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

[Title of the Thesis]

Supervisor: [Name]
Co-supervisor: [Name]

M.Sc. Thesis by:
[Name], matriculation number [nnn]
[Name], matriculation number [mmm]

Academic Year 200N-200N+1

About this template

With this template I want to give you some input on how to structure your thesis if you develop your thesis with me in Politecnico di Milano. Next to the pure structure, which you should reuse and adapt to your own needs, the document also contains instructions on how to approach the different sections, the writing and, sometimes, even the work on your thesis project itself. Sometimes you will also find boxes like this one. These are meant to provide you with explanations and insights or hints that go beyond the mere structure of a thesis.

I hope this template will help you do the best thesis ever, if not in the World, at least in your life.

Florian Daniel
October 12, 2017

Disclaimer: Sometimes I may make statements that are general, if not over-generalized, personal considerations, or give hints on how to do work or research. Be aware that these are just my own opinions and by no way represent official statements by Politecnico di Milano or its community of professors. If something goes wrong with your thesis or presentation, you cannot refer to these statements as a defense. You are the final responsible of what goes into your thesis and what not.

Acknowledgements: The original template for this document was not created by me. I would love to acknowledge the real creator, but I actually do not know who it is. The template has been passed on to me by a former student, who also didn't know the exact origin of it. It was circulating among students. However, to the best of my knowledge at the time of writing, it seems that Marco D. Santambrogio and Matto Matteucci may have contributed at some point with considerations on structure and funny citations. Both were helpful and enjoyable when preparing this version of the template. I will be glad to add more precise acknowledgements if properly informed about the origins of this template.

Supervisors and co-supervisors

If the supervisor is internal to Politecnico di Milano (a professor or researcher), then on the first page use "Supervisor" plus the titles "Prof." and "Dr." for professors and researches, respectively. If the work was co-supervised by someone else, refer to him/her as the "Co-supervisor." If the work was supervised by someone external to Politecnico di Milano, use "External supervisor" for the external supervisor plus "Internal supervisor" for the internal supervisor that mandatorily must co-supervise the work with the external supervisor.

Optionally, here goes the dedication.

Abstract

The abstract is a small summary of the thesis. It tells the reader in few words (up to one/one and a half page of total text) everything he/she needs to understand:

- ☐ the *context* of the work (e.g., chatbots),
- ☐ the specific *problem* approached by the thesis (e.g., the development of personal bots by non-programmers),
- ☐ if applicable, clearly state the *research questions* you would like to answer (e.g., "is it possible to enable non-programmers to do X using A?"),
- ☐ the three/four *core aspects of the proposed solution* (e.g., use predefined rules, use machine learning, assisted development, etc.),
- ☐ the *concrete outputs* produced by the thesis (e.g., a state of the art analysis, a conceptual/mathematical model, an application, middleware or API, an empirical study with/without users, etc.), and
- ☐ the *findings and conclusions* that one can draw from the evaluation of the approach (e.g., that under some very specific conditions non-programmers are indeed able to implement own chatbots effectively using the proposed technique).

Checklists
Now and there I propose checklists with items, such as the one just above this box. They are meant for you to check if you included all the content that is relevant and that should be included, in order to make your text complete. When reading your thesis, I will look for all these items.

Writing style

This is a M.Sc. thesis. It's neither Facebook nor Twitter nor an email. This is going to be an official document with legal value that will decide on the final mark of your yearlong university career and perhaps even on your future work perspectives. So, you surely don't want to be judged badly because of grammar errors, flawed/wrong vocabulary or superficial layout and/or text structure. It is a must that what you write is always *correct* content- and language-wise (no false statements or claims, no language mistakes), *readable* (no sentences that cannot be understood) and targeted at the *average-skilled reader* (professors, but also your own colleagues).

Plagiarism

This is a M.Sc. thesis. It's neither Facebook nor Twitter nor an email. This is going to be an official document with legal value that will decide on the final mark of your yearlong university career and perhaps even on your future work perspectives – yes, I plagiarized myself here a little bit. So, you surely don't want to copy/paste material from scientific articles, online resources, books, and similar without adequately acknowledging the holders of the respective intellectual property rights. If you do so, it is a must that you properly *cite* each source where you take text or inspiration from. It is fine to do so – actually, citing someone is a compliment! – but it becomes a crime if the source is not cited. Not only M.Sc. titles but also Ph.D. titles have been withdrawn for fraudulent "reuse" of others' intellectual property. Be aware that Politecnico di Milano, like most higher educational institutions that issue university degrees or scientific publishers, may use specialized software to automatically detect plagiarism.

Sommario

Here goes the translation into Italian of the abstract. If the thesis is written in Italian, no translation into English is needed. Hence, one of the following must be checked:

☐ Thesis written in *English*, properly proofread translation needed

☐ Thesis written in *Italian*, no translation needed, chapter omitted

Acknowledgements

If you would like to thank somebody for given support, this is the right place to do so.

Contents

Chapter 1

Introduction

The introduction is one of the core chapters of your thesis. It expands what has already been said in the abstract with additional details on the content and contribution and on the structure of the thesis. It is meant to introduce the reader to the work he/she will be reading in the rest of the document and, most importantly, to get the reader curious about reading on, knowing more about your work.

1.1 Context: [topic]

This thesis is about describing the work you are doing in your final thesis project. You have been working on it for months, and nobody knows the work better than you do. This is great and exactly how things should be: by doing your thesis project you became an expert – if not *the* expert – in this specific field of research and/or technology.

But attention: being the expert is also dangerous when it comes to explaining others what you did and why you think you did a great work that deserves attention (I give it for granted that you work does so). There are only very few people around you (your supervisor and possible co-supervisor, some friends, maybe someone else) who are as expert as you are in this topic. So, if you start in a full-impact fashion to tell that you implemented an extraordinarily cool, new algorithm to solve X, or that you discovered this extremely surprising finding Y, or that you mathematically proofed that Z, etc. (you got it), your reader will not understand anything. Therefore, before talking about what you actually did, you need to introduce the reader to the context of your work, provide the necessary core definitions that are needed to understand the terminology you will be using in the rest of the thesis (if it's not standard IT terminology).

Therefore:

- ☐ Tell the *research area(s)* your work/project focuses on. If you are doing your thesis with me, likely candidates of research areas are Web Engineering, Data Science, Crowdsourcing, Service-Oriented Computing, Business Process Management.
- ☐ Tell possible *sub-areas* that are more specifically related to what you are doing. Again, if you are doing your thesis with me, likely candidates of sub-areas are chatbots, social knowledge extraction, business process matching/modeling, quality control in crowdsourcing, etc.
- ☐ Make the *heading* of your context section self-explaining by substituting "[topic]" in heading 1.1 with the sub-area most relevant to your work. It should read like "Context: quality control in crowdsourcing" or similar.
- ☐ If needed, introduce some *key definitions* (no need to introduce everything here, but be sure that the introduction does not use terminology the reader may not be familiar with). For instance, if you are working on chatbots, this is definitely a term that needs to be introduced here; it's not yet commonly known but it's crucial for the understanding of the rest of the thesis and introduction.
- ☐ Use *examples* to make definitions and ideas concrete and clear.
- ☐ Throughout, make *references* to the relevant literature.

Use of tenses and pronouns

Writing a thesis is writing a scientific document like scientific articles or research publications. There are two conventions that are usually applied in this kind of publications (admittedly, they may seem somewhat odd if not used to):

First, the most used tense is the *simple present*. The thesis is meant to describe a piece of work, from problem statement, to the conception of a solution, its implementation and evaluation. Yet, it's not a novel about your life, and it's not meant to provide a chronological story about what you did and didn't do. Content is presented in an order that is most effective to convey its message, not in time order. In this spirit, it's much more effective to say "in order to get result A, first we do X, then we do Y and then Z," instead of saying "in order to get result A, we did Y after having done X, then we went on doing Z." The order of actions, their interconnections, inputs and outputs already tell the dependency – if properly described. Most of the times, the most effective way to describe a solution or

methodology only becomes clear after trial and error. It's enough to explain the result, not how you got there chronologicallly.

Second, the *pronoun* used to talk about the own work is "our" (work). That is, it is custom to say "we" instead of "I," even if you are writing your thesis alone. However, don't forget about all the people that helped you get there: your supervisor, co-supervisor, colleagues, etc. This may sound strange at the beginning, but, at the other hand, using "I" too often risks to convey the impression that you are self-focused and egoistic, which is never good.

1.2 Scenario and Problem Statement

Now that the reader got the general context of your work and has an intuition of the problem you will be solving in the rest of the thesis, it's time to be clear about which *specific problems* your thesis project is going to solve. One way of doing so is by describing a *scenario* (a description of a real situation, with all its actors, roles, tasks, instruments, etc.) that provides evidence that there are one or more real problems right now that, with the current technology and understanding of the domain, are hard to solve or not solvable at all. If instead the problem(s) can be solved already, it should be evident from the scenario that this is possible only at a prohibiting cost or with unsatisfying guarantees on the quality of the result or not within useful time for the target user.

It's important that the scenario is written in such a way that the reader, after reading it, agrees with you that the problem you are focusing on is a relevant one, one that deserves being studied and solved. Consider that if you convince the reader here that your thesis is needed (after all, that's what this section is about), he/she will be very open to possible solutions and happy to see how you solve it. If instead you fail to convince the reader – let me be harsh – the whole rest of your thesis is useless in the eyes of the reader. This is the worst outcome you want.

Conclude this section by explicitly stating which of the problems evident in the scenario you are approaching. Don't raise false expectations! Never ever tell the reader there are five core problems and then solve only two of them in the thesis, without telling upfront that this is what you intended to do in the first place. As soon as you list problems, the reader wants to see a solution, unless you stop him/her immediately from thinking so by telling that out of the described problems you focus on a subset only, usually because this subset is already a huge research and development problem in its own.

In summary:

- ☐ Describe a *real scenario* that provides evidence of *real problems*.
- ☐ Convince the *reader* that the problems need to be solved.
- ☐ Use an *illustration* or *figure* to help the reader understand.
- ☐ If possible, provide *references* to literature that backs your assessment of the problem.
- ☐ Provide a clear *problem statement* that summarizes what came out of the scenario and your specific focus.

1.3 Methodology

Fixed the problem(s) you want to approach, you can approach it/them in thousands of different ways. Your way is just one of the thousands, and the reader may have (and very likely will have) a very different intuition of how to solve the problem(s) you just pointed out. So, clarify how you intend to proceed:

- ☐ Tell if you follow an existing *methodology* or not; if yes, name it and provide a reference to literature, if available. For example, Design Science [**?**] is a likely methodology to cite here.
- ☐ Tell which of the following *procedures*, *techniques*, *methods* you use in your work and for which purpose (put them also into the right order, so that their application or use makes immediate sense to the reader):
  - ☐ *Systematic literature review, survey*
  - ☐ *Statistical hypothesis formulation and testing*
  - ☐ *Software prototyping*
  - ☐ *Iterative development*
  - ☐ *Participatory design*
  - ☐ *Performance evaluation*
  - ☐ *Comparative studies*
  - ☐ *User studies*
  - ☐ *Expert interviews*
  - ☐ *Simulation/emulation*

- ☐ *Live experiments*
- ☐ *Case studies*
- ☐ *Mathematical theorem proofing*
- ☐ *Mathematical modeling*
- ☐ *Pseudocode*
- ☐ *Graphical modeling* (e.g., UML, ER)
- ☐ *Model-driven development*
- ☐ *Automatic code generation*
- ☐ ...

☐ Tell if you use some special *software instruments* that help you in your work. We are of course not talking about Word or Google Search. Perhaps you can tell that you used R for data analysis or some specific modeling instrument for automated code generation or simulation.

1.4 Contributions

Now that the reader knows what you want to solve and how you intend to proceed, you can anticipate the contributions your thesis makes to the state of the art. Attention, a thesis project may produce lots of different *outputs* (e.g., a software prototype, a set of registrations and transcripts of interviews, datasets collected during experiments) and *contributions* (e.g., a demonstration that some software solutions solves a given problem under well defined conditions, a formal proof that some property holds, empirical evidence that something works as expected). The former are all the artifacts produced throughout the work. The latter refer to *new knowledge* (if you are doing a full thesis) or the most important, *final output* (if you are doing a tesina). Sometimes, outputs and contributions overlap, but not necessarily.

Typical contributions are (multiple choices may apply to your thesis):

☐ A *systematic literature review* of the state of the art providing evidence for some argument

☐ The design of a *model* (mathematical, graphical, algebraic, etc.) describing how to solve a real world problem in a reusable fashion

☐ The drawing of *conclusions* (findings) from the analysis of a dataset describing some physical or virtual phenomenon

☐ The implementation of a *software prototype* solving a real world application problem

☐ The design of a *language* (textual, graphical) enabling others to solve own problems or to solve them easier

☐ *Formal proofs* of correctness, completeness or other properties of the proposed models or theorems

☐ *Objective evidence* from empirical studies (e.g., performance analyses or simluations) that demonstrate that the proposed prototype or solution works / works better than existing software or solutions that solve the same/similar problem(s)

☐ *Subjective evidence* from user studies or expert interviews backing the claims of viability of the proposed problem or solution/artifact

☐ A reasoned *argumentation*, e.g., based on a detailed case study, supporting the viability of the proposed problem or solution/artifact

Thesis vs. Tesina

Let me spend some words on the difference between these two. Before that, however, it is important to clarify the very purpose of your final project, be it a thesis or a tesina (a small thesis). The purpose of it is giving you the possibility to show that, after years of attending classes and giving exams, you are also able to *apply* the knowledge you acquired during your studies. In short, it's all about you showing that you are *mature*. Mature form a knowledge perspective, mature from an application perspective, mature from a work/teamwork perspective, mature from an ethical perspective.

It is common that a thesis project is not very well defined in its beginning and that even the supervisor does not really know how to approach a given problem or which problem to focus on in the first place. This may even be annoying to you, but attention: there is no intention behind it. Your supervisor is not withholding information from you to test you or to see if you get something. It's just the nature of real *problem solving*. If things were clear from the beginning, there wouldn't be any problem! Fledging out the problem and agreeing on a solution and methodology is a core part of you demonstrating your maturity – if not the most important one. *How* you proceed from the inception of the thesis idea to the final solution is as important as *what* you find and/or produce in the end.

This being said, a *thesis* in Politecnico di Milano usually requires you to make a contribution to the literature (the so-called state of the art). Making a contribution – from a science point of view – means creating new *knowledge*, that is, finding something that nobody knew before, demonstrating a property that nobody showed

before, improving the performance of a given system with a new algorithm, and similar. For a thesis, it is therefore not enough to produce a perfectly engineered solution. It is key that you also demonstrate, provide empirical evidence or proof that your solutions performs as claimed. Well, for a *tesina* this last demonstration is usually not required, and the focus is on the engineering of the solution. In addition, perhaps in the case of the tesina the solution to be engineered is also less complex then for a thesis, but this depends on the context and on how you want to measure complexity.

1.5 Structure of Thesis

Here you explain the structure of the thesis, so that the reader knows how to read it. Consider that not every reader wants to read through the whole thesis to find some specific information. Actually, only few will do so (your supervisor and co-supervisor, and the possible reviewer for sure). Many more will just leaf through it and look for specific types of information (e.g., the context of your work, your findings, how you implemented something, which technologies you used). It is your duty to accommodate them all. How? By telling them how your thesis is structured.

Therefore, in this section you provide a brief description (2-3 sentences) for *each* chapter that follows this introduction. Use an itemized or numbered list to structure the text, like this:

- ☐ Chapter 2 introduces the state of the art and...
- ☐ Chapter 3 provides...
- ☐ ...

Structuring text
Besides telling the reader how the content of your thesis is organized into chapters, it is important that you master some basic text structuring techniques. To organize your text there are lots of instruments you can use: chapters, sections, sub-sections, paragraphs, itemized lists, numbered lists, code examples, figures, images, screen shots, captions below figures, tables, and so on. Use them all! Don't write text without structure. Never.

Be aware that the structure of your text, that is, how you present your work, conveys a lot of information about how well you actually understand what you are writing about, how much you care about being clear and helping your reader understand, and how much value you give yourself to your own thesis. A well

structured presentation of content that the reader can understand and agree with is a huge plus in this respect. Text that lacks proper paragraphs, does not use lists where needed, etc. is a minus and also much harder to read (think about how much a well structured text can help you go back ten pages and find concepts you know you read about compared to a text that comes without an easy to memorize formatting and structure). When writing, think about some of your textbooks. Since you are doing an engineering degree, I'm sure these are textbooks that make exemplary use of the different formatting instruments available.

Chapter 2

State of the Art

This chapter discusses the state of the art that is relevant for your own work. What does that mean? It means that it provides the reader with all the relevant references he/she may need to know in order to understand better three things: (i) the context of your work, (ii) the problem and the need for a solution, and (iii) the value of your contribution. You achieve this by citing works or scientific papers that solved the same or similar problems in the past. Citing does not just mean adding a references to the bibliography and printing a number here; it means you tell the reader about the merits and possible demerits of each of the references you feel relevant. Of course, doing so requires you to first read each reference and, most importantly, to understand it. There should be lots of references in this chapter.

It is advisable that you structure the chapter into sections in function of the topics you treat. If you do so, before starting with the first section of the chapter, explain the reader how you structure your discussion in one paragraph.

- ☐ *Read* relevant literature and or *test* related software or tools.
- ☐ *Summarize* your reading.
- ☐ Provide correct *references* (the bibliography in the end of this document).

2.1 [Topic one]

...

2.2 [Topic two]

...

2.3 Summary

Close the state of the art chapter with some words that connect the discussion of the references to your thesis. Pay attention that the reader understands why you discussed the works/topics you discussed and how they are related to what you do.

- ☐ Show that in the state of the art the *problem* you want to solve has not yet been solved or not been solved in an as efficient / effective / easy to use / cost-saving fashion as you target with your work.
- ☐ If your work has similarities with some *specific references*, point them out here and explain why these are particularly important to you. Perhaps you started your investigation from the outputs of a specific paper or you want to improve the performance of an algorithm studied earlier; it's good to mention this here.
- ☐ Attention: this is not yet the place where to anticipate *your solution*. You may give hints, but it's too early to make a comparison between your work and the state of the art, as the reader does not yet know anything about your work. This discussion can go into the final chapter.

Chapter 3

Data Collection

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

- ✏ Data from existing datasets
- ✏ Data collected with different approaches
- ✏ Hand-labeled data

3.1 Tools

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

3.1.1 Tweepy

Tweepy is a python wrapper for the Twitter API. Two main methods were used to collect all the dafault 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 datails of a single tweet, such as the full text, number of retweets and replies.

3.1.2 Botometer

Botometer [4] checks the activity of a Twitter account and gives it a score based on how likely the account is to be a bot.

Using their dataset they tested most classification algorithm and highlighted Random Forest as their final classifier, since it gained the highest accuracy. Its accuracy was 98.42% and 0.984 F1 measure [6].

Botometer provides API to check Twitter accounts genuinity.

3.1.3 Hoaxy

Hoaxy is a tool that visualizes the spread of articles online. Articles can be found on Twitter, or in a corpus of claims and related fact checking.

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

3.2 Datasets

3.2.1 Caverlee-2011

This dataset is composed by content polluters, detected by a system composed by 60 social honeypots, which are Twitter accounts created for serve the purpose of tempting bots; and genuine accounts, randomly sampled from Twitter. They 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, they 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 statuses about marketing, business, and so on
Friend Infiltrators	Their profiles and tweets seem legitimate, but they mimic the mutual interest in following relationships

They manually checked each cluster to asses their validity. For each content polluter, they save their 200 most recent tweets, their following and follower graph, and the temporal and historical profile informations.

In order to collect genuine users too, they randomly sampled 19,297 Twitter ids and monitored them for three months, to see if they were still active and not suspended by the social platform moderation service [6].

They 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, they chose 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 is composed by genuine accounts and bots. In this dataset there is a deeper differentiation for the bots, 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 spambots #1	retweeters of an Italian political candidate	991	1,610,176	2012
social spambots #2	spammers of paid apps for mobile devices	3,457	428,542	2014
social spambots #3	spammers of products on sale at Amazon	464	1,418,626	2011
traditional spambots #1	training set of spammers used by Yang [2]	1,000	145,094	2009
traditional spambots #2	spammers of scam URLs	100	74,957	2014
traditional spambots #3	automated accounts spamming job offers	433	5,794,931	2013
traditional spambots #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 automatization.
- 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 spambots #1**.
- **Social spambots #2** are accounts that promotes mobile app, using popular hashtags for months.
- **Social spambots #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 spambots #1** dataset is the training set used in [2].
- **traditional spambots #2** are users that mention other users in tweets containing scam URLs. They usually invite users to claim a prize.
- **Traditional spambots #3** and **traditional spambots #4** are bots that continously tweet job offers.
- **Fake followers** are account involved in increasing popularity of other users. In order to collect them, they bought followers from fastfollowerz.com, intertwitter.com and twittertechnology.com. [13]

The intent of this project was to find a methodology useful to detect sophisticated spambots on Twitter. This novel category differentiates from the traditional spambot type, due to the ability of those content polluters to mimic the human interactions on the social platform. They relied on both crowdsourcing and machine learning experiments to compare the accuracy of the detection of such spambots. The experiments highlighted the lack of ability to detect this new wave of spambots, even with the help of human annotators.

3.2.3 Varol-2017

This dataset contains a list of Twitter accounts, labeled as bots (1) or humans (0).

The construction of the Varol dataset starts with the indentification 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 or breadth-first need an initial users set. During the observation window about 14 million user accounts has been gathered. All the collected users must 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 [6], they 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 with the newer 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 3000 extracted accounts were manually labeled by some volunteers. They analyzed users profile, friends, tweets, retweets and interactions with other users. Then they assingned a label to each user. Of course the final decision is conditioned to personal opinion [9].

3.2.4 BotBlock

Botblock (https://github.com/dansarie/Botblock) is a Twitter block list containing the user ids of a large number of known porn bot accounts. They are mainly used to aggressively market porn sites.

3.3 Varol clustering

At first try, we wanted to undestand if different kinds of bots were easy to distinguish, using their profile features only. er, we didn't know what kinds of bots populate Twitter for the most. So, an unsuperised 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 [9], 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 browser all the possible features. 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 analisys based on a hierarchical clustering with single linkage and euclidean distance. The data must had been preprocessed and cleaned.

Here we illustrate what kind of preprocess operations were performed for that purpose:

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)

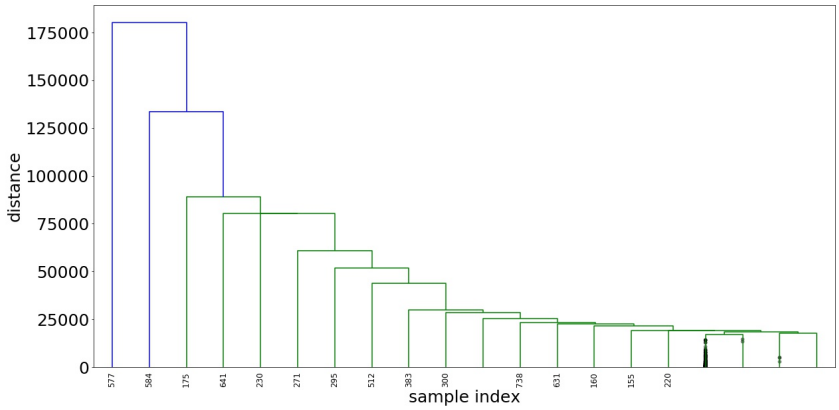

Figure 3.1: Hierarchical Clustering Dendrogram (truncated to the last 20 merged clusters)

Since we didn't know how many categories of bots were listed in this dataset, the first step consisted in understanding wich was the optimal number of clusters to look for. To achieve that goal, we applied *hierarchical clustering*. 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.

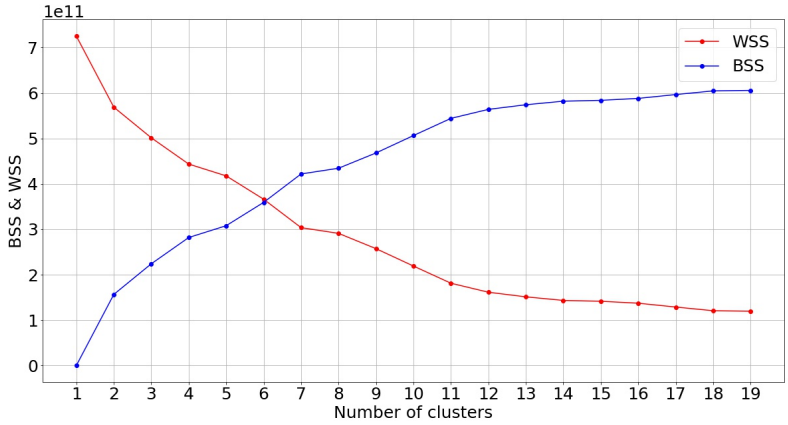

Figure 3.2: Hierarchical Clustering - knee-elbow

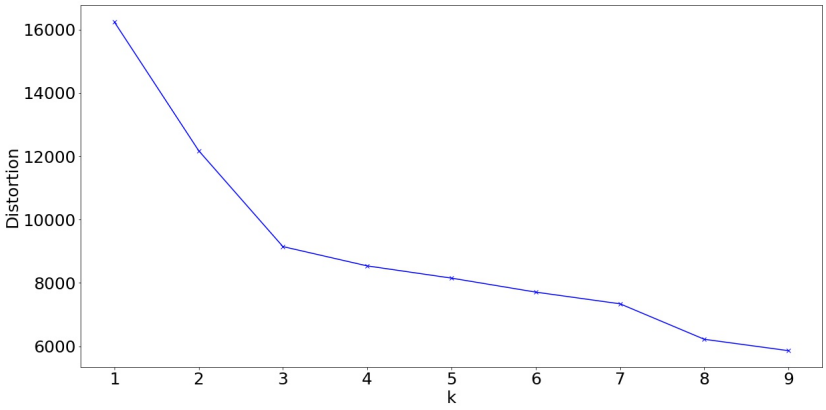

Figure 3.3: The Elbow Method showing the optimal k

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 methond (3.3). In this figure there is an elbow beetween 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 correlation among accounts in the same cluster, so this technique didn't seem to fit the speeding up of the labeling process, nor to create useful features for a classifier.

3.4 Collection

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-spreader
- Spambots
- Fake-followers

We started with a hand-labeling of the Varol dataset [9]. 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. In these cases, tehy were temporarily signed as "general purpose".

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

category	labeled account
NSFW	31
news-spreader	71
spambots	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 choise brought some noise to our data, that 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 [9] 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.)" [9]. 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 Varol research, but they are important in [13], 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 perform 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 6000 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 setp, because we gathered both more long-lived accounts and more extreme accounts (which probably have shorter lives). We finally obtained 6903 users and 198378 tweets.

3.4.2 News-spreader

Many bots on Twitter are news-spreader. The goal of these users is to spread politics, sports or actuality 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 the 80% of these "pre-elections bots" are still alive [11]. We think that part of our news-spreader dataset includes some of those accounts.

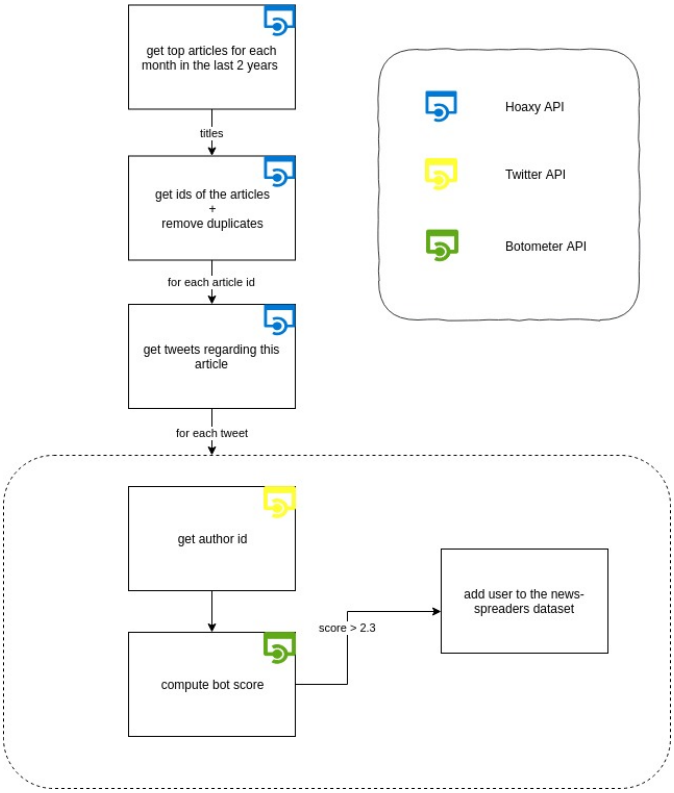

Figure 3.4: Collection of news-spreading bots, approach 1

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 willing 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.

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

that users who didn't tweet enough statuses to be included in this class. This hand-made analisys 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 3590 accounts and 333699 tweets.

3.4.3 Spam-bots

As seen before, spambots were already collected in [13]. Authors allowed us to access to their dataset, so we obtained the spambots 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 spambots 1
- social spambots 2
- social spambots 3

We chose this categories because they contain the most popular kinds of spambots, that are the ones who advertise products, services or mobile applications. We ignored *"social spambots 1"*, since they are italian news-spreaders and *"traditional spambots 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 spambots. Finally we gathered 4943 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 Cresci research [13]. 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 6307 users and 41683 tweets.

Figure 3.5: Collection of fake-followers bots

3.4.5 Genuine

Finally we needed genuine accounts. We used again the Cresci dataset [13] and we filled it with all the Varol users labeled as humans. We again performed scraping on the existing accounts and we collected 3661 users and 263240 tweets in total. Morover, we gathered all the Genuine users from the Caverlee dataset [6]. these are many more than other classes, but they will be necessary for an future classification between bots and humans. We perfomed scraping and we were able to obtain 15701 users and 1278852 tweets.

3.4.6 Bots

We needed a set of unlabeled bots in order to train a binary classifier, that is able to classify among bots and humans. We again scraped the Caverlee dataset [6] and we gathered 15539 and 1276457 tweets.

3.4.7 Final Datasets

At the end of the collection, we set up two datasets. The first one is the *Multiclass* dataset, that contains only bots labeled by their behaviors, its composition is the following:

category	# users	# tweets
NSFW	6903	198378
news-spreader	3590	333699
spambots	4943	458809
fake-followers	6307	41683

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

category	# users	# tweets
Bots	15539	1276457
Genuine	15701	1278852

3.5 Data visualization

As the collection of the data was completed, we explored our final dataset. In this section we will only focus on different cateogories of bots, which is the most iteresting 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 less tweets than the others, while news-spreaders have a lot of tweets, but we collected less profiles.

category	target id
NSFW	0
news-spreader	1
spambots	2
fake-followers	3

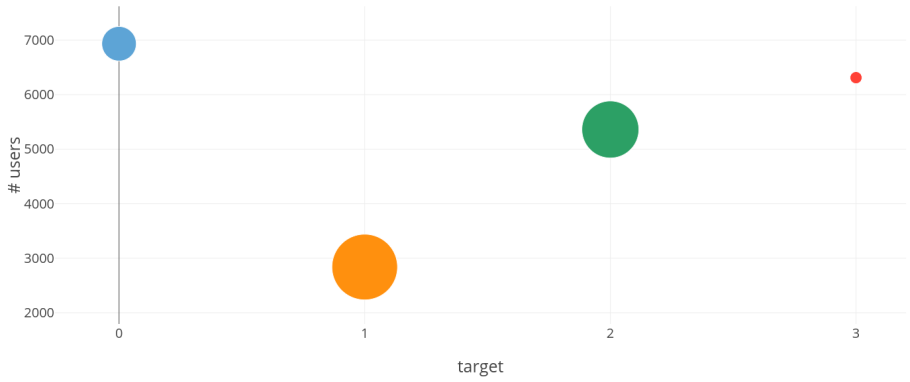

Figure 3.6: users amount and tweets

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

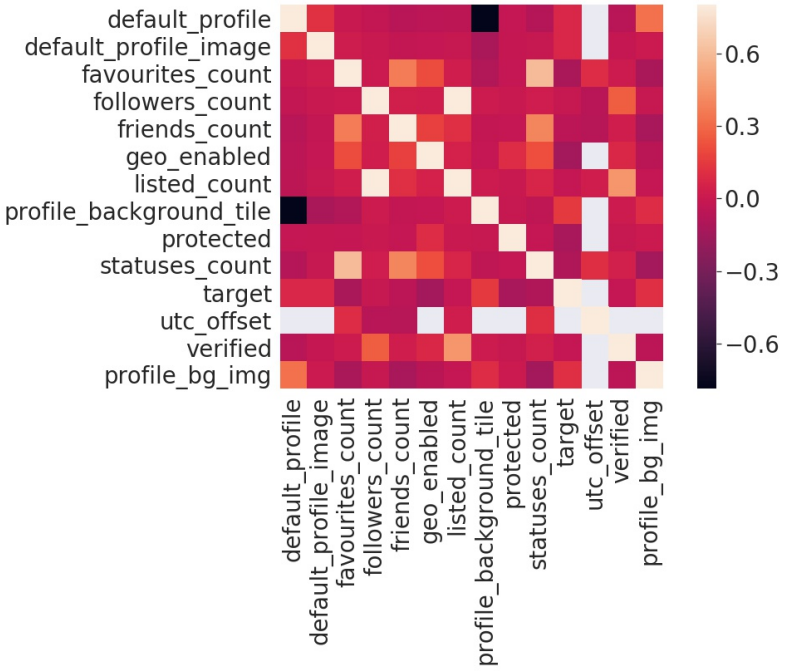

Figure 3.7: heatmap

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

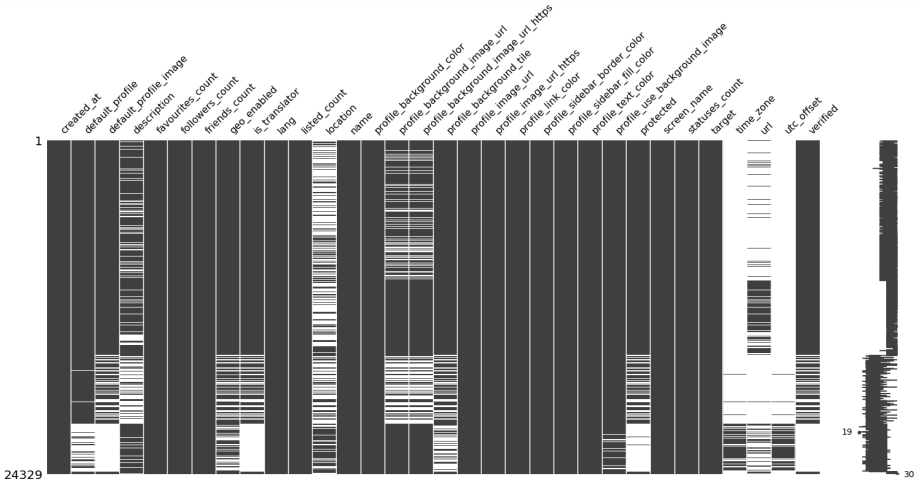

Figure 3.8: heatmap

Finally we performed three further analisys. With the heatmap we could not detect which features were more important. Anyway, during the hand-labeling step, we understood that some of these features were instead very usefull 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 boxplot. It is a method used to represent groups of numerical data through their quartiles. In Figure 3.9 we can analyze the statuses count 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 behavior: news-spreaders and spambots are the classes with more tweets, while fake-followers have less 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.

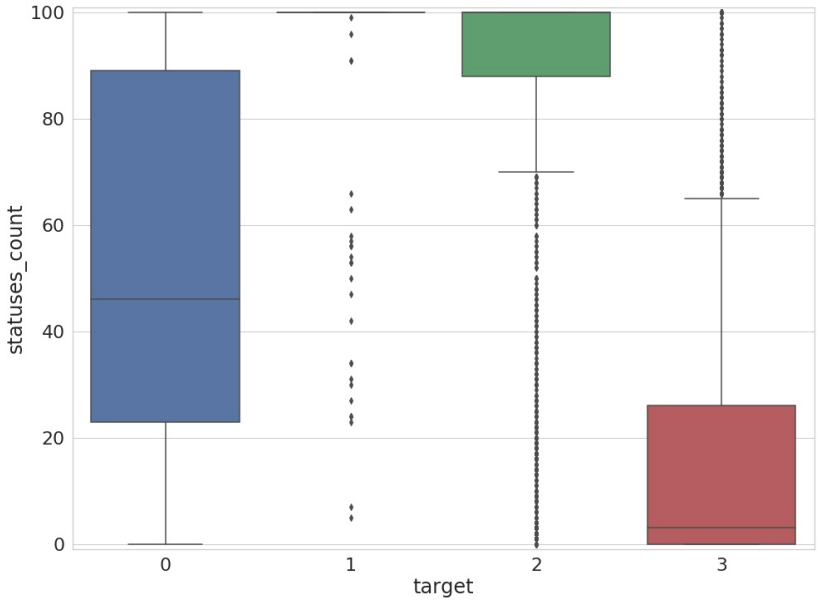

Figure 3.9: boxplot statuses_count

In Figure 3.10 and 3.11 we performed the same analisys considering the *"friends_count"* and *"followers_count"* features. We limited the charts between 4000 and 2000 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 behavior between them. For the genuine accounts we need to make an extra observation. Of course there are a lot of human users on Twitter and they have very different behaviours.

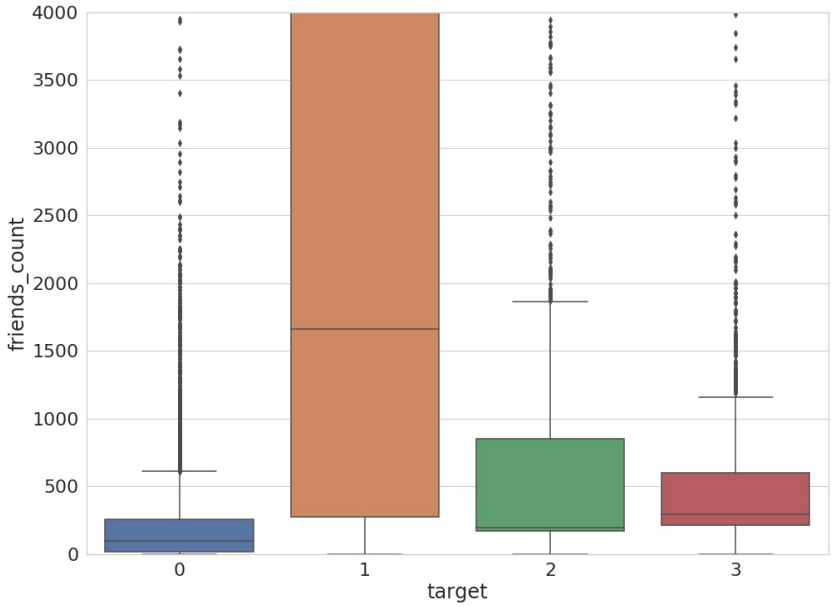

Figure 3.10: boxplot friends_count

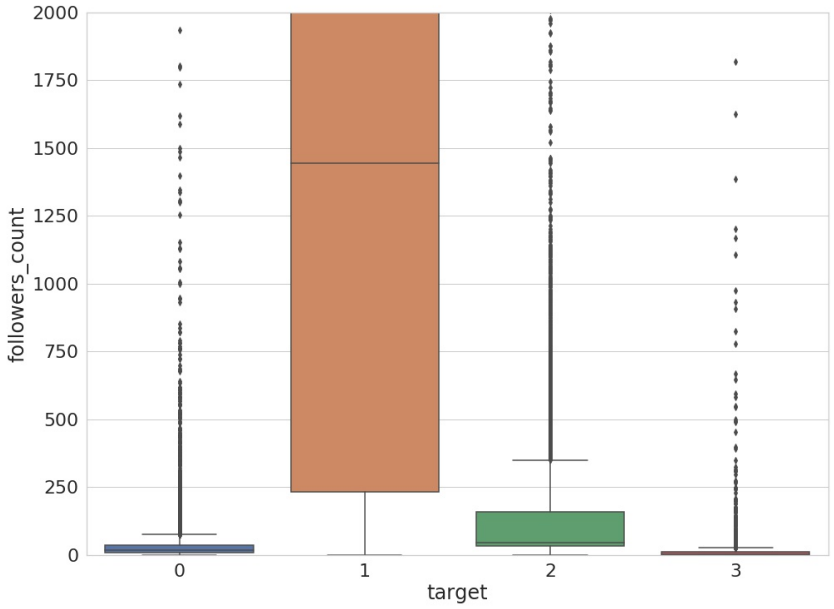

Figure 3.11: boxplot followers_count

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.

Twitter APIs 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 informations and enhance our data with them. This was the bulk of the work, but it helped us to catch characteristic behaviors of some user. We created many features. Some of them are descriptive, like the lenght of strings (name, description ecc) or the count (miminimum, 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, because further material led us to a slower, but equivalent, prediction over test samples.

At the end of this stage, the resulting - and final - feature vector will include 38 features.

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 analyze 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 was in tuples with target 3 or 4. In order to uderstand the behaviour of this feature, we printed its values count for each indicted target.

target	3
value	mean
0	2868
1	228

target	4
value	mean
0	181
1	19

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

- ✏ **profile_background_tile**: As for "default_profile_image", all the missing values belonged to target 3 and 4. We used the same approach and we obtained:

target	3
value	mean
0	3086
1	99

target	4
value	mean
0	1347
1	147

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

✏ **profile_use_background_image**: All the missing values are still in the last two classes.

target	3
value	mean
0	12
1	4983

target	4
value	mean
0	25
1	3246

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 were been added to the users feature vector, in order to append new informations for each account. Here is the list of this first 18 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

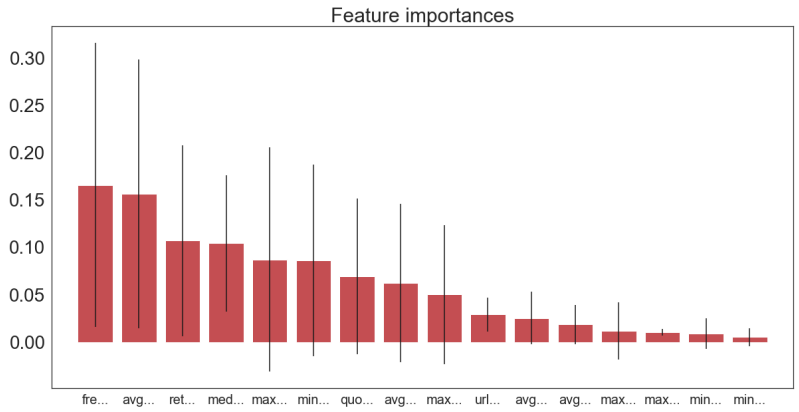

Figure 4.1: Descriptive features ranking with Random Forest

As shown in Figure 4.1, this first group has been performed a ranking function on, with the following result:
1. freq (0.148675), 2. avg_len (0.140074), 3. media_perc (0.102189), 4. min_len (0.098582), 5. ret_perc (0.095440), 6. max_len (0.087577), 7.

avg_ret (0.082658), 8. max_ret (0.063234), 9. quote_perc (0.060392), 10. url_perc (0.030648), 11. avg_hash (0.030057), 12. avg_fav (0.016788), 13. max_fav (0.013613), 14. min_hash (0.012777), 15. max_hash (0.011260), 16. min_ret (0.004959), 17. min_fav (0.001077)

4.4 Intrinsic features

Due the multiclass 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.
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 encapsule this distinctive behaviour 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

The features were tested to fit a

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 each user a measure of the average intra-distance of its 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.

4.4.3 Ranking

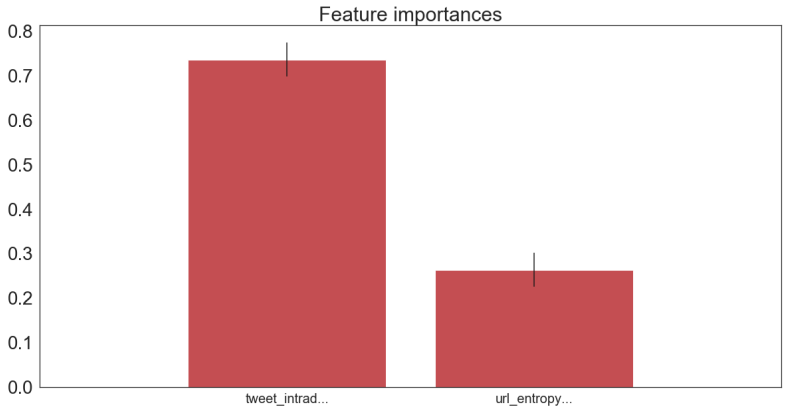

Figure 4.2: Intrinsic features ranking with Random Forest

Figure 4.2 shows the inner ranking of these two new features, that have these scores: *1. tweet_intradistance (0.736293), 2. url_entropy (0.263707)*

4.5 Extrinsic features

Once we have modelled the personal twitting 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

Multiclass Dataset

After the dictionaries related to the multiclass 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 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 words of every tweets and match them with all the dictionary. 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
p*à*	0.160
...	

News-Spreader 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 Multiclass 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 ths 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 dataset (starting from the multiclass 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. Figures 4.4 and **??** list 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 Figures 4.5, **??** represent the situation with half of the words included in the Safe Areas. So do Figures 4.6 and **??**, with the 75% of the safe terms, and, finally, Figures 4.7 and **??**, 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.

✏ 100 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.

We needed another confrontation term, which could tell us something about the goodness of the classifier, in relation with the number of safe words used for the computation of the features. We validated a Random Forest multiclass classifier, with increasing number of trees in the forest. Figure 4.3 shows the results.

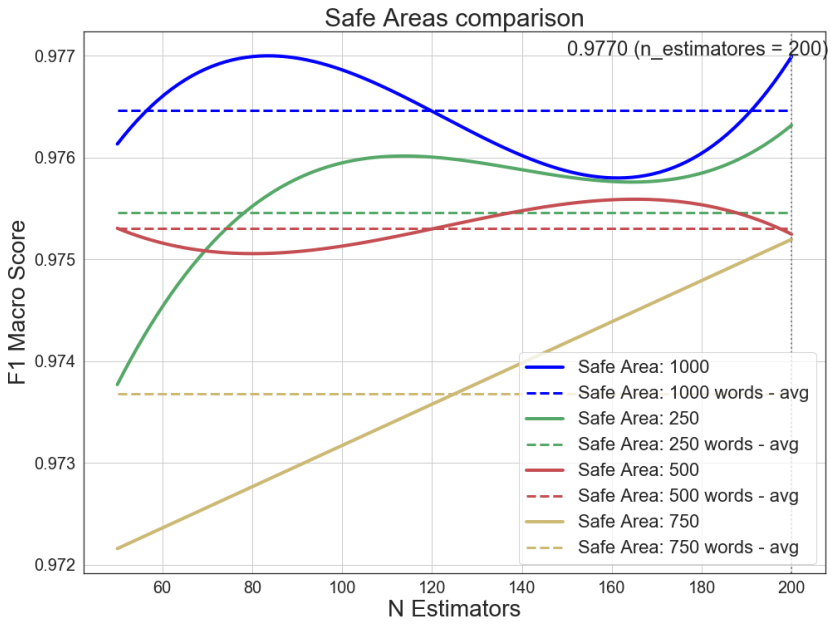

Figure 4.3: Safe Areas comparison

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.

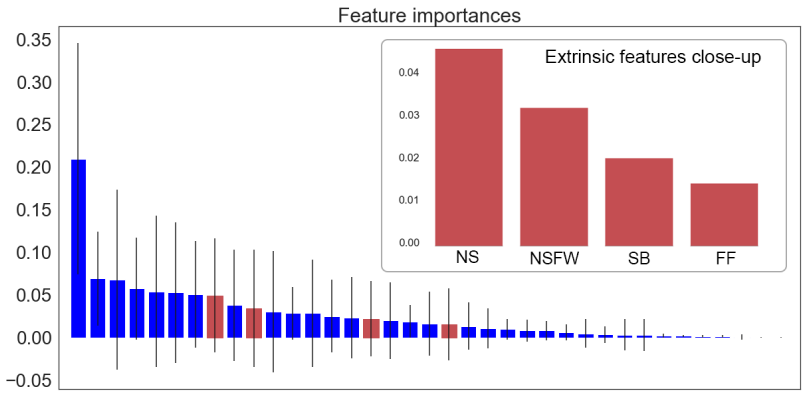

Figure 4.4: Features ranking - Safe Area = 250

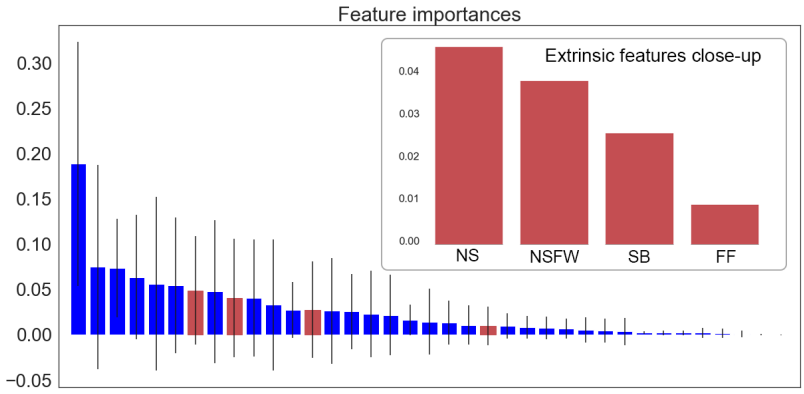

Figure 4.5: Features ranking - Safe Area = 500

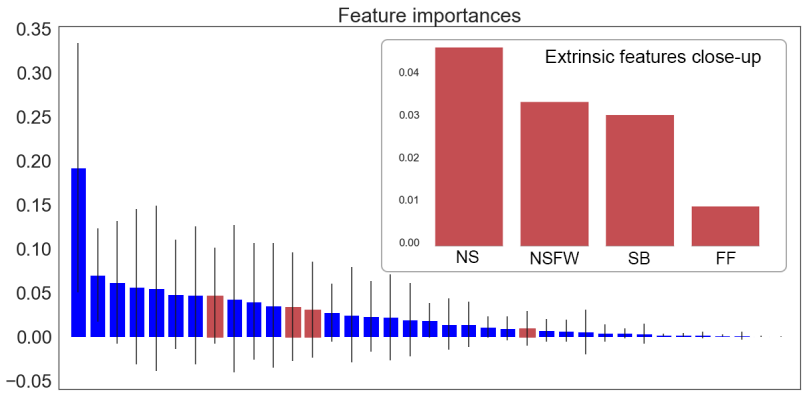

Figure 4.6: Features ranking - Safe Area = 750

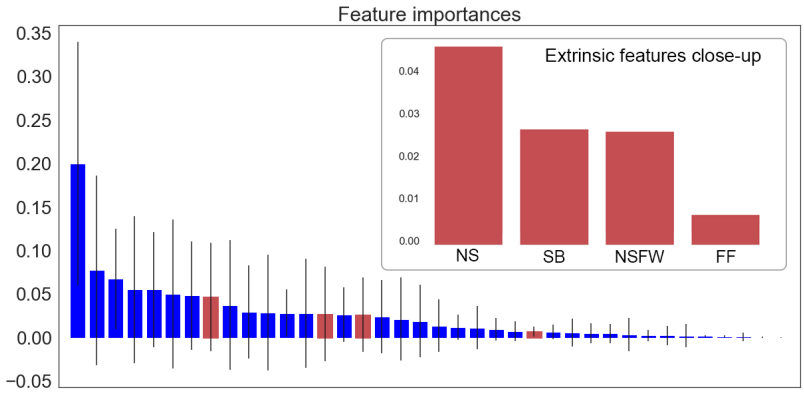

Figure 4.7: Features ranking - Safe Area = 1000

According to our user-based, descriptive and intrinsic features, both classes act in a similar way: they spam similar links, have high tweet frequency and their contents are often repeated.
What could be the difference?

We started with "blind" classifier, that was the main problem. 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 [3] 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 output layer. The repeated stacking of such models makes the architecture much 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 by a collection of clean pictures of random users, already collected in the Selfie Dataset [7] . This dataset contains 46,836 images, wiht metadata about genders, ages and so on; it's built for research purposes. We just randomly picked the same number as for the other class, 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. We thought that the bias introduced with this new features would have been under control and that it wouldn't compromise the final results of our models.

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

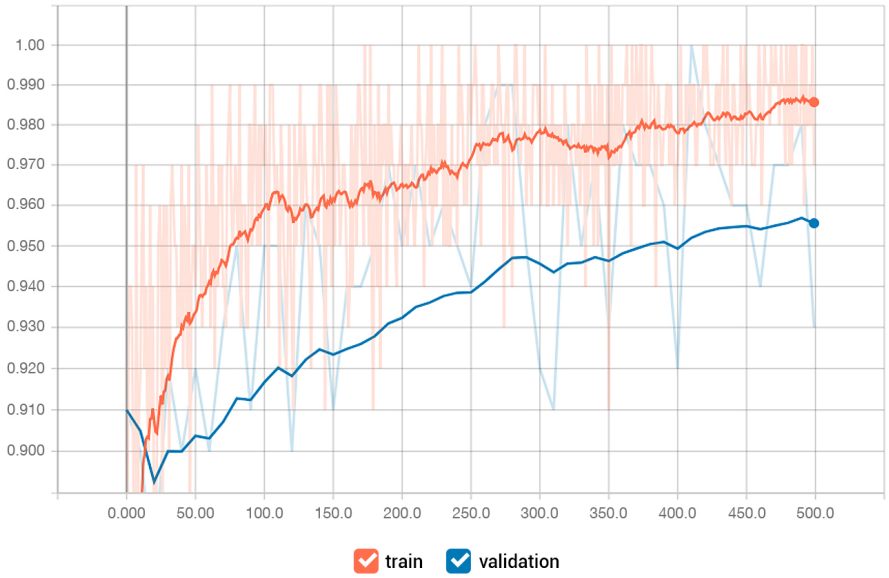

Figure 4.8: Train and validation accuracy through learning steps

tweets.

Obviously, the prediction time has been affected by this new preprocessing stage, but we think that this number of pictures analysed (up to eleven), makes the generalization duration still reasonable.

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 Multiclass Dataset

At the end of this process we obtained a feature vector composed of 37 elements, 12 based on the user and 26 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 33 elements, 12 based on the user and 21 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

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 handcrafted 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 three 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 on their tweets only
- a K-neares-neighbors classifier, used over the four bot classes, based on the user's features only

The algorithm were used together into a pipeline workflow, 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 multiclass models.

5.1 Baselines

The choices explained in this section were made at the same time of the ones listed in the Baseline section of the last chapter.

This is, basically, the same stage of the above-mentioned, but in a model-driven perspective. The features involved are the ones described in section 4.1, but we started from that base, to try different classifiers over it. We chose to evaluate the performances of raw classifiers, for both the binary and the mutliclass problem. Each type of classifiers has been tested with the respective dataset, but considering only the baseline features of those data.

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.

$$InformationGain(X) = Information(D) - Information_X(D)$$

where

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

and

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

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 multiclasses 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, it 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 multitarget 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 multiclass problem and we aimed to maximize AUC score for the binary case.

