Engenharia Informática
Final Project Report

# Minerva - Social Network Mining

**University of Aveiro**
**Departamento de Eletrónica, Telecomunicações e Informática**

Diogo Silva
Pedro Escaleira
Pedro Oliveira
Rafael Simões

2020

b

University of Aveiro

# Minerva - Social Network Mining

Authors:
Diogo Silva
Pedro Escaleira
Pedro Oliveira
Rafael Simões

Supervisor:
Diogo Gomes,
Professor Auxiliar at Departamento de Eletrónica, Telecomunicações e Informática

# Keywords

Social networks, Data mining, Information dissemination, Network graphs, Machine learning, Network analysis, Twitter, Private user accounts, Big data, Twitter user information scouring, Automated agents, Natural language comprehension and replies, Bot management, Twitter bots, Deep neural networks.

# Abstract

The paradigm of social network is paramount to the dissemination of information nowadays. Encompassing millions of users from the most diverse backgrounds, cultures and ethnicities, one such network of particular relevance is Twitter. With one of the highest user counts, an unfathomable number of tweets of all natures are shared in this platform every day. As such, it should be of no surprise that Twitter exerts a major influence in our current daily lives, be it direct or indirectly. Unfortunately, and this is a recurring problem within the web itself, there is no way to control or even analyze the way information is passed around on these platforms. There is no easy way to visualize the flow of information as it runs rampant and, due to the very nature of social networks, is shared between the deep network of users. Truly the only way to analyze these phenomena would be to personally infiltrate and navigate the network as a user. But one can not be expected to be able to dedicate all hours of a day to perusing twitter, noting how a tweet is being passed around. This is where bots come into play. Bots can mimic human behaviour, travel twitter, gain followers, generate controversy through retweets, infiltrate private account circles and most importantly, scour the flow of information and store data. Above all, they can do this all day, everyday with no hindrances.

# Acknowledgements

As our biggest, and arguably most important, project yet, and taking into consideration the context of it being the culmination of three years of hard-work and study by all of our team's members, we believe some acknowledgements should be made.

Firstly we would like to thank our parents for having been so kind as to make the sacrifices to allow us the necessary resources to pursue a higher education and for preparing us as human beings with morals and manners. Truly we are who we are due to their never ending and relentless support.

Secondly we should thank our project's main coordinator, Prof. Diogo Gomes, both for his continuous help, suggestions and resources all the way through the development process, but also for having permitted us to pursue this endeavour, showing enthusiasm for our work since the very start.

Thirdly we would like to thank Prof. Mário Antunes for being so kind as to help further our knowledge of machine learning models and artificial intelligence.

Finally we would like to thank all our friends and families, for having the patience to stick with us through all these years of stress and anxiety.

# Abbreviations

**ACID**: Atomicity, Consistency, Isolation, Durability

**API**: Application Programming Interface

**CRUD**: Creation, Read, Update, Delete

**DBMS**: Database Management System

**GUI**: Graphical User Interface

**NLC**: Natural Language Comprehension

**PDP**: Policy Decision Point

**PEP**: Policy Enforcement Point

**PWA**: Progressive Web Application

**REST**: Representational State Transfer

**UI**: User Interface

# Contents

# List of Figures

# CHAPTER 1

# Introduction

Serves this present report to go into detail about both the context, implementation and succinct discussion of the results obtained during the production of the **Minerva - Social Network Mining** project, made under the pretense of the *Projeto de Informática* class within the *Licenciatura em Engenharia Informática* course at *Universidade de Aveiro*. The knowledge exposed in this report should allow any reader to understand both what it is that our work aims to achieve, its purpose, as well as our design decisions and implementation nuances. They should also leave with an understanding of what our product can accomplish, and how they can interact with it. All in all we want any reader of the following pages to be able to perceive the current state of the project and have enough insight to be able to pursue similar goals.

Taking this into consideration we will start by, on *Chapter 2.*, discussing the current state of the art of some of our project's concepts, as well as presenting some other products that present certain similarities to our own. Following we have *Chapter 3.* which talks about the project on a more conceptual level, exposing the problem we are trying to solve, what our main users are, and so on. *Chapter 4.* is a more technical chapter that discusses our product's architecture and gives insight into what each of our utilized technologies do, and why they were chosen for integration as well as explaining the actual implementation of our work showing how each of our modules works, how they interact with each other, how our bots function and so on. In *Chapter 5.* we take to talk about the results obtained in terms of final product achieved, alongside presenting the data network our bots were capable of scouring, showcasing how our problem quickly evolved into one of *Big Data*. Afterwards we have *Chapter 6.* which aims to present our more user-oriented product. This section describes and presents our web-app, both in terms of design but also in terms of possible interactions and features. At last, *Chapter 7.* serves as a conclusion for this report and project. We will be making some final remarks on what was learned and what the overall experience was like, serving as a closure to these past months of hard labor.

# CHAPTER 2

# State of the art

In this chapter we will be presenting the state of the art of the concepts that Minerva tackles. We will start off by presenting some of the most recent technologies related to natural language comprehension, twitter bot creation and network analysis and will then move on to talking about projects that show some similarities to our own.

## 2.1 Olivia Taters Twitter Bot

**Olivia Taters** is a **Twitter Bot** created on 2013 that mimics the behaviour of a regular teenage girl on social networks like **Twitter** [17]. The way it works is by creating or replying to tweets using samples of other tweets that it had previously found. These generated responses are made in a way that makes it so it blends in with the normal flow of the twitter thread its replying to, or, in case of original tweets, are consistent with the bot's previously generated content. In figure 2.1 we have an example of a tweet published by this bot.

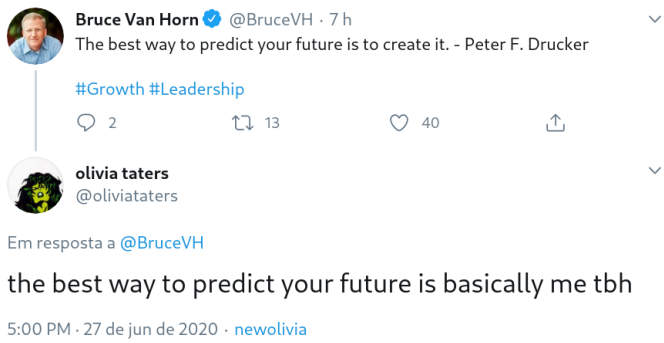


**Figure 2.1:** One of Olivia Taters tweets. Obtained from Twitter.

This type of behaviour and functioning was a good source of inspiration for our own bots' algorithms as we expect them to be able to respond to tweets in a similar way that **Olivia Taters** does, i.e, we too want our bots to be able to reply to tweets in a human-like fashion in order to be able to fool Twitter's users.

## 2.2 Bot Sentinel

**Bot Sentinel** is a public, free-use, tool that can be used to find bots, *trolls*[1] or groups manipulating conversations on Twitter in order to fight disinformation and targeted harassment. In order to do that, this service uses machine learning models to find toxic tweets. Like in our project, this platform also digs Twitter trying to find the maximum number of accounts and tweets and has the possibility to analyse and study networks of users [2]. We differ from this platform in terms of purpose, since they're main goal is to target potential sources of harassment and bots, we aim to be able to follow the flow of information and present it for further analysis.

## 2.3 Study on Twitter disinformation operations

This study was made with the goal of visualizing **disinformation operations** made on **Twitter** [19]. To achieve that, the author of the investigation made use of a public **Twitter dataset** of accounts and content associated with potential information operations [3].

To visualize and study the patterns on this data, the researcher organized it in **temporal networks**, as is seen in figure 2.2.

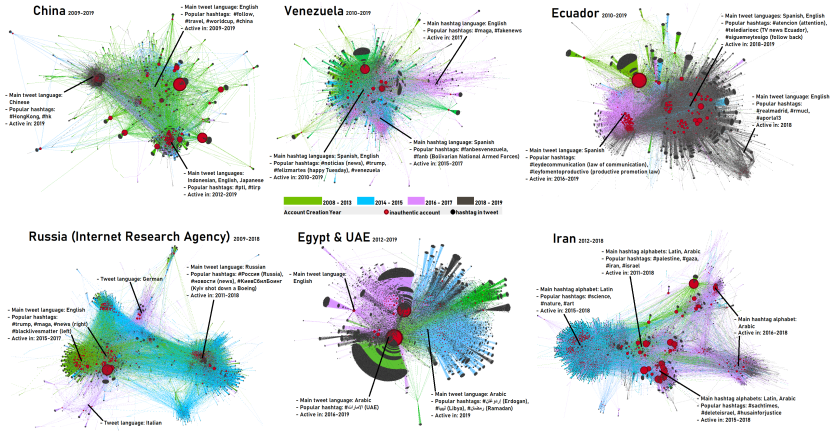


**Figure 2.2:** Temporal networks of Twitter data, organized by country. Obtained from [19].

[1] Internet slang which refers to a person who purposely starts flame wars and arguments with the purpose of upsetting a person or group of people on a given platform. They accomplish this by posting inflammatory, digressive, extraneous and, or, off-topic messages in order to stir up the community.

The way this work relates to our project is the data visualization. As in this study, our project offers a way of visualize Twitter data our bots capture using **directed graphs**, to study possible patterns on this social network.

CHAPTER 3

# Conceptual modelling

In this third chapter we will be taking a look at the main motivations that drove Minerva's development. We will be reiterating on the problem that we attempted to face, how we tried to solve it, as well as listing our system's requirements and main actors for whom our platform has been built.

## 3.1 Problem Introduction and Our Solution

Twitter, and in more broadening terms, social networks, play a major role in the way people consume both media and news in the current times, being an outlet of information accessed by millions of real users on a daily basis. Nowadays no event goes unnoticed, no drama goes unrecorded, and the dissemination of information runs at an astonishing, unfathomable and unprecedented high speed. One may look at this statement and see its positive side, people are more connected to each other, more informed about not only what happens in their local communities but in the world at large, different cultures can more easily interact with each other and more ideas and ideals can be shared. One might, