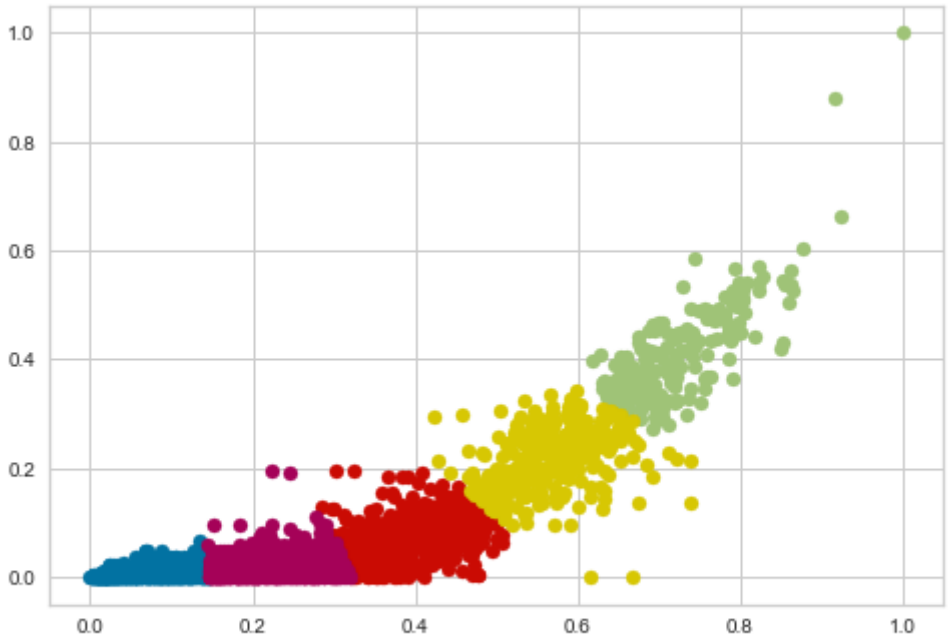
In order to define the ideal number of clusters, we perform the following analysis:

```
from yellowbrick.cluster import SilhouetteVisualizer

fig, ax = plt.subplots(2, 2, figsize=(15,8))
for i in [2, 3, 4, 5]:
      '''
      Create KMeans instance for different number of clusters
      '''
      km = KMeans(n_clusters=i, init='k-means++', n_init=10,
max_iter=100, random_state=42)
      q, mod = divmod(i, 2)
      '''
      Create SilhouetteVisualizer instance with KMeans instance
      Fit the visualizer
      '''
      visualizer = SilhouetteVisualizer(km, colors='yellowbrick',
ax=ax[q-1][mod])

      visualizer.fit(x)
```



It is evident that the ideal number of clusters is five. So, we will apply again K-means using five clusters.

```
kmeans = KMeans(n_clusters = 5, init = 'k-means++', max_iter = 1000,
n_init = 100, random_state = 0)
kmeans.fit(x)
yhat = kmeans.predict(x)
# retrieve unique clusters
clusters = unique(yhat)
# create scatter plot for samples from each cluster
for cluster in clusters:
    # get row indexes for samples with this cluster
    row_ix = where(yhat == cluster)
    # create scatter of these samples
    plt.scatter(x[row_ix, 0], x[row_ix,1])
# show the plot
plt.show()
print(f'Silhouette Score: {silhouette_score(x, yhat,
metric="sqeuclidean")}')
```




```
Silhouette Score: 0.6818398274130316
```

We observe that the Silhouette Score has been improved (0.6818) and, since the (normalized) features refer to the time between consecutive posts, we can distinguish the following groups:

- Blue: Extremely Active
- Maroon: Very Active
- Red: Active
- Yellow: Slightly Active
- Light Green: Inactive

Evaluation in Real Accounts

Now that our models are ready we can test them using some real accounts.

In this section, a comparison between 4 models (gbc, lightgbm, rf, ada) that had great results is going to be made with emphasis given to the fact that only real users' accounts are going to be tested. First, in a similar fashion seen before the appropriate data frame is created and the machine learning algorithms are declared.

```
#preprocess data from real accounts
test_data = pd.read_csv('final_features_real.csv')
test_data.drop(columns = {"Unnamed: 0.1",'Unnamed: 0'}, inplace =True)
new_test = pd.DataFrame(test_data, columns=range(164))
new_test.dropna(inplace=True)
test_data.columns=new_test.columns.values
```

```
#Create the models to test on the real accounts
gbc = create_model('gbc')
lightgbm = create_model('lightgbm')
rf = create_model('rf')
ada = create_model('ada')
```

With the above code some results about some important metrics are going to be returned just like in the development of the supervised models sections. These results are not presented here since the important comparison has to be made to the final overall metrics of the models coming from their predictions. These metrics are the Accuracy, Precision, Recall and F1 scores. The code for the evaluation of one of those models is presented below.

The status of the account (bot/human) was identified using **bot-detectivev2** and is presented in the following table.

Account	**Bot/Human**
@Aristoteleio	bot
@ylecun	bot
@nportokaloglou	human
@greeceinfigures	human
@big_ben_clock	human
@adonisgeorgiadi	human
@kostasvaxevanis	human
@vanitysthasmin	bot
@akurkov	human
@ve10ve_ghost	human
@adarshburman2	human
@bassaces1	bot

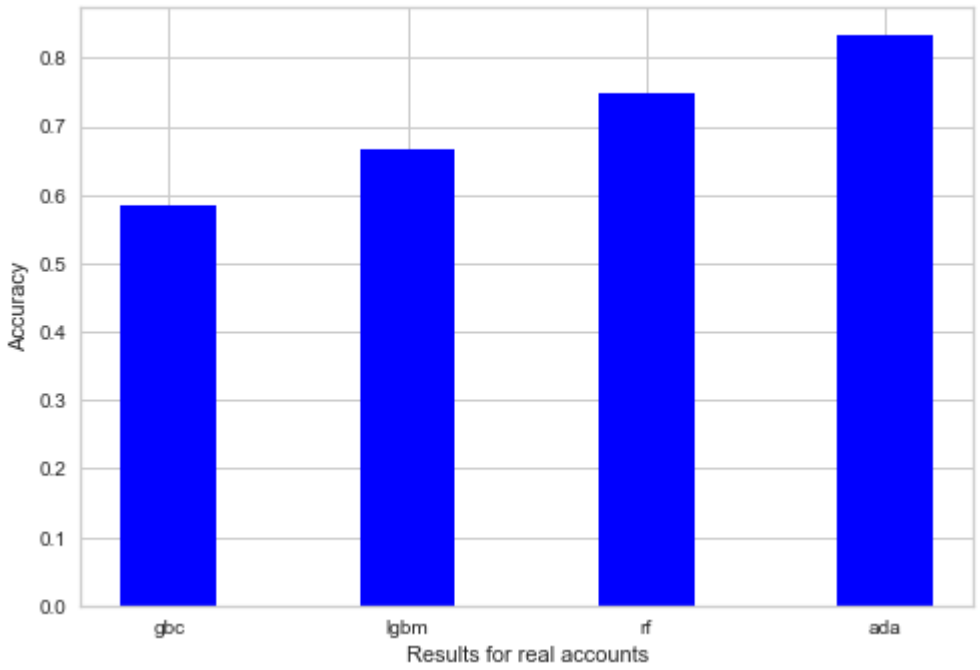
```
#get results from each model
from sklearn import preprocessing, metrics
status = [0,0,1,1,1,1,1,0,1,1,1,0]#1 human 0 bot

le = preprocessing.LabelEncoder()
#label encode the status
gbc_pred = predict_model(gbc,test_data)
gbc_pred
```

```
gbc_pred['Label'] = le.fit_transform(gbc_pred['Label'])

y_test = status
y_predicted = gbc_pred['Label']
#get the metrics
ac1= metrics.accuracy_score(y_test, y_predicted)
p1 = metrics.precision_score(y_test, y_predicted, average ='macro')
r1 = metrics.recall_score(y_test, y_predicted,average ='macro')
f1 = metrics.f1_score(y_test, y_predicted,average ='macro')
print("Accuracy",  ac1)
print("Precision", p1 )
print("Recall",r1)
print("F1", f1)
```

```
Accuracy 0.5833333333333334
Precision 0.6142857142857143
Recall 0.625
F1 0.5804195804195804
```

- Comparison of **Accuracy** scores:

```
# creating the bar plot
plt.bar(['gbc','lgbm','rf','ada'], [ac1,ac2,ac3,ac4], color ='blue',
        width = 0.4)

plt.xlabel("Results for real accounts")
plt.ylabel("Accuracy")
plt.show()
```

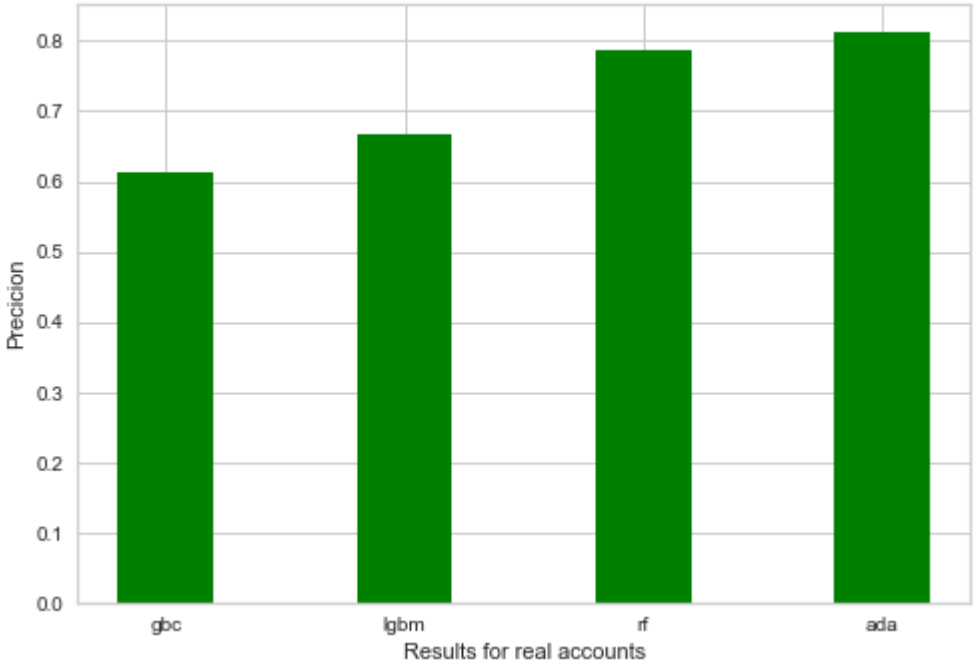


It can be commented that the ada model boosts the best accuracy score followed by the rf model.

- Comparison of **Precision** scores:

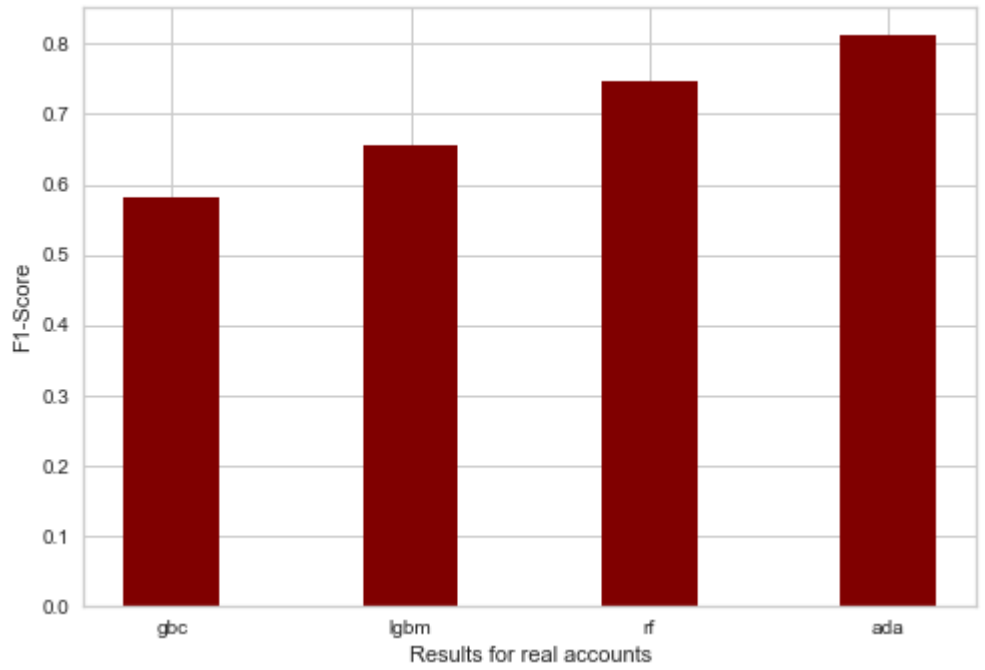
```
plt.bar(['gbc','lgbm','rf','ada'], [p1,p2,p3,p4], color ='green',
        width = 0.4)

plt.xlabel("Results for real accounts")
plt.ylabel("Precicion")
plt.show()
```

Again the ada model takes the lead followed by the rf model.



- Comparison of **F1** scores:

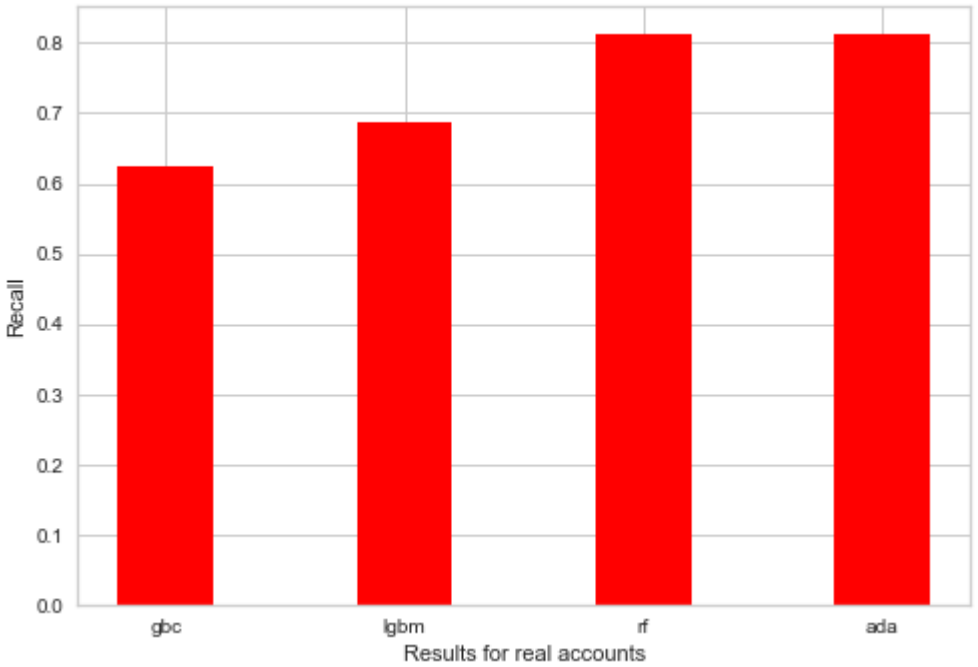
```
plt.bar(['gbc','lgbm','rf','ada'], [f1,f2,f3,f4], color ='maroon',
        width = 0.4)

plt.xlabel("Results for real accounts")
plt.ylabel("F1-Score")
plt.show()
```



The ada model once again outperforms the other models.

- Comparison of **Recall** scores:

```
plt.bar(['gbc','lgbm','rf','ada'], [r1,r2,r3,r4], color ='red',
        width = 0.4)

plt.xlabel("Results for real accounts")
plt.ylabel("Recall")
plt.show()
```



Overall the ada model boosts the best overall performance and thus it would be recommended over the others. The rf model came second in all the comparisons, except for the recall comparison, and thus would also be a strong model to be used.