POLITECNICO DI MILANO
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Detection and Classification of Harmful Bots in Human-Bot Interactions on Twitter

Supervisor: Prof. Florian Daniel
Co-supervisor: Prof. Cinzia Cappiello

M.Sc. Thesis by:
Lorenzo Cannone, matriculation number 875802
Matteo Di Pierro, matriculation number 876179

Academic Year 2017-2018

Abstract

This thesis describes a study and a web tool for the identification and classification of bots, according to the potential harm or threat they may cause to humans in online conversations. The focus is on Twitter, as social network platform, and bots are meant as algorithmically driven accounts that act like humans in interactions. The problem has so far been addressed with a special attention to the detection of such automated entities among legitimate ones. The method and tool described in this thesis propose a finer level of granularity and show that it is further possible to classify bots also into potential types of harm – bots with adult content (*Not Safe For Work*), bots propagating news and potential misinformation (*News-Spreaders*), bots that spam product sales or job offers (*Spam-Bots*) and bots who mimic interest (*Fake-Followers*) – in function of the content they share and their behaviour.

The methodology followed involves the creation of two datasets, the engineering of features and the selection of the models. Starting from binary-labelled (bot or not) accounts lists, the thesis shows how the training sets have been collected, following different approaches, yielding a stratified multi-class dataset. The work done focuses on different bot behaviours, and specific features have been crafted to capture those ways to act. The final solution involves a predictive pipeline algorithm, composed of a first binary machine learning classifier, aimed to detect bots among genuine accounts and a multi-class ensemble classifier, which goes deeper into the aforementioned categories and classifies the potential threat. Each model of the classifiers pool has been evaluated individually and inside the ensemble. The validation stages measured a 95% of AUC score for the binary classifier, and an average of 98% in F1 score, for the multi-class ensemble model.

The tool implementing the method, BotBuster, is a web application running the prediction script, that is accessible for free.

Sommario

In questa tesi è descritta una metodologia e uno strumento per l'identificazione e la classificazione di bot, in termini di potenziali comportamenti minacciosi o dannosi, nei confronti di umani, nel contesto delle interazioni sui social network. In questo specifico caso, l'attenzione è stata rivolta alla piattaforma online di Twitter, e i bot sono da considerarsi come account presenti sul social, gestiti da algoritmi che ne automatizzano le interazioni con gli umani. Il problema dei bot è, fino ad ora, stato affrontato con un'ottica improntata all'identificazione di quest'ultimi tra la totalità delle entità che popolano il web. Il metodo esposto, così come lo strumento proposto, puntano ad aggiungere un livello di profondità alle soluzioni fino ad ora esplorate, classificando la natura di questi bot e i possibili danni che possono recare ad utenti legittimi – bot che condividono contenuti per adulti (NSFW), bot che fanno propaganda politica, diffondendo notizie e possibile disinformazione (News-Spreaders), bot che propinano offerte di lavoro o vendite di prodotti di dubbia validità (Spam-Bots) e bot che simulano interesse per i contenuti condivisi da altri account (Fake-Followers) – in funzione dei contenuti da loro condivisi e dal loro comportamento.

Il metodo studiato prevede la costruzione di due dataset, un processo di creazione di attributi aggiuntivi e la selezione dei modelli di classificazione. Partendo da liste di utenti con target binario (bot o umani), la tesi mostra come siano stati creati i training set per gli algoritmi, seguendo diverse piste, portando alla luce dei dataset stratificati, binario e multi-classe. Il lavoro svolto si concentra sui diversi comportamenti imputabili ai bot, sono state quindi assemblate delle feature specifiche per distinguere tali modi di interagire. La soluzione finale prevede una pipeline predittiva, composta da un primo modello di apprendimento automatico in grado di classificare bot e umani, e da un ensemble di classificatori che si occupano di andare a fondo nella distinzione di quelle summenzionate categorie di bot e delle potenziali minacce che possono rappresentare. Ogni modello è stato valutato in maniera isolata e nell'insieme dell'ensemble. In fase di validazione sono

stati misurati un 95% di AUC score, nel modello binario, e una media del 98% per la metrica F1, nel modello multi-classe.

Lo strumento che implementa il metodo descritto, BotBuster, è una applicazione web che esegue lo script per la classificazione, ed è disponibile per chiunque su internet.

Contents

Chapter 1

Introduction

1.1 Context

Many technologists consider chatbots one of the hottest technologies in recent time (https://bit.ly/2od0Tdw), an opinion fueled for example by Facebook's release of its Messenger API in 2016. In April 2017, Facebook reported 100,000 monthly active bots on the Messenger platform. In March 2017 Varol et al. [1] estimated that between 9% and 15% of active Twitter accounts ar bots (29-49 million accounts out of 328 millions, https://bit.ly/2v3AT6O). Gartner estimates that by 2020 85% of customer requests will be handled by bots, while Inbenta estimates 1.8 billion unique customer chatbot users by 2021. [2]

The technological advancements of chatbots undoubtedly produced a hype on its own, yet bots today are by far not limited to instant messaging only. Bots permeate all kinds of on-line conversations in Twitter, Facebook, Instagram, Q&A sites, on-line newspapers, emails, and the like. They are everywhere where there are humans conversing with each other via the Internet, legitimately or illegitimately. For example, Messenger explicitly allows bots in its chats, while WhatsApp states that it blocks phone numbers generating bot traffic (https://bit.ly/2HhW9wG).

Inspired by Bessi and Ferrara, [3] we understand bots generically as algorithmically driven entities that on the surface act like legitimate human users in on-line conversations.

Daniel et al. [4] has started asking the question whether this increasing presence of bots may lead to harmful communication patterns that may hurt the human participant in the conversation, and we found a variety of anecdotal evidence that this may indeed happen. Of course, bots are not harmful in general. But sometimes, intentionally or unintentionally, software-driven conversations may just break common conversational rules, etichette, or even law. It is important to acknowledge the problem, so as to be able to provide countermeasures and to prevent people from getting hurt.

A **harm** occurs when someone suffers an injury or a damage, but also when someone gets exposed to a potential adverse effect or danger. In a prior work [4], we identified examples for the following types of harm caused by bots:

- *Psychological harm* occurs when someone's psychological health or wellbeing gets endangered or injured; it includes feelings like worry, depression, embarrassment, shame, guilt, anger, loss of self-confidence, or inadequacy. An example of a bot causing psychological harm is Boost Juice's Messenger bot that was meant as a funny channel to obtain discounts by mimicking a dating game with fruits but used language that was not appropriate for children (`http://bit.ly/2zvNt0E`).

- *Legal harm* occurs when someone becomes subject to law enforcement or prosecu-tion; it includes for example the breach of a confidentiality agreement or contract, the release of protected information, or threatening. A good example is the case of Jeffry van der Goot, a Dutch developer, who had to shut down his Twitter bot, which generated random posts, after the bot sent out death threats to other users (`http://bit.ly/2Dfm71P`).

- *Economic harm* occurs when someone incurs in monetary cost or looses time that could have been spent differently, e.g., due to the need to pay a lawyer or to clean one's own social profile. A concrete example of an infiltration by a bot happened on Reddit in 2014, where the bot wise shibe provided automated answers and users rewarded the bot with tips in the digital currency dodgecoin, convinced they were tipping a real user (`http://bit.ly/2zu2b6r`).

- *Social harm* occurs when someone's image or standing in a community gets affect-ed negatively, e.g., due to the publication of confidential and private information like a disease. An example of a bot causing social harm was documented by Jason Slotkin whose Twitter identity was cloned by a bot, confusing friends and followers (`http://bit.ly/2Dfq4DH`).

- *Democratic harm* occurs when democratic rules and principles are undermined and society as a whole suffers negative consequences, e.g., due to fake news or the spreading of misinformation. Bessi and Ferrara [3], for instance, showed that bots were pervasively present and active in the on-line political discussion about the 2016 U.S. Presidential election (predating Robert S. Mueller III's investigation into the so-called Russian meddling).

These types of harm may happen while bots perform **actions**, such as posting a message or commenting a message by someone else, that are not harmful per se and that also human users would perform. What needs to happen is the verification of some condition of inappropriateness, which turns the action into an **abuse**. Abuses that can be found are, disclosing sensitive facts, denigrating, being grossly offensive, being indecent or obscene, be threatening, make false allegations, deceive users, spam, spread misinformation, mimic interest, clone profiles, and invade spaces that are not meant for bots [4]. Some of these may be subject to legal prosecution (e.g., threatening people), others only breach moral, ethical or social norms, yet they still may be harmful to unprepared, human users.

1.2 Problem Statement and Contributions

In this thesis, we focus on **Twitter** as online communication environment, as Twitter is freely accessible through its open API (`https://developer.twitter.com`) and does not require special friend privileges to access content of users. We specifically study content shared via *tweets*, where a tweet t is a short text message of up to 280 characters, including URLs to external content or images, mentions (annotated references of other Twitter accounts), and hashtags. We neglect direct messages (similar to chats) between users, as these are private and not available for inspection.

Bots manifest themselves in Twitter as regular users: they have an own account and user profile and are able to tweet (post a message), re-tweet (forward a tweet), reply to tweets, like tweets. Through the API of Twitter, it is possible to retrieve all tweets of a given account, that is the tweet timeline of the account, for inspection. For the purpose of this thesis, we focus on the tweets produced (tweets, re-tweets, replies) and interpret a generic user as a tuple $u = \langle P, T \rangle$ where P is the set of user profile parameters (e.g., name, bio, location) and $T = [t_i, ..., t_n]$ is the chronologically ordered sequence of tweets.

Out of the different types of **abuses** listed above, in this thesis we concentrate on: *Fake-Followers* that follow a user to pretend interest in the user; bots that spread *adult content* like pornographic material or lures

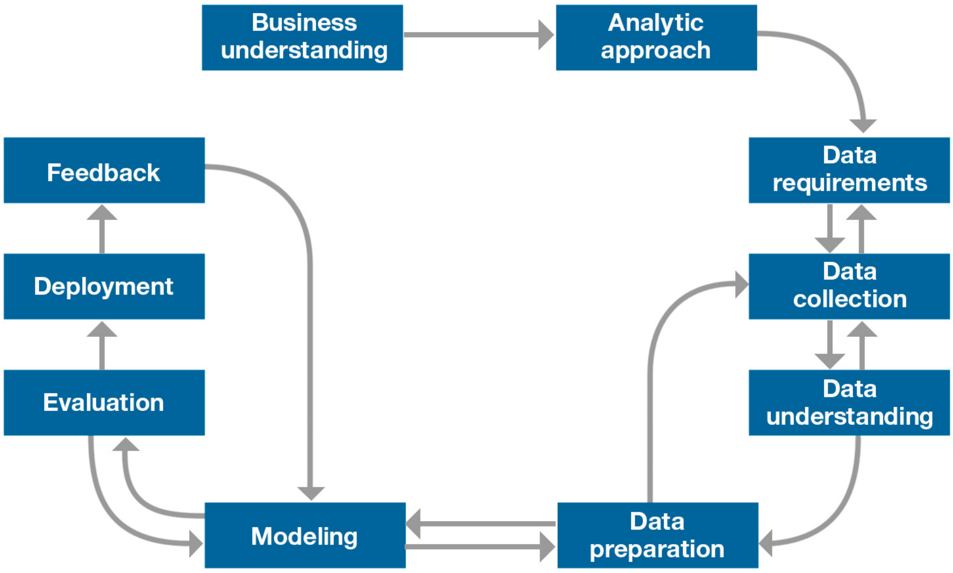


Figure 1.1: Foundational Methodology for Data Science by IBM

(Not Safe For Work); bots propagating news and potential *misinformation*, i.e., high-frequency publishers of news of ambiguous quality; and *Spam-Bots* distributing commercial advertisements. The selection of these four abuse types is motivated by the common method we propose for their analysis, i.e., feature-based classification.

If we call the four classes of bots *NSFW, News-Spreaders, Spam-Bots, Fake-Followers*, respectively, and use the class *Genuine* for genuine accounts, the **problem** we address in this thesis is how to classify a given account u by analysing the account's profile P and tweet timeline T: $u = \langle P, T \rangle \mapsto \{NSFW, NS, SB, FF, GEN\}$.

The thesis follows the typical steps of the Data Science methodology outlined in Figure 1.1 and makes the following **contributions**:

- We construct and share *four labeled datasets* containing NSFW, News-Spreaders, Spam-Bots and Fake-Followers accounts in Twitter.
- We train and study different *classification algorithms* for the categorization of accounts into the five classes of harm identified by our dataset.
- We compare the *performance* (precision, recall, f-measure) of the methods against that of a baseline classifier that uses only simple profile data for classification and demonstrate the *viability* of the approach.

- We compare different *ensemble methods* in order to combine the best classifiers and compute the final predictions over users.

- We describe two *applications* of the best method: an implementation of a harmful bot detection tool, BotBuster, whose performance we compare with that of our reference state-of-the-art tool, i.e., Botometer (`https://botometer.iuni.iu.edu`); and an estimation of the global bot ecosystem of Twitter, starting from a random sampling of 12616 accounts.

- BotBuster is accessible for free at `http://botbuster.it` for testing.

1.3 Structure of Thesis

The thesis has the following structure:

- Chapter 2 introduces the state of the art.

- Chapter 3 explores existing datasets and tools for data collection. Then it shows how the data were gathered and it ends with the *creation of our datasets.*

- Chapter 4 explains the features engineering step. Here we create new features based on users and tweets, that allow the classifiers to better understand the users' behaviours. At the end of this chapter we will have two *feature vectors*: one for the binary problem (bot-or-not) and one for the multi-class problem.

- Chapter 5 compares different classifiers for both the *binary* and the *multi-class* problems and performs the tuning of the parameters. In the end we will have three bot-behaviour-specific classifiers and a bot-or-not classifier.

- Chapter 6 identifies the best way to combine the three multi-class classifiers and performs the tuning of the ensemble parameters. Then it explains the final *prediction pipeline* that, starting from a user ID, returns the probabilities of membership for each class (bot, genuine) and bots sub-class (NSFW, News-Spreaders, Spam-Bots, Fake-Followers).

- Chapter 7 describes the development of the *web application (BotBuster).* This tool provides the account classification service, according to the *prediction pipeline.*

- Chapter 8 draws conclusions, examines possible future works and estimates the global distribution of users on Twitter, by classifying randomly sampled users with our tool.

Chapter 2

State of the Art

In telling bots and humans apart, lots of effort have been spent to find the most accurate detection system. In the past years, several interesting approaches have been tested to build bot classifiers, most of these strategies involved supervised machine learning algorithms, to highlight the hidden patterns characterizing automated behaviours. Although, some researches relied on other methods, such as graph-based inspections, human annotators and stacked models on different tiered data. All these approaches share the goal of better identifying algorithmically-driven entities on social networks.

2.1 Feature-based approaches

The work that is most closely related to this thesis is *Botometer*, formerly known as BotOrNot [5, 6], an online tool that computes a bot-likelihood score for Twitter accounts and allows one to *tell bots and genuine user accounts apart*. The tool builds on more 1000 features among network, user, friends, temporal, content and sentiment features, and uses a random forest classifier for each subset of features. The training data used is based on bot accounts collected in prior work by Lee et al. [7], who used several Twitter honeypots to lure bots and collected about 36,000 candidate bot entities following or messaging their honeypot accounts. Botometer is a machine learning classifier system based on engineered features of heterogeneous nature. It considers attributes stemming from sentiment analysis, timing, user's network, user content and meta-data. This feature-based approach led to an overall 0.94 score in AUC metric, measured with the Random Forest algorithm on a merged dataset, composed of the 36K aforementioned accounts and 3,000 new samples, manually annotated. The goal of BotBuster is to build on the results of Botometer and not only to tell bots and humans apart, but also to distinguish different types of bots based on their potential of harming the people they interact with.

Cresci et al. [8] specifically focused on the problem of *fake followers*. They constructed a dataset of human accounts (manually and by invitation of friends) and bought fake followers from online services like `http://fastfollowerz.com`. The work compares two types of classifiers, classifiers based on expert-defined rules and feature-based classifiers (machine learning), and shows (i) that fake followers can indeed be spotted and (ii) that black-box, feature-based classifiers perform better than white-box, rule-based classifiers. In addition, the work also produced a publicly available, labelled dataset that can be used for research purposes.

Other features-based studies took Botometer as a baseline, like the work on *Bot-Hunter* [9]. In terms of *datasets* to be analysed in the bot detection process, Beskow and Carley [9] specifically propose four *tiers* of data for the classification of Twitter accounts: single tweet text (tier 0), account + one tweet (1), account + full timeline (2), and account + timeline + friends timelines (3). They focused on the training set quality, recognizing the magnitude of this component inside a data science project, especially when data doesn't exist and must be collected with tricky methods. Their dataset was composed of 19,221 accounts labelled as bots, collected with event-based methods. The humans were gathered by Twitter APIs, retrieving *normal* data. They collected 70,000 random users and sampled from them. Past researches [1] asses that about 8% of the Twitter population is automated. Beskow and Carley thought that the possible percentage of bots included in the human set was an acceptable amount of noisy data. With their tool Bot-Hunter, they however only study different machine learning techniques for tier-1, with performances comparable to similar models, as Botometer.

2.2 Network-based approaches

The problem of telling bots and humans apart has been investigated already before Botometer. For instance, Ratkiewicz et al. [10] studied the phenomenon of *astroturfing*, i.e., political campaigns that aim to fake social support from people for a cause, and showed that bots play a major role in astroturfing activities in Twitter. This approach differs from the aforementioned ones, due to the graph-based structure involved to analyse the network. In order to represent the information flow on the social network, the researchers built a directed graph, in which nodes represent single accounts and edges stand for interactions, such as retweets or quotes. Edges are incrementally weighted, as new interactions between the same pair of users are observed. This research aimed to exploit a network-basd system to build a following feature-based classifier. The network-based component's goal was to detect relevant memes, and then build some features on them; while the classifier, named *Truthy*, allows users to visualize memes and their related meta-data. As application of that system, the research led to a binary classifier built to label legitimate and truthy memes. The final model was chosen with the AdaBoost algorithm, 366 manually labelled memes, from which has been performed resampling, in order to balance the unbalanced dataset. The performance hit was 0.99 of AUC score.

Another network-based study is the one brought out by Cao et al. [11] describe *SybilRank*, a tool for the detection of sybil accounts (bots) in social networks that analyzes the social graph to compute sybil-likelihood scores. Also differently from the other approached, the work studies the social graph of Facebook, not that of Twitter. The study highlights how, by that time, the feature-based approaches with machine learning models failed to be effective in social network intrusion detections [12]. Cao et al. provided an useful tool for profiles ranking, based on the likelihood of an user to be a fake account, that can facilitate the manual verification of such users and the eventual suspension. It works on large scale, with a computational cost of $O(nlog(n))$.

A side project by the OSoMe (http://osome.iuni.iu.edu) researchers, which is also graph-based, is Hoaxy [13, 14, 15]. Hoaxy is a web tool whose purpose is to show the spreading of misinformation through Twitter contents and spreaders. It performs queries via APIs and follows the graph of interactions flow, for a required topic or keyword. Interactions are meant as retweets, quotes and direct tweets. Every visualized spreader is given a colour, based on the Botometer score. A completed trusted profile si shown as a blue dot and a alleged bot is represented with a red circle. The diffusion network is also displayed as a graph, which has users on its nodes and interactions on its edges.

2.3 Human-based approaches

Most of the aforementioned works relied on a partial manual labelling process, in order to enhance their data and to asses the value of the automate classifiers. Cao et al. describe the human annotation ineffective, when it comes to large scales evaluations, due to the effort required for a single classification [11].

The research made by Cresci et al. [16] highlighted the lacking of capability, by crowdsourcing annotators, to distinguish new waves of spambots from legitimate users. Indeed, the manual labelling of such bots has been performed with an accuracy that measured less than 24%, with more than 1000 accounts misclassified as humans (false negatives). This experiment shows like some particular social bots are hard to detect, even by humans, making the automatic classification models more challenging to build.

However, when it comes to more traditional social bots, manual annotation is still trusted. Going beyond merely telling bots and humans apart, human-based methods are often used as ground-truth, like with the work done by Chu et al. [17]. They coined the term *cyborg* to refer to bot-assisted humans in social networks and used a manually labelled dataset of 6000 randomly sampled Twitter accounts and a random forest classifier plus entropy measures to classify accounts into bots, cyborgs and humans.

Another finer-grained classification of bots is proposed by Varol et al. [1], who however propose a bottom-up approach to the identification of bots with similar online behaviour. The classifier used is the one adopted by Botometer, while the dataset used also included a manually annotated collection of Twitter accounts. After classifying users into bot or not, the authors further clustered the bot accounts into three types of bots: *spammers*, *self promoters*, and accounts that *post content from applications*.

2.4 Summary

We learned that human-based approaches are expensive, in terms of time and individuals involved, and often don't reach the desired performance, due to the quality of new waves of social bots. Graph-based methods are good for large scale evaluations, using low computational complexities, but are usually build to get insights for moderators, as well as features for automatic classifiers. We are beginning our work without a trusted multi-class dataset, it is reasonable to think that we won't need a graph-based structure to process our informations. The feature-based methods seem to be more suitable to infer over bots and humans, even if the newborn content polluters are harder to detect, making the work more challenging. We face the problem with an automatic classification system, based on engineered features.

In this thesis, we study different feature-based analysis techniques with the specific aim of distinguishing bots based on the potential harm they may cause in Twitter. The datasets we use can be understood as tier-2 datasets (account + full timeline of tweets), using the terminology by Beskow and Carley [9]. It is important to note that the work focuses on bots that interact with humans in online conversations; bots used for cyber attacks or for the automation of generic tasks are out of the scope. Our goal is to propose an online tool, as it was done with Botometer, that provides a deeper classification for the examined accounts. Acknowledging the works done before ours, we will start building the datasets following different leads and drawing by the available data that the previous researches collected. The methodology will be the same as the ones described in this chapter, sticking to the modus operandi for data science projects. By studying the proposed papers, we learned that some models fit this kind of study better than others. We will try several classifiers and see if our needs will be fulfilled by the same algorithms seen in the state of art.

We hope that this thesis will find its space among the discoveries found in telling bots and humans apart, adding new points of view in this research.

Chapter 3

Data Collection

In this chapter we present all the available datasets containing bot accounts, together with all the tools and methodologies used to collect the respective data. The final dataset contains:

- Data from already existing datasets
- Data collected with different automated approaches
- Hand-labelled data

3.1 Tools

Different tools were used in order to both collect the data and to enrich existing datasets. This stage was essential to gather additional features. Here we present all the instruments involved in this task.

3.1.1 Tweepy

Tweepy is a Python wrapper for the Twitter API. Two main methods were used to collect all the default features of users and tweets:

Method	Input	Output
API.get_user	[id/user_id/screen_name]	User object
API.user_timeline	[user_id][, count]	list of Status object

A User object contains all that features that describe the user's profile and his utilization of Twitter. A Status object contains the details of a single tweet, such as the full text, number of retweets and replies.

3.1.2 Botometer

Botometer [5] checks the activity of a Twitter account and gives it a score based on how likely the account is to be a bot.

Authors et al [5] tested most classification algorithm with their dataset and highlighted Random Forest as their final classifier, since it gained the highest accuracy. Its accuracy was 98.42% and 0.984 F1 measure [7].

Botometer provides an API to check Twitter accounts authenticity.

3.1.3 Hoaxy

Hoaxy is a tool that visualizes the spreading of articles online. Articles can be found on Twitter, or other sources of misinformation and fact-checking.

For each fake news, Hoaxy plot a graph showing the users who tweeted the news on the nodes and the retweet activity on the edges. In this graph it is possible to figure out who invented the news and how the news spread.

Hoaxy only checks the news sources and compares them with a list of unreliable URLs.

3.2 Datasets

3.2.1 Caverlee-2011

The dataset et al. [7] is composed by content polluters, detected by a system composed of 60 social honeypots, which are Twitter accounts created to serve the purpose of tempting bots; and genuine accounts, randomly sampled from Twitter. The authors observed the content polluters that used to interact with their honeypots, during a 7-months time-span. The accounts that weren't deleted, by the policy terms of the social platform, were clustered with the Expectation-Maximization algorithm for soft clustering. At the end of this process, the authors found nine different clusters, which were grouped in four main categories:

Cluster	Description
Duplicate Spammers	Accounts that post nearly the same tweets with or without links
Duplicate @ Spammers	Similar to the Duplicate Spammers, but they also use Twitter's @ mechanism by randomly including a genuine account's username
Malicious Promoters	These bots post tweets about marketing, business, and so on
Friend Infiltrators	Their profiles and tweets seem legitimate, but they mimic the mutual interest in following relationships

Each cluster has been manually checked, in order to asses its credibility. Finally, 22,223 bots were included in the dataset. For each content polluter, the authors collect the 200 most recent tweets, following and follower graph, and the temporal and historical profile information.

In order to collect genuine users too, 19,297 Twitter ids were randomly sampled and monitored for three months, to see if they were still active and not suspended by the social platform moderation service.

The authors subsequently built a classifier framework trained on their dataset, which uses crafted features grouped in four clusters: User Demographics (**UD**), User Friendship Networks (**UFN**), User Content (**UC**) and User History (**UH**). After testing several classification algorithms, the most performing algorithm was Random Forest with boosting sampling, which accomplished 98.62% of accuracy, 0.986 of F1-measure and 0.995 in the AUC score.

3.2.2 Cresci-2017

Cresci Dataset et al. [16] is composed of genuine accounts and bots. In this dataset there is a further differentiation of bots in sub-classes, which are precisely marked according to different categories.

Dataset	Description	#Users	#Tweets	Year
genuine accounts	verified accounts that are human-operated	3,474	8,377,522	2011
social Spam-Bots #1	retweeters of an Italian political candidate	991	1,610,176	2012
social Spam-Bots #2	spammers of paid apps for mobile devices	3,457	428,542	2014
social Spam-Bots #3	spammers of products on sale at Amazon	464	1,418,626	2011
traditional Spam-Bots #1	training set of spammers used by Yang [18]	1,000	145,094	2009
traditional Spam-Bots #2	spammers of scam URLs	100	74,957	2014
traditional Spam-Bots #3	automated accounts spamming job offers	433	5,794,931	2013
traditional Spam-Bots #4	automated accounts spamming job offers	1,128	133,311	2009
Fake-Followers	accounts inflating followers of other accounts	3,351	196,027	2012

- **Genuine accounts** are those users who correctly answered to a simple question, posed in natural language, so they represent accounts with no automation.

- During Rome majoral election in 2014, one of the candidates used a set of automated accounts to publicize his policies. These accounts were gathered to be part of the **Social Spam-Bots #1**.

- **Social Spam-Bots #2** are accounts that promotes mobile app, using popular hashtags for months.

- **Social Spam-Bots #3** promotes products on sale on Amazon, by tweeting products URL and descriptions.

All these accounts were manually checked to verify their automated nature.

- The **Traditional Spam-Bots #1** dataset is the training set used in [18].

- **traditional Spam-Bots #2** are users that mention other users in tweets containing scam URLs. They usually invite users to claim a prize.

- **Traditional Spam-Bots #3** and **traditional Spam-Bots #4** are bots that continuously tweet job offers.

- **Fake-Followers** are account involved in increasing popularity of other users. In order to collect them, Cresci et al. [16] bought followers from fastfollowerz.com, intertwitter.com and twittertechnology.com.

The intent of this project was to find a methodology useful to detect sophisticated Spam-Bots on Twitter. This novel category differs from the traditional Spam-Bot type, due to the ability of those content polluters to mimic the human interactions on the social platform. The authors et al. [16] relied on both crowdsourcing and machine learning experiments to compare the accuracy of the detection of such Spam-Bots. The experiments highlighted the difficulty of detecting this new wave of Spam-Bots, even with the help of human annotators.

3.2.3 Varol-2017

The Varol dataset et al. [1] contains a list of Twitter accounts, labeled as bots (1) or humans (0).

The construction of this dataset starts with the identification of a representative sample of users, by monitoring a Twitter stream for 3 months, starting in October 2015. Thanks to this approach it is possible to collect data without bias; in fact other methods like snowball (a technique that nominee new samples, starting from the social networks of an initial pool of users) or breadth-first (a graph exploration technique) need an initial users set. During the observation window about 14 million user accounts were gathered. All the collected users had to have at least 200 tweets in total and 90 tweets during the three month observation (about one tweet per day).

Using the classifier trained on the honeypot dataset in [7], the authors et al. [1] computed the classification scores for each of the active accounts, obtaining 0.85 in the AUC metric score, a lower score then the one obtained, by the same model, on the honeypot's data (0.95 AUC). This difference was justified by the lower ages of the manually annotated bots, with respect to the ones collected by the social honeypots. Then the samples were grouped by their score and 300 accounts from each bot-score decile were randomly selected. The 3,000 extracted accounts were manually labeled by volunteers. The authors analyzed users profile, friends, tweets, retweets and interactions with other users. Then a label was assigned to each user. Of course the final decision is conditioned to personal opinion.

3.2.4 BotBlock

Botblock (https://github.com/dansarie/Botblock) is a Twitter block list (on Twitter users can use these lists to block any interaction with the accounts listed) containing the user ids of a large number of known porn bot accounts. They are mainly used to aggressively market porn sites.

3.3 Unsupervised labeling of datasets

At first try, we wanted to understand if different kinds of bots were easy to distinguish, using their profile features only. We didn't know what kinds of bots populate Twitter for the most. So, an unsupervised approach could have helped us to highlight different categories. We relied expectations on clustering techniques, hoping to get a solid help in automatizing the labeling process of the data.

We used the Varol dataset [1], that contains a plain list of bots and humans.

It was not possible to use all the data, because we needed to scrape from the web all the possible features and some of the listed accounts were already been deleted. So, for this work, we had to consider only those accounts that were still active.
The first step was the knee-elbow analysis based on a hierarchical clustering with single linkage and euclidean distance. We used this technique in order to determine the optimal number of clusters. The data must had been preprocessed and cleaned.
Here we illustrate what kind of pre-process operations were performed for that purpose; features marked with '—' don't need to be preprocessed:

feature	type	preprocess operation
id	int	delete
name	str	replace with len(name)
screen_name	str	replace with len(screen_name)
statuses_count	int	—
followers_count	int	—
friends_count	int	—
favourites_count	int	—
listed_count	int	—
url	str	replace with hasUrl (0/1)
lang	str	one hot encoding
time_zone	str	one hot encoding
location	str	one hot encoding
default_profile	int	replace with hasDefaultProfile (0/1)
default_profile_image	boolean	boolean to int (0/1)
geo_enabled	boolean	boolean to int (0/1)
profile_image_url	str	delete
profile_use_background_image	boolean	boolean to int (0/1)
profile_background_image_url_https	str	delete
profile_text_color	str	delete
profile_image_url_https	str	delete
profile_sidebar_border_color	str	delete
profile_background_tile	boolean	boolean to int (0/1)
profile_sidebar_fill_color	str	delete
profile_background_image_url	str	delete
profile_background_color	str	delete
profile_link_color	str	delete
utc_offset	int	delete
is_translator	boolean	boolean to int (0/1)
follow_request_sent	int	delete
protected	boolean	boolean to int (0/1)
verified	boolean	boolean to int (0/1)
notifications	boolean	delete
description	str	replace with hasDescription (0/1)
contributors_enabled	boolean	boolean to int (0/1)
following	boolean	delete
created_at	str	string to int (year)

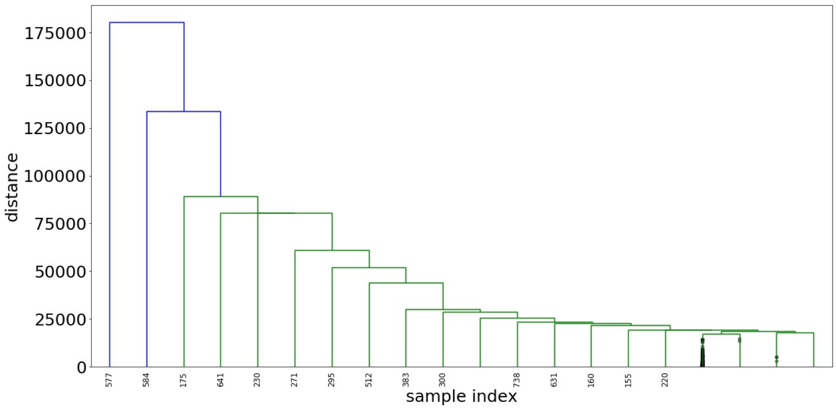


Figure 3.1: Hierarchical Clustering Dendrogram (truncated to the last 20 merged clusters)

Since we didn't know how many categories of bots were listed in this dataset, the first step consisted in understanding which was the optimal number of clusters to look for. To achieve that goal, we applied *hierarchical clustering*. This is a method of cluster analysis which build a hierarchy of clusters. The results of hierarchical clustering are usually presented in a dendrogram, that illustrates the arrangement of the clusters produced. In figure (3.1) you can see the dendogram of the algorithm.

In order to select the optimal number of clusters, we plotted the knee-elbow figure (3.2). It shows the variation of WSS (within cluster sum of squares) and BSS (between cluster sum of squares) as the number of clusters increase.

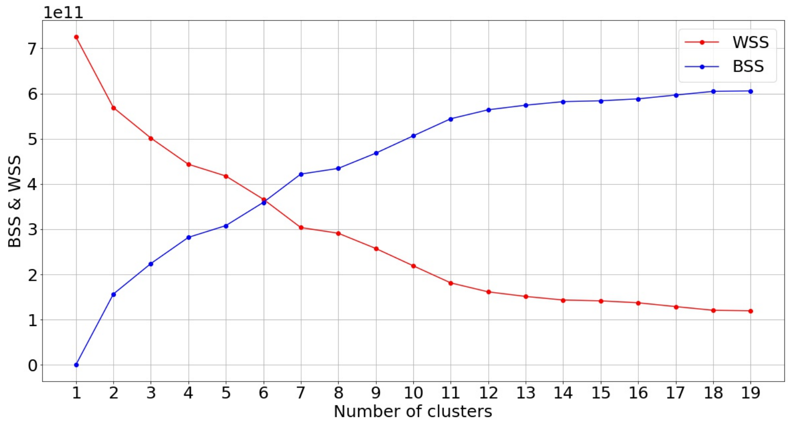


Figure 3.2: Hierarchical Clustering - knee-elbow

It is clear that there are no well-defined elbows or knees, both curves seems to be "smooth", so it were hard to pick a reasonable k (number of clusters) for the algorithms to come.

Then we tried another approach. We applied the *K-means* algorithm and we plotted the elbow method (3.3). In this figure there is an elbow between k=3 and k=4, so the most accurate solution were represented by four clusters.

As this process came to an end, we manually inspected the resulting clusters.

cluster	size
cluster 1	82
cluster 2	648
cluster 3	2
cluster 4	16

Cluster 2 contains most of the samples, while the others have fewer elements. We observed the Twitter profile of all the elements belonging to cluster 1, 3, 4 and a small sample of profiles for cluster 2.

Unfortunately there was no behavioural correlation among accounts in the same cluster, they just had similar values in the default features (i.e. number of followers, number of tweets, etc). So this technique didn't seem to fit the speeding up of the labeling process, nor to create useful features for a classifier.

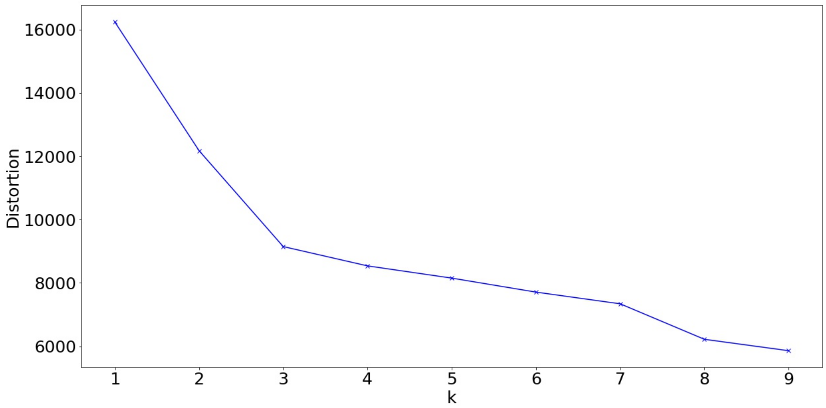


Figure 3.3: The Elbow Method showing the optimal k

3.4 Supervised labeling of datasets

The clustering approach did not help us, but the manual inspection of the clusters allowed us to get in touch with some existing bots, making us understand which categories of bots are most common on the social network. In particular we detected 4 main classes:

- NSFW bots
- News-Spreaders
- Spam-Bots
- Fake-Followers

We started with a hand-labeling of the Varol dataset [1]. For each account we analyzed its profile and tweets and we assigned it a label according to the categories we identified. We have faced some unexpected behaviors among bot accounts, that didn't fit the above-mentioned categories. For example, we found harmless bots, whose goal was to improve their ability of formulating sentences. In these cases, they were temporarily signed as "general purpose".

We also found genuine users, who we thought had been incorrectly added to the dataset. This task resulted into the collection of the following bots:

category	labeled account
NSFW	31
News-Spreader	71
Spam-Bots	418
Fake-Followers	5
general purpose	63
genuine	104

"General purpose" accounts are sometimes bots with no goal, they aim to emulate human behavior and often they were recognizable just because their description informs other users about their own nature.

Sixtythree users were not enough to represent a class and it was not possible to find a large list of those accounts who act like them, so we added all these ids to the "genuine" group. Even if this choice brought some noise to our data (since we added some bots in a humans dataset), it allowed us to provide our data more heterogeneity.

"NSFW" accounts are only thirtyone elements, anyway the problem of pornography is a known issue on Twitter. In [1] they clusterize users too. "These bot clusters exhibit some prominent properties: cluster C0, for example, consists of legit-looking accounts that are promoting themselves (recruiters, porn actresses, etc.)" [1]. This kind of users are often banned by Twitter, so it is likely that the accounts that we were not able to scrape, used to belong to this category. Therefore we firmly believed that obtaining further accounts of this class was fundamental.

Finally it is clear that even Fake-Followers were few, since they were not considered in the research by Varol, but they are important in the Cresci one [16], so we decided to expand this category too.

All these samples were not enough to train a classifier, hence we needed to collect more data. We performed this task by focusing on one category at a time.

3.4.1 NSFW

Not safe for work is a tag used on internet to mark all that URLs, web pages, e-mails that contain nudity, intense sexuality, profanity or violence. In particular we wanted to collect a specific sub-category: the pornbots. In order to collect them, we used the BotBlock dataset.

BotBlock contains thousands of pornbot ids. We wanted to gather about 6,000 samples, an amount that would have been enough for the final dataset, with regards to its balance. Since they were sorted according to their creation date, we shuffled the whole list. Then, using the Twitter API, we looked for accounts that weren't deleted yet. We needed to scrape profile features and tweets, so we couldn't consider banned accounts. The user list was initially shuffled to allow us to collect users with different ages. This emerged as a very useful step, because we gathered both more long-lived accounts and more explicit accounts (which probably have shorter lives). We finally obtained 6,605 accounts and 198,378 tweets.

3.4.2 News-Spreaders

Many bots on Twitter are News-Spreaders. The goal of these users is to spread politics, sports or other news. Often their behavior is not harmful, they just retweet statuses from newspapers accounts. However, there are users created to diffuse fake news. In the last few years Twitter has been used to boost politics propaganda. During elections or political campaigns, ad hoc accounts are created to divulgate specific political idea.

As a recent study highlighted, about 80% of these "pre-elections bots" are still alive [19]. It is possible that some of them have been included in our News-Spreader dataset.

We started gathering these ids by exploring Hoaxy. We used two different approaches. The first way (in figure 3.4) consisted in collecting the twenty top popular fake news, for each month, in the last two years. We performed this task using the Hoaxy APIs. Thanks to this service, we obtained all the tweet ids that have spread the considered claim. With the official Twitter APIs we collected all the users involved in this spreading activity. We finally passed all these accounts to the Botometer API, since many of the retrieved users were humans.

We set a threshold, in order to classify a user as a bot. That threshold is 2.3 due to the willingness of including some false positives in our data, increasing the heterogeneity in their behaviors and the challenge level for the classifiers. We think that a high intra-homogeneity among classes could lead the models to perform well on the training data, but worse over unseen ones; so we set a lower threshold to lower the intra-homogeneity of the News-spreaders dataset.

The second approach just consisted in collecting the most popular News-Spreaders according to Hoaxy. In consistency with the mentioned threshold, profiles with a Botometer score lower then 2.3 were still discarded.

Finally we checked every profile added to this dataset and removed all those users who didn't tweet enough statuses to be included in this class.

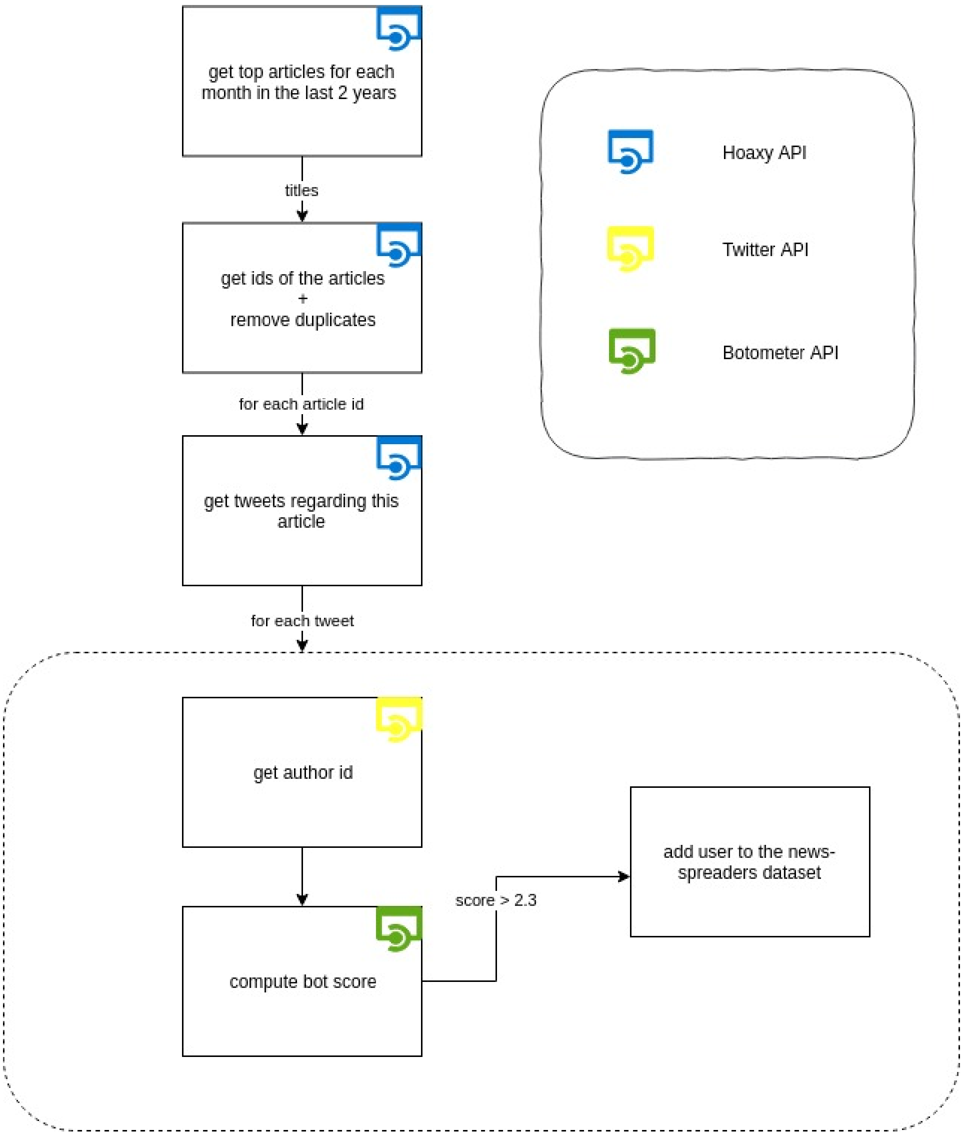


Figure 3.4: Collection of News-Spreading bots, approach 1

This hand-made analysis made us know that there are no bots who only spread fake news. Usually they tweet a lot of verified news and some fake ones, to keep their credibility. We reached 3,590 accounts and 333,699 tweets.

3.4.3 Spam-Bots

As seen before, Spam-Bots were already collected in the work of Cresci et al. [16]. Authors allowed us to access their dataset, so we obtained the Spam-Bots list by sampling their data. Due to homogeneity reasons, we needed to perform scraping again, since we needed different features compared to the available ones. We selected users from:

- traditional Spam-Bots 1
- social Spam-Bots 2
- social Spam-Bots 3

We chose this categories because they contain the most popular kinds of Spam-Bots, that are the ones who advertise products, services or mobile applications. We ignored *"social Spam-Bots 1"*, since they are italian News-Spreaders and *"traditional Spam-Bots 3 and 4"*, since we retrieved enough job-offer spammers during the hand-made labeling. If we had stored too many bots of this category, we would not have been able to generalize on generic Spam-Bots. Finally we gathered 4,943 accounts and 458809 tweets.

3.4.4 Fake-Followers

The collection of this class was quite easy. We initially performed scraping of the data collected by the work of Cresci et al. [16]. Many of these accounts had already been banned, so we could not collect their features. In order to enrich our dataset, we created a new Twitter account (figure 3.5). Then we bought Fake-Followers from two different services:

- instakipci.com/
- rantic.com/buy-legit-twitter-followers/

Instakipci provides low-quality followers. Usually they have no tweets, no followers and a they have a lot of followings.

Rantic, on the other hand, ensures more realistic followers. They seem to have a real network of friends and, sometimes, they tweet too.

By using both services, we gathered a more miscellaneous dataset. We collected 6,307 users and 41,683 tweets for about 20$.

Figure 3.5: Collection of Fake-Followers bots

3.4.5 Genuine

Finally we needed genuine accounts. We used again the Cresci dataset [16] and we filled it with all the Varol users labeled as humans. We again performed scraping on the existing accounts and we collected 3,661 users and 263,240 tweets in total. Morover, we gathered all the Genuine users from the Caverlee dataset [7]. these are many more than other classes, but they will be necessary for a future classification between bots and humans. We perfomed scraping and we were able to obtain 15,701 users and 1,278,852 tweets.

3.4.6 Bots

We needed a set of unlabeled bots in order to train a binary classifier, that is able to classify bots and humans. We again scraped the Caverlee dataset [7] and we gathered 15,539 and 1,276,457 tweets.

3.4.7 Final Datasets

At the end of the collection, we set up two datasets. The first one is the *multi-class* dataset, that contains only bots labeled according to the type of threat, its composition is the following:

category	# users	# tweets
NSFW	6,605	198,378
News-Spreader	3,590	333,699
Spam-Bots	4,943	458,809
Fake-Followers	6,307	41,683

The second dataset is the *Binary* dataset, containing bots and genuine users.

category	# users	# tweets
Bots	15,539	1,276,457
Genuine	15,701	1,278,852

3.5 Descriptive statistics of datasets

As the collection of the data was completed, we explored our final multi-class dataset. In this section we will only focus on different categories of bots, which is the most interesting part.

A first look (3.6) shows us how many user accounts we collected (y axis) for each class and how many tweets we could scrape (diameter). It is easy to observe that Fake-Followers and NSFW bots have fewer tweets than the others, while News-Spreaders have a lot of tweets, but we collected fewer profiles.

category	target id
NSFW	0
News-Spreader	1
Spam-Bots	2
Fake-Followers	3

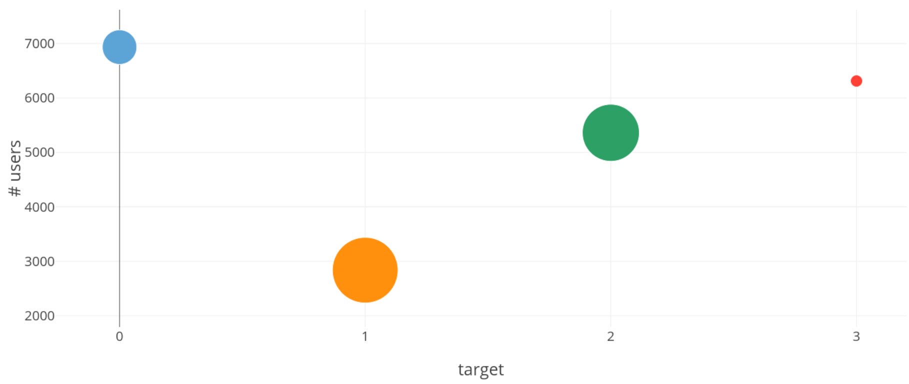


Figure 3.6: Users amount and tweets

Then we plotted the heatmap of the correlation matrix (3.7). We wanted to undestand if some feature was more useful to predict the correct target. This plot suggested us that there were no features highly correlated to the target.

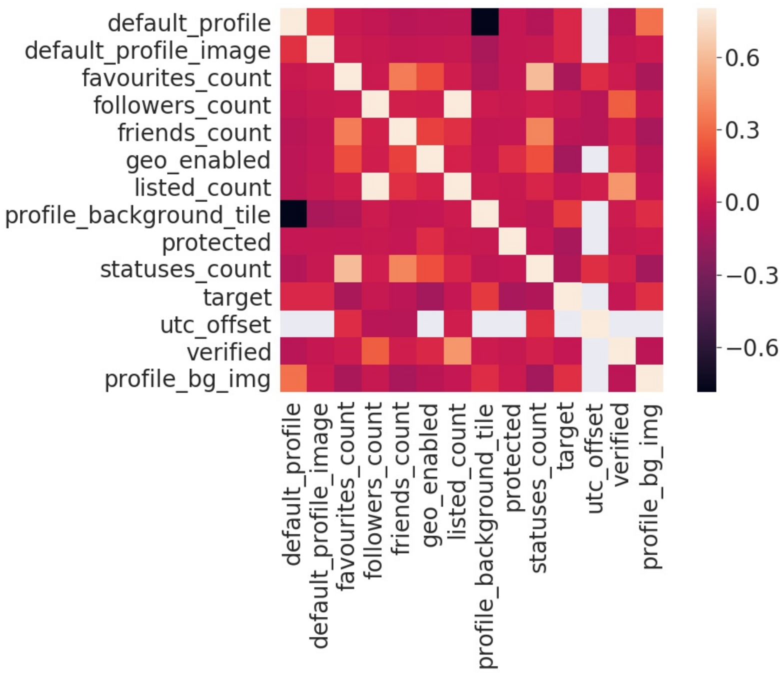


Figure 3.7: Heatmap that highlights the correlation between the features

Moreover in figure (3.8) it is possible to see the distribution of the missing values. This is fundamental for the features engineering step.

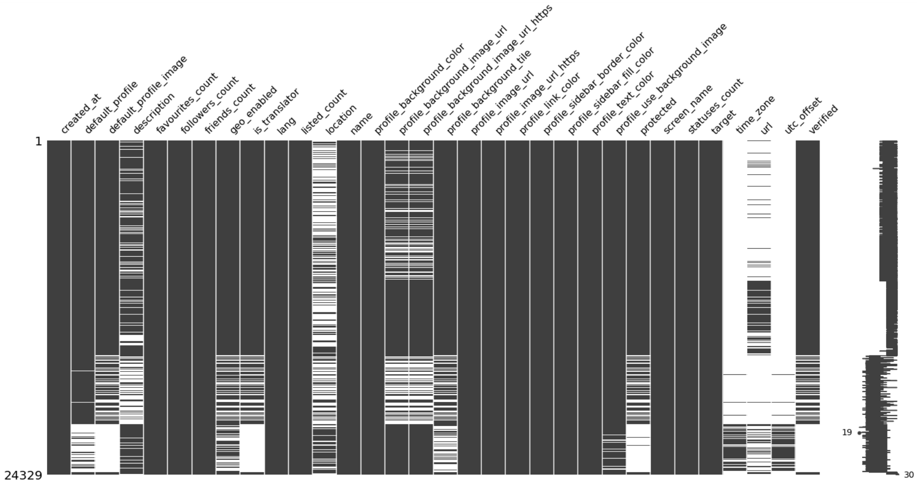


Figure 3.8: Missing values distribution for each feature

Finally we performed three further analyses. With the heatmap we could not detect which features were more important. Anyway, during the hand-labelling step, we understood that some of these features were instead very useful to identify a few classes of our dataset. These attributes are *"followers_count"*, *"friends_count"* and *"statuses_count"*. For each of them we plotted a box-plot. It is a method used to represent groups of numerical data through their quartiles. In Figure 3.9 we can analyze the statuses count (number of tweets) for each target. We collected up to 100 tweets for each user, so this chart is limited to 100. Here we can notice an interesting behaviour: News-Spreaders and Spam-Bots are the classes with more tweets, while Fake-Followers have fewer statuses. By reflecting on the gols of the bots, this result is exactly what we expected to see. In fact Fake-Followers don't need to tweet, they just need to exist, while other types of bots have to publish many statuses, to draw attention on their contents.

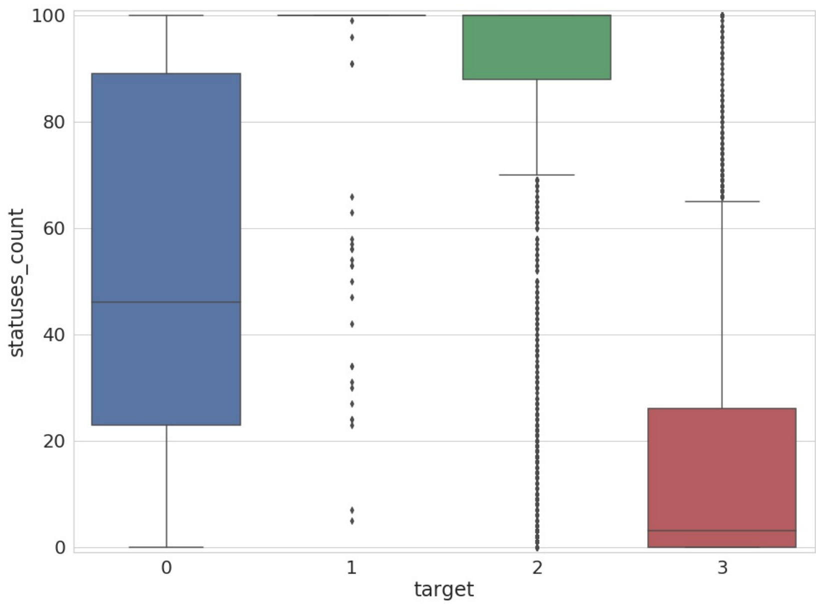


Figure 3.9: Boxplot statuses_count

In Figure 3.10 and 3.11 we performed the same analysis considering the *"friends_count"* and *"followers_count"* features. We limited the charts between 4,000 and 2,000 to keep them understandable. This two figures show us that News-Spreaders usually have a bigger network, while Fake-Followers just follow few users. News-Spreaders network may depends on the popularity of the news media. Other categories are more balanced and their differences can be attributed to the data and not to a different behaviuor between them.

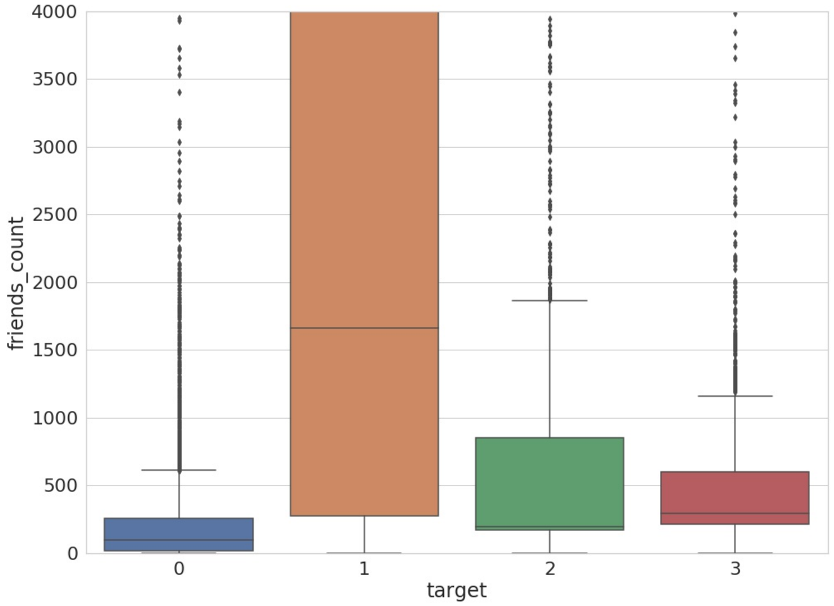


Figure 3.10: Boxplot friends_count

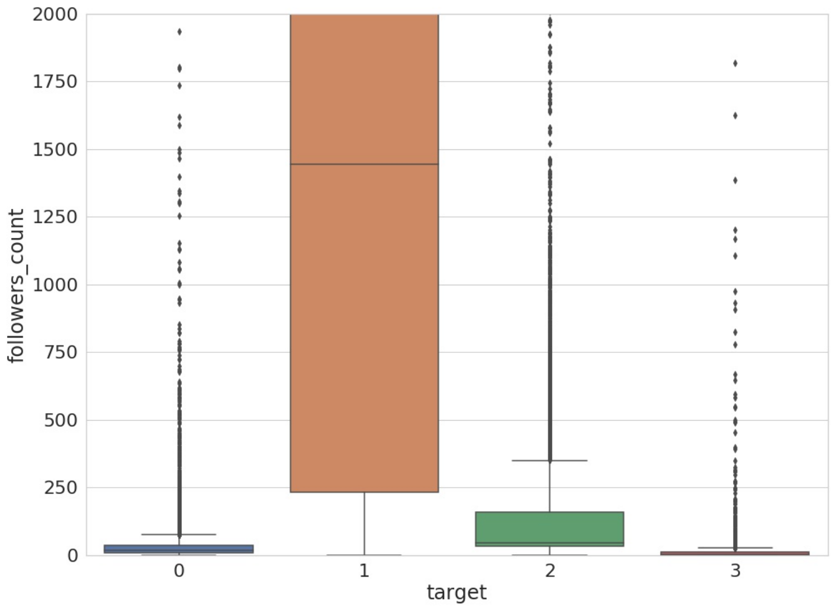


Figure 3.11: Boxplot followers_count

Chapter 4

Features engineering

This chapter can be seen as one of the most important of the whole project.

We wouldn't have hit such performances, if it wasn't for feature engineering. We had a large pool of models to pick for our purpose, and we tried different assets for them, but the difference were made with the reasoning behind the construction of the final feature vector.

The Twitter API provides us two kinds of features: the user attributes and the tweet attributes. We knew that user attributes weren't enough to infer on targets, so we started planning how to include tweet information and enhance our data with them. This was the bulk of the work, but it helped us to catch characteristic behaviours of some user. We created many features. Some of them are descriptive, like the lenght of strings (name, description ecc) or the count (minimum, maximum, average) of other aspects like hashtags per tweet and tweet's lenght. Features describing the tweeting activity (frequency, how often a tweet contains a media or a url or it is just a retweet) have been considered too. Other kind of features are more behaviour-oriented, like the monotony of different tweets of the same user, while others are oriented to the most used words in tweets. Finally we also added features related to image analysis.

The choice over the amount of tweets that would have been considered was a trade-off between performance and prediction speed. We finally chose to retrieve up to the latest 100 tweets for each user.

At the end of this stage, the resulting - and final - feature vector for the multi-class dataset will include 37 features; while the binary dataset will have 34 attributes.

Each resulting feature group has been used to fit a Random Forest with Entropy criterion and 250 tree estimators. This phase allowed us to rank, within each feature group, the crafted attributes, thanks to the inner ranking method of the model. The ranking of an attribute is performed by computing the entropy brought with that feature, when it comes to split the data for the creations of sub-trees. The higher is the feature's entropy, the lower is its rank.

4.1 Baseline

In this section we analyse the complete set of default profile features and which kind of pre-processing operation we applied. With this default set we trained several classifiers to define some baselines for the upcoming work and this allowed us to evaluate the improvements made by the features engineering step.

Some features are ready to be used in a classifier, while other ones need to be pre-processed, in order to allow them to be more expressive. We wanted to rely on user features only for this experiment, in order to have a large improvement margin to exploit, once we would have gone deeper in the study.

During the data exploration stage, we identified the most and least meaningful attributes to trace a baseline, so we started from that. We simply tried to improve and to homologate the features highlighted in the previous chapters.

We faced a lot of missing values as well as non-numeric ones. Even if the goal was to build a raw model with semi-raw data, we needed feasible and manageable attributes to work with.

Her we list all the pre-processing operations applied to each feature belonging to the ones provided by the method *get_user()*, of the official Twitter APIs.

feature	type	preprocess operation
id	int	delete - useless feature
name	str	delete - non-numeric feature
screen_name	str	delete - non-numeric feature
statuses_count	int	—
followers_count	int	—
friends_count	int	—
favourites_count	int	—
listed_count	int	—
url	str	replace with hasUrl (0/1)
lang	str	delete - non-numeric feature
time_zone	str	delete - too many missing values
location	str	delete - too many missing values
default_profile	int	delete - too many missing values
default_profile_image	boolean	boolean to int (0/1)
geo_enabled	boolean	delete - too many missing values
profile_image_url	str	delete - non-numeric feature
profile_use_background_image	boolean	boolean to int (0/1)
profile_background_image_url_https	str	delete - non-numeric feature
profile_text_color	str	delete - non-numeric feature
profile_image_url_https	str	delete - non-numeric feature
profile_sidebar_border_color	str	delete - non-numeric feature
profile_background_tile	boolean	boolean to int (0/1)
profile_sidebar_fill_color	str	delete - non-numeric feature
profile_background_image_url	str	delete - non-numeric feature
profile_background_color	str	delete - non-numeric feature
profile_link_color	str	delete - non-numeric feature
utc_offset	int	delete - too many missing values
is_translator	boolean	delete - too many missing values
follow_request_sent	int	delete - relative feature
protected	boolean	delete - too many missing values
verified	boolean	delete - too many missing values
notifications	boolean	delete - relative feature
description	str	replace with hasDescription (0/1)
contributors_enabled	boolean	delete - too many missing values
following	boolean	delete - relative feature
created_at	str	delete - useless feature

Features processed as “delete - relative feature“ are those ones related to the user who performed the scraping. So we didn’t need them.

4.2 Missing values filling

Features with few missing values was not deleted from the dataset, but we needed to fill that fields. In this section we analyse how we performed this task for each features.

- **default_profile_image**: Thanks to the figure 3.8 we could see that all the missing values was at the bottom, in particular, all of them were in tuples with target 3 (Fake-Followers). In order to understand the behaviour of this feature, we printed its values count for that target.

Fake-Followers	
value	mean
0	2868
1	228

  In both cases the value "0" is more frequent then "1", so we filled all the missing values with the mode (0).

- **profile_background_tile**: As for "default_profile_image", all the missing values belonged to Fake-Followers. We used the same approach and we obtained:

Fake-Followers	
value	mean
0	3086
1	99

  In this case we decided to fill these fields with the mode (0). Since most of the data are 0, this choice allowed us not to dirty the dataset.

- **profile_use_background_image**: All the missing values are still in the last class.

Fake-Followers	
value	mean
0	12
1	4983

  This data is really unbalanced, so filling the null fields with the mode (1) is still the better solution.

4.3 Descriptive features

In order to enrich our attributes and to provide support to our algorithms, we decided to add some descriptive “meta“ features, such as synthesis statistics and counters.

Their purpose is to describe the tweets in a statistical way, adding ranges and means to the attributes provided by the official APIs.
Each of these new values have been added to the users feature vector, in order to append new information for each account. Here is the list of these first 17 brand new features, introduced by our work:

feature	description
avg_len	average lenght of the tweets (words)
max_len	lenght of the longest tweet (words)
min_len	lenght of the shortest tweet (words)
avg_ret	average amount of retweets (by other users) per tweet
max_ret	highest amount of retweets (by other users) on a tweet
min_ret	lowest amount of retweets (by other users) on a tweet
avg_fav	average amount of favourites (by other users) per tweet
max_fav	highest amount of favourites (by other users) on a tweet
min_fav	lowest amount of favourites (by other users) on a tweet
avg_hash	average amount of hashtags involved in tweets
max_hash	highest amount of hashtags involved on a tweet
min_hash	highest amount of hashtags involved on a tweet
freq	amount of tweets per day (up to 100)
ret_perc	percentage of retweets, made by the user, over its retrieved tweets
media_perc	percentage of media content encorpored in tweets
url_perc	percentage of URL links placed inside tweets
quote_perc	percentage of quotes, made by the user, over its retrieved tweets

4.3.1 Ranking

As shown in Figure 4.1, this first group has been performed a ranking function on, with the following result:
1. *freq* (**0.148**), 2. *avg_len* (**0.140**), 3. *media_perc* (**0.102**), 4. *min_len* (**0.098**), 5. *ret_perc* (**0.095**), 6. *max_len* (**0.087**), 7. *avg_ret* (**0.082**), 8. *max_ret* (**0.063**), 9. *quote_perc* (**0.060**), 10. *url_perc* (**0.030**), 11. *avg_hash* (**0.030**), 12. *avg_fav* (**0.016**), 13. *max_fav* (**0.013**), 14. *min_hash* (**0.012**), 15. *max_hash* (**0.011**), 16. *min_ret* (**0.004**), 17. *min_fav* (**0.001**)

The scores assigned to each feature represent the percentages of importance for the model, in terms of Information Gain provided.

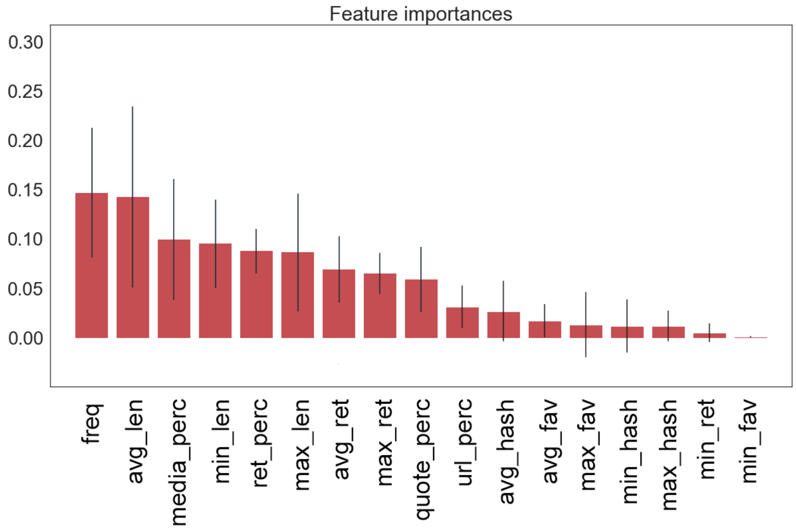


Figure 4.1: Descriptive features ranking with multi-class Random Forest

These meta-features were evaluated on the multi-class Random Forest, highlighting how the tweeting frequency is most dominant attribute. It seemed to be the most helpful feature of its group, when it came to support the choices for decision splits. The average tweet length appeared to be important as well, since it almost matched the first ranked attribute. It is reasonable to think that each category of bots are characterized by specific tweeting activities. Fake-Followers, for instance, usually have few tweets on their timeline, they are often built to follow other accounts, without performing any content sharing. Instead, bots like News-Spreaders and Spam-Bots are thought to act in the opposite way: they have to keep sharing scamming URLs or political posts. in order to do this, they must be active on the social network, with relative high-valued *freq* and *avg_len* features.

4.4 Intrinsic features

Due the multi-class nature of our dataset, it was impossible to rely on the descriptive meta features only.

We faced the need of better capturing some behaviours, that could have helped us distinguish between targets. These ways to act could be content-driven and, so far, we did not have a set of specific features to analyse the user content. We spent a lot of time analysing Twitter timelines by ourselves. This was one of the most useful phases of our work. Indeed, we have learnt a lot about bots acting like humans on the social platform. One thing that was easy to notice was the monotony, in terms of words or URLs involved in tweets, met with Spam-Bots, as well as the opposites, for Genuine accounts or Fake-Followers.

We tried to encapsulate this distinctive content posting by adding two intrinsic features to the training vector, along with the descriptive ones.

How to portray such monotony?
We though about different approaches, like complex sentiment analysis or entity recognition, but then, we chose to rely on two methods: the TF-IDF weighting technique and the information entropy.

These two different approaches application produced two brand new features:

feature	type
tweet_intradistance	float
url_entropy	float

4.4.1 Tweet intradistance

We looked inside every retrieved tweets for each user, then we encoded each of them with TF-IDF weighting.

Every term (word) of every tweet was represented by a numeric weight, according to TF-IDF.
This weighting formula is a combination of Term Frequency (TF) and Inverse Document Frequency (IDF).

$$TF_{i,j} = \frac{n_{i,j}}{|d_j|}$$

The term frequency factor counts the number n of the i_{th} term inside the j_{th} document (the tweet, in our case), dividing it by the lenght of the latest, in order to give same importance to both short and long collections of texts.

$$IDF_i = \log \frac{|D|}{|\{d : i \in d\}| + 1}$$

Where d is the document (tweet).

The inverse document frequency factor aims to highlight the overall magnitude of the i_{th} term in the collection which it belongs. The collection D, in our work, is represented by all the gathered tweets of the examined user.

$$(TF - IDF)_i = TF_{i,j} \times IDF_i$$

After the encoding process, we wanted to map the resulting vectors into an euclidean space, in order to compute the distance of each weighted text, from the total centroid of the collection.

We decided to add to each user a measure of the average intra-distance of her tweets.
In order to accomplish that, we relied on the WSS metric used in K-means clustering, but trying to soften its magnitude. We didn't want huge ranges in our features, minimizing the normalizations along the process.

The resulting formula for this brand new attribute is the following:

$$\textbf{tweet\_intradistance}(U) = \frac{1}{N} \sum_{\mathbf{x} \in U} \|\mathbf{x} - \boldsymbol{\mu}\|^2$$

Where N is the number of tweet for user U, x is the encoded tweet and μ is the centroid of the tweet collection for that user.

This feature turned out to be pretty relevant in supporting the detections of NSFW and Spam-Bots, due their repetitive natures.

4.4.2 URL entropy

For this attribute, we decided to exploit the entropy of the information of a messages source. For each user U, has been computed the following:

$$\textbf{url\_entropy}(U) = \sum_{w \in W_u} -\frac{c(w)}{|W_u|} \log(\frac{c(w)}{|W_u|})$$

where W_u is the collection of URLs retrieved from the user's tweets. The term w represent a single URL, belonging to the collection, and *c(w)* is the function counting the occurrence of that URL. Finally, $|W|$ stands for the cardinality of the collection.

The higher the number of the different URLs inside the tweets, the higher the url_entropy for that user. A monotony URL spammer, with just on link in its argumentations, would lead to a zero-valued url_entropy:

$$url\_entropy(U) = -1 \log(1) = 0$$

In order to treat those URLs, we had to detect them inside the tweet texts, with some regular expressions. After the tweet was cleansed and represented only by its embedded link (if present), we wanted to handle just the domain, so we stripped all the sub-paths of the root. We didn't care about which page or sections the bots were interested to spam, as long as the domain were the same.

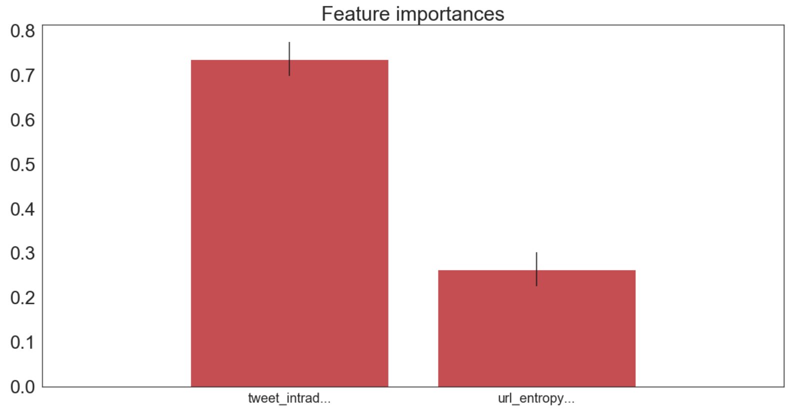


Figure 4.2: Intrinsic features ranking with Random Forest

4.4.3 Ranking

Figure 4.2 shows the inner ranking of these two new features, that have these scores: *1. tweet_intradistance (0.736), 2. url_entropy (0.264).*

From the chart it is clear that the tweet content monotony plays a more important role than the analysis of the links posted. This could mean that bots tend to repeat themselves in terms of words used in tweets, more than in terms of URLs shared. Although, the *url_entropy* feature may lacks in performances, when it has to examine shortened URLs. These links must be expanded to retrieve the actual URL, that may coincide with some of the ones already posted by the same bot, but without passing through shortener services. In terms of induced entropy, the same actual URL, posted twice, could be interpreted as two different endpoints, if one of the posts embedded it in the shortened way. It is computationally expensive, and often useless, to expand all these shrunk links. So, we decided to keep the feature computation as it was when the ranking was performed, and that the bias incorporated was acceptable.

4.5 Extrinsic features

Once we have modelled the personal tweeting actions, in terms of words and links dissimilarity, we needed to look for those parameters that could be compared with all the users in our dataset. We wanted the users to get out of their shells, and to match their timelines with each others.

Once again, sentiment analysis came to our mind. We found lots of paid or limited services that could have only partially supported us during this stage. We couldn't think about implementing our own semantic analysis, as it is meant to be, due to the effort and time it would had taken. For simplicity sake, we had the idea to look for the most meaningful words in tweets, that are common to all the users belonging to the same class. We tried this approach, hoping to find a robust help in separating topics among targets.

The idea was to build various partially-non-overlapping dictionaries, one for each class of users, containing the most popular words used by them in their retrieved tweets, stripped of stopwords. Each dictionary is ordered according to the occurrence of each word, in order to have a proper ranking for the terms and to give each of them a score. The magnitude of the non-overlapping portions of these lists will be shown soon, in this section.

We gathered the 1000 most common words for each category of accounts.

4.5.1 Scores computation

Multi-class Dataset

After the dictionaries related to the multi-class dataset were filtered, to prevent (partial) overlapping, we had four collections, composed by the tuples *(word, score)*:

dictionary	size
NSFW_main_words.csv	253
news_spreders_main_words.csv	424
spam_bots_main_words.csv	325
fake_followers_main_words.csv	469

The overall scores are normalized, so that the most common word, for each class, is associated with a unitary weight, the least common one with a value similar to zero (it depends on the final amount of words included in the dictionary).

In terms of representation, this *keywords score* is built with 4 different values, one for each class.

If some user hit a word that is placed inside more than on a dictionary, we wanted to let the relative weights speak and assign scores to each of the targets that contain that word, knowing they are less relevant, as they don't fall in the highest positions.
So we have

feature	type
NSFW_words_score	float
spam_bots_words_score	float
news_spreders_words_score	float
fake_followers_words_score	float

When we process a new user and infer on him, we scan every word of every tweet and match them with all the dictionaries. Every time that we hit a listed word, we assign the user the score of that word.

For instance, If the word we are comparing matches with the most frequent for NSFW accounts (the top position in its dictionary), we update the *NSFW_words_score* of our user, summing 1 to its current value for that feature.

The followings are examples of the top three words used by our bot categories:

NSFW dictionary	
word	*score*
bio	1.000
photos	0.167
pȧ	0.160
...	

News-Spreaders dictionary	
word	*score*
obama	1.000
g7	0.723
potus	0.690
...	

Spam-Bots dictionary	
word	*score*
talnts	1.000
developer	0.511
engineer	0.467
...	

Fake-Followers dictionary	
word	*score*
iPhone	1.000
cheap	0.993
bounty	0.856
...	

We expected to capture patterns about the choice of the words involved in tweets, for each of our categories. This extrinsic attributes revealed themselves as very useful, lately.

Binary Dataset

As for the multi-class dataset, we computed two dictioraries based on the Binary dataset classes.

dictionary	size
bots_main_words.csv	315
genuine_followers_main_words.csv	315

The result seemed to be acceptable for the bot dictionary, while the genuine one contains many non-english words. The top three words (with their scores) for both dictionaries are:

bot dictionary	
word	*score*
weight	1.000
loss	0.744
traffic	0.661
...	

genuine dictionary	
word	*score*
nao	1.000
pra	0.563
uma	0.493
...	

4.5.2 Safe Area

Making a step back, before the overall computation of this group of features, we had to decide the portions of the dictionaries that weren't supposed to overlap with each others. Once we gathered the main words for each bot class, we had to chose the size of what we called the *Safe Area*. The Safe Area is the collection of the first N words, in each dictionary, that are not overlapping with the others listed in the other dictionaries. This method has been applied to ensure that some words resulted as strictly category-characterizing. We had to pick the number of the top N words that had to be representing, for each category.

We tried with four different portions of the collections: the 25% of the words, the 50%, the 75% and the total non-overlapping solution, the 100% of the terms.

We computed four different datasets (starting from the multi-class collection), one for each parameter, and tested the outcomes of this process on them. The testing phase was composed by the features ranking, like done before, but considering the whole feature vector, and by a *10-fold-crossvalidation* scoring. This last technique has been widely used during all our work, and will be explained in details in the following chapter. Figure 4.3 lists the results for the first tested Safe Area, with 250 words involved in that zone, and the dictionaries that are overlapping for the 75%. The following Figure 4.4, represents the situation with half of the words included in the Safe Areas. So does Figure 4.5, with the 75% of the safe terms, and, finally, Figure 4.6, with totally disjointed dictionaries.

Looking at these graphics, we could infer that the most meaningful feature, for the Random Forest, had been computed with the score of the NSFW_words list. The overall placements in the rankings are similar among the different datasets.

- ✏ 250 words in the Safe Area - overall ranking placement:
  8. news_spreaders_words_score, **10.** NSFW_words_score,
  16. spam_bots_words_score, **20.** fake_followers_words_score.

- ✏ 500 words in the Safe Area - overall ranking placement:
  7. news_spreaders_words_score, **9.** NSFW_words_score,
  13. spam_bots_words_score, **22.** fake_followers_words_score.

- ✏ 750 words in the Safe Area - overall ranking placement
  8. news_spreaders_words_score, **12.** NSFW_words_score,
  13. spam_bots_words_score, **24.** fake_followers_words_score.

- 1000 words in the Safe Area (disjointed lists) - overall ranking placement:
8. news_spreaders_words_score, **11.** NSFW_words_score,
14. spam_bots_words_score, **25.** fake_followers_words_score.

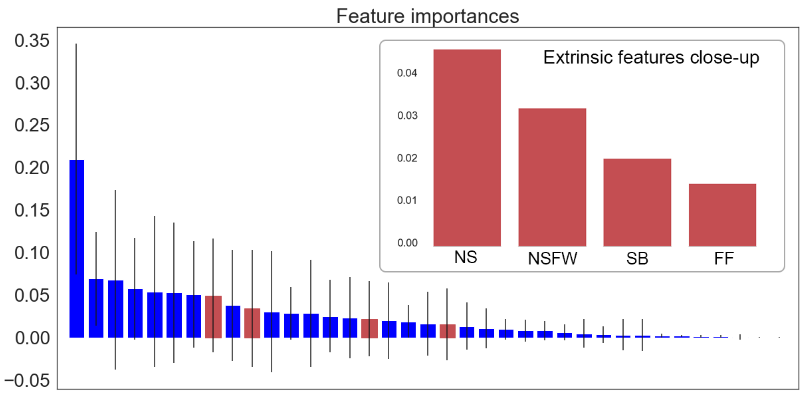


Figure 4.3: Features ranking - Safe Area = 250

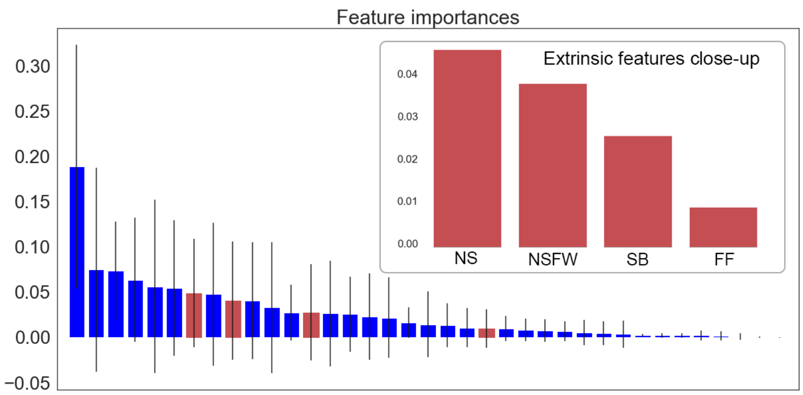


Figure 4.4: Features ranking - Safe Area = 500

The inner ranking remains still with all the sizes for the Safe Areas. The dictionary drawn from the News-Spreaders set seemed to be the most relevant list, for scores computations. The Fake-Followers, as expected, brought a weak effort due to their noisy use of words, along our dataset. They usually tweet less then the other categories, and often in heterogeneous

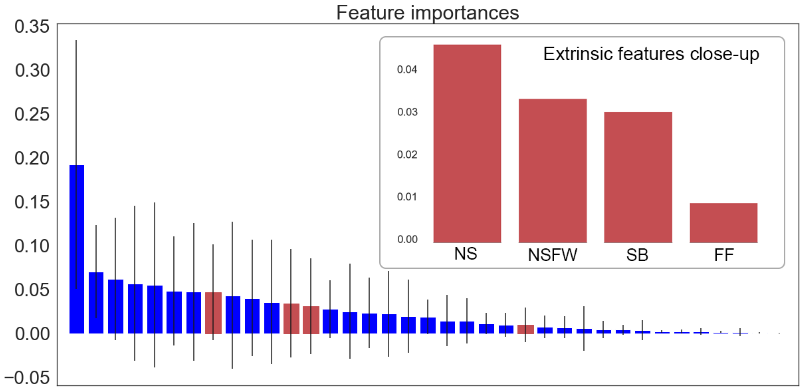


Figure 4.5: Features ranking - Safe Area = 750

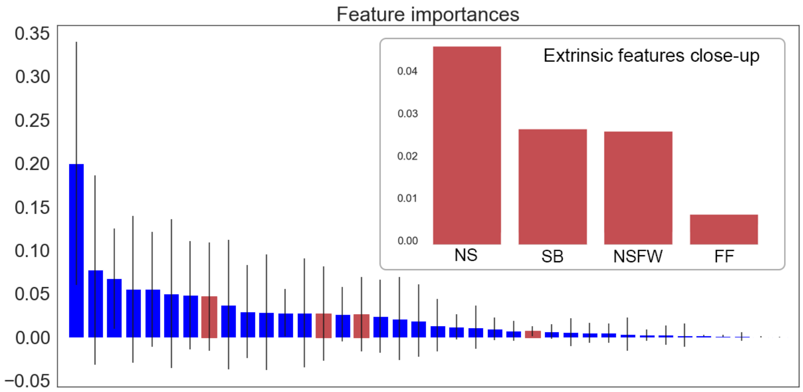


Figure 4.6: Features ranking - Safe Area = 1000

languages, as we couldn't find a specific origin for those bots. Although, the overall placements changes with different Safe Areas lengths.

We needed another confrontation term, which could tell us something about the goodness of the classifier, in relation with the Safe Area used for the computation of the features. We validated a Random Forest multi-class classifier, with increasing number of trees in the forest. Figure 4.7 shows the results.

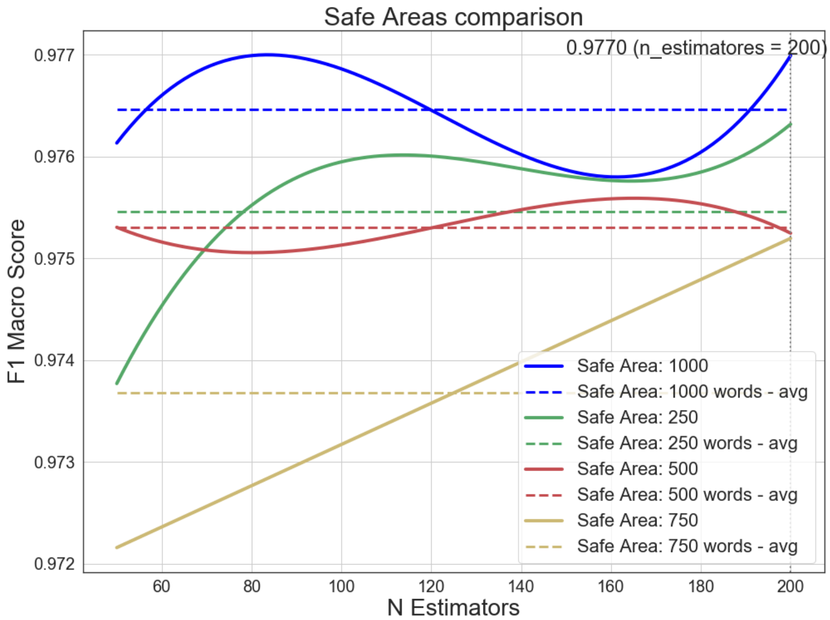


Figure 4.7: Safe Areas comparison

The best looking number of words to save was 1000. The totally non-overlapping solution would have provided us the best F1 score, once we would have started developing the final models. We decided to go with that configuration to build the extrinsic features.

4.6 Image features

One of the main issues of our work was to make the NSFW class different, in terms of classifiers vision, with respect of Spam-Bots.

According to our user-based, descriptive and intrinsic features, both classes act in a similar way: they spam similar links, have high tweet fre-

quency and their contents are often repeated.
What could be the difference?

We started with “blind“ classifier, that was the main problem. a blind model, in this case, is meant to not be able to understand visual contents. Indeed, we just analysed text-based information and meta-data on text posts. Our feature vector lacked in visual components. A Spam-Bot could have been detected as a NSFW, as its tweets involved media and URLs. We needed to go deep and actually see what kind of media were broadcast. This is the reason for presence of the upcoming features.

We've found a small versatile project on GitHub for NSFW detection. The projects involves a pre-trained TensorFlow neural network for image recognition, that looks for adult and violent content inside pictures. It assigns the media a probability to be not safe for work.

The model exploits the Inception v3 [20] architecture. It takes a 299x299x3 input, representing a 299 pixels squared visual field and 3 color channels (R,G,B). It has five convolutional layers, with two max-pooling operations between them. Successively, it stacks multiple “Inception modules“and it ends with a softmax activation function for output layer. The softmax function is a generalization of the sigmoid function used for the Logistic Regression model. It is used to calculate the probabilities distribution of the event over N different events, and it is very common for neural networks with more than one target. The repeated stacking of such models makes the architecture deep, allowing each module to detect features on multiple scales, using convolution operations with different kernel sizes.

The Inception model has been fed with pictures representing Not Safe For Work contents, as well as Safe For Work contents. The NSFW images has been retrieve by crawling the browser tabs, with the help of Fatkun batch download Images, a Google Chrome extension for batch downloads. We gathered 1,548 elements for that class. The SFW class content is composed of a collection of clean pictures of random users, already collected in the Selfie Dataset [21]. This dataset contains 46,836 images, with meta-data about genders, ages and so on; it's built for research purposes. We just randomly picked the same number of NSFW pictures, from the SFW collection, in order to have a balanced training set.

The network has then been trained with 500 iteration steps, leading to a 95.7% of accuracy in validation and 99% in training stage, when distinguish NSFW contents from SFW ones. The accuracy graph is shown in Figure 4.8. These attributes were computed with a machine learning model which, for definition, has an error in its functioning. These features differs from the others, because don't rely on 100% accurate methods, like the computation of means and statistic indices. Thus, we knew we were introducing a bias in our feature vector, and that it was bounded to the Inception module test error. Although, we thought that the error brought with this new features would have been under control and that it wouldn't compromise the final results of our models.

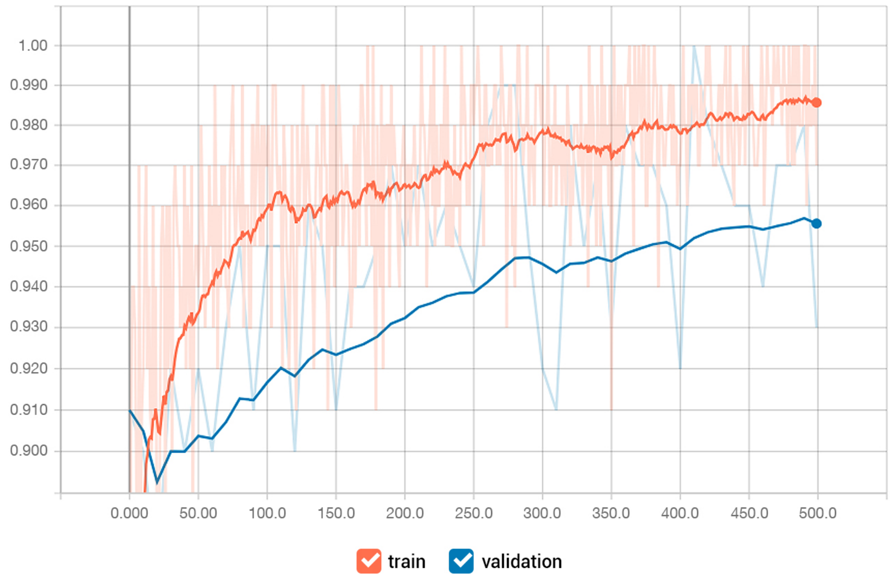


Figure 4.8: Train and validation accuracy through learning steps

We decided to scan our dataset with this new component and to give a NSFW score to both profile pictures and tweets. For time complexity reasons, we couldn't imagine to scan all the retrieved tweets for each user and to look for embedded media. We limited the process to the latest ten tweets.

The brand new attributes that helped us in better separating NSFW class are:

feature	type
NSFW_profile	float
NSFW_avg	float

4.7 Final feature vectors

4.7.1 Multi-class Dataset

At the end of this process we obtained a feature vector composed of 37 elements, 12 based on the user and 25 based on his tweets.

User features

Features
age
default_profile
description_len
favourites_count
friends_count
followers_count
listed_count
profile_use_background_image
name_len
screen_name_len
statuses_count
url

Tweets features

Descriptive features
freq
min_fav
avg_fav
max_fav
min_hash
avg_hash
max_hash
min_len
avg_len
max_len
min_ret
avg_ret
max_ret
media_perc
quote_perc
ret_perc
url_perc

Intrinsic features
tweets_intradistance
url_intradistance

Extrinsic features
NSFW_words_score
news_spreaders_words_score
spam_bots_words_score
fake_followers_words_score

Images features
NSFW_profile
NSFW_avg

4.7.2 Binary Dataset

At the end of this process we obtained a feature vector composed of 34 elements, 12 based on the user and 22 based on his tweets.

User features

Features
default_profile
description_len
favourites_count
friends_count
followers_count
listed_count
profile_use_background_image
name_len
screen_name_len
statuses_count
url
verified

Tweets features

Descriptive features
freq
min_fav
avg_fav
max_fav
min_hash
avg_hash
max_hash
min_len
avg_len
max_len
min_ret
avg_ret
max_ret
media_perc
quote_perc
ret_perc
url_perc

Intrinsic features
tweets_intradistance
url_intradistance

Extrinsic features
bots_words_score
genuine_words_score

Images features
NSFW_profile

Chapter 5

Bot classifiers

In this chapter we will show the choices and stages behind the final model. Starting from baseline models, we enhanced the chosen classifiers with hand-crafted features coming from the last chapter.
We saw and studied the performance improvements with validation approaches, and this phase led us to our current solution.

The result involves four models:

- A first Random Forest classifier that has been used to provide an early filter on the separation between genuine accounts and bots.

- A second Random Forest that gives a classification among the four studied categories of bots only.

- A Naive Bayes classifier, used over the same classes of the second Random Forest, but which reads and labels the users, according to their tweet texts only.

- A K-Nearest Neighbours classifier, used over the four bot classes, based on the user features only.

The algorithms were used together into a pipeline work-flow, whose first step is the detection of bots from humans, thanks to the first binary Random Forest. Then, the percentage of membership in the bot category is further split into four sub-percentages, which represent the prediction over the inner bot categories. This last partition is performed by a stacking ensemble, whose goal is to combine the predictions of the three multi-class models.

5.1 Baselines

In order to build a classification system, we needed to test different algorithms with raw settings. We decided to perform several validations on the entire feature vector, with four different machine learning classifiers, for both the binary and the multi-class problem.

Furthermore, no parameters tuning has been applied, in order to minimize the results of our baselines classifier, with their standard settings.

5.1.1 Random Forest

Random forest is an ensemble learning method used in classification tasks and prediction ones as well.

The algorithm builds several *decision trees* and the resulting output is provided by the mode of the predictions coming from the estimators in the forest.

Each decision tree is trained on a subset of the original data, formed by sampling with replacements the whole training set. They share the same splitting criterion, in order to build subtrees, which is the entropy:
Every tree computes the Information Gain of each feature, which is the difference, in terms of entropy, between the information gained on the data D, before splitting on the attribute X, and the one gained after the split, which provides n subsets of D.

$$\mathit{InformationGain}(X) = \mathit{Information}(D) - \mathit{Information}_X(D)$$

where

$$\mathit{Information}(D) = -p_1 \log p_1 - ... - p_n \log p_n$$

and

$$\mathit{Information}_X(D) = \frac{|D_1|}{|D|} Information(D_1) + ... + \frac{|D_n|}{|D|} Information(D_n)$$

The attribute providing the highest InformationGain, against the others at the same level of the tree, is chosen to perform a split.

The feature set considered by each tree is a random subset of the original pool.

Due to its ability to face overfitting and to the feature importance ranking that it can provide, this tool is often preferred over other models belonging to the same category.

The advantage of preventing overfitting usually comes with a slower prediction time, because it needs enough estimators for this task. But, for our purpose, there were enough estimators to face the variance problem without affecting the generalization speed.

5.1.2 Logistic Regression

Logistic regression is a common statistical model, that uses a sigmoid function to map the output of a linear regression on a normalized score, giving the probability, for each sample, to belong to the positive class, given its features and a weighting vector:

$$P(\hat{y}_i = +1|\vec{x}_i, \vec{w}) = \frac{1}{1 + e^{-\vec{w}h(\vec{x}_i)}}$$

Where $\hat{y}_i$ is the predicted target, over the i_{th} sample, $\vec{x}_i$ is the feature vector of that sample, $\vec{w}$ represents the weighting vector that has to be learned and h is the activation function of the linear regression.

Logistic Regression searches for the weighting vector that matches the highest likelihood and, in order to do that, it minimizes a cross-entropy error function, provided by the negative log of the likelihood:

$$\mathbf{L}(\vec{w}) = -\ln \prod_{i=1}^{n} P(\hat{y}_i = +1|\vec{x}_i, \vec{w})$$

In multi-classes tasks, there are two possible approaches to face the problem:

- a more general *softmax* function to replace the logistic sigmoid, which assigns the probability, for the i_{th} sample, to belong to the class C:

$$P(\mathbf{C}_i|\vec{x}_i, \vec{w}) = \frac{e^{-\vec{w}h(\vec{x}_i)}}{\sum_{j=1}^{n} e^{-\vec{w}h(\vec{x}_j)}}$$

- "One-vs-Rest" method, which for each class builds a model that predicts the target class against all the others.

We decided to stick with the default settings of the libraries involved, so OvR was the approach used for the baseline.

5.1.3 K-Nearest Neighbors

K-Nearest Neighbors is an instance-based model used for classification, regression and pattern recognition. It is considered as a lazy learning algorithm, because all the computation is deferred until the prediction phase.

When it performs a classification over a new point, it looks for the K nearest samples in the training set, according to a chosen metric, and it assigns, to the unseen sample, the mode of the targets of the retrieved neighbors.

The choices to make are the ones regarding the number K of neighbors to consider, the weights to assign to them and the metric to calculate the distance with. We used the default settings for the metric (*Euclidean distance*) and for the weighting technique (*uniform*), but we chose to consider 10 neighbors, because the automatic setting was $K = 5$, which is the number of our possible targets. We chose a K that is large enough to make the model not too sensible to outliers, and restricted enough to sharpen the classes boundaries.

We first normalized the training data and then we fitted the algorithm on them, in order to simplify the distance computations.

5.1.4 Support Vector Machine

Support Vector Machine is a smart way to do instance-based learning. It can be seen as a generalization of the weighted KNN algorithm, with an arbitrary and feasible *kernel function*, instead of the more generic dot product.

It can be summarised with a support vector $\tilde{\mathbf{x}}$ (a subset of the training set), a weighting vector $\tilde{\mathbf{w}}$ for them and a **kernel** *K(x, x')* (a similarity function).

In order to make it work properly, three choices must be made:

- a proper kernel, which is often selected according to experience and domain knowledge of the problem. We wanted to make things simple in this stage, so we used the default kernel function, which is the Radial Basis Function:
$$K(x, x') = exp(-\frac{||x - x'||^2}{2\sigma^2})$$
with σ as a free parameter

- the weights $\vec{w}$, which are obtained by maximizing the margin that splits the records belonging to different classes. Each samples are mapped into a space, thanks to what is known as the *kernel trick*. The "trick" helps a linear classifier to work on a non-linear problem, applying the kernel function in the prediction phase.
This process highlights the boundary that separates the points belonging to different classes. SVM aims to draw the boundary for the classes, in order to maximize the "margin" formed between the closest points that have different targets

- the support vector $\vec{x}$, which comes as a consequence of choosing weights

Since we were still facing a multitarget problem, the binary nature of SVM must had been adapted to our needs. We decided, once again, to stick with the default setting for non-binary classifications, in order to have only raw baselines to compare.

The multi-target classification is handled with "One-vs-One" approach. It considers all possible pairwise binary classifiers and so it leads to $\frac{N(N-1)}{2}$ individual binary classifiers, where N is the number of the classes in the problem.

In comparison with "One-vs-Rest" approach, "One-vs-One" is less sensitive to an imbalanced dataset, but it's more computationally expensive then the the other, which only builds N binary classifiers. Despite our choices over methods and parameters weren't accurate in this stage as they were in the other ones, we decided to stick with this setting for SVM, because otherwise it would have led us to an irrelevant algorithm, in comparison with the above-mentioned.

5.1.5 Comparison and baseline selection

Different tasks imply different evaluation metrics. Every classifier was validated and selected according to certain indices of goodness. In particular, we followed a triple of metrics that involves Precision, Recall and F1, for the multi-class problem and we aimed to maximize AUC score for the binary case.

Multi-class metric

The selected baseline models were tested with a holdout approach at first, then with a crossvalidation method. We built a Confusion Matrix for each model, in order to bring out goodness indices for each class, such as *True Positive* (TP), *False Positive* (FP) and *False Negative* (FN). The evaluation metrics considered are *Precision*, *Recall* and *F1 score* and they work on the mentioned indices.

- $Precision = \frac{TP}{TP+FP}$
  It measures the proportion of positive identifications, for a given target, that was actually correct.
- $Recall = \frac{TP}{TP+FN}$
  It measures the proportion of actual positive classifications that was identified correctly.

✏ $F1score = \frac{2(Precision \times Recall)}{Precision + Recall}$
It calculates the harmonic mean of the previous metrics.

Every metric is adapted to fit a multi-class problem. For each class, it has been computed this set of measures, and then they were averaged without weights (macro average), in order to not take label imbalance into account.

Binary metric

Since this classifier was built for a different purpose, with respect to the multi-class models, the *Area Under the Curve* score (AUC) is the metric we followed, both for baselines and the final model evaluation. Area Under the Curve represents the goodness of a classifier, in terms of the integral of the *Receiver Operating Characteristic* (ROC curve), defined over the variation of a decision treshold.

The ROC curve lies in a bi-dimensional space, which has the *True Positive Ratio* ($TPR = \frac{TP}{TP+FN}$) on the Y-axis, and the False Positive Ratio ($FPR = \frac{FP}{FP+TN}$) on the X-axis. In general, a classifier should accomplish more than 0.5 in AUC score, because that threshold represents a random guesser, which has the 50% of probabilities to detect the actual class. The more the AUC score tends to 1, the better is the ability of the classifier to distinguish among classes.

The motivation behind the adding of this new metric is that we had a balanced binary dataset, and this metric is a good fit for this kind of problem. Moreover, Botometer claims to have accomplished an AUC of 0.95, on a 10-fold crossvalidation test. We wanted to get close to that score, using our crafted features.

5.1.6 Holdout evaluation

The holdout evaluation is performed separating the samples of the dataset into training and test subsets. The splitting process is randomized and it requires a bigger portion of the original dataset to be inserted in the training data, comparing to the amount of samples that will form the test set. A common choice is to use a third of the data to evaluate the model.

5.1.7 Multi-class

In this case, we decided to use 75% of the data for the training set and 25% for the test set. This choice is a little bit different from the most common one, which builds the training set with 2/3 of the whole data, because we didn't dispose of a huge amount of records, so we preferred this ratio and then trying an other validation method for comparison. Here we list the algorithms and their parameters, as they were written according to the Scikit-learn library for Python, their confusion matrix and their scores:

- ✏ *RandomForestClassifier(n_estimators = 10, criterion = 'entropy')*
  Confusion matrix:

		Predicted class			
		NSFW	NS	SB	FF
Actual class	NSFW	1690	27	12	6
	NS	27	785	30	0
	SB	14	39	1280	5
	FF	12	7	18	1215

  Precision: 0.957
  Recall: 0.958
  F1 score: 0.957

- ✏ *LogisticRegression(fit_intercept=True, max_iter=100, penalty='l2')*
  Confusion matrix:

		Predicted class			
		NSFW	NS	SB	FF
Actual class	NSFW	1274	213	197	51
	NS	24	740	75	3
	SB	29	87	1144	78
	FF	204	49	46	953

  Precision: 0.793
  Recall: 0.807
  F1 score: 0.794

- ✏ *KNeighborsClassifier(n_neighbors=10)*
  Confusion matrix:

		Predicted class			
		NSFW	NS	SB	FF
Actual class	NSFW	1578	36	60	61
	NS	81	685	63	13
	SB	81	32	1187	38
	FF	104	6	55	1087

  Precision: 0.883
  Recall: 0.869
  F1 score: 0.875

✏ *SVC(kernel='rbf', decision_function_shape='ovo')*
Confusion matrix:

		Predicted class		
		NSFW	SB	FF
Actual class	NSFW	1735	0	0
	NS	842	0	0
	SB	1114	223	1
	FF	483	0	769

Precision: 0.603
Recall: 0.445
F1 score: 0.408

From these evaluations it is possible to see how the Random Forest algorithm outperforms Logistic Regression and Support Vector Machine. This difference colud be driven by the multi-class task. Logistic Regression and SVM need to be adapted to this purpose. Another factor that can discriminate the performances is the choices of the features and their magnitude. Random Forest doesn't require attribute normalization to top its scores, additionally, every feature has been ranked and tested with the inner feature ranking provided by the algorithm. It is possible that some of these attributes need further processing to better support the other models tested.

Binary

This evaluation was made with the common splitting ratio between train and test set. Since we had 31,212 samples available, well balanced, we used two thirds (20,808) for the training set and the remaining (10,404) for the test set. We wanted a first term of comparison, so, in the beginning, we evaluated the AUC metric with the holdout technique.

✏ *RandomForestClassifier(n_estimators = 10, criterion = 'entropy')*
Confusion matrix:

		Predicted class	
		BOT	GEN
Actual class	BOT	4314	336
	GEN	554	4160
	AUC	0.905	

✏ *LogisticRegression(fit_intercept=True, max_iter=100, penalty='l2')*
Confusion matrix:

		Predicted class	
		BOT	GEN
Actual class	BOT	3124	1526
	GEN	558	4156
	AUC	0.776	

✏ *KNeighborsClassifier(n_neighbors=10)*
Confusion matrix:

		Predicted class	
		BOT	GEN
Actual class	BOT	3698	952
	GEN	1129	3585
	AUC	0.777	

✏ *SVC(kernel='rbf')*
Confusion matrix:

		Predicted class	
		BOT	GEN
Actual class	BOT	4625	25
	GEN	4676	38
	AUC	0.501	

Once again, Random Forest has the best scores, even in the binary problem. Although, e can see a worsening in the KNN performances in this test. It can be imputable to an "unlucky" holdout set or to the fixed parameters tested.

5.1.8 Crossvalidation

This approach is based on repeated holdouts. It is performed by splitting the whole data in K non-overlapping folds, leading to K different holdout evaluations. The results for each step are stored and the final evaluation is given by the mean of the K evaluations. For each evaluation, one fold is used for testing, the other ones for training the models. A common practice is to set $K = 10$ and thus averaging 10 different evaluations. This method is also known as *K-fold crossvalidation.* We used a stratified approach, which takes care about keeping the labels balanced on each fold.

Due the need of performing ten steps, it is computationally more expensive then a simple holdout validation. In our case, it was feasible, in term of speed, because of the models complexity and the data amount. This situation held for both the binary and the mutliclass tasks.

The obtained scores are also more meaningful, with regards to holdout, because they are less sensitive to “lucky“ or “unlucky“ splits.

Here is the results for every baseline model:

Multi-class

- *RandomForestClassifier(n_estimators = 10, criterion = 'entropy')*
  Mean precision: 0.947
  Mean recall: 0.945
  Mean f1 score: 0.943

- *LogisticRegression(fit_intercept=True, max_iter=100, penalty='l2')*
  Mean precision: 0.827
  Mean recall: 0.815
  Mean f1 score: 0.815

- *KNeighborsClassifier(n_neighbors=10)*
  Mean precision: 0.878
  Mean recall: 0.858
  Mean f1 score: 0.862

- *SVC(kernel='rbf', decision_function_shape='ovo')*
  Mean precision: 0.573
  Mean recall: 0.456
  Mean f1 score: 0.413

As the results show, the Random Forest algorithm is the one that achieves the best performances, even with default settings, on both holdout and 10-fold crossvalidation. We thus decided to consider it as the main tool to build our bot categories classifier.

Binary

- *RandomForestClassifier(n_estimators = 10, criterion = 'entropy')*
  Mean AUC: 0.916

- *LogisticRegression(fit_intercept=True, max_iter=100, penalty='l2')*
  Mean AUC: 0.792

- *KNeighborsClassifier(n_neighbors=10)*
  Mean AUC: 0.835

Mean precision: 0.779

✏ *SVC(kernel='rbf')*
Mean AUC: 0.583

Even in the binary cases, the Random Forest had the best performance, and it could be imputed to the similar features involved in both problems. Moreover, we could see that the Support Vector Machine emerged as a lightly improved random guesser.

5.2 Binary Classifier

Since our dataset was pretty balanced and we couldn't retrieve many more genuine accounts, we didn't want our instrument to treat this category of users just as one the other bot kinds. It was important to perform a previous filter that was able to give importance to the separation between bots and genuine accounts.

We were inspired by the work made with Botometer [5], which involved a binary labelled dataset, with bot and genuine accounts. The researchers built their features, grouped them in six main categories, then they ran a Random Forest algorithm per group.

We already had our feature engineering done, so we decided to test it on this new task.

In order to not to build a poorer version of our multi-class model, we didn't want to use a reduced copy of our dataset, stratifying it by stripping random bots from it. We needed a balanced dataset, with about the same amount of genuines and bots. So, we started from the same dataset used by the Botometer project, in order to have a baseline comparison.

5.2.1 Dataset

The dataset we used for this classification was composed of part of our collected records and of some entries from the Caverlee-2011 dataset, which contains 22,223 content polluters and 19,276 legitimate users, both collected through a social honeypot, as described in their paper [7].

We used the APIs to retrieve the ids for both genuines and bots, from the Caverlee list. The process provided us 15,687 legitimate user ids, and 15,525 general bot ids (without inner classifications), for a total number of 31,212 samples. The difference from the original number of entries is due to the age of the dataset. Since 2011, the year of the creation of the list, a lot of accounts have been deleted or suspended.

The feature vector we used is the same that came out from the feature engineering process, except for the specific characterizing features, that weren't considered, because crafted for the inner separation among bots. We excluded the *NSFW_avg* image feature, as we noticed it didn't bring much performance boosting with the multi-class models. The extrinsic features must had been adjusted with new dictionaries, so we had two features: *bots_words_score* and *genuine_words_score*. Both the features have been computed as for the multi-class case, with up to 1000 non overlapping words in each dictionary.

5.2.2 Model

The model chosen for the purpose was the best performer of the tested baselines: the Random Forest binary classifier. The algorithm has had its parameters tuned during the validation phase. We decided to stick with 10-fold crossvalidation, as it was done for the baselines.

After several Grid Search runs, the last round computed had this hyperparameters to combine together:

- *n_estimators* = [150, 200, 250, 300, 350, 400, 450, 500]
- *max_depth* = [None, 26, 28]
- *criterion* = 'entropy'

As we can see in Figure 5.1, the AUC is increasing with the number of the estimators in the forest. We decided t stop at 450, which corresponds to the highest AUC score, since this phase was aimed to find a comparison term with Botometer, but it didn't represent the final model. In their paper [1], the Botometer group claims to reach 0.95 in AUC score.

The AUC obtained with our arrangement is equal to 0.96, as shown in Figure 5.2, which is a positive accomplishment, considering that it will be used only as support for the identification of humans among bots, but we didn't crafted specific features as the ones involved in the Botometer project and we didn't have the same amount of data neither.

The model has then been fitted with the hole data, with this settings: *n_estimators* = 450, *max_depth* = 26 and *criterion* = 'entropy'.

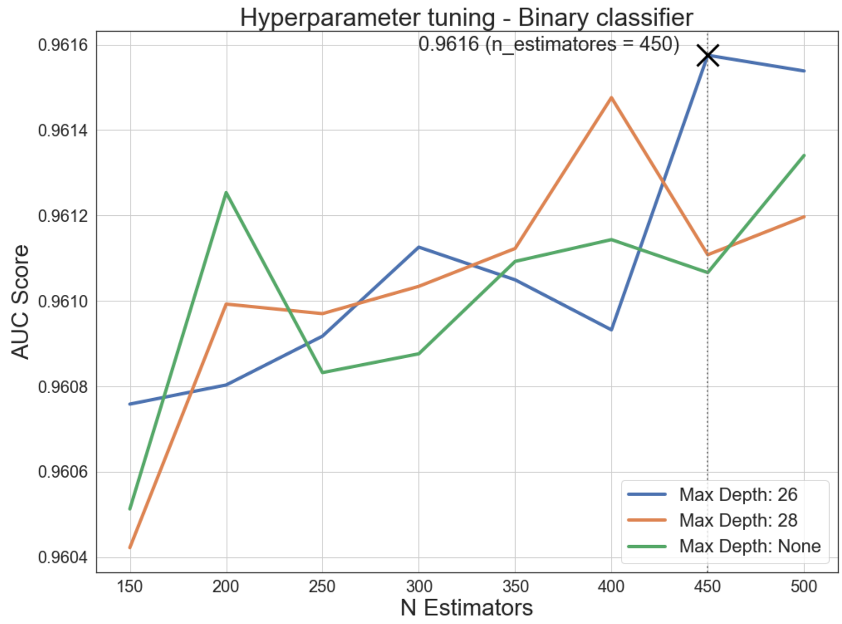


Figure 5.1: Grid search results - close-up view

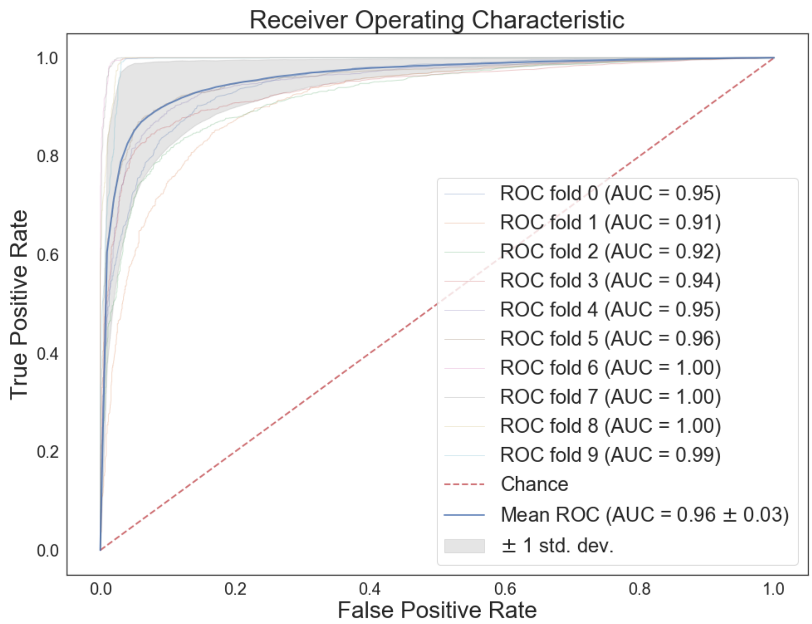


Figure 5.2: Bot-or-not ROC curve

5.2.3 Validation

We had an interesting amount of data that were not involved in this task, because of the comparison with the same Botometer's dataset. Since this unseen data had a further discrimination among bots, it was easy to sample some records randomly, replacing their multi-class targets with binary values. We performed this job to validate the newborn model on unseen and fresher data. We were interested in testing a model that were trained over "old" accounts, with consequent different attributes values and different behaviours on the platform, with younger accounts.

This validations would had given us a preview of the real performance of the model, once it would had been deployed on the internet. The account that a user would test with our application could be younger than the ones included in the Caverlee's list.

We sampled 6,000 accounts, divided in 3,000 genuine and 3,000 bot ids, randomly picked by our multi-class dataset.

The binary model were fitted with its data and it was ready to perform new predictions.

Looking at the most relevant features for the classifier, as shown in Figure 5.3, we could find the **age** field at the top position.

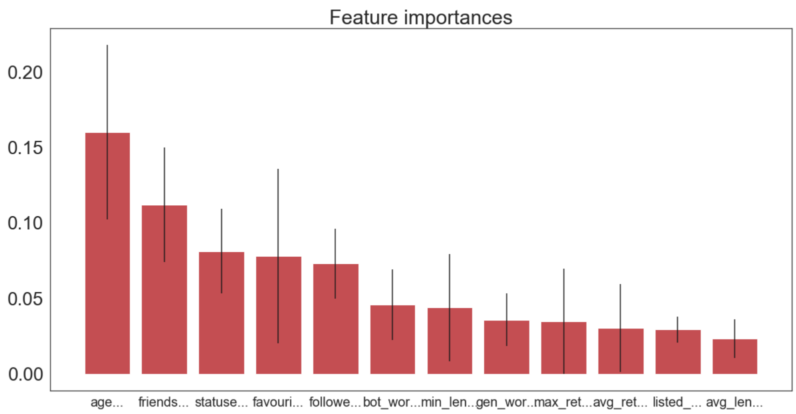


Figure 5.3: Binary Random Forest features ranking

This was the first warning of a validation performance worsening. A said before, the age of the accounts in the Caverlee's dataset were higher than the ones in our dataset. In particular, we examined the *age* field of the training set, and the one coming from our validation samples, picked from the mutliclass dataset.

Table 5.1: Age field comparison among bot accounts

Training Bots		Validation Bots	
age		*age*	
mean	8	mean	4.50
std	0.66	std	2.69
min	4	min	0
max	12	max	11
25%	8	25%	3
50%	9	50%	4
75%	9	75%	6

Table 5.2: Age field comparison among genuine accounts

Training Genuine		Validation Genuine	
age		*age*	
mean	9.22	mean	6.40
std	0.54	std	1.94
min	4	min	3
max	12	max	11
25%	9	25%	5
50%	9	50%	6
75%	9	75%	8

Like Tables 5.1 and 5.2 show, there is a clear differences in the age attribute, between training and validation set.

We went forward to check if this diversity would had led us to a bad validation performance, or if the model would had handled the predictions in other ways.

The 10-fold crossvalidation on the validation set produced the following confusion matrix, with the correlated AUC score:

		Predicted class	
		BOT	GEN
Actual class	BOT	558	2442
	GEN	158	2842
	AUC	0.566	

The worsening were real, and it highlighted the short-sighted training phase we performed, trying to top the Botometer performance.

We tried to mitigate this performance loss, by excluding the main suspect from the features set. Here is the validation performance, without considering the accounts' ages.

		Predicted class	
		BOT	GEN
Actual class	BOT	2920	20
	GEN	1589	1411
	AUC	0.721	

The improvement was encouraging, but still not enough to rely on this basic solution. Considering the bot target as the positive class, we still had too many False Positive in our confusion Matrix. The binary classifier used to tend to identify an user as a bot, with too much confidence. We had to reduce that number, in order to provide a reliable filter in the final prediction pipeline system.

5.2.4 Data extension

The idea we had was to use some data from our multi-class dataset to enrich the binary training set, in order to make the algorithm handle younger and different types of samples from the Twitter population.

In order to perform the extension, we sampled 3,000 genuine accounts and 8,000 bots (2,000 content polluters for each class), all coming from our dataset, and added them to the Caverlee's dataset. The new training set was composed by 42,212 samples.

We performed a 10-fold-crossvalidation to see the effect of this data refill, sticking to the same hyperparameters found by the last Grid Search. The AUC score measured with these data was 0.963. We could see a slight improvement of the performances, with this data extension. However, we wanted to take a look inside the inner ranking performed by the algorithm, to check if the age field represented an important splitting point. Figure 5.4 shows that the age attribute was still the most considered when the trees had to perform the first splits.

We couldn't blindly follow the AUC score through Grid Searches, without making considerations about what will happen when we will allow people to classify data coming from outside our collected samples. The age feature would have driven the Random Forest to misclassification over accounts with low *age* values. Even if the exclusion of that attribute would had made the overall AUC score worse, we had to strip it from the features vector, in order to better generalize on real test cases.

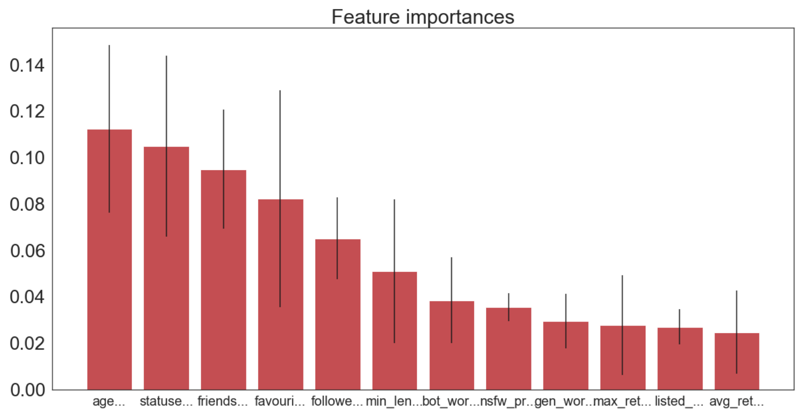


Figure 5.4: Features ranking with augmented data - Top 12

While cross-validating the model, we tested the complete features vector (with and without the extension from our dataset) and the one stripped by the age values. The crossvalidation was performed with the same hyperparameters settings of the model fitted with the Caverlee's dataset only.

	Fitted data		
	original - with *age*	extended - with *age*	extended - without *age*
AUC	**0.961**	**0.963**	**0.948**

The age filed removing made thing worse, but we decided to perform it anyway, because of the good score reached without it, and the flexibility we were giving to the Random Forest. The slight worsening could be also imputed to the biased extrinsic features of that data: those samples came from the multi-class dataset and they originally had the extrinsic features based on the four bot categories' dictionaries. In order to refill the binary dataset with these new samples, we had to recompute the extrinsic features, applying the analogous method used for the binary purpose. We didn't recompute the entire dictionaries, we just assigned the scores to the new samples we were introducing, basing the calculations on the already listed words. Those had been exposed in chapter 4. This approach aimed to force the algorithm to identify bots and humans, basing its comparisons on the online computations of those features, like in a real-case generalization.

This last configuration was used to performed a further tuning of the parameters. A new Grid Search brought us the configuration for the hyper-parameters shown in Figure 5.5, leading to the new AUC score, exposed in Figure 5.6. The binary classifier has then been fitted with 42,212 samples

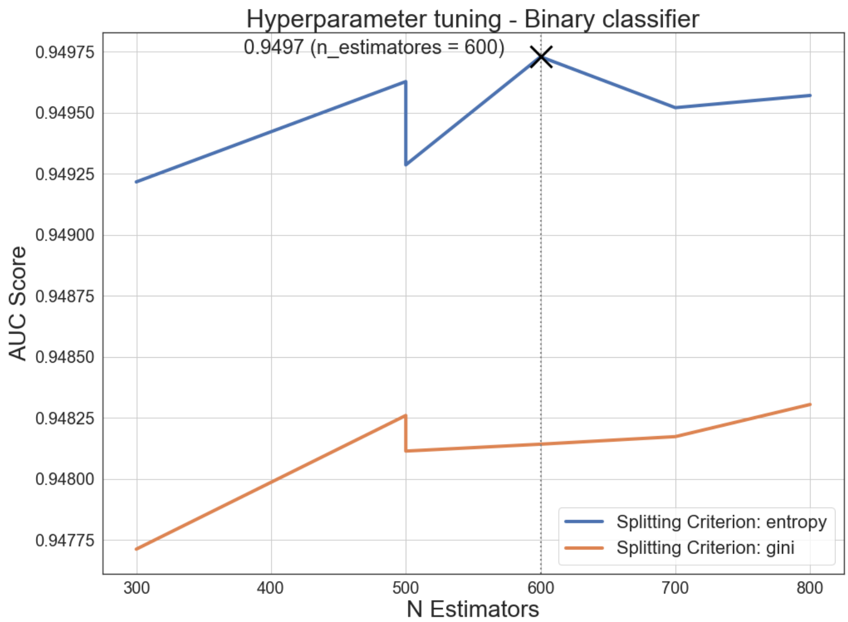


Figure 5.5: Grid Search with extended data - close-up view

with 34 features, 600 estimators, entropy splitting criterion and 26 levels of maximum depth.

5.3 Multi-class ensemble classifier

It somehow represents the core of our thesis, it models the starting idea: go deep inside bot identification and search and classify similar behaviours among them.

In this section we will expose the model involved in the multi-class ensemble. In this process, we used a Random Forest algorithm, working on all the crafted features; a KNN model, operating on the user attributes only;

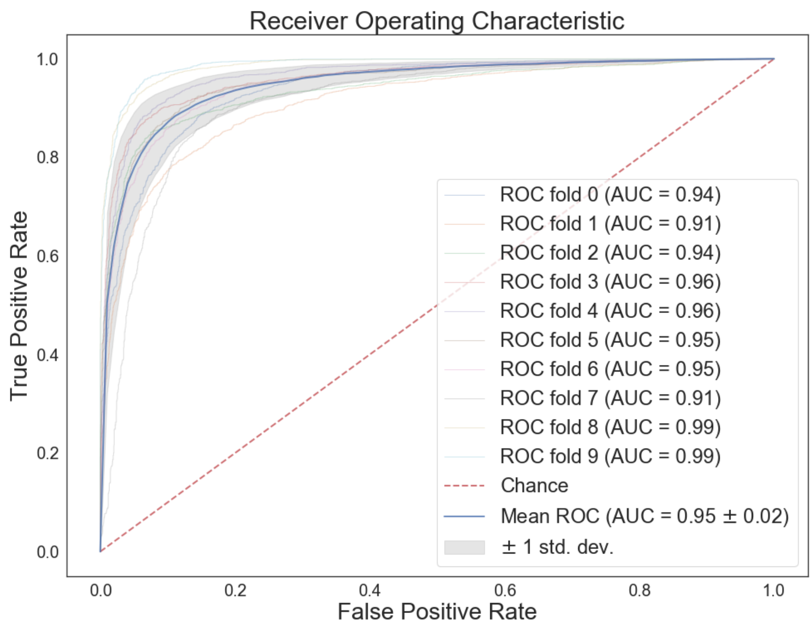


Figure 5.6: Bot-or-not with extended data ROC curve

a final text-based Naive Bayes classifier, which reads the tweets' texts and classifies them.

At first, this ensemble of these three models should have been blended with the prediction of the binary classifier. That means that the genuine class was part of the labels we were trying to classify, even in the multi-class models. Then, we found an issue in this approach: the binary classifier itself wasn't enough, even including it into the ensemble, to give the right importance to the genuine accounts. This problem emerged because of the others classifiers, as they were trying to classify the genuine class too. They lacked in data with that target, so, basically, they used to treat that category as one other of the bot types.

Even if the results on our validation sets were still good (we accomplished a F1 measure of 0.973), for the final ensemble method, we knew that this method would had yielded a poor bot vs genuine detection tool. We couldn't accept that situation, because, in order to go deeper than other works, in bot behaviour classifications, we had to provide a solid previous discrimination between humans and automated accounts.

The ensemble method with all the classifiers blended together were replaced with a pipeline, and the multi-class models were trained on bot categories only. These last classifiers had been put together inside a ensemble, which returns the final mutliclass probability prediction, based on the opinions of those models, as shown in Figure 5.7

The different nature of the classifiers, and the feature subsets as well, is one of the strengths of the stacking approach: it combines different opinions about the samples, driven by different classifiers, considering different parameters and attributes; basing on those unlike classifications, it builds its own.

It differs from other ensemble methods as bagging and boosting, because of this miscellaneous schema, and it can be a robust method to exploit the different characteristics of the classifiers stacked together.

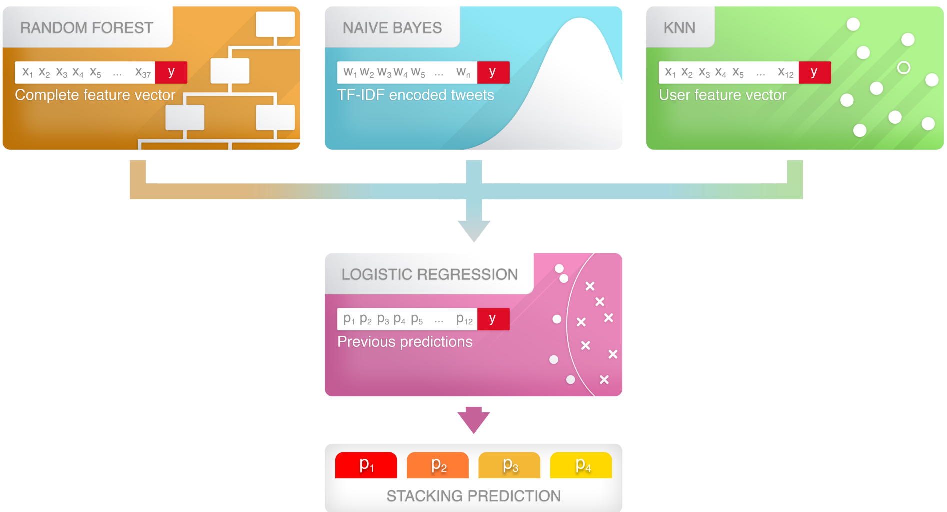


Figure 5.7: Multi-class ensemble schema

In the following subsections there are the detailed explanations of the three classifier announced before.

5.3.1 All-features-based Random Forest classifier

Dataset

During this phase, we used the previously described dataset 3.4.7 with its four different labels. The algorithm was fed with 21,445 samples and 37 features. the amount of records were light enough to consider K-fold cross-validation, without slow the validation down too much.

Model

We found ourselves in the situation in which we had some brand new features and we didn't know how useful they were. Obviously, we could appeal to heat-maps or other tools, to highlight the correlations among variables and targets. However, the model we wanted to develop was the Random Forest, which proved to perform well with F1 score. Since this kind of model exploits its criteria to employ the features, we needed to prove them with a direct approach.

Features selection

A useful advantage of the Random Forest algorithm is the ability to provide a feature ranking, according to its splitting criterion. We retrieved this standing, in order to see if we would have found some of the ones coming out from feature engineering at the top positions. The algorithm ranking ranked the features this way: 1. *favourites_count* (**0.179**), 2. *nsfw_profile* (**0.068**), 3. *freq* (**0.061**), 4. *tweet_intradistance* (**0.060**), 5. *news_spreaders_words_score* (**0.058**), 6. *statuses_count* (**0.053**), 7. *avg_len* (**0.051**), 8. *followers_count* (**0.051**), 9. *NSFW_words_score* (**0.043**), 10. *ret_perc* (**0.041**), 11. *min_len* (**0.038**), 12. *spam_bots_words_score* (**0.035**), ... 37. *min_fav* (**0.0001**).

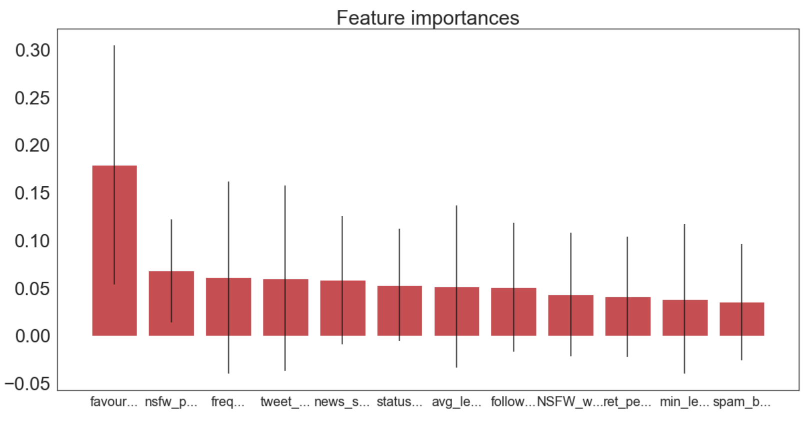


Figure 5.8: Random Forest top-12 feature ranking

As Figure 5.8 shows, we could find some of our crafted features inside this list: lots of tweets descriptive attributes (*avg_len, freq, ret_perc*, etc...), as well as the *tweet_intradistance* feature and three of the four extrinsic features, like *news_spreaders_words_score*, *NSFW_words_score* and the *spam_bots_words_score*. This picture confirmed us that the idea behind those features was useful.

Since those attributes were thought to belong to different clusters, we decided to try several combinations of those feature clusters, validating the model on them with a crossvalidation. The purpose of this stage was to see if some groups of features were enough to describe the real problem, or if some group would shown up as irrelevant. To face this evaluation, we performed a light-weighted Grid Search, which is a method that takes desired ranges of hyperparameters and tries all the possible permutations of them, looking for the best combination, in terms of a certain metric.

We are talking about a light-weight version of this tool, because we just went through different numbers of tree estimators in the forest. The different feature groups are not considered as hyperparameters and are not handled by the Scikit-learn implementation of the Grid Search. We had to manage the different training by our own, looking how the test score would have changed along with the increasing number of estimators and the different set of features.

Grid Search uses crossvalidation to find the better estimators for the models, and this approach was right for our situation. Due to the multi-class nature and some imbalances with the labels, we decided to follow the F1 score metric to asses the value of our model.

The features were organized in clusters, as described in Chapter 4. We had the user features, the descriptive features, the intrinsic features, the extrinsic and the image features. Then we tried the model with the entire set of 38 attributes. As shown in Figure 5.9, the best configuration seems to involve the whole set of features, as it reaches these scores, with 100 estimators: *Precision* = **0.978**, *Recall* = **0.976**, *F1* = **0.977**.

The model has been tested with the default value for the maximum depth in the trees, which is set to 'None'. It means that the trees are expanded until every leaf is pure, or all leaves contain one sample.

In order to try all the alternatives, we setted a test involving the performance of the model, when it was working on an increasing number of features. We had the ranking provided by the forest itself, so we started by testing only the most important attribute, adding one feature at time, until the least important was included. We were looking for some changing in the scores, that would have pointed to a lighter model, with the exclusion of some features. Figures 5.10, 5.11, 5.12 show the trends of the Precision, the Recall and the F1, respectively, along with the number of features tested.

As all the Figures show, the best solution possible, looking at both the three metrics, is the one involving all the 37 components of the feature

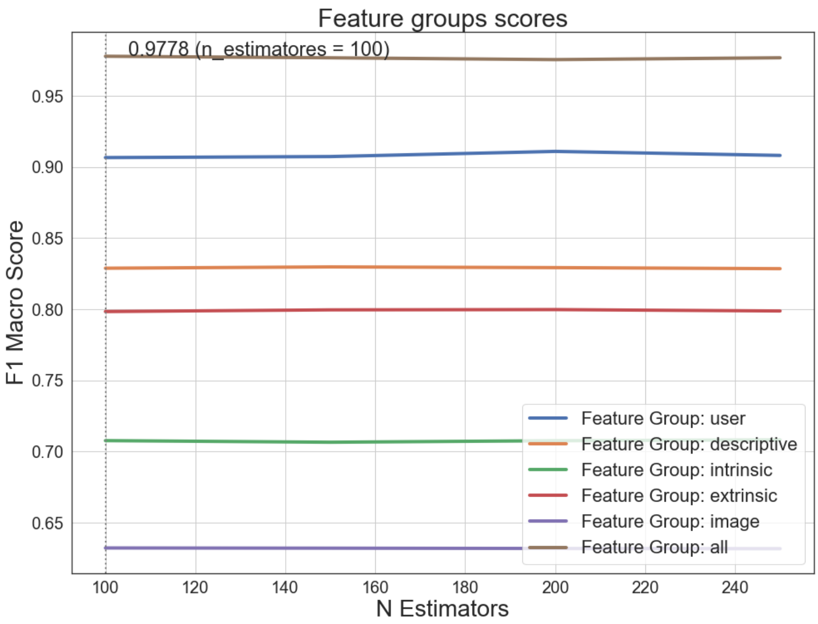


Figure 5.9: Performance over different feature clusters

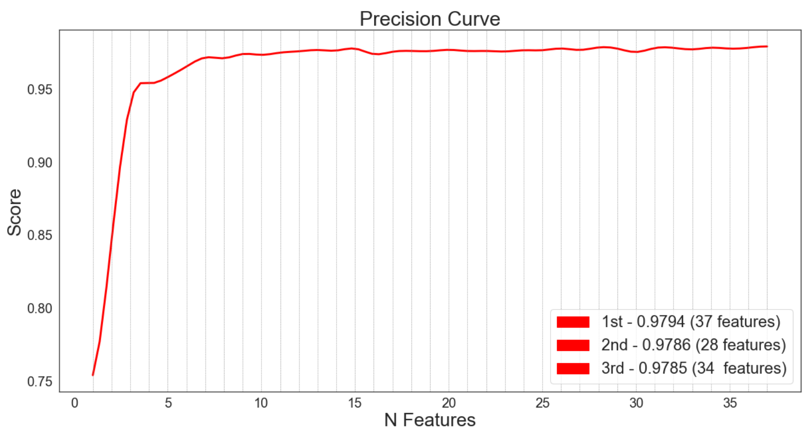


Figure 5.10: Precision trend along with number of features tested

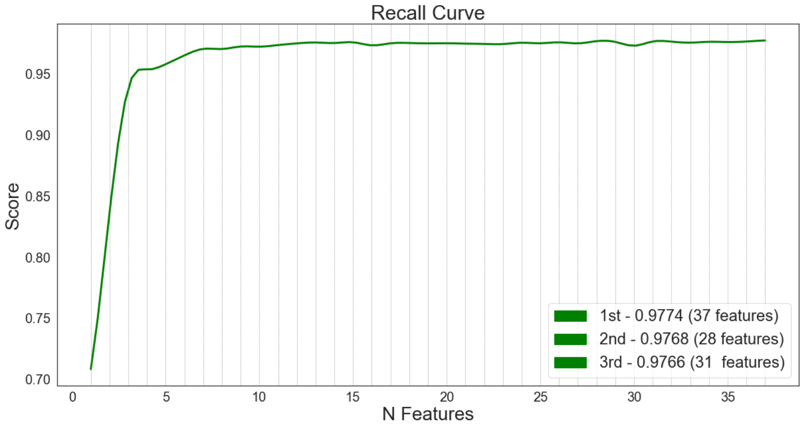


Figure 5.11: Recall trend along with number of features tested

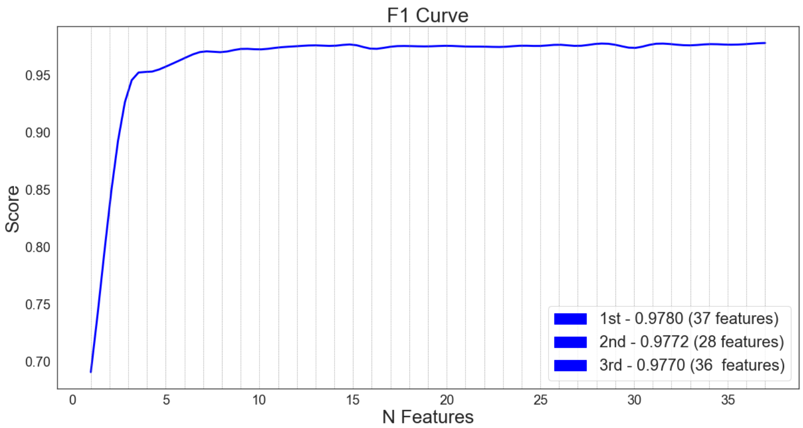


Figure 5.12: F1 trend along with number of features tested

vector. There was the possibility to choose the second result, which wanted only the first 28 features, in terms of importance for the Random Forest. However, we weren't struggling with heavy models or long prediction times and the Random Forest algorithm handles the overfitting problem properly, even with complex models. Thus, we moved on with the entire feature vector as support for the classification goal.

We then continued with a proper Grid Search over the whole number of features.

Hyperparameters Tuning

The algorithms rely on parameters in order to fit a problem. Once the number of featured was picked, as well as the model, we needed to consider the possible hyperparameter ranges. The Grid Search method from Scikit-learn helped us, once again, during this exploration. Since we were testing a Random Forest, we wanted to play with the number of estimators (trees) to include in the pool, as well as the maximum depth of each tree and the splitting criterion.

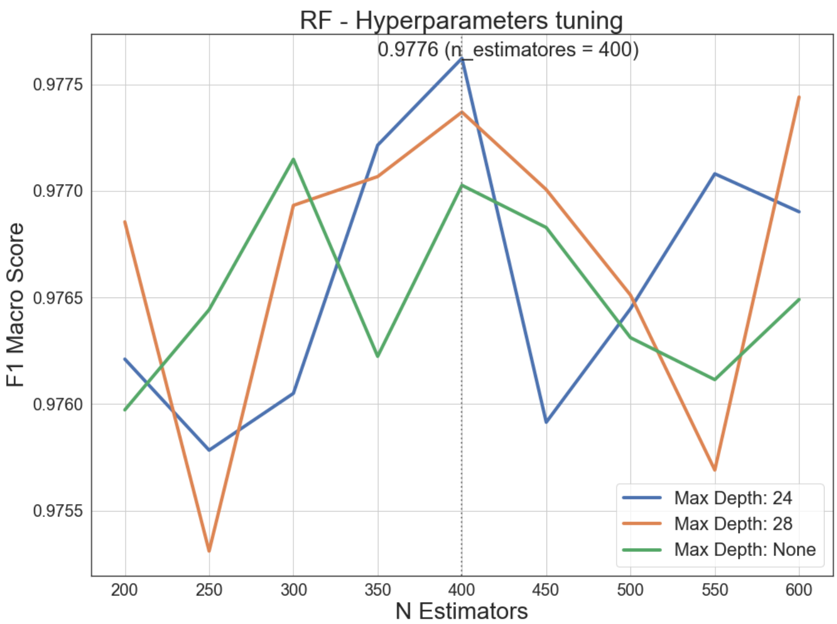


Figure 5.13: F1 scores with "Gini" criterion - close-up view

The Figures 5.13, 5.14 show how the average F1 score, measured on 10-

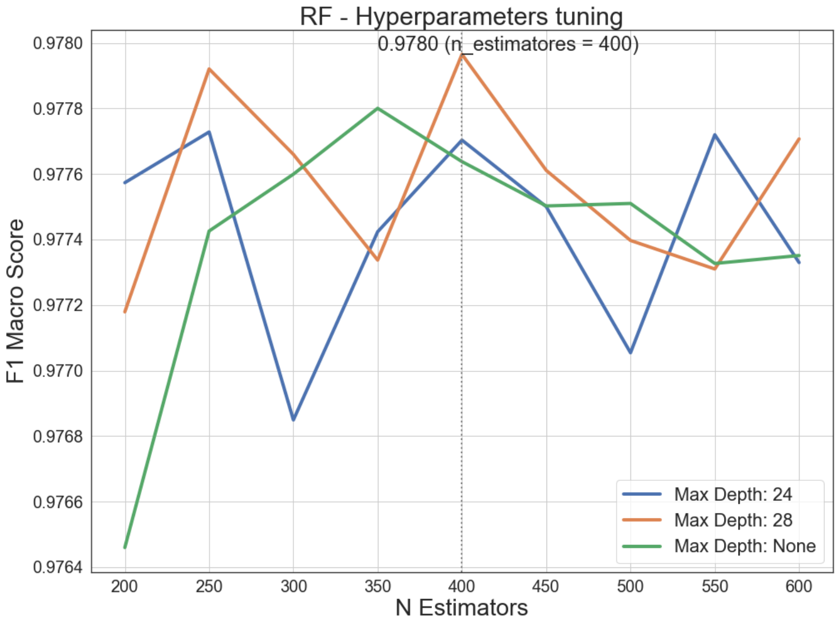


Figure 5.14: F1 scores with "Entropy" criterion - close-up view

fold crossvalidation, changes with the increasing of the number of estimators in the forest. The different coloured lines represent the *max_depth* hyper-parameter. The first Figure (5.13) shows the Grid Search results, with the *gini* splitting criterion. The second one (5.14) represent the situation having *entropy* as a splitting choice. We combined nine numbers of estimators (from 200 to 600), together with three different maximum depths for the trees (26, 28, None) and the two above-mentioned splitting criteria.

We could observe a peak, for both criteria, in correspondence with 400 estimators. Although the Gini-based forest's score didn't seem a bad point, we went with the Information Gain splitting criterion, which si also the same we used to rank the features of our data.

The final configuration involves 400 trees, the Entropy criterion and the maximum depth (for each estimator) equal to 28 levels, yielding the following scores in crossvalidation:

- *Precision*: **0.979**
- *Recall*: **0.977**
- *F1 score*: **0.978**

Figure (5.15) shows an example of one of the estimators of the final model, plotted with Matplotlib library for Python. It has been represented with the first two levels of depth, for visualizations reasons.

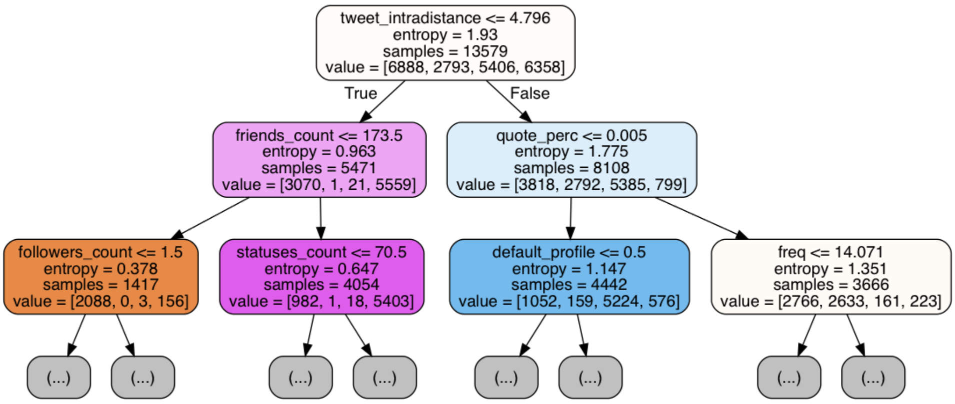


Figure 5.15: Tree estimator of the Random Forest model

As the picture shows, this tree used the *tweet_intradistance* feature as root, in order to perform its first split on that attribute.

The first algorithm of the multi-class ensemble was completed and ready to be combined with the following models.

5.3.2 User-based KNN classifier

The Random Forest model represents somehow the core of the ensemble, as it was trained on the entire feature vector, with all the data we had for the purpose. A massive attention for parameters and features were given for that classifier. However, we wanted to put it into an ensemble, not to improve its already strong stability over outliers, but to support it with different perspectives.

We noticed, as shown in the previous Figure 5.9, that the user features were a good group to build a classifier on. Thus, we started thinking how to implement such support, and we basically looked at our baselines.

The model that had the best performance, not considering the Random Forest, was the K-Nearest Neighbors algorithm.

We didn't want this model to work the entire feature vector, because we knew that it would had been overlooked by the Random Forest. Instead, we wanted it to concentrate on the features that describe the users, without the information driven by their tweets. Therefore, we relied on the user features, with the extension of the image feature that assesses the NSFW score to the profile picture. This extension was due to the fact that such feature doesn't need the user's tweets to be computed; it can be seen as one of the user features as the others of that group.

We hoped that treating the data before, or during, the training phase, would had brought to a good sustain for the first multi-class model, where needed.

Dataset

The dataset is composed of the same number (21,445) of samples of the first multitarget Random Forest, but preserving only the twelve features belonging to the user group, plus the NSFW_profile attribute coming from the image features group.

Feature vector
default_profile, favourites_count, followers_count
friends_count, listed_count, screen_name_len
statuses_count, url, description_len, NSFW_profile
name_len, profile_use_background_image, age

Model

This model is quite simple and doesn't require much effort in interpolating several hyperparameters. But this doesn't mean that it is a closed box algorithm. It can be improved by paying attentions to some details. In particular there had been done three considerations, and they were regarding

- ✏ **Hyperparameters**
  The main hyperparameters that can be combined together are the number k of neighbours ti consider, when performing a prediction, and the distance metric used by the calculations of the distances. The Scikit-learn implementation of the algorithm uses the *Minkowski* metric, which describes the $\mathbf{L}_\mathrm{p}$ norm:

  $$L_p = (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p}$$

  This metric is a generalization of the $\mathbf{L}_2$ norm, also known as the Euclidean distance. The library we used allowed us to tune the p hyperparameter, which indicates the metric used: $p = 1$ means L_1 norm, the Manhattan Distance, when $p = 2$ stands for the Euclidean. Higher values of p are available, with other metric involved.

We tried several values for both p and k, obtaining different results. However, the overall decision had to take in consideration also other two factors, the features weighting and the kernel function.

- **Features Weighting**
  The KNN algorithm aims to map the samples into a space and then it looks for the distances among them. In order to have best mapped space, it is reasonable to perform a weighting over the data. With this approach, the mapping would change and some points could result both as closer or more distant from each others.

  We performed some tests on four configurations. The first one didn't involve a weighting vector, while the second one was a standard normalization of the features. This last option provided weak results, with the same configurations of parameters, and thus were discarded from further analysis.

  The third option, which was the most interesting, was built by computing the Information Gain for each feature, using the inner ability of the Random Forest classifier, and then by applying the weights on the attributes, basing the coefficients on the scores provided by the ranking. This method emerged as the most effective, in the overall score.

  The last one was implemented as a Gaussian Kernel, which exploits the above-explained Radial Basis Function 5.1.4. This attempt fell shorter then the standard normalization, and it could be imputable to an overestimation of the ability to approximate the probability density of our data. It seemed that the Gaussian-based weighting wasn't a good fit for the problem.

We ran a Grid Search session over an increasing number of neighbours and the first four coefficients of the Minkowski distance. All the combinations were tested with both the non-weighted data and the Entropy-based weighted data. The results obtained are highlighted in Figure 5.16

The different colours represent the Minkowski distances tested, as well as the different line styles, dotted and continue, represent the weighted and the non-weighted solutions, respectively.

As visible in the picture, the best score, in the usual 10-fold crossvalidation, has been obtained with 5 neighbours, the Manhattan distance, and the weighted solution, which is averagely better than the other one.

The final model has been created with the following call: *KNeighborsClassifier(p=1, n_neighbors=5)* and it has been fitted with the weighted data,

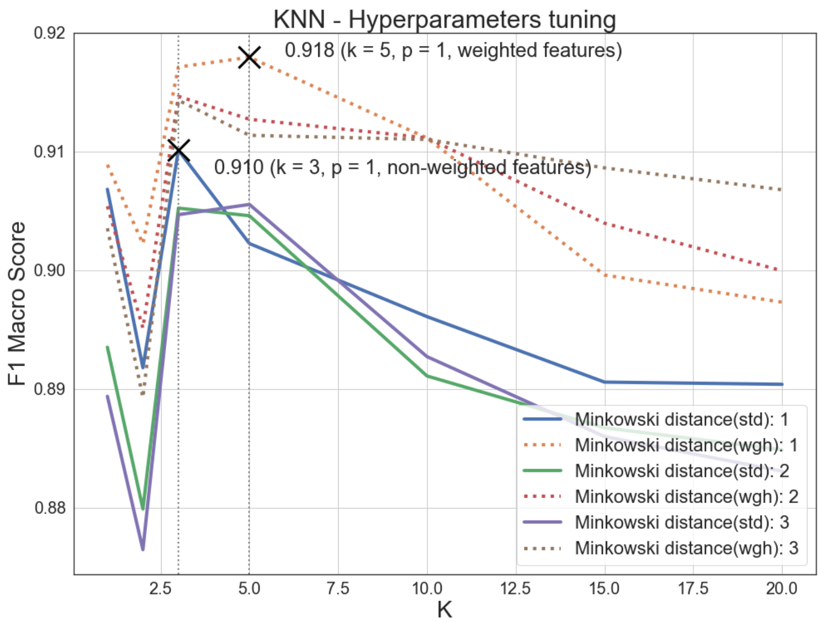


Figure 5.16: Hyperparameters p, k tuning, with different weightings

obtaining the following scores in 10-fold-crossvalidation:

- *Precision*: **0.924**
- *Recall*: **0.917**
- *F1 score*: **0.918**

5.3.3 Text-based Naive Bayes classifier

In chapter 4 we created some useful features that helped the above-mentioned classifiers. Anyway we didn't consider enough the tweet texts. A first idea consisted in add a feature that describe the context of the tweets. This task was easily viable using *Google Cloud Natural Language*. Unfortunately, this service only provides a few free calls, and we would not be able to tag all the tweets in our dataset. Moreover, the Context Classification is very specific, and we would have risked to have a too large domain of values. We tried to classify some tweets and we noticed that it was not possible with many of them, since they were just exclamations or contextless sentences. We therefore decided to train a proprietary text classifier. This allowed us to classify texts according to our targets, and our final classifier would have been self-sufficient, without external services.

Dataset

Since we wanted to classify texts instead of users, we needed to create a specific dataset. It had to contain a list of tweets and all the related target. We composed it by binding the target of each tweet to the label of its author. This produced some noise; often bots tweet something different from their goal, to seem more humans. We could accept this problem, since most of the tweets are aimed to a goal. The shape of the dataset is the following:

category	#tweets
NSFW	196712
News-Spreaders	280300
Spam-Bots	453719
Fake-Followers	41316

It seems unbalanced, anyway it is appropriate to keep all the data. Some Spam-Bots tweets just contain links and they will be removed in the next steps. Differently, Fake-Followers usually don't tweet, or they never tweeted.

Model

In order to classify texts, we decided to use a Naive Bayes approach. This algorithm consists in a probabilistic classifier based on the Bayes' theorem. "Numerous researchers proved that it is effective enough to classify the text in many domains [22]. Naive Bayes models allow each attribute to contribute towards the final decision equally and independently from other attributes" [23].

Tweets can't be processed as they are. Since this model aims to classify texts basing on its words, we had to clean the dataset from all those parts of the text that are not real or useful words. For example articles, smiles, punctuation should not be taken into consideration. Moreover it is fundamental to reduce inflected words to their word stem. Finally, we need to clean all those noisy parts of the tweets, which is important to allow the final text classifier to consider only real words, in order to identify the context.

Tweets without this pre-processing step look like this:

'RT @SteveSchmidtSES: TRUMP disgraced the Presidency and the United States at the G-7 summit. From his slovenly appearance to his unprepared... https://t.co/KiT29FvJw5'

In order to perform these tasks, we used a *sklearn Pipeline*. It is a method that allows to build a custom predictive algorithm. In our case, it contains intermediate steps of data transformation and finally the machine learning algorithm. It works like a generic model, but it perform all the included transformations before processing a data, both in training and prediction steps.

We added to the pipeline the following operations:

- **Remove retweet information:**
  Delete the texual pattern which indicates that the current status is a retweet. It consists in a "RT @original_author:".

- **Remove punctuation:**
  With regular expressions we removed everythink different from characters and numbers. In this step also smiles and other symbols are removed.

- **Remove stopwords:**
  stopwords are the most common words that are always used in a language and that can not help to classify a context. Some example of words belonging to this category are articles or prepositions. We removed them, by using a stopwords dictionary of *NLTK* libraries.

- **Transform uppercase characters into lowercase:**
  Before tokenize words, we transformed every character into lowercase, in order to be sure that every word is considered only in one form.

- **Apply stemming:**
  This is the step where words are reduce to their word stem. Since the text classifier is based on the occurrences of words in the texts, we don't need a correct grammar in our tweets. Instead, words at their basic form are more useful for the target. In order to perform this transformation we used *SnowballStemmer* from *NLTK* libraries.

- **Apply TF-IDF encoding:**
  Finally we applied to each word a TF-IDF encoding. Since we had a huge amount of tweets, without this step we would have risked to give too much importance to the overused wordsand almost nothing

importance to the others. Moreover, without using TF-IDF we had a worse performance.

- **MultinomialNB:**
  This is the final classification algorithm. Thanks to the pipeline it always receives cleaned data, performing a better training and predictions. We selected a Naive Bayes classifier for multinomial models since we are dealing with a multi-class problem.

Holdout evaluation

The final text classifier has the following performance:

- *F1 score:* **0.71** with TF-IDF
- *F1 score:* **0.64** without TF-IDF

Since the other models classify users and not single tweets, we could not use a classifier for texts only. In order to get a prediction on users, based only on their tweets, the final classification script compute the resulted probabilities for each tweet. Then, for each user, the final prediction consists in the mean of the predictions over his tweets.

Chapter 6

Prediction

6.1 Stacking meta-classifier

In order to classify bots, we had three models, each with different purposes, but they had to cooperate for the bots' behaviour identification. The initial idea was to use only the multi-class Random Forest to classify the bot categories, using the other two models as meta-models to build extra features with their outcome. Those features would have had the dataset enhanced, but their meaning would have been bounded to the multi-class classifier limits. We wanted to give the right importance to each model, hoping they would help each other to better distinguish the patterns end to better model the real problem.

We thought about several methods to exploit their strengths and combine them. In particular, we thought about a genetic approach and a stacking ensemble with a meta-classifier. We wanted to evaluate the performance obtained by these methods and chose the one that fitted our need.

Both the genetic and the meta-model were trained with holdout technique, splitting the whole dataset into training and test sets. The 70% of the samples ended up into the training set, the 30% in the validation set. We had a training set for the ensemble models that contains 6,434 entries. The data that fed the stacking methods were the predictions of the tree classifiers, over the validation set. In order to make those prediction without cheating, we couldn't use the models that were already fitted with the hole data. We had to retrain them with the 70% of the records. We didn't perform further Grid Search to find the best hyperparameters in this stage, because the final script that we were going to assemble was taking into account the entire dataset to train the models. Furthermore, this small variation, in terms of amount of training data, wouldn't had led us into a misinterpretation of the problem, if we had kept the same hyperparameters found earlier. We decided to stick with the configurations already found and to train the model with fewer data.

Once the models were fitted, we used the *predict_proba()* method of the Scikit-learn implementations of the classifiers, in order to retrieve "soft classifications". This method differ from the standard *predict()* function, because it doesn't return the predicted target for that sample, but it returns the computed probabilities for each class. The way a probability is given depends on the classification algorithm, i.e. Random Forest assesses the probability by retrieving the votes that its leaves give to each class. The *predict()* method, instead, after the probability computation, returns the most probable target. We didn't want our model to assign a strict label to an unseen sample, indeed, we were interested in the percentage of categories membership. We used this method to construct the output vectors needed to train the stacking models.

Each sample of this new dataset contains 12 elements, 4 soft predictions (one for each category) for each classifier (complete Random Forest, text-based Naive Bayes and user-based KNN).

New sample

KNN prob.				**NB prob.**				**RF prob.**			
p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}

A new training set was born and it was built with the soft classifications of the models in the pool, over the validation set. It was ready to proceed and to serve the ensemble models.

The pipeline for perform the training of the ensemble models is resumed in Figure 6.1.

After several parallel attempts were done, the blending system was built with the meta-classifier.

6.1.1 Genetic algorithm

This approach started as a side way, when we were already testing the stacking ensemble.

The idea behind genetic programming, is to emulate the natural species evolution, by encoding the the *chromosomes* in the process with data structures. The chromosomes represent the possible solutions for the problem and they have to "evolve", in order to get fitter and fitter for the goal. Several

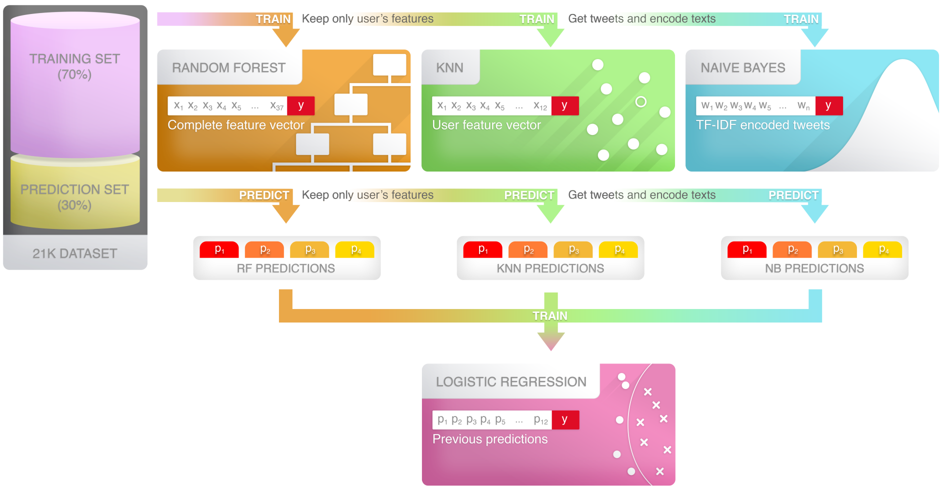


Figure 6.1: Training pipeline of the stacking ensemble

operators must be determined to perform this evolution. Once a first *generation* of feasible chromosomes has been formed, they have to be evaluate according to a *fitness function*, which asses how well a chromosome faces the problem. The best portion of chromosomes are picked to be part of the next generation, and this is called *elitism*. The solutions left are given a probabilities to join the elite ones, in order to form a new generation with about the same size as the previous. This step is called *selection*. The chromosomes picked in the selection stage are assigned a high *crossover* probability. The crossover operator handles the "born" of new chromosomes, mixing parents alleles in a certain way. The mixing method is highly correlated to the chosen encoding strategy. Each newborn is give a low probability to undergo a mutation. This step often seems useless, but it's pretty important, in order to explore a higher spectrum of solutions, which couldn't be expanded by the mating operators only. After the new population has been accepted, it is ready to be validated through the previously define fitness function. The loop holds, until a solution is found, or, like in our case, the process sticks to a local or global maximum.

Genetic operators

We setted the genetic algorithm with the support of the Deap library for Python, setting these operators:

- *Encoding*: each chromosomes represented a weighting vector for the outcome of our three classifiers. Each allele of the chromosome was float valued, with numbers between 0 and 5, generated randomly, with a uniform distribution. We started with normalized weights, but the spectrum of the solution explored was way too poor to fit the needs. This range was given after observing the weights that the Logistic Regression model were assigning to the inputs received, that was wider and involved even negative values. We randomly generated 200 chromosomes for the initial population, with this form.

Chromosome

KNN weights				**NB weights**				**RF weights**			
w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}

✏ *Fitness evaluation*: the fitness function that assessed the value of the solutions was somehow similar to the one used in the other stacking method. We applied the weights of our chromosomes to the samples in our dataset.

For each sample, we made pairwise additions, among the outputs of different classifiers, multiplied by the chromosome's weights, for the same category:

User-based components
$p_0 * w_0 \ldots p_3 * w_3$

$$+$$

Text-based components
$p_0 * w_0 \ldots p_3 * w_3$

$$+$$

All-features-based components
$p_0 * w_0 \ldots p_3 * w_3$

$$=$$

Resulting prediction
$\mathbf{p_0\ p_1\ p_2\ p_3}$

In order to stick to the probabilities nature, the computed prediction had been normalized.

That prediction has been compared with the known real target for the examined sample. Since the targets of our dataset aren't soft valued, we took the maximum probability of the computed prediction to make the comparison with the actual class. Our fitness function aims to favourite those solutions which maximizes the F1 macro score, as it has been for the validation of the classifiers, until this stage.

During this process, the problem we had to face was that we wanted to produce soft classifications, because we knew that our collected data presents similar patterns within the same categories. This means that the algorithms easily classify our test set, because of the distinctive traits found for each target. In order to mitigate the real test error, over unseen samples, we wanted the prediction to be as smooth as possible, without confusing the F1 score interpretation.

We faced the problem involving a smoothing factor to our fitness function.
When computing a sample, we populated a Confusion Matrix of the prediction, using the above-mentioned method to match predicted and actual classes. The matrix helped us computing the F1 macro score easily. At the same time, we counted every hard classification, marking as 'hard' every computed prediction that contained a probability greater or equal to **0.8**, among its five stored values. This count was used as a penalty, it has been averaged for the number of samples, and then subtracted to the computed F1 score of that chromosome. In order to privilege the maximization of the F1 factor, instead of the minimization of the penalty, the final fitness function assigned this score to each chromosome:

$$Fitness = 3 \times F1\_score - Penalty$$

This way to operate didn't affect the overall F1 of the sample, since penalizing hard classifications didn't discourage the system to look for values high enough to have a dominant category in the prediction.

Once every chromosome has been evaluated, they could proceed to the next steps of the algorithm.

✏ *Selection*: the selection phase handles the choice over which chromosomes pick for mating. Several pre-implemented methods are available, but we used the tournament method. It works selecting the size K of the tournament, which we chose to be 3. Then, it randomly selects K (3) chromosomes from the population and places it inside a pool.

Then it compares their fitness. The chromosome with the best fitness has probability p (the crossover probability) to be selected for mating. The second has $p*(1-p)$ chance to get selected, the third $p*((1-p)^2)$.

- *Crossover*: The crossover probability has been setted to 95%. The crossover operator wasn't something already implemented by the library, as for the fitness function. Our operator used to produce two brand new chromosomes for the next generations. The first child is the unweighted mean of its parents:

  $$[x_0, x_1, ...x_{11}] \oplus [y_0, y_1, ...y_{11}] = [\frac{x_0 + y_0}{2}, \frac{x_1 + y_1}{2}, ... \frac{x_{11} + y_{11}}{2}]$$

  The second child is the weighted mean of its parents, computing the weights with respect to the fitness of the two mating chromosomes:

  $$f_x = \frac{fitness_x}{fitness_x + fitness_y}$$

  $$f_y = \frac{fitness_y}{fitness_x + fitness_y}$$

  $$[x_0, x_1, ..., x_{11}] \oplus [y_0, y_1, ..., y_{11}] = [x_0 * f_x + y_0 * f_y, ...x_{11} * f_x + y_{11} * f_y]$$

  The retrieved children used to be part of the upcoming generation.

- *Elitism*: This part was necessary, in order to not lose the best solutions found so far. It is a sort of insurance, which guarantees to keep, at least, the best situation until this stage, and to let it take part of the next generations of solution. We preserved our three best chromosomes for each generations and move them to the next stages.

- *Mutation*: The mutation probability is generally setted to low values, like what happens in nature. It represent the error in DNA replications from the parents and it shouldn't reach the 1% of probability to occur. Although, we wanted to force some mutation, because, as said before, we needed a wider space of solutions and the elitism helped us in containing the damages of such mutations. In the worst cases, all the chromosomes have had been damaged and resulted as useless, but the elitism had preserved the best ones and kept it untouched. So we imposed a 45% of mutation probability, for each newborn solutions, before entering the pool.

  Our mutation operator was a decoration of the value changing method already implemented: we randomly used to pick three elements from the chromosome and set them to zero.

Results

After several runs of the genetic program, with boosted starts (the best solutions found at the previous run were placed inside the first generations of the following runs), we stuck in a maximum of the score. In the last run, from the 5_{th} generation there was no improvements in the fitness of the best solution. We selected the fittest chromosome, whose scores were:

- *Weights*:

KNN Weights			
2.452	4.104	0.0	0.0
NB Weights			
4.766	1.0	0.0	0.0
RF Weights			
3.790	1.0	1.08	2.506

- *Fitness*: 0.768
- *F1 score*: 0.44
- *Percentage of hard classifications*: 55.3%

The results were discouraging, compared to the singular scores of the models involved. It was worth to try this approach, but we were aware that a "simple" weighted mean of the outcomes of the classifiers weren't enough to describe the problem.

Thus, we built a different and more sophisticated stacking method.

6.1.2 Logistic Regression

The reason behind the choice of a meta-classifier is that we wanted a more complex way to perform inner weighting of the outcomes that we had from other models. A simple weighted mean wasn't enough for this purpose. Furthermore, implementing a logistic-like loss to evaluate the fitness of a genetic algorithm would have mean to apply the Logistic Regression training model, without performing gradient descent, but with a genetic approach. It would have been just unnecessary and computationally expensive. Thus, we discarded the idea of using the genetic programming to emulate a Logistic model, even if the smoothing factor used for that try was a good insight for our task. In order to mitigate the lacking of soft classifications, we chose to rely on the regularization factors that belong to the training algorithm of the Logistic Regression. This kind of models is often involved in stacking other classifiers, with a binary purpose.

Dataset

The same training set has been used for train both the Genetic and Logistic models. Since we were managing a multi-class datasets, we knew that the ensemble meta-model would had been adapted to this job. The most common tool used for stacking purposes is the Logistic Regression, which performs well on binary separations. We decided to test this model on a multinomial approach, with a softmax activation function, instead of trying the already visited One-vs-Rest method.

Comparison with Random Forest

We didn't want to blindly select this model over some others tool, especially over Random Forest, which proved us to perform well in multi-class classifications. Thus, we tested these two algorithms with the new dataset. We ran some default configurations of the models, in order to have a raw comparison to trace a line between them.

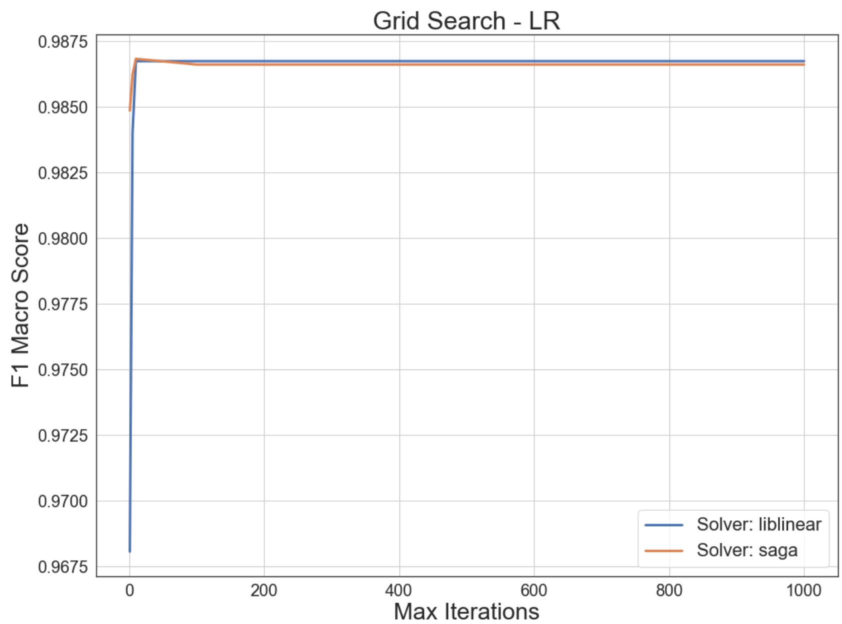
Figure 6.2(a) shows the early convergence of the Logistic model's F1 score, with low maximum iterations. The model has been tested with Lasso penalty and two different solvers, but the results, over the increasing of the training epochs, are way similar. Although the Random Forest, as shown in Figure 6.2(b), tops the performance of the Logistic Regression, the scores were close enough ($\mathrm{F1}_{\mathrm{LR}} = 0.9868$, $\mathrm{F1}_{\mathrm{RF}} = 0.9874$) to give a chance to the Logistic model, in order to try its regularization terms.

Hyperparameters tuning

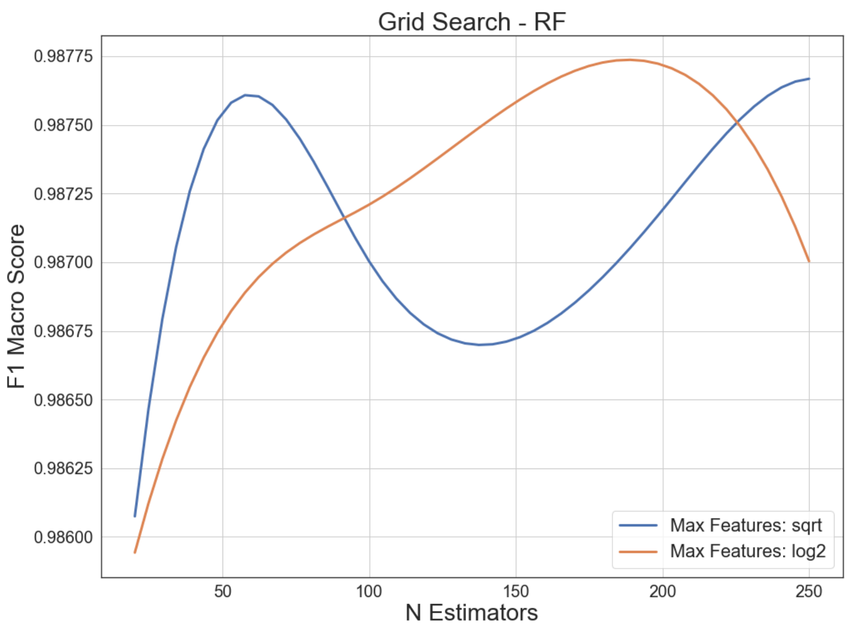
We tried two regularization terms for the Logistic model and several numbers of maximum iterations for the training algorithms. The regularization terms are parameters computed in addition with the minimization of the characteristic loss function. Their purpose is to avoid the weights to explode and the model to become more sensitive to noisy data. In other words, they are involved to prevent overfitting. The idea is that the loss function, gets modified as follows

$$\mathbf{L}(\mathbf{w}) = L_D(w) + \lambda L_W(w)$$

Where $L_D(w)$ represent the error on the data and $L_W(w)$ is the term representing the model complexity. In general, smoother weights implies lower model complexity. The lighter the complexity, the lower the variance of the



(a) LogReg with raw settings



(b) Random Forest with raw settings

Figure 6.2: Stacking models comparison

model and the risk perform overfitting. The parameter λ has to be tuned with a validation method.

The penalties that we explored were:

- ✏ *Lasso (L_1)*:

$$\mathbf{L_1}(\mathbf{w}) = \frac{\lambda}{2}||\mathbf{w}||_1$$

  where $||\mathbf{w}||_1 = \sum_{i=1}^{N} |w_i|$

  This regularization function is non-linear and doesn't provide a closed-form solution. It tends to cut out some features from the model, yielding to sparse and lighter model. It can be seen as an implicit way to apply features selection.

- ✏ *Ridge (L_2)*:

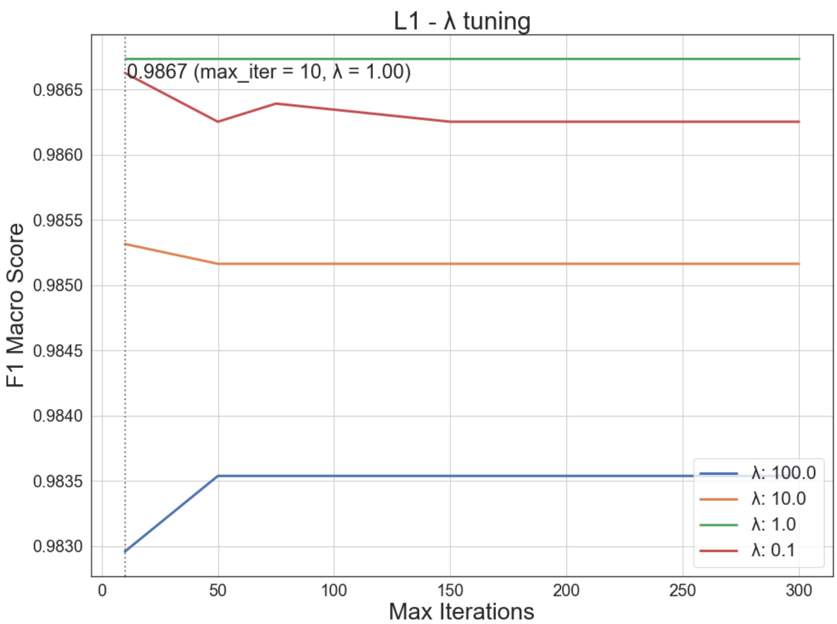
$$\mathbf{L_2}(\mathbf{w}) = \frac{\lambda}{2}||\mathbf{w}||_2^2$$

  where $||\mathbf{w}||_2^2 = \sum_{i=1}^{N} w_i^2$

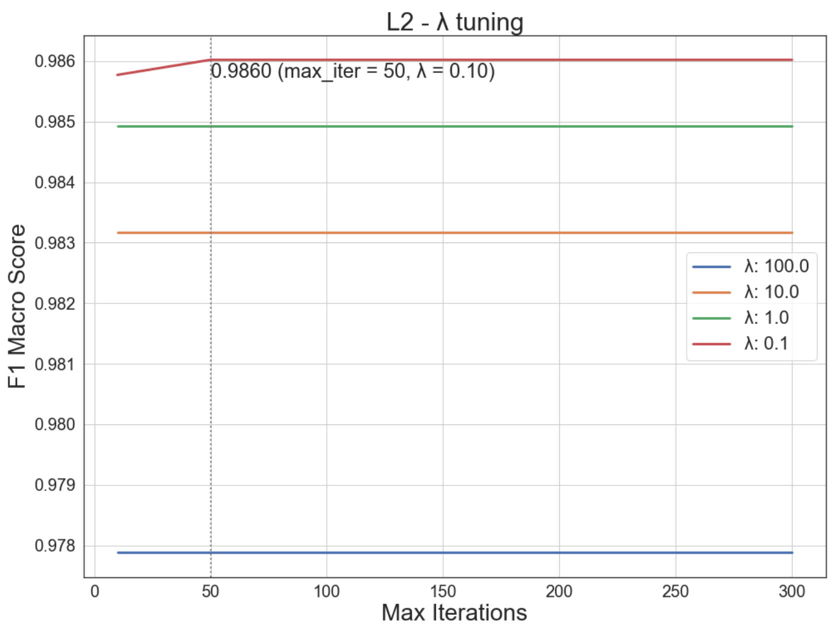
  This softer term tends to shrink the weights, keeping the loss function quadratic in $\mathbf{w}$ and closed forum solution exists.

Figure 6.3(a) highlights the slightly better results obtained with the Lasso penalty, with unitary λ coefficient (Lasso F1 score: 0.973). As Figure 6.3(b) shows, the ridge penalty needs to be weakened ($\lambda = 0.1$) in order to get close to the Lasso performance, which is a compromise hard to deal with. The smaller is the regularization coefficient, the higher is the model complexity, as said before. Moreover, we decided to gather further consideration, by looking inside the weighting applied by those two terms. Since we didn't have a lot of data for the training, we wanted to keep the regularization high enough to not fit the noise in the model.

We took a look inside the weighting performed by the model, with both L_1 and L_2 regularizations, in order to catch some insight from them. The weights are composed by four vectors of twelve elements each: each vector represent the weights applied for a One-vs-Rest target classification, and each element of the vectors are mark the features the model has been fitted on. As said before, each of the three groups of four features represents the probabilities, for a sample, of membership to the classes. In order to give a good representation, we considered just one vector of twelve elements, computed by averaging all the weights applied in all the OvR predictions.



(a) Lasso penalty, $\lambda = (0.1, 1, 10, 100)$



(b) Ridge penalty, $\lambda = (0.1, 1, 10, 100)$

Figure 6.3: Regularization coefficient tuning

Ridge regularization

KNN mean weights			
NSFW	*NS*	*SB*	*FF*
$3.4e^{-16}$	$1.7e^{-15}$	$3.2e^{-15}$	$-1.1e^{-14}$
Naive Bayes mean weights			
NSFW	*NS*	*SB*	*FF*
$2.8e^{-15}$	$-1.1e^{-14}$	$9.8e^{-15}$	$4.8e^{-15}$
Random Forest mean weights			
NSFW	*NS*	*SB*	*FF*
$8.7e^{-15}$	$-8.3e^{-15}$	$1.0e^{-14}$	$-1.7e^{-14}$

The Ridge regularization leads to very small weights, and negative ones too. Even with unitary λ coefficient, it is hard to distinguish a discrimination among features. This approach would have yielded a smoother model, but with the ability to give a chance to every classifier to distinguish among targets. We wanted to get some further insights from the other weighting.

Lasso regularization

KNN mean weights			
NSFW	*NS*	*SB*	*FF*
-0.660	-0.185	-0.465	-0.830
Naive Bayes mean weights			
NSFW	*NS*	*SB*	*FF*
0.578	-0.054	0.072	0.0
Random Forest mean weights			
NSFW	*NS*	*SB*	*FF*
1.993	2.147	2.049	2.571

The L_1 term, as expected cut out some features from the model. Looking at the excluded attributes, we noticed that the regularization caught the strenghts and the weaknesses of the classifiers. The text-based Naive Bayes classifier seemed to be useless when it came to detect Fake-Followers. It seemed legit, as the dictionary used by that category is the smallest and the most heterogeneus (lots of non english words are involved) in our dataset. Lasso seemed to understand this behaviour and decided to not consider the opinion of that classifier, when it has to give its opinion about that bot cateogry. Another insight got from L_1 is coming again from the Naive Bayes algorithm. The model couldn't distinguish with certainty Spam-Bots, since they act in a similar way, with respect to NSFW accounts. They just tweets click-baiting links, with catchy captions. A "blind" classifier struggles in understand the nature of those links. This is the reason that the contributes of the tex-based model had been almost discarded from the stacking meta-classifier.

Since we knew our dataset and we were aware of the bias it might contain, we preferred a lighter and sparser model, over a more complex one, even when the F1 scores used to match. We wanted our model to infer on new unseen data and to be ready to give a representative statistical description of the actual situation on Twitter. We had to be far-sighted and not to recline on the accomplishments of the 10-fold crossvalidation. We thought that the Lasso model would have been performing better in out-of-box predictions.

We kept the L_1 penalty, with $\lambda = 1$ and proceeded with the hyperparameters tuning.

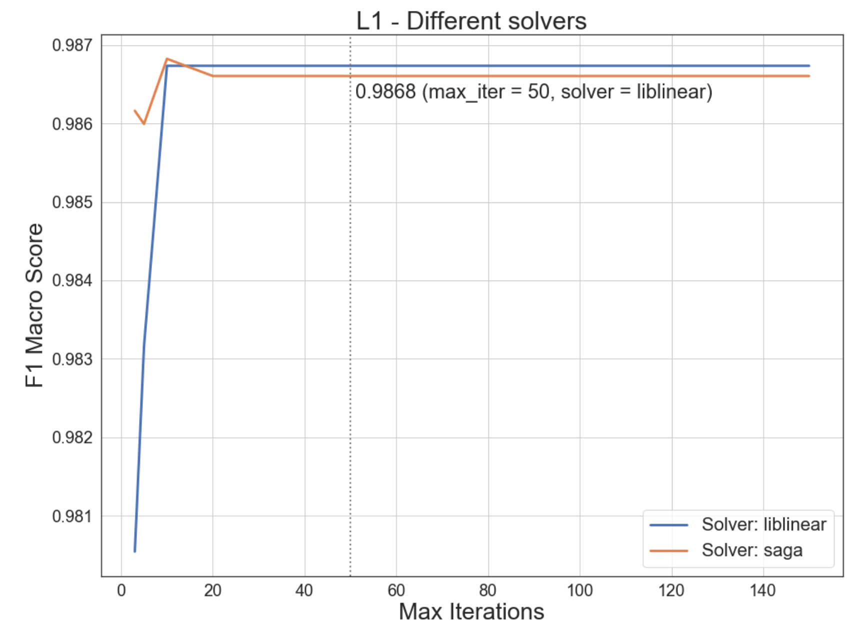
Figure 6.4(a) shows the trend in the F1 score, along with the increasing number of iterations, applying the *SAGA* [24] solver (a variant of the *Stochastic Average Gradient* [25] optimization that supports Lasso penalty) and the *LIBLINEAR* [26], an open source library for large-scale linear classification.

As it can be seen in Figures 6.4(b), by increasing the number of maximum iterations, up to 5000, the performances remain stable with every solver. The algorithms seem to not improve after 75 maximum iterations setted. Moreover, the LIBLINEAR solver gains slightly better results, in terms of F1 score, as it reaches 0.9867 in this metric, compared with the score obtained with LIBLINEAR solver, which is 0.9867.

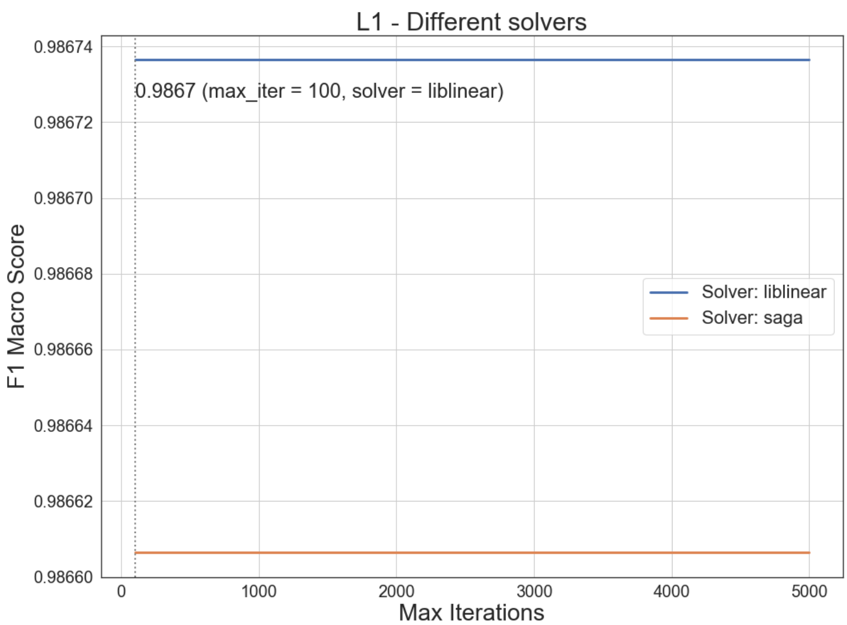
The final Logistic Regression meta-classifier has been fitted with the Scikit-learn library, with this setting:
LogisticRegression(max_iter = 100, penalty = "l1", solver = "liblinear", C = 1, multi_class = "multinomial", fit_intercept = True).
The C parameter stands for $\frac{1}{\lambda}$, the regularization coefficient. This setting obtained the following scores in a 10-fold crossvalidation:

- ✏ *Precision*: **0.987**
- ✏ *Recall*: **0.986**
- ✏ *F1 score*: **0.987**

The stacking ensemble performance is resumed in Figure 6.5, where all the components are first measured individually to show that the combination of them improves the overall F1 score.



(a) Up to 150 iterations



(b) Up to 5000 iterations - close-up view

Figure 6.4: Lasso Logistic Regression scores over solvers

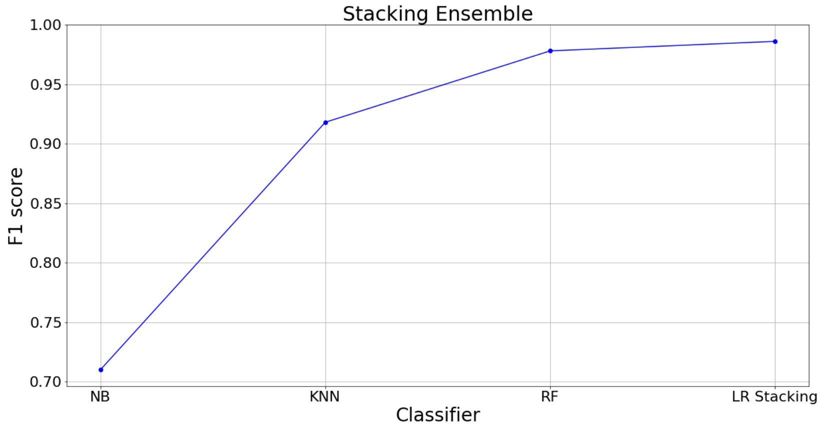


Figure 6.5: Stacking ensemble performance

6.2 Prediction pipeline

The final model is represented by an execution pipeline, involving a first binary classifier and then a mutli-class ensemble.

As described by figure 6.6, In order to perform a prediction over a new sample the process is the following:

1. User and tweets data retrieving with Twitter APIs
2. Prepare data with binary extrinsic features and strip image attributes, in order to output binary probability prediction (**Binary Random Forest**)
3. Prepare data and perform features computation to output multi-class probability prediction (**All-features multi-class Random Forest**)
4. Strip all attributes, except for the user features, weight them with Information Gain-driven proportions to output multi-class probability prediction (**User-based multi-class KNN**)
5. Prepare and treat text to perform text-based probability prediction (**Text-based multi-class Naive Bayes**)
6. Build new features vector with the stacked outcomes of the mutliclass classifiers
7. Compute the final mutliclass probabilistic prediction with the meta-classifier (**Multinomial Logistic Regression**)

8. Use the multi-class division to repartition the bot probability provided by the binary model

The binary classifier returns two values: the membership probability for the bot category and the one for the genuine class. The percentage that marks the bot nature of the examined account gets partitioned by the outcome of the multi-class stacking ensemble. The pipeline will be performed by a web application, in order to provide a useful classification tool for every internet user.

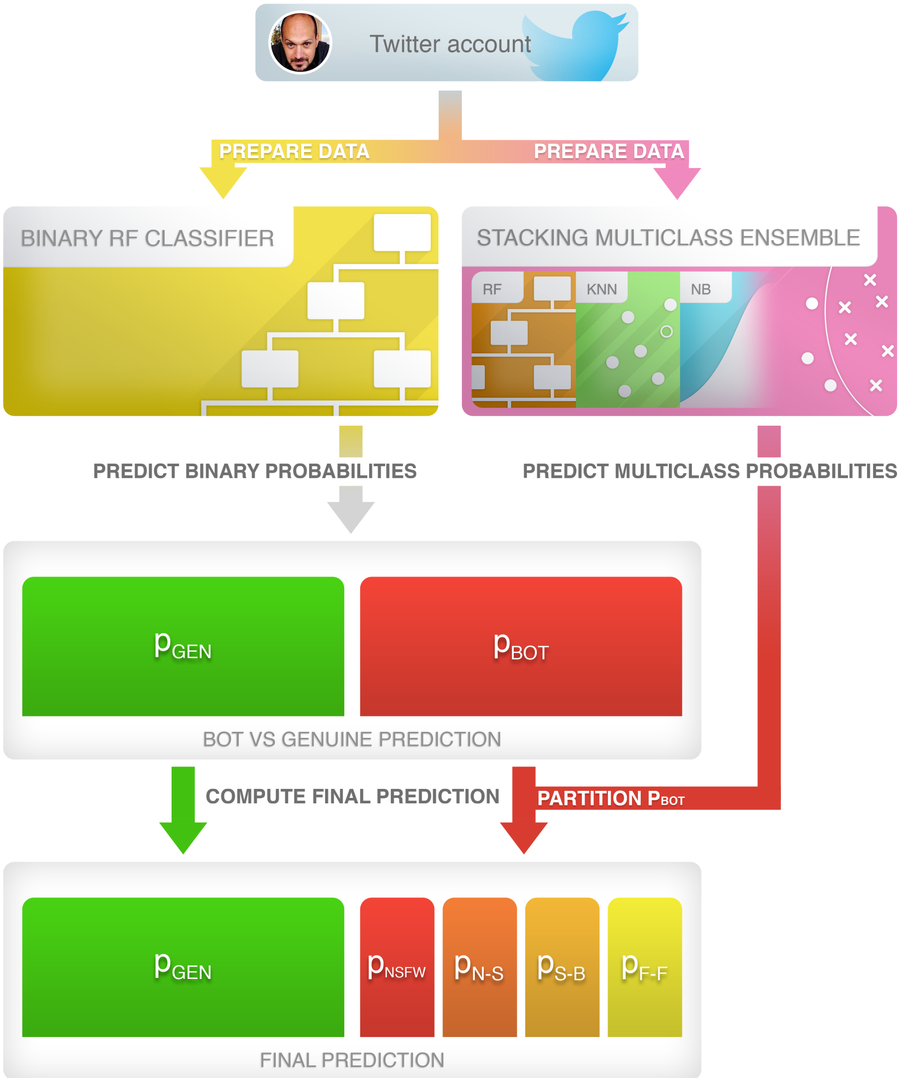


Figure 6.6: Final prediction pipeline

Chapter 7

Web application - BotBuster

This chapter presents the practical application of our work. We wanted to provide a tangible proof of the viability of the thesis, an instrument available to every user in search of classification, among the accounts met on Twitter. We thought it is useful to allow people to be aware about of the nature of the users that populate the social network. Bots often don't claim themselves as automated accounts, that makes their detection hard, even to an experienced utilizer.

BotBuster is the name of the project, whose goal is to provide the probability-based classification of the desired Twitter user. The logo used to represent the application tool is shown in Figure 7.2 The probabilities are shown in a histogram-shaped graph, with different colours, one for each target, as displayed in Figure7.1.

The web application is freely accessible through a public URL. It is currently available on *www.botbuster.it*.

Basically, the functioning of BotBuster can be resumed in getting a Twit-

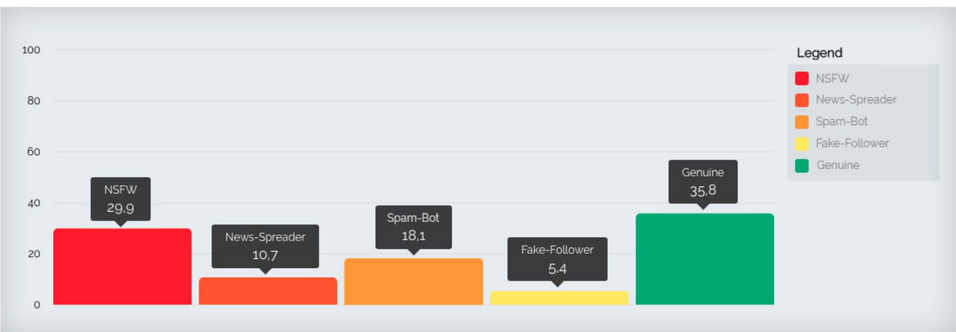


Figure 7.1: Probability classification diagram on BotBuster

Figure 7.2: BotBuster logo

ter user name through an input field on the home page, and then executing the prediction pipeline described in section 6.2. The input field filling triggers the engine operating in the background, which runs a Python script performing the prediction.

7.1 Architecture

The web application works under a client - server paradigm. The front-end program is executed on the client's web browser, while on the server runs the back-end application. Every time a user search for a Twitter user name, the HTTP protocols is involved to perform POST and GET requests between client and server. The front-end and the back-end applications handle the request and exchange data, as shown in Figure7.3.

7.2 Back-end

The back-end side of the web application is the core of the whole tool. Its goal is to summarize and perform the methods explained in the previous chapters and to classify Twitter accounts, whose user name or id is provided by the BotBuster user. The prediction time may vary, and it depends on several factors, like the condition of the internet connection, the current utilization of the CPU, the amount of data retrieved for the required user, as well as other factors. We can say that, testing the web application on local machines, the average computation time for a single prediction, for those users that has at least 100 tweets on their timelines, is up to 12 seconds. This amount of time is due to several factors, as said before, but it can be imputable to a specific element: the nsfw_avg feature computation. It may takes up to 7 seconds, in order to process and classify 10 images with the

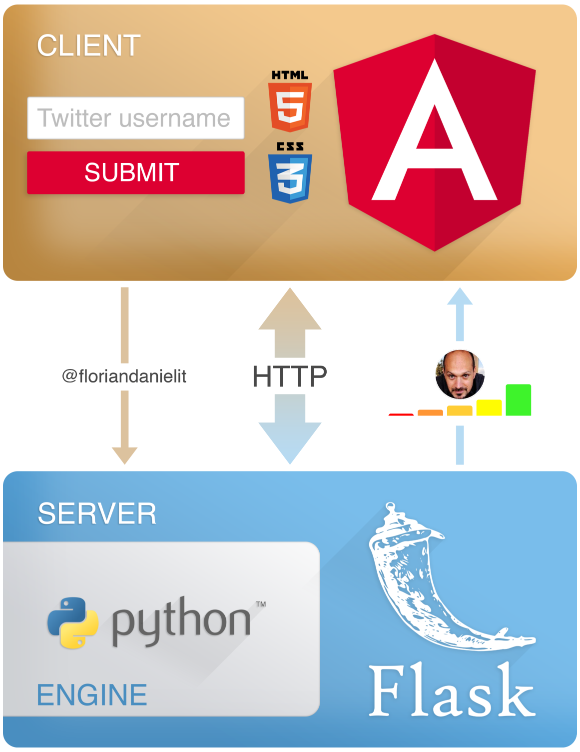


Figure 7.3: Client - Server architecture

Inception neural network. Users with less than 10 tweets, or less than 10 tweets with embedded images, require a shorter timespan to be classified, which is, in average up to 5 seconds. Figure 7.4 shows the repartition of computational time, along with the processes involved. It is based on the timespan needed to classify users with at least 100 tweets and with at least 10 tweets with images.

7.2.1 Engine

The engine of the web application is a Python 3 script. It performs all the steps described in the pipeline execution section 6.2. The models, that have been previously fitted with data, serialized and stored, are now loaded by the Python script. They have to perform a single prediction at a time. In addition to the models we built for the classification, the pre-trained convolutional neural network for NSFW recognition has been introduced to the pipeline, in order to infer on the media contents posted by the examined user. The first step consists in calling the Twitter APIs to retrieve user's data and its most recent tweets, up to 100. The script, then, handles the

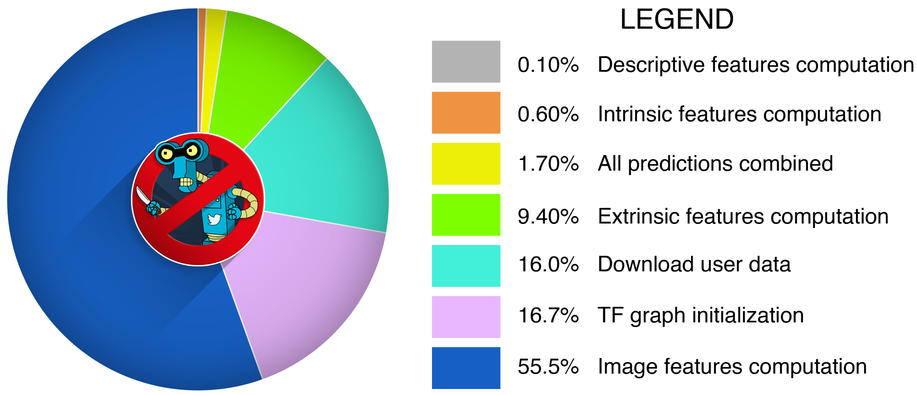


Figure 7.4: Prediction time pie chart

preprocessing stages and the data preparations needed by the different classifiers. Two final predictions are computed: the binary and the mutliclass classification. The probability given to the bot target, by the binary classifier, is used to weight the mutliclass prediction provided by the stacking ensemble. The final classification is composed by five probabilities: $\mathbf{P_{NSWF}}$, $\mathbf{P_{News\text{-}Spreader}}$, $\mathbf{P_{Spam\text{-}Bot}}$, $\mathbf{P_{Fake\text{-}Followers}}$, $\mathbf{P_{Genuine}}$.

7.2.2 Flask

Flask is a web framework for Python. It is a good fit for our tool, due to the Python-based script of the engine. The idea behind the Flask back-end is to map URL paths to some logic we want to execute. In the flask main script, we used three endpoints to trigger the computation of some code.

- *@app.route*('/', methods=['GET']):
  This first endpoint is used to catch a HTTP GET request from the client browser. It gets triggered as the user navigates to *www.botbuster.it*. The code that is being executed simply handles the rendering of the static files that compose the front-end, in order to make the website visible for the user, at that URL. The homepage appears as shown in Figure 7.5

- *@app.route*('**/api/user**', methods=['POST']):
  Everytime a user fills the Twitter username input filed and clicks on "**Bust it**" button, a HTTP POST request is made and this endpoint is involved to execute the first part of the classification script, which

Figure 7.5: BotBuster - homepage

retrieves the data regarding the requested Twitter account, with the official APIs. This first stage returns informations such as the profile picture, the nickname, the extended name, the number of tweets of that user, its following and follower accounts and its number of given likes, to someone else's contents. All these data are wrapped into URL links, which bring the user to the Twitter pages dedicated to those data. together with the meta-data bar of the requested account, a GIF is displayed to inform the user that the classification is in pending status, as shown in Figure 7.6.

✏ *@app.route*('**/api/classify**', methods=['POST']):
This last endpoint is used subsequently to the previous one and it launches the classification script. It takes some seconds to compute the probabilities and to give them back to the client front-end. The prediction time may vary, as it depends on the number of media found in the account's tweets: the Inception convolutional neural network takes few seconds for each pictures it has to analyse. Once the computation is done, the resulting probabilities are returned as a JSON file to the client, in order to be rendered in the browser window, by the Angular front-end. The response provided by the classification script is represented as shown in Figure 7.7.

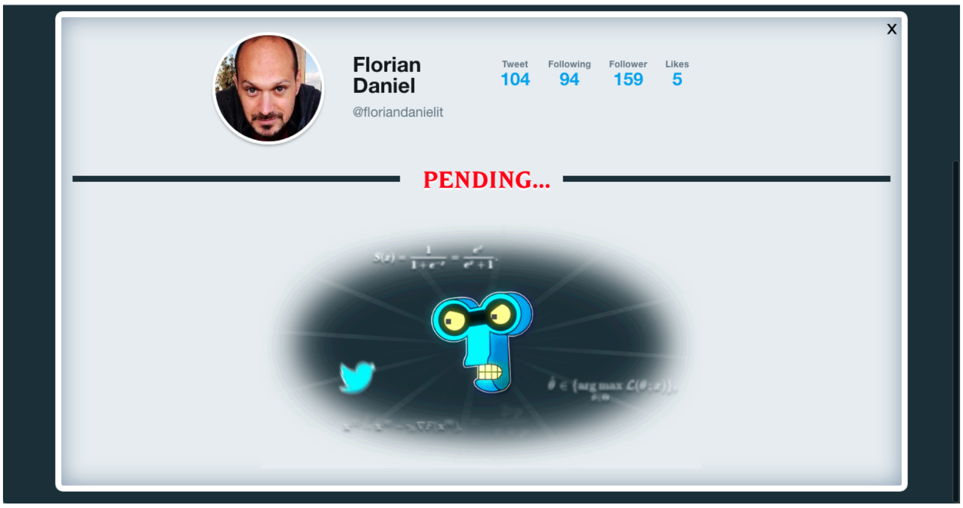


Figure 7.6: BotBuster - pending classification

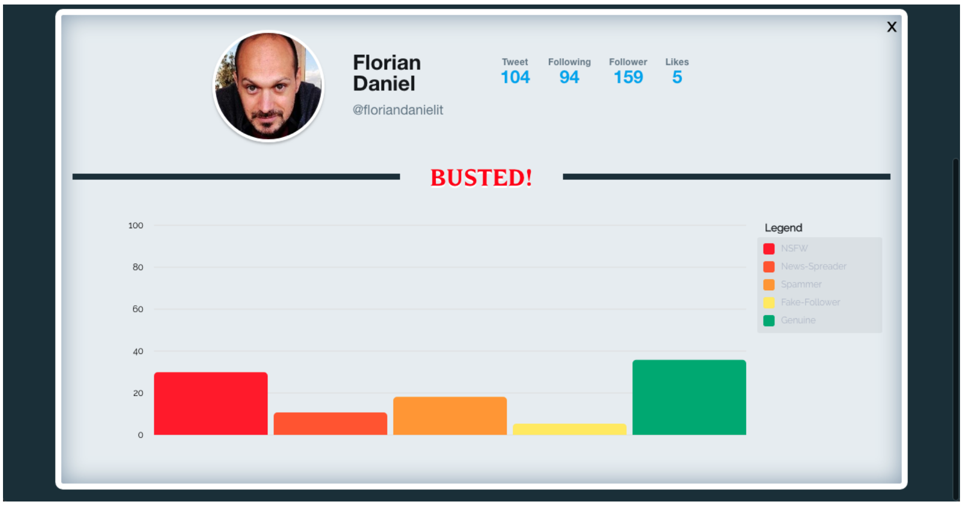


Figure 7.7: BotBuster - classification

7.3 Front-end

The client-side application relies on Angular framework, a TypeScript-based open source, front-end web application platform, developed for the most by Google and some individuals and developer corporations. It is a rebuild of the previous AngularJS framework. BotBuster fronted project is composed by a single HTML page, with TypeScript files creating complex types for the data and handling the HTTP responses. The Angular component operating on the homepage contains HTML tags that are being toggled with the responses provided by the web server application.

As soon as the user submits a Twitter username trough the input field, the search bar gets hidden and the meta-data of the requested account appears, together with the pending GIF. Once the last POST (the classification request) gets a response, the pending GIF disappears to be replaced with the charts representing the classes and their assigned probabilities. Angular framework makes TypeScript files work with the web browser, thanks to Javascript language, and it loads the components without loading the entire pages to do it. It fits our purpose, since we didn't need many pages to be displayed and we wanted a light visualization of the script results.

7.4 Deployment platform

The platform used for the deployment is the App Engine of the Google Cloud Platform. The application is being deployed, along with the needed packages and dependencies, on a Python flexible environment. Then, the domain **www.botbuster.it** has been redirected to point to the address of the web application provided by Google, in order to have a user friendly URL visible on the web.

7.5 Comparison with Botometer

Our work differs from the Botometer project due the effort involved in deepening the classifications among bots and the detection of their harmful behaviours. However, since the first element of our prediction pipeline is built also with the same data used for Botometer, we wanted some comparison terms on real world cases.

We analysed some specific accounts, in order to highlights the main difference between our predictions and Botometer's, as well as the similarities. The comparison in Figure 7.8 shows the differences between our methods and Botometer's, with a verified account, such as an official editorial user. In this case, we examined the CNN account. Although this account is marked as verified by Twitter itself, it is reasonable to believe it is managed by automations. These kinds of profiles are directly linked with the official editors' websites and they tweet the same contents found in their online newspapers. Such linked behaviour could be easily programmed, in order to guarantee a high rate of tweeting activity. The verified mark has been used as a feature even in our binary model, but we lacked lots of values for that attribute, so, in our classification, it doesn't play an important role in assessing the authenticity to the profiles. We think that Botometer relies heavily on the

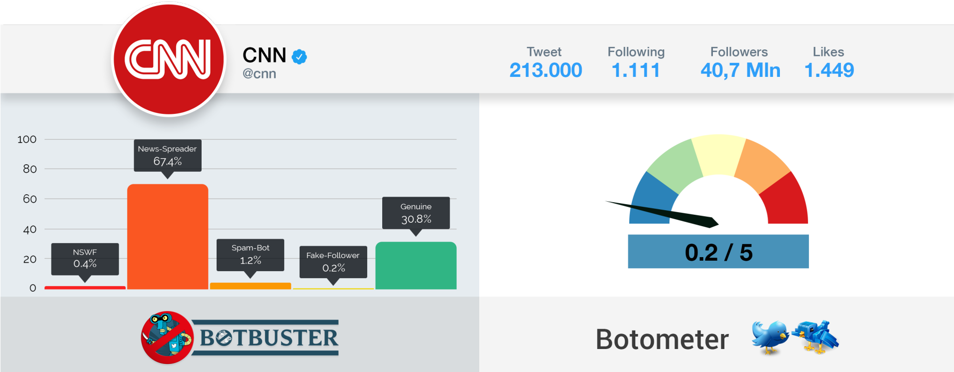


Figure 7.8: BotBuster vs Botometer - CNN's account comparison

verified feature to detect genuine accounts, since lots of the verified accounts tested with that tool have been evaluated as legitimate. Figure 7.9 shows this particular behaviour of the Botometer web application. All the accounts shown are likely automated or act in a way that is not recognizable as a human interaction on the social platform. Their tweeting frequency should discourage the classifier to blindly trust the verified attribute. We couldn't be sure that verified accounts are handled by bots neither, but we chose to let the Random Forest weight that feature basing it on the information gain, as it does with all the other attributes. The inner mutliclass classification matches our expectations, since CNN account's goal is to spread its news, and we identified it as a News-Spreader, with 67.4% of confidence. As we get deeper in our prediction, we can observe the feature values computed for that examined profile. In particular, the intrinsic and the extrinsic features seem to frame the profile nature well:

Extrinsic features

NSFW score	**NS score**	**SB score**	**FF score**
0.002	0.034	0.011	0.011

Figure 7.9: Verified accounts classification with Botometer

Intrinsic features

Tweet intradistance	**URL entropy**
18.25	0.0

The News_spreader_words_score is the highest among the multi-class extrinsic features, this means that the words used by the CNN accounts matches, for the most, the ones collected in our News-Spreaders dictionary. Moreover, we can see that the URL entropy is equal to zero, and this means two possibilities: the account has no tweets with links on its timeline or it tweets the same URL in each post. The last option is the one standing for this case. The last 100 tweets retrieved contain the link to the CNN official website.

Another difference between our work and Botometer is represented in Figure 7.10, where a user that we detect as a Fake-Follower can't be classified by Botometer, because of the absence of tweets on its timeline. This could mean that the researcher behind that project thought that the user features weren't enough to perform a proper prediction, without data coming by the tweeting activity. We chose to not use the tweeting mechanism as a support feature, being aware that user driven classification could be less precise, due to the limited information obtained by the user profile only.

In Figure 7.11, instead, Botometer seems to perform better with this type of accounts. We examined the official profile of PoliMi. Botometer detects it as legitimate, but we see it as a Spam-Bot, principally, with a good percentage of genuine nature. This difference cannot be imputed to the verified check, as it is missing, in this example. Looking at the partial scores assigned by the feature groups that Botometer computes, we can see

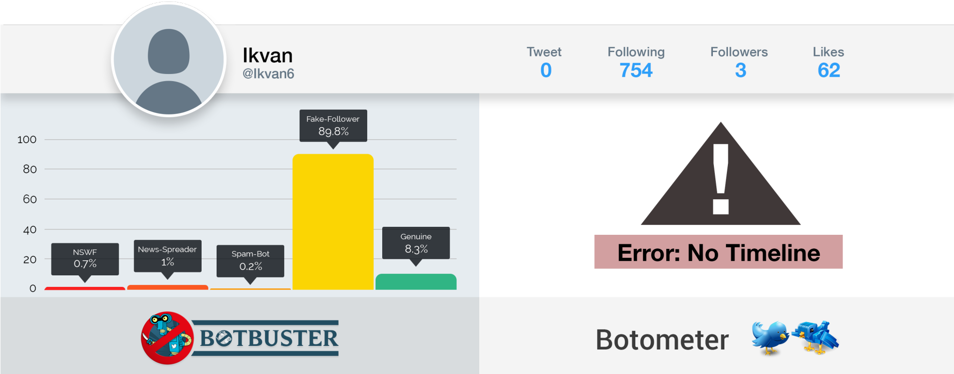


Figure 7.10: Classification of users without tweets on their timelines

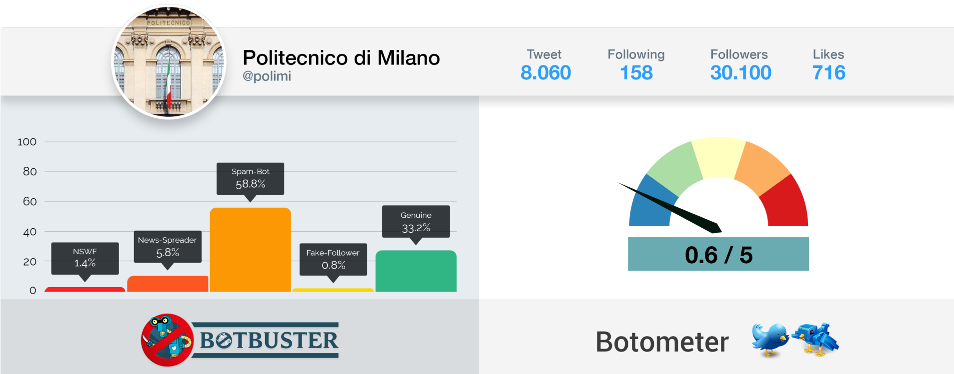


Figure 7.11: Comparison of PoliMi account classifications

that it assigns very low bot-likely values in the Sentiment field, as well as in the Network lookup. Figure 7.12 shows the partial scores assigned to PoliMi account. They implemented *English-specific features* and *Language-independent features* too; we didn't perform this distinction, every textual feature, as well as the words used for the text-based classifier, can be seen as their Language-independent attributes. The issue with this approach is the majority language collected in our training data. Most of the accounts gathered are English speakers, this leads to English-oriented dictionaries and textual features, making the classification of foreign accounts harder for our tool.

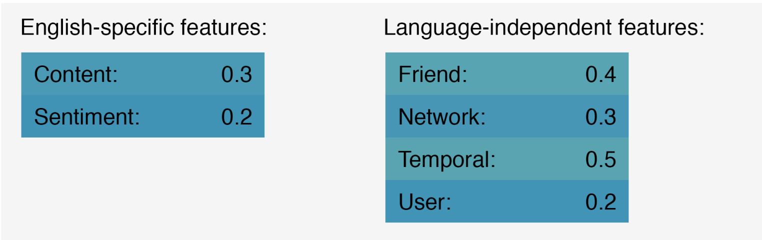
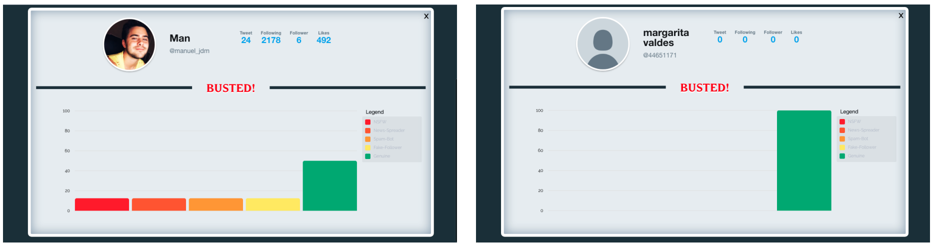


Figure 7.12: Partial Botometer scores for PoliMi account

Since it is legitimate to think that this profile is managed by a real person, we consider those feature introduced by Botometer as a valid contribute to the binary classifier. It could be considered in future implementations, in order to better distinguish human profiles, even with very high tweeting activities.

During the tool testing, we noticed some particular types of users that we haven't been able to include in the training sets. These types are the *protected* users and the ones which never had interactions on the social, at all. No interactions means that these unused users have no tweets, no followings, no followers, no likes and often no profile picture. Protected users choose to hide their sharing activities to everyone, and to reveal their timelines only to those accounts who send them the request to see their tweets. Obviously, the timelines aren't accessible via Twitter APIs, so it is hard to infer the nature of such accounts, since it is not possible to asses if they post contents or not. For this kind of users, we decided to assign a standard classification, that represents the uncertainty, as shown in Figure 7.13(a). The classification assesses a 50% probabilities to both bot and genuine categories, and partitions the bot behaviours in equal parts. Instead, the prediction output of unused accounts has been fixed with a 100% chance to be genuine. We cannot assess the real nature of those users, nor their purpose, but we can state that their not harmful, since they do not have interactions with other accounts. That makes them genuine accounts, like legitimate users, they don't represent threats or harmful entities. An example of such classification is shown in Figure 7.13(b).



(a) Protected account

(b) Unused account

Figure 7.13: Classification of specific account types

Chapter 8

Conclusion and Future Work

We can see this thesis as a proper journey, in which we learnt a lot about the users and data that populate social networks, and how they can be tricky to deal with. The legacy of our work is a tool that can be used by anyone and that can prevent undesired or unlucky interactions with potentially harmful bots on Twitter.

The development of such tool doesn't imply that perfection has been reached, not even at the last step of this journey. We are aware of the limitations our work is subject to, due to the limited time we could invest and to the lack of experience in this field, experience that we have grown, by cleaning and processing these data and by trying to give them meaning, in terms of empirical considerations.

This chapter summarizes the obtained results, as well as the weaknesses faced and the possible future extensions that could strengthen the work done so far.

8.1 Summary and Lessons Learned

The overall considerations can be summarized along the main steps involved in the thesis development.

- ✏ We had to face the challenge of inserting a new piece of research into the prolific puzzle of social bots detection. We studied the work done in this field and decided to go deeper inside the classification methods available so far. It wasn't easy to make us space inside research works, especially considering the time needed to gather the proper amount of data, for this kinds of purpose. The job was challenging and we tried to exploit the time and the tools we had, to give the best contribution to the research as possible. We are pleased with the results obtained, in terms of dataset creations and tools developed. Obviously, some limitations came to visit, starting from the timespan of the data observations, moving forward with the computation power we invested, and so on.

- ✏ Our methodology involved the data collection, whose required effort was one of the biggest of the entire project. The main - and the first - issue faced, indeed, regarded the data itself. We did not have a custom dataset to draw samples from. We needed five different targets for our classifiers, because five were the account types we wanted to focus on.

  We followed several leads, in order to build a proper dataset for the multiple classes classifiers, and to enrich the training set of the binary model. After all the data were clean and homogeneous, we crafted useful features to support the decision boundaries among targets. Multiple models were created and combined into an execution pipeline. They were selected and enhanced by evaluating their performance scores, with test sets coming from our datasets. At the end of the whole process, an online web application was build to wrap the final prediction Python script. We encapsulated it into a client/server paradigm, in order to be available to everyone.

- ✏ During the stages needed to complete our work, we learned lots of things about models, tools, methodologies and related works. However, the most important thing we have seen is the significance of the data. Data plays the leading role in a data science project, without a wide, consistent, stratified and concept-representative dataset, the models won't be able to approximate the actual targets correctly. This is the main issue met with our approach. We gathered a nice amount of stratified data, but the problem is inherent to that stratification. Every class differs from the others with significant distinctive traits, leading to a relatively effortless model selection. Our classifiers fits our data very well, with pleasing metric scores, but are they really capable of generalizing over unseen samples?
  What happens when we feed our classifiers with a brand new, borderline entry, without characterising marks? Here the nature of the datasets used emerges.

  We don't have much instruments to be compared with ours, since the available works put the effort into bots and humans detection. Lately in this chapter, we provide a wide scale projection of the Twitter population, and use the outcome to compare our work with other public studies that made the same study on large numbers.

8.2 Outputs and Contributions

The contributions of our thesis can be summarized with three main points:

- ✏ The construction and the release of a multi-class dataset, composed of 21,743 bot samples, labelled with four targets: *NSWF* (**0**), *News-Spreaders* (**1**), *Spam-Bots* (**2**) and *Fake-Followers* (**3**). The dataset was built following different approaches and assembling freely accessible data from previous researches, as discussed in Chapter 3. It is accessible for free at `http://botbuster.it/dataset`.
- ✏ The overall methodology followed to build our classification system. After the samples gathering, we studied how to build some features from raw data, in order to better describe the concept of the classification problem. Several models have been tested with raw settings and that led to a final pool of three multi-class classifier and a binary model. The bot-or-not Random Forest was build following part of the methodology used for Botometer, starting from the same training set, and then enhancing the data with newer samples. The performances of our model and Botometer have been compared, in order to have a reliable binary classifier to start our pipeline with. The second step involves the bot behaviour classifier. It is composed of three stacked models: a Random Forest algorithm fitted with the complete feature vector, a K-nearest neighbours model that considers only user data, without tweets meta-data, and a Naive Bayes text-classifier which, complementarily to the KNN model, classifies bots by reading their tweets only. The models stacking is performed with a Logistic Regression meta-model that takes as input the outputs of the previous algorithms, and performs the final probability prediction. Such stacking model is trained with bots data only and it is used to better defining the bot behaviours, repartitioning the probability that an examined account is a bot, according to the bot-or-not model. This prediction pipeline was tested with a 10-fold-crossvalidation, peaking a 0.987 score in the F1 measure, 0.987 of Precision and 0.986 of Recall metrics.
- ✏ **BotBuster**, the web application. It is an instrument aimed at classifying every Twitter user, with no discriminations about their tweeting activities. It is the implementation of the method discussed in our

thesis. The tool provides “soft“ classifications, which means that the outcome returned by the engine is a probability vector, and not a specific target. It assigns, to the examined account, the membership likelihood to each target involved in our study. The prediction rendered with vertical bars contains probabilities for NSFW, News-Spreaders, Spam-Bots, Fake-Followers and Genuine category. BotBuster is accessible for free at `http://botbuster.it` for testing.

- ✏ A wide scale projection for the Twitter population. We provide an estimation of the bots distribution on the social network, sampling and classifying 12k random accounts. The process and the results are discussed in the following section.

In addition with the usefulness provided to the internet users, with the development of such instrument, we hope that the study we made could become a starting point for those who are interested in building more sophisticated tools for automatic recognition of social bots, and the identification of their potentially harming behaviours, even in cross-platform environments.

8.3 Projection on wide scale

Being now able to automatically classify Twitter accounts into genuine users or different types of bot accounts, we wanted to give and estimation about the bots distribution over the full Twitter population. In order to be able to do that, we needed to sample unseen users from the social network and to classify them with our tool. Starting from 200K randomly generated IDs, within a feasible ranges of numbers, we checked each identifier with the Twitter APIs, to verify if they were actually bounded to existing accounts. This filtering process left us with just 12,616 retrieved profiles. Then, we analysed every one of those users with BotBuster, and the results are shown in Figure 8.1.

What we can infer from this chart is that the most representative category of accounts is the Genuine one. It is reasonable to assess that the legitimate accounts populate Twitter for the most, but, according to our analysis, there may be even up to twice as many bots on Twitter as identified in by Varol et al. [1]. The researchers of that paper claim to have sampled 14Mln accounts and to have found a bot percentage that is up to 15% of the entire population. The classification of such amount of users would have taken us an incredible timespan to be executed, since the average prediction takes 12 seconds to be computed. One other thing that may

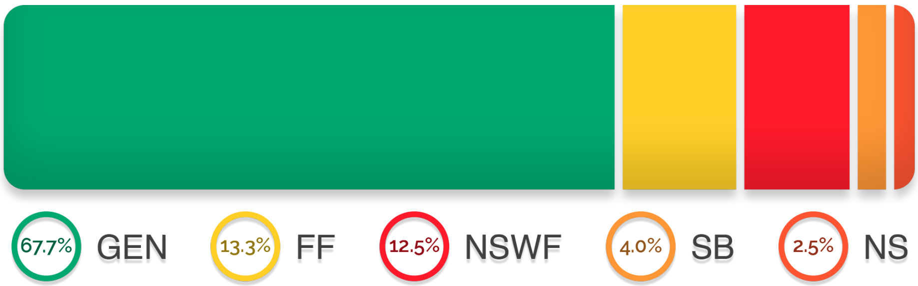


Figure 8.1: Projection on Twitter population

have led us to different results is that we took into account totally random samples, which means including also inactive users. Users are marked as active if they logged at least once into the system in the last six months. According to Twitter, the current population of active users is composed of about 326Mln of accounts 8.2. We managed to reach 12K profiles and lots of them are most likely inactive, given the high ratio of Fake-Followers found with BotBuster. Fake-Followers are is characterized by low tweeting activity, along with other factors, such as the number of accounts that follow them. BotBuster seems to classify inactive users as Fake-Followers, because they usually lack tweets. It is a limitation imputable to the nature of the training sets, since we didn't have enough samples with such low activity, marked as legitimate.

What really emerged from this experiment is the high ratio of inactive users found by randomly sampling Twitter accounts. They have been distributed among Genuine, Fake-Followers and NSFW, depending on two factors: their level of interaction and their profile pictures. Each of these inactive users are characterized by a very small number of tweets, but the ones which have no tweets, no followings, followers, custom profile pictures or likes have been marked as genuine by the tool. The ones that have at least one of those aforementioned fields with non-empty value are classified according by the rules of our models. The issue in this approach is represented by the Inception module for the image features computation. If the neural network fails in classifying a picture correctly, the examined user could be marked as NSFW, even with an almost-complete lacking of interactions, just by looking at her profile photo. This behaviour explains the high ratio for both Fake-Followers and NSFW classifications, with the retrieved samples.

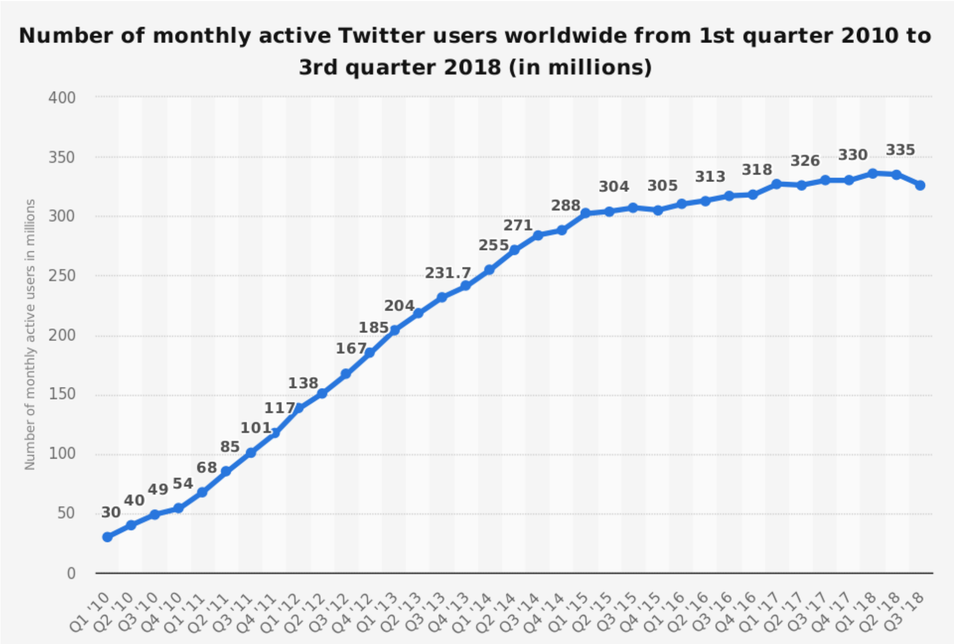


Figure 8.2: Number of monthly active Twitter users worldwide
Release date: October 2018
https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users

8.4 Limitations

As said before, we met several limitations in our thesis, and they came both from external factors and internal choices that affected the final results.

The external factors are, for the most, the time and the computational power. We can impute to the data the most limitations, but the fact is that, with the right amount of time, we could had collected a better dataset, exploiting different and non tested methodologies. For instance, we could have used social honeypots, crafted to trigger specific kinds of bots, maybe capturing more diversity in our data, involving new waves of social bots, as well as more institutional accounts, like editorial or politics users. We lacked verified profiles in our training sets, we think that adding more accounts with the verification mark could be helpful to extend the legitimate behaviour's spectrum that we have analysed. We can observe the struggle of our classifiers, when it comes to such verified accounts.

Even the other categories could have been extended with borderlines accounts, in order to make more flexible decisions. Once again, it is a matter of time, mostly. The computational power didn't represent a major issue, but the lack of such power led to a general slow-down of the work, which denied us the possibilities to do further experiments, like the ones described above. The calculations of some features, like the extrinsic attributes and the image classification, required lots of time, with the tools we had. Some algorithms had to be run for hours, entire nights or days. The Grid Searches used to find the best hyperparameters configurations took us much time, and they have been repeated very frequently. Every little change, every little different choice, brought us to recalculate the best models' settings, which means Grid Searching over the parameters space, repeatedly. It could have been faster with cloud computing solutions.

When it comes to internal choices, we have to start with something we lacked, especially at the beginning of this journey: the experience.
Inexperience made us chose some moves that, in operative terms, wasted precious time. For instance, the programming skills with Python and the machine learning frameworks weren't fulfilling the necessities, at the first stages of the work. We spent a lot of time struggling with programming paradigms and handling the dataset, with information losses that forced us to repeat basic operations and restore entire processes. These limitations have been mitigated with the experience on practical coding, and the entire workflow had a speed-up, as the time passed. Some inexperience-driven limitations, that not necessarily represented a bad handling of the time, were the attempts that revealed to be failures, but that made us learn something more about methods and contexts. For instance, one of the first hands-on tries on the data was to apply an unsupervised labelling to a binary dataset, in order to catch useful patterns and to simplify the targeting process. By that time, we thought that there weren't other ways but handling labelling the datasets. Fortunately, and thanks to that unlucky approach, we found out new ways to collect the labelled data.

Due to inexperience, some choices we made, when we were building the first models, have been rushed, in comparison with the decision taken once the overall structure was clear. Sometimes, we went back to old decisions and we tried different possibilities, in order to better adjust some features to a pipeline, instead that to the previous stacking method. This revisiting process took much time, and we are aware that there are still things that could be refined better.

For instance, the extrinsic features have been passed under different tests, before choosing the final configuration. This is how we arrived to totally disjointed dictionaries. But we did not play with the number of initial words retrieved, taking 1000 as fixed (older versions had less words).

We could have applied empirical methods to choose the right amount of terms to consider.

Also, we did not consider language-dependent features. This process was a consequence of the first adaptations to the data retrieved. We were gathering only American accounts, and the textual features were initially thought to fit that language. With the extension of the dataset, especially with the Fake-Followers category, we introduced different parts of the world in our data, with their languages. This made the data heterogeneous, without adjusting the attributes. Geolocalization is not taken into account as a feature, as well as the nationality.

Some attributes were deleted after observing a large number of missing values. One example is the colour chosen by the user to fill the profile tile (a rectangular-shaped space that lies under the profile picture and that can be filled with photos or plain colours). Since it is represented as a string (RGB hexadecimal code), in order to be handled by the algorithms we built, it should have been encoded with techniques as One-Hot encoding, that would have made the number of features explode. We could have found better way to handle the missing values and the encoding of such features.

Another thing that could have been done, and that we actually considered, is sentiment analysis on tweets. We could have relied on external services, but sticking to a low number of calls for the APIs, because these are paid services that offer limited free solutions. Implementing our own sentiment classifiers would have been expensive, in terms of time and effort. We preferred to spend that time to refine the features we already had and to add a classifier to the ensemble.

8.5 Future Work

Several things could be implemented by future works. The most important ones could be the dataset extension, as well as the detection of new targets. Since we studied the Twitter platform, we stuck to a narrow range of harmful behaviours. This range could be extended with new classes to perform classifications on. Offensive bots, like the ones that incite to hate or racism, are outside the domain of our study, because we did not find a proper method to collect a reasonable amount of them. The missing sentiment analysis discouraged us to identify the level of hate or offensive contents in tweets. This insertion could make the target space wider.

Another useful work that could be done is the evolving training of our models. It could be implemented a way to receive trusted feedbacks over the performed classifications. Such indications, along with the classified sample, could feed the models by adding that entry to our training set, with the indicated target. Obviously, feedbacks should be trusted and not biased. A heuristic method to choose which feedback consider should be implemented too, as well as reinforcement learning algorithms, to improve the evolving training.

A more sophisticated image recognition system can be included. NSFW category is a well defined sphere of the bots ecosystem, it would be very easy to detect it with the right instruments. By now, we have an Inception convolutional neural network, used to compute two features bounded to the images. It would be interesting getting some insights from video contents too, in feasible computation time. The image features aren't 100% accurate, due to the validation error found and to the data used to fit the network. It can be improved, with more and diversified samples. The time needed to process ten images discourages us to look for more media content to analyse. It would be a robust support to be able to lookup more tweets with media contents embedded.

Some features that look outside the user's box could be added, like the ones which get clues from the user's network, from her friends or the accounts that quote and retweet her. We had our focus on the user's routine, with meta-data about her relationships, but we don't get inside those interactions.

Bibliography

[1] Onur Varol, Emilio Ferrara, Clayton A. Davis, Filippo Menczer, and Alessandro Flammini. Online human-bot interactions: Detection, estimation and characterization. 2017.

[2] Inbenta Technologies Inc. The ultimate guide to chatbots for businesses. tech. rep., 2016.

[3] A. Bessi and E. Ferrara. Social bots distort the 2016 us presidential election online discussion. first monday 21, 2016.

[4] Florian Daniel, Cinzia Cappiello, and Boualem Benatallah. Bots acting like humans: Understanding and preventing harm. *IEEE Internet Computing, accpted for publication*, 2019.

[5] Clayton A. Davis, Onur Varol, Emilio Ferrara, Alessandro Flammini, and Filippo Menczer. Botornot: A system to evaluate social bots, 2016.

[6] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and AlessandroFlammini. The rise of social bots, 2016.

[7] Kyumin Lee, Brian David Eoff, and James Caverlee. Seven months with the devils: a long-term study of content polluters on twitter. 2011.

[8] Stefano Cresci, Roberto Di Pietro, Marinella Petrocchi, Angelo Spognardi, and Maurizio Tesconi. Fame for sale: Efficient detection of fake twitter followers. 2015.

[9] Beskow, David, Carley, and Kathleen. Bot-hunter: A tiered approach to detecting and characterizing automated activity on twitter, 2018.

[10] J. Ratkiewicz, M. D. Conover, M. Meiss, B. Gonçalves, A. Flammini, and F. Menczer. Detecting and tracking political abuse in social media, 2011.

[11] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. 2012.

[12] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. 2010.

[13] Chengcheng Shao, Pik-Mai Hui, Lei Wang, Xinwen Jiang, Alessandro Flammini, Filippo Menczer, and Giovanni Luca Ciampaglia. Anatomy of an online misinformation network. 2018.

[14] Chengcheng Shao, Giovanni Luca Ciampaglia, Alessandro Flammini, and Filippo Menczer. Hoaxy: A platform for tracking online misinformation. 2016.

[15] Chengcheng Shao, Giovanni Luca Ciampaglia, Onur Varol, Kaicheng Yang, Alessandro Flammini, and Filippo Menczer. The spread of low-credibility content by social bots. 2018.

[16] Stefano Cresci, Roberto Di Pietro, Marinella Petrocchi, Angelo Spognardi, and Maurizio Tesconi. The paradigm-shift of social spambots: Evidence, theories, and tools for the arms race. 2017.

[17] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. Detecting automation of twitter accounts: Are you a human, bot, or cyborg?, 2012.

[18] C. Yang, R. Harkreader, and G. Gu. Empirical evaluation and new design for fighting evolving twitter spammers. 2013.

[19] John S. and James L. Knight Foundation. Disinformation, 'fake news' and influence campaigns on twitter. 2018.

[20] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[21] Mahdi M. Kalayeh, Misrak Seifu, Wesna LaLanne, and Mubarak Shah. How to take a good selfie?, 2015.

[22] T. Joachims. Text categorization with support vector machines: Learning with many relevant features, 1998.

[23] S. Ting, W. Ip, and A. H. Tsang. Is naive bayes a good classifier for document classification?, 2011.

[24] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. 2014.

[25] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. 2017.

[26] Rong-En Fa, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. 2017.