Multiclass 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 multiclass 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 of the multiclass 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, Botmoter 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 stage is performed separating the samples in 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 Multiclass

In this case, we decided to use the 75% of the data for the training set and the 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 algorithm 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

Binary

This evaluation was made with the common splitting ration 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	

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:

Multiclass

✏ *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 much 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 was inspired by the work made with Botometer [4], which involved a binary labelled dataset, with bot and genuine accounts. They 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 multiclass 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 by part of our collected records and by 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 [6].

We setted 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 4.7, 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 multiclass 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 multiclass 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 hyper-parameters 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 [9], 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 Botmoter 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'.

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

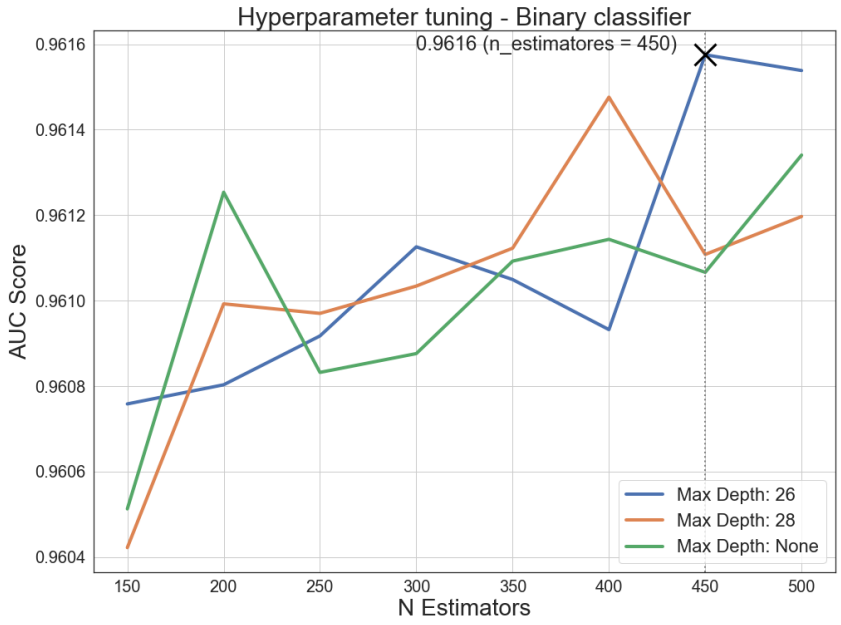

Figure 5.1: Grid search results

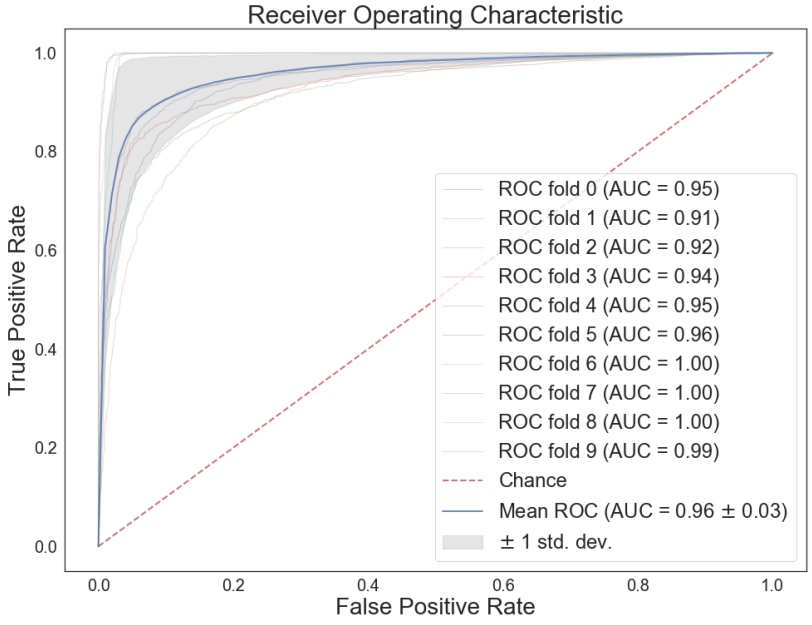

Figure 5.2: ROC curve

unseen data had a further discrimination among bots, it was easy to sample some records randomly, replacing their multiclass 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 multiclass 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.

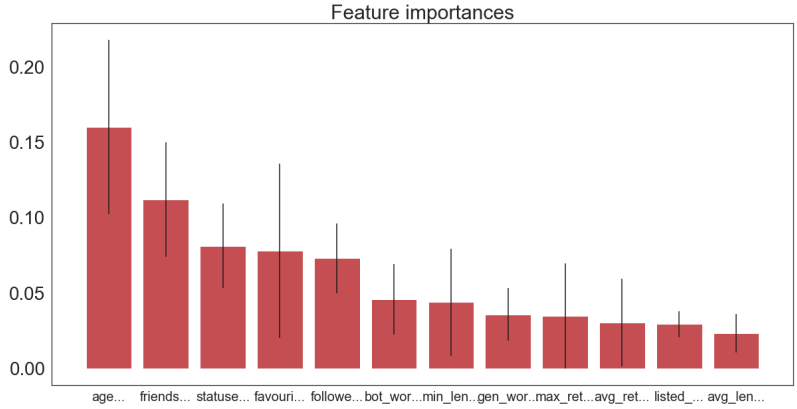

Figure 5.3: Binary Random Forest features ranking

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.

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

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

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 tied to mitigate this performance loss, by excluding the main sus-

pect 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 multiclass 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 Serch. 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 make considerations about what will happen when we will allow people to classify data coming from outside our collected samples. The age feature would had drove 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.

While cross-validating the model, we tested the complete features vector

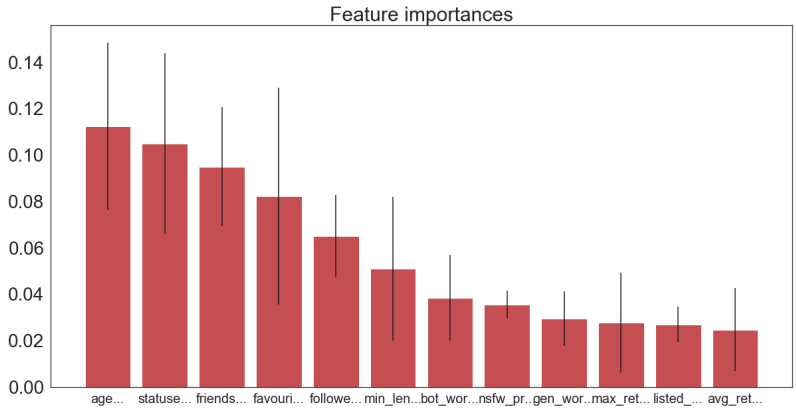

Figure 5.4: Features ranking with augmented data - Top 12

(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 multiclass 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 hyperparameters 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 with 34 features, 600 estimators, entropy splitting criterion and 26 levels of

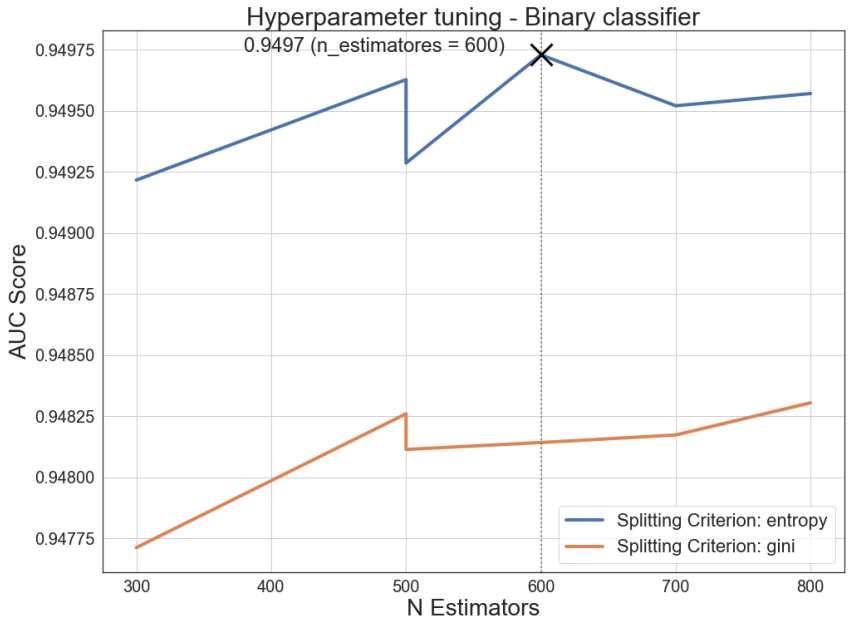

Figure 5.5: Grid Search with extended data

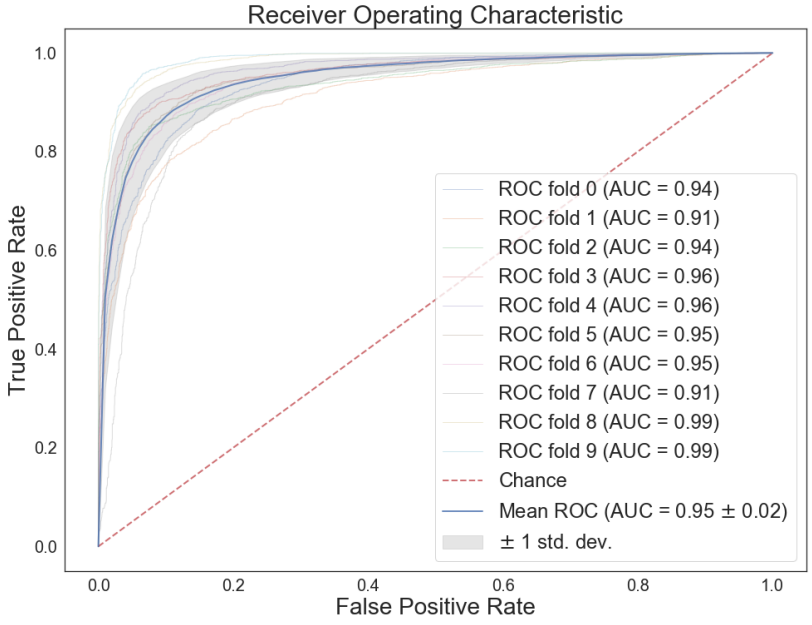

Figure 5.6: AUC score with extended data

maximum depth.

5.3 Multiclass ensemble classifier

This is the first piece of the thesis we worked on.
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 multiclass 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; a final text-based Naive Bayes classifier, which reads the tweets' texts and classifies them.

At first, this ensemble of those three models should had 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 multiclass 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 to a poor bot vs genuine detection tool. We couldn't accept that situation, because, in order to go deeper than other works, in bots' behaviours 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 multiclass 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.

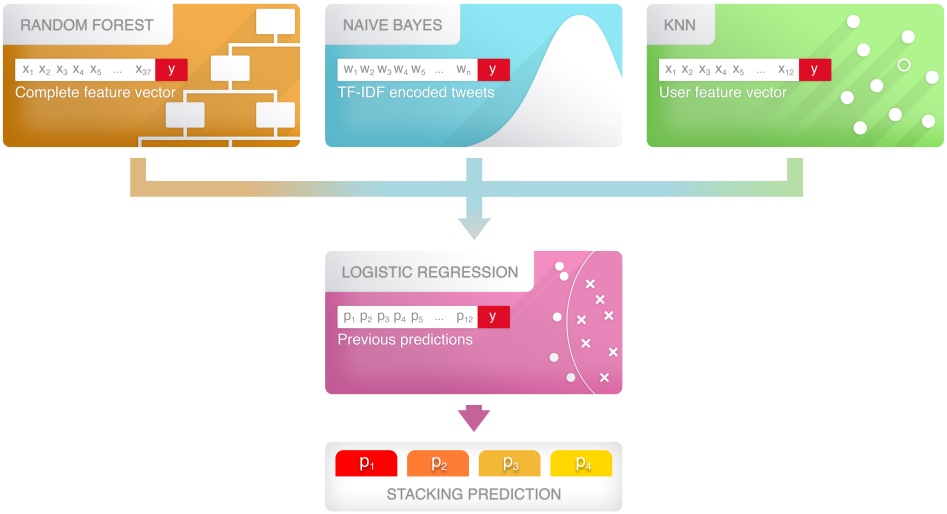

Figure 5.7: Multiclass ensemble schema

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 with its five different labels. The algorithm was fed with 21,445 samples and 37 features. the amount of records were light enough to consider K-fold crossvalidation, 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 heath-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* (0.179115), 2. *nsfw_profile* (0.068165), 3. *freq* (0.061246), 4. *tweet_intradistance* (0.060451), 5. *news_spreaders_words_score* (0.058583), 6. *statuses_count* (0.053364), 7. *avg_len* (0.051733), 8. *followers_count* (0.051187), 9. *NSFW_words_score* (0.043496), 10. *ret_perc* (0.041200), 11. *min_len* (0.038870), 12. *spam_bots_words_score* (0.035445), ... 37. *min_fav* (0.000146).

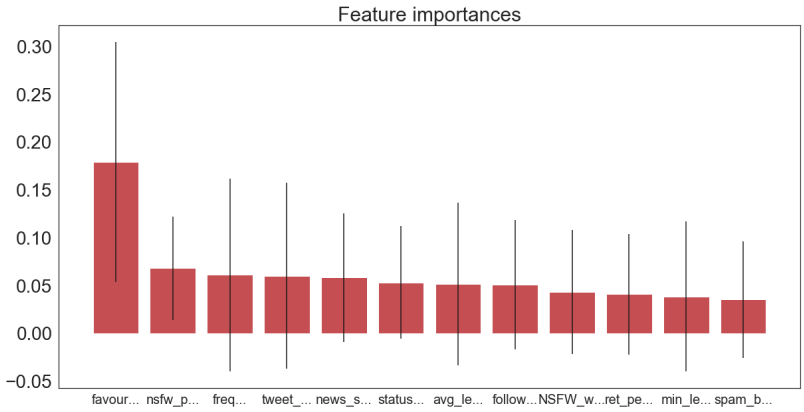

Figure 5.8: Random Forest top-12 feature ranking

As Figure 5.8 shows, we could find some of our crafted features inside this list: lots of tweets descriptive features (*avg_len, freq, ret_perc*, etc...), as well as the *tweet_intradistance* attribute 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 multiclass 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.

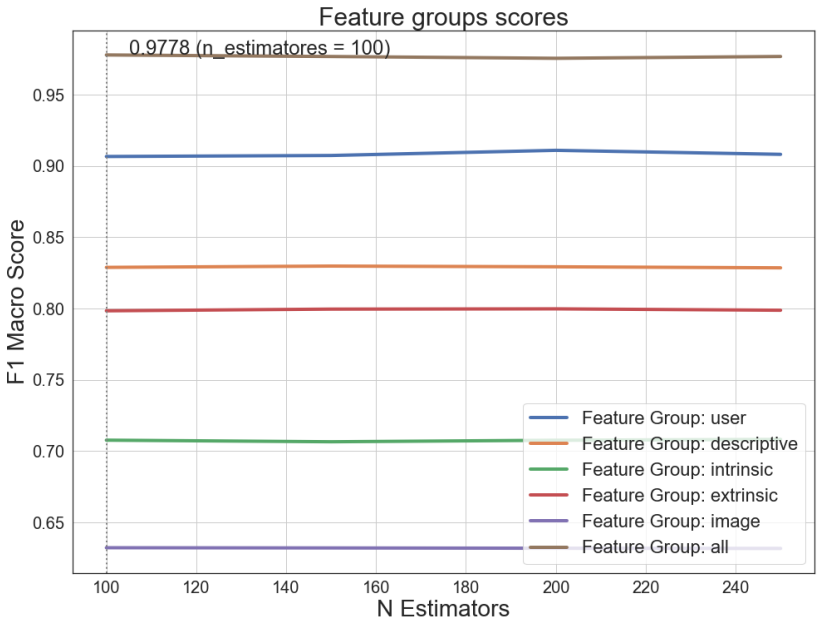

Figure 5.9: Performance over different feature clusters

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 had 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.

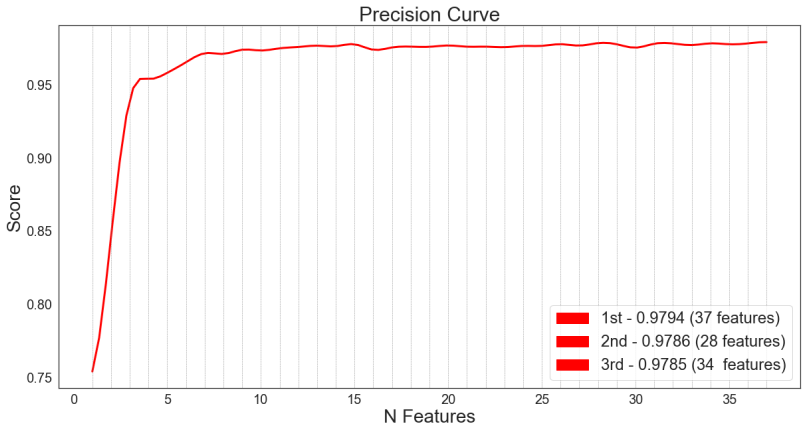

Figure 5.10: Precision trend along with number of features tested

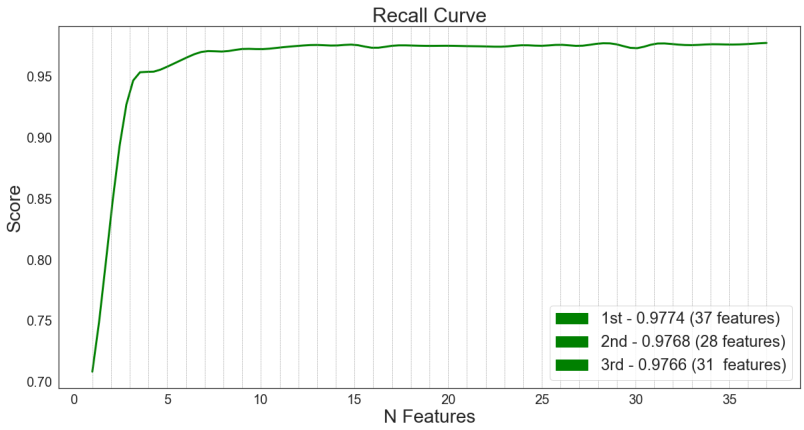

Figure 5.11: Recall trend along with number of features tested

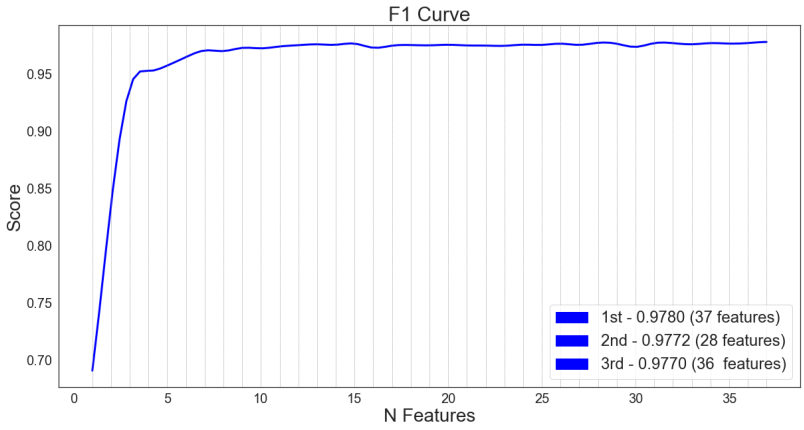

Figure 5.12: F1 trend along with number of features tested

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 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, one again, during this exploration. Since we were testing a Random Forest, we wanted to play with the number of estimators (tree) to include in the pool, as well as the maximum depth of each tree and the splitting criterion.

The Figures 5.13, 5.14 show how the average F1 score, measured on 10-fold crossvalidation, changes with the increasing of the number of estimators in the forest. The different coloured lines represent the *max_depth* hyperparameter. The first Figure (5.13) shows the Grid Search results, with the *gini* splitting criterion. The second one (5.14) represent the situation

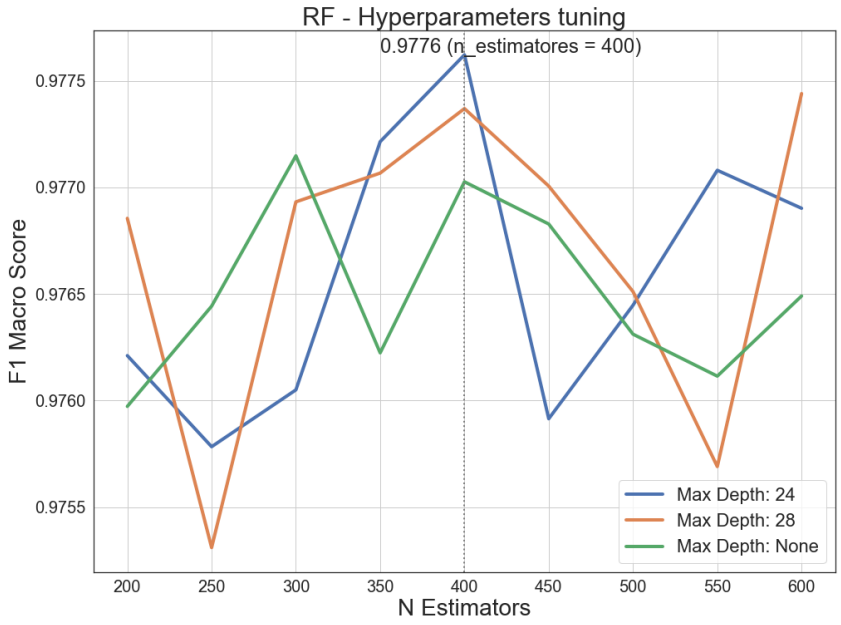

Figure 5.13: F1 scores with 'Gini' criterion

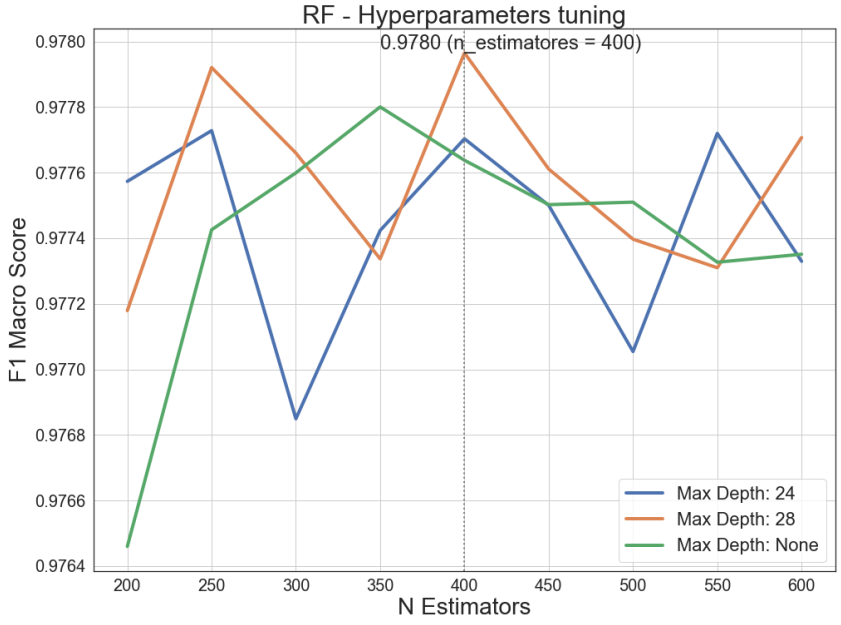

Figure 5.14: F1 scores with "Entropy" criterion

having *entropy* as a splitting choice. We combined nine numbers of estimators (200,250,300,350,400,450,500,550,600), 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 Information Gain criterion and the maximum reachable depth (for each of the 400 estimators) equal to 28 levels.

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.

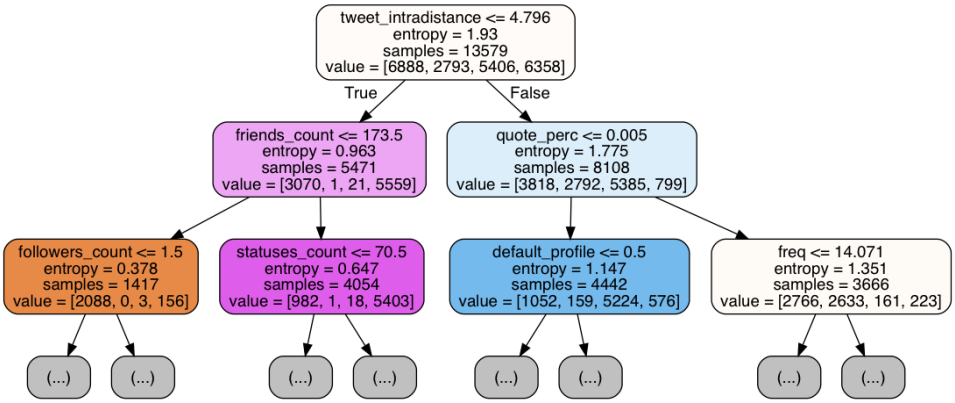

Figure 5.15: Tree estimator of the Random Forest model

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 multiclass 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 wa 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 multiclass model, where needed.

Dataset

The dataset is composed by 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 was a standard normalization of the features. This last option provided low 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 gGin 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.

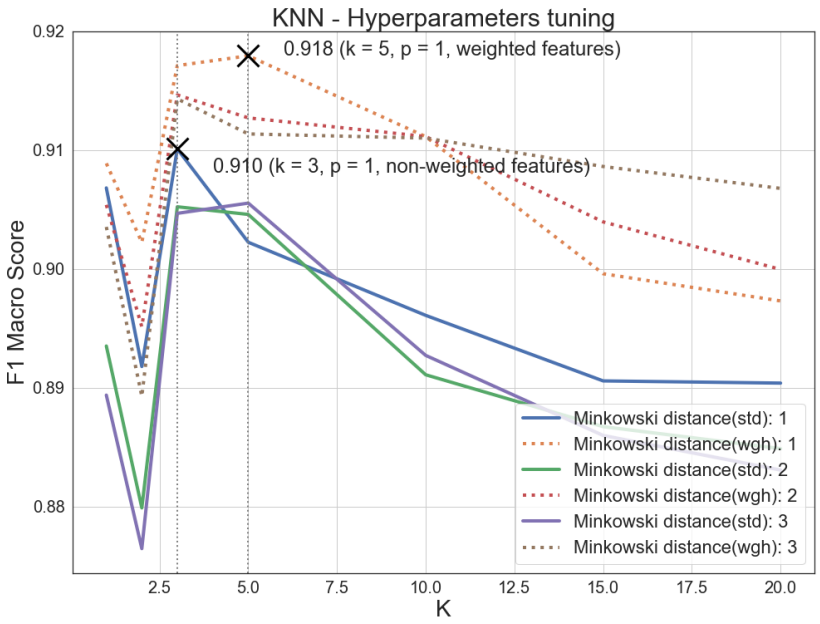

Figure 5.16: Hyperparameters p, k tuning, with different weightings

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.

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 was just exclamation 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 labelling each tweet with the target 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 bot's tweets are aimed to a goal. The shape of the dataset is the following:

category	# tweets
NSFW	196712
news_spreaders	280300
spam	453719
fake_followers	41316
genuine	261233

It seems unbalanced, anyway it is appropriate to keep all the data. Some spambot's 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 [5]. Naive Bayes models allow each attribute to contribute towards the final decision equally and independently from other attributes" [12].

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 usefull words. For example articles, smiles, punctuaction 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 this tasks, we used a *sklearn Pipeline*. It is an object that contains intermediate steps of 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 multiclass problem.

Holdout evaluation

The final text classifier has the following performance:

✏ **F1 = 0.71** with TF-IDF

✏ **F1 = 0.64** without TF-IDF

Anyway, 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

Final 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 multiclass 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 multiclass 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 model were fitted, we used the *predict_proba()* method of the Scikit-learn implementations of the classifiers, in order to retrieve "soft classifications". 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. The predict_proba() method computes the probability, for a sample, to belong to the highlighted class, without assigning the most probable target. 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.

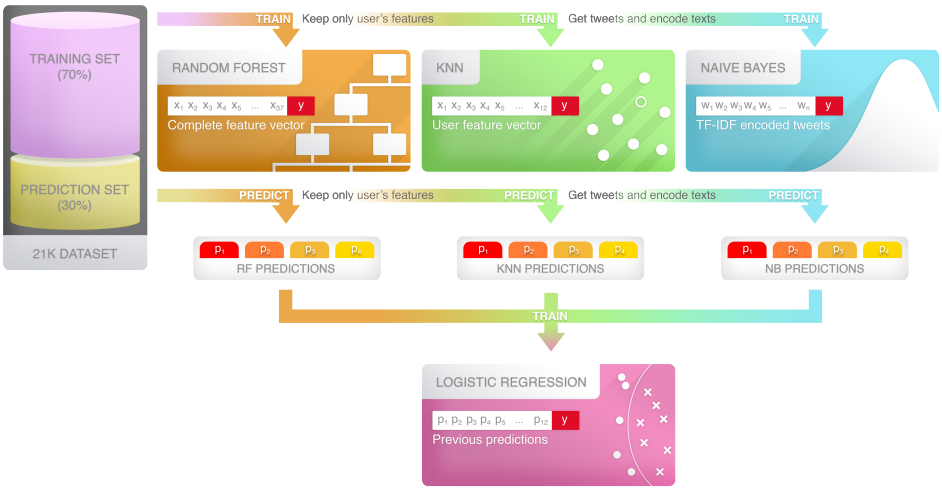

Figure 6.1: Training pipeline of the stacking ensemble

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 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
$p_0 * w_0 \dots p_3 * w_3$

+

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

+

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

=

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 multilabel 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 wanted to blindly select this model over some others tool, especially over Random Forest, which proved us to perform well in multiclass 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.

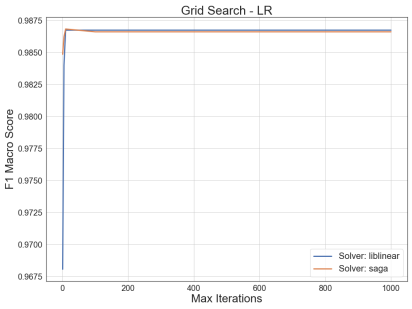

Figure 6.2: LogReg with raw settings

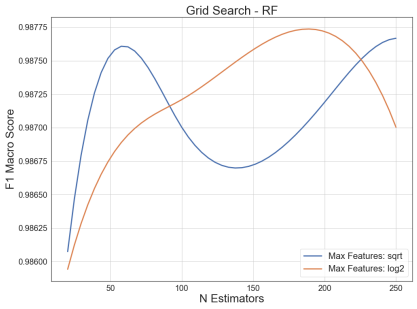

Figure 6.3: Random Forest with raw settings

Figure 6.4: Stacking models comparison

Figure 6.2 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 6.3, tops the performance of the Logistic Regression, the scores were close

enough ($F1_{LR} = 0.9868$, $F1_{RF} = 0.9874$) 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 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 6.5 highlights the slightly better results obtained with the Lasso penalty, with unitary λ coefficient (Lasso F1 score: 0.973). As Figure 6.6 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,

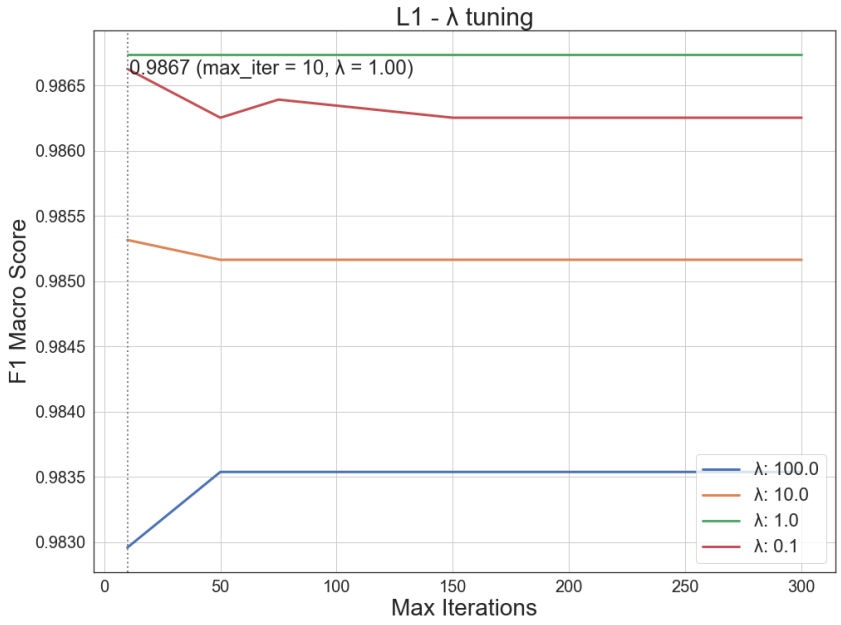

Figure 6.5: Lasso, λ = [0.1, 1, 10, 100]

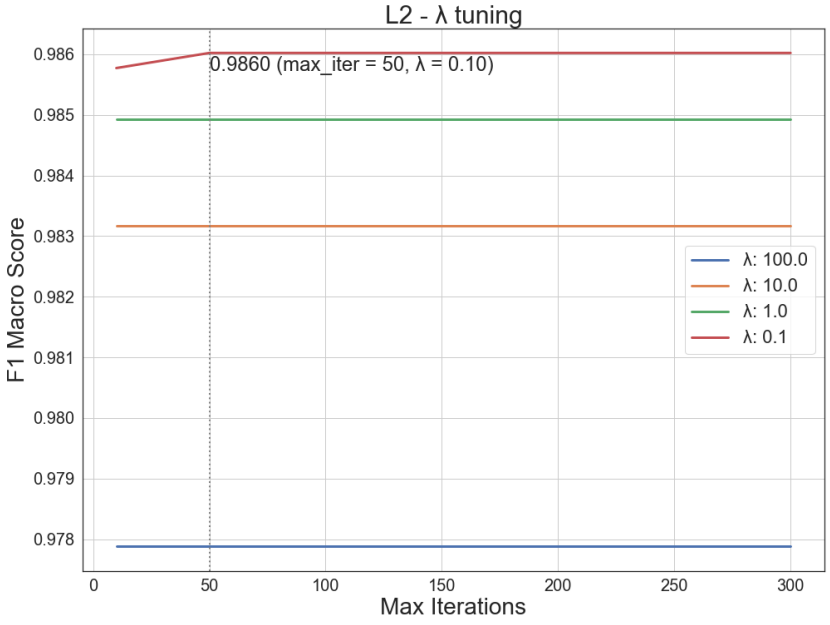

Figure 6.6: Ridge, λ = [0.1, 1, 10, 100

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 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.

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 had been performing better in out-of-box predictions.

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

Figure 6.7 shows the trend in the F1 score, along with the increasing number of iterations, applying the *SAGA* [1] solver (a variant of the *Stochastic Average Gradient* [8] optimization that supports Lasso penalty) and the *LIBLINEAR* [10], an open source library for large-scale linear classification.

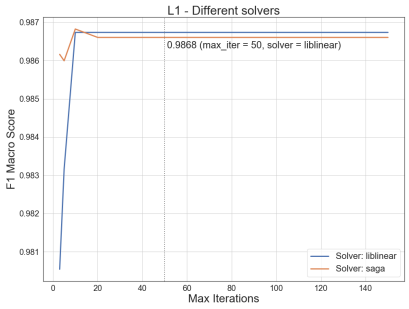

Figure 6.7: Up to 150 iterations

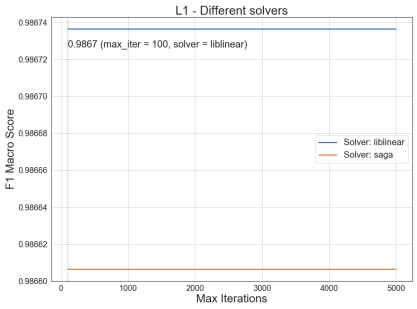

Figure 6.8: Up to 5000 iterations

Figure 6.9: Lasso Logistic Regression scores over solvers

As it can be seen in Figures 6.8, 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*: 0.986

6.2 Prediction pipeline

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

As described by figure 6.10, 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 engineering to output multiclass probability prediction (**All-features multiclass Random Forest**)

4. Strip all attributes, except for the user features, weight them with Information Gain-driven proportions to output multiclass probability prediction (**User-based multiclass KNN**)

5. Prepare and treat text to perform text-based probability prediction (**Text-based multiclass 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 multiclass 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 mutliclass stacking ensemble. The pipeline will be performed by a web application, in order to provide a useful classification tool for every internet user. The engine of this web application is mainly composed by a python script, which, given a Twitter user name, resumes this prediction pipeline and executes it, providing the classification.

This last lines anticipate the content of the following chapter of our thesis.

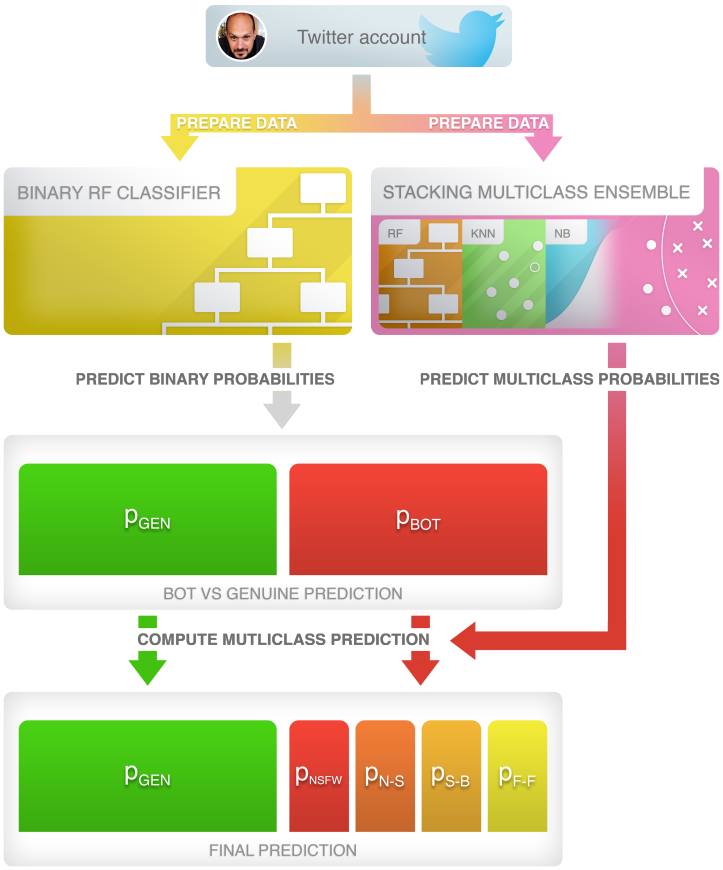

Figure 6.10: Final prediction pipeline

Chapter 7

Web application - BotBuster

This chapter represents the practical application of our work. We wanted to provide a tangible proof the thesis, an instrument available to every users in search of classification, among the accounts met on Twitter. We thought it was useful to allow people to be aware about the nature of the users that populate the social network. Bots often don't claim themselves as automated accounts, that makes hard their detection, even for 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 had to be visible on a public URL. It is currently available on *www.botbuster.it*.

Basically, the functioning of BotBuster can be resumed in getting a Twitter user name through an input filed on the home page, and then executing

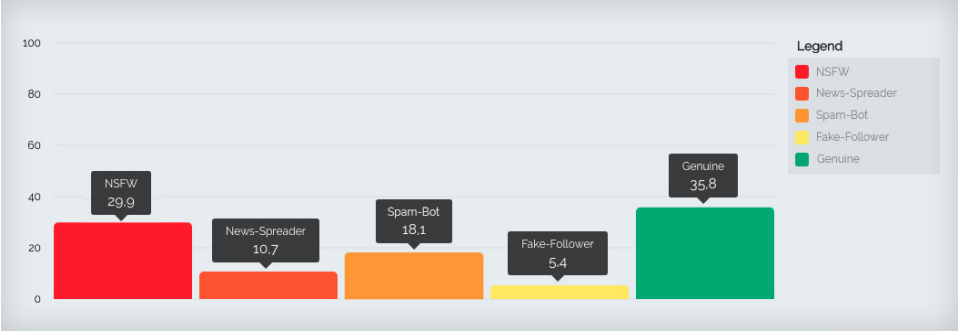

Figure 7.1: Probability classification diagram on BotBuster

Figure 7.2: BotBuster logo

the prediction pipeline described in chapter 6.2. The input filed 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 Angular program is executed on the client's web browser, while on the server runs the Flask 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 frontend and the backend applications handle the request and exchange, as shown in Figure7.3.

7.2 Backend

7.2.1 Engine

The real 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 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

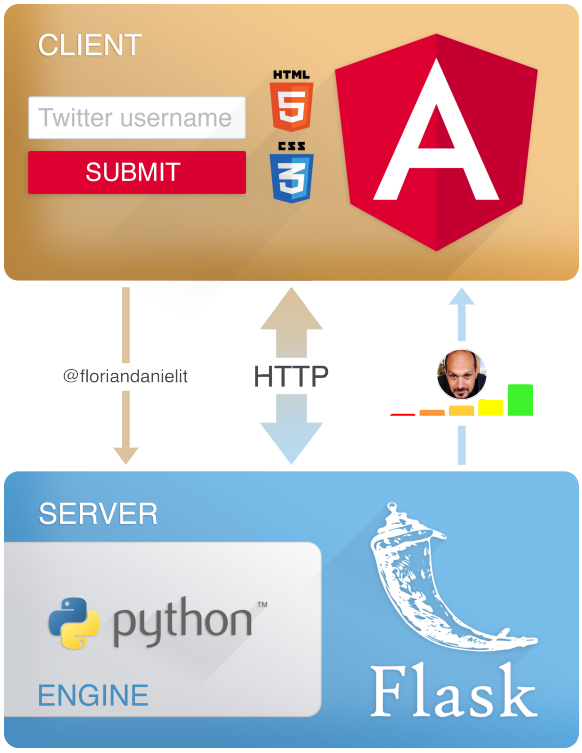

Figure 7.3: Client - Server architecture

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 backend 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 http://www.botbuster.it. The code that is being executed simply handles the rendering of the static files that compose the frontend, in order to make the website visible for the user, at that URL. The homepage appears as shown in Figure 7.4

Figure 7.4: BotBuster - homepage

- *@app.route*('**/api/user**', methods=['POST']):
  Everytime a user fills the Twitter username input filed and click 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 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 7.5.

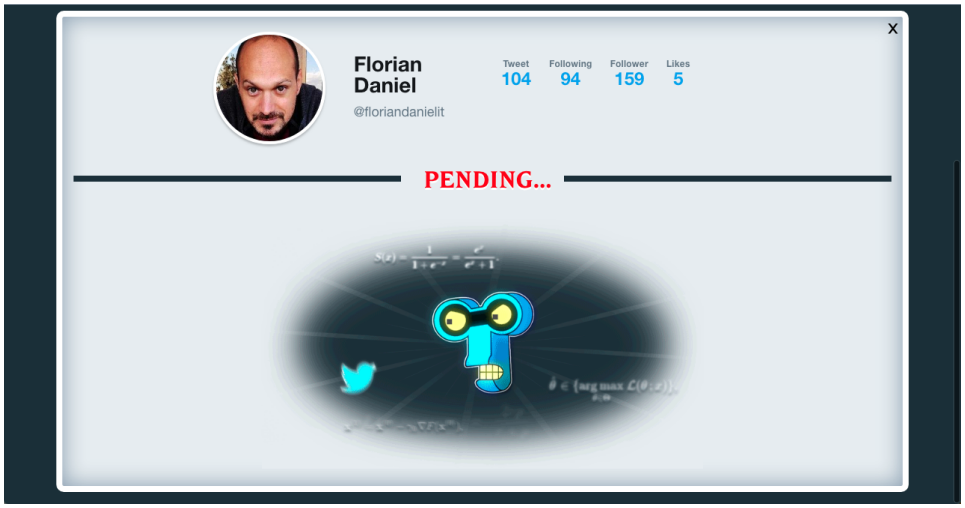

Figure 7.5: BotBuster - pending classification

✏ *@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 frontend. 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 frontend. The response provided by the classification script is represented as shown in Figure 7.6.

7.3 Frontend

The frontend application relies on Angular framework, a TypeScript-based open source, frontend web application platform, developed for the most by

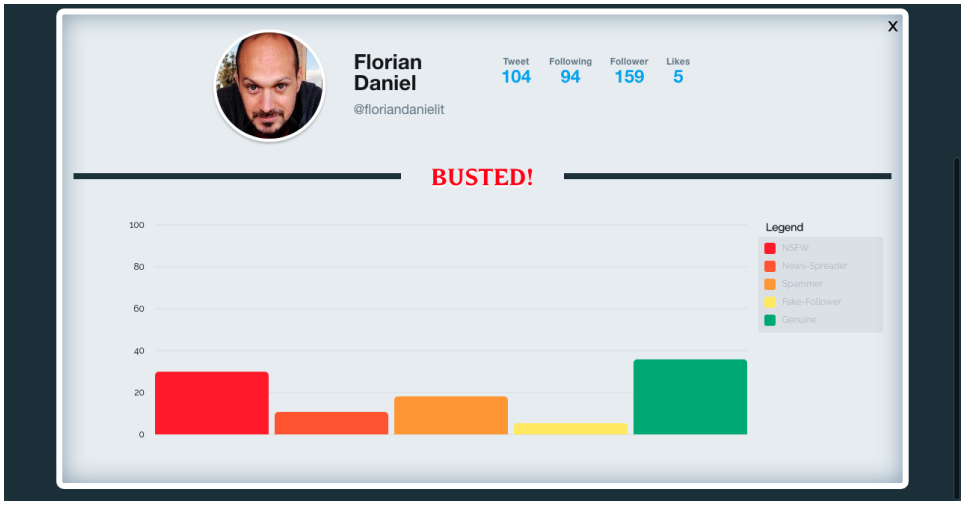

Figure 7.6: BotBuster - classification

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 theirs, as well as the similarities. The comparison in Figure 7.7 shows the differences between our methods and theirs, 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 attributes, 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

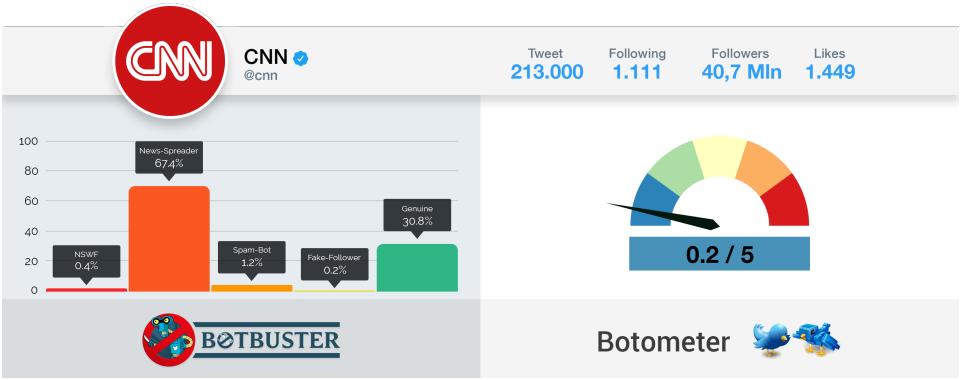

Figure 7.7: BotBuster vs Botometer - CNN's account comparison

verified feature to detect genuine accounts, since lots of the verified account tested with that tool have been evaluated as legitimate. Figure 7.8 shows this particular behaviour of the Botometer web application. All the account 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

Figure 7.8: Verified accounts classification with Botometer

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 this 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

Intrinsic features

Tweet intradistance	**URL entropy**
18.25	0.0

The News_spreader_words_score is the highest among the multiclass 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 7.9, 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.

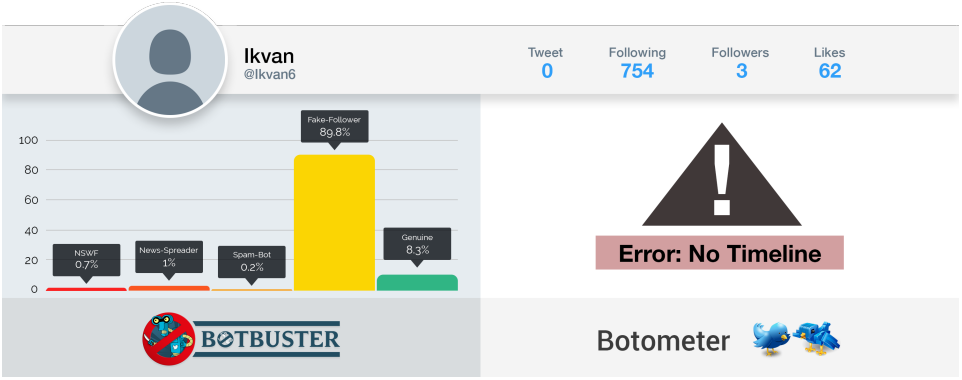

Figure 7.9: Classification of users without tweets on their timelines

In Figure **??**, 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 sees 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 that it assigns very low bot-likely values in the Sentiment field, as well as in the Network lookup. Figure 7.10 shows the partial scores assigned to 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.

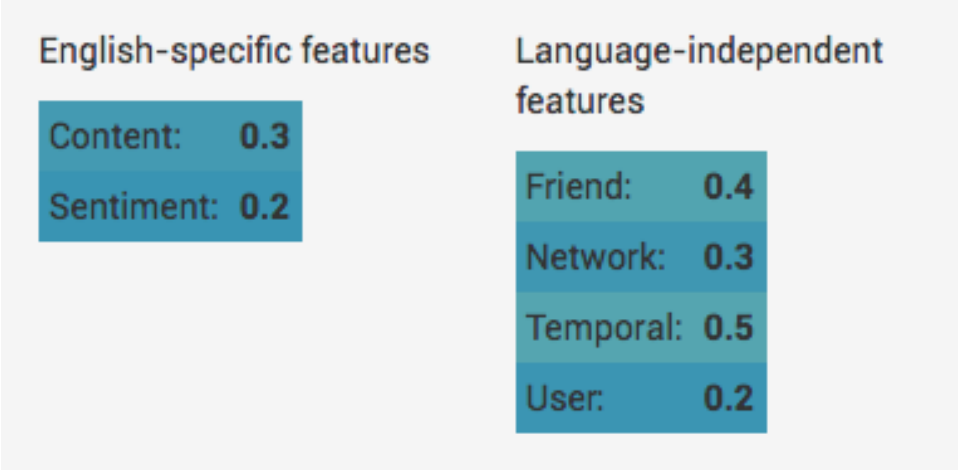

Figure 7.10: Partial Botometer scores for PoliMi account

Chapter 8

Conclusion and Future Work

So far so good. We are almost done. What is left is, well, just one of the most important chapters of the whole thesis, i.e., the conclusion. The purpose of this section is not to "conclude" the thesis in the sense to "stop" here. It's rather to draw conclusions, that is, tell how well your work actually meets the requirements identified, answers the research questions, advances the state of the art. As such, this is perhaps the most important section! It may seem easy to just summarize a bit what you did and tell again what your objectives were when starting the work. But be aware that this can be much more difficult than it sounds, and you can expect your supervisor iterating with you several times over this same chapter. It is important that you show again your personal and professional maturity and your understanding of the topic. As you will see, some healthy self-criticism too is needed to make this chapter good.

8.1 Summary and Lessons Learned

Summarize here your work in about one page.

- ☐ Start from the initial *problem statement* or *research questions*.
- ☐ Summarize your *approach* and *methodology*.
- ☐ Recap the *lessons learned*.

8.2 Outputs and Contributions

Provide an overview of the outputs your project/work produced and then state what you think are the (research) contributions that advance the state of the art.

- ☐ List all the concrete *outputs* you produced (remember the discussion in Section 1.4).
- ☐ Copy/paste here the *list of contributions* you already anticipated in Section 1.4 (attention: outputs and contributions are two different lists; don't mix them).
- ☐ For each of the contributions, provide suitable *evidence*, drawing from the body of your thesis. For instance, if you claim that you did a formal proof of something, provide the exact number or name of the proof. If you promised subjective evidence for something, link this claim to the user studies you did. Etc. One or two sentences are enough for each of the contributions.

8.3 Limitations

This is where your self-criticism is needed. By now, I am confident you did a great work with your project and the writing of your thesis. So, compliments for that! You're almost done. But let's be frank: the work is not perfect. It simply cannot be, it never is. If it is, then not only I but also the whole commission of your defense will give you a standing ovation (I really would like to see this once). But in general there are just so many aspects of a research/thesis project that one would have to control or test, and with the limited time and resources available for these kinds of final projects it is just not possible to do everything.

In this section, you therefore tell the reader which aspects of your work may limit the impact or generalizability of your findings or contributions. As said, be frank. If you tell that you did a user study with only 10 people instead of 30 (which would make the findings stronger), you don't risk to give the impression you didn't do it well enough. Actually the opposite is true: if you don't tell it, your reader, who by now will anyway have gotten that there were 10 and not 30 people involved in the study, will instead think either (i) that you *didn't know* that a higher user involvement would have been better to back your claims or (ii) that you intentionally want to *hide* information or even *cheat*. None of these are good for you, and for sure worse then telling straightaway. Keep this in mind.

Here some typical limitations of research. Check if any of them apply to your work:

- ☐ Small *sample size* (e.g., the number of users in the study or the amount of data collected in an experiment).

- ☐ For experiments that involve multiple *indipendent variables*, likely you will not have tested them all (e.g., in a crowdsourcing experiment, you fixed a reward for all experiments and did not study if that too affected your results).
- ☐ You may have *promised* something in the beginning of the thesis; if you didn't achieve everything either you drop the very promise or you mention it here as a limitation.
- ☐ When you collected data, there may have been some *bias* in the data (e.g., if you implement a prototype and do a user study yourself where there participants know that you actually implemented the software, they will give you biased answers, typically better ones).
- ☐ Collected data many have turned out being *incomplete* or of *lower quality* then initially expected. How does this impact your findings?
- ☐ Your prototype may have *crashed* or *not worked properly* in some experiments; it's important you tell the reader and explain possible implications of this on the validity of your conclusions.
- ☐ Due to time restrictions, you may have *not been able to complete* all experiments planned initially; again, explain the possible implications.
- ☐ People participating in a user study may have *dropped out* of the study, for whatever reason; if the reason is related to what you did or not did, you should mention it.
- ☐ Sometimes it is *not possible to compare* an own algorithm with other, similar algorithms, e.g., because their code is not available; this too may limit the viability of the findings.
- ☐ ...

8.4 Future Work

Finally, here you tell the reader which aspects you think would deserve further study or development. A good starting point for this is of course the list of limitations you just discussed. Not all of them may be worth investing more effort, but some will. The idea of this section is to identify where possible new effort should be invested, in order to make the work complete. Again, be frank and don't be afraid of identifying also new research directions. It's not you who will be doing what you propose here. It's meant for

the reader, the community. Everybody understands that after your defense you won't be working any longer on this project. It's all about suggesting future work, not telling that *you* will be doing it.

Bibliography

[1] LIENS MSR INRIA)-Simon Lacoste-Julien (INRIA Paris Rocquencourt LIENS MSR INRIA) Aaron Defazio, Francis Bach (INRIA Paris Rocquencourt. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. 2014.

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

User Manual

If you implemented a piece of software that is meant to be used by somebody else than you, then here you can provide a brief user manual that tells the target user how to use it. Part of this is the possible installation of the software and its operation and trouble shooting.

Appendix B

Dataset

If your work was based on a dataset that can be considered an output of the project, here you can describe it in detail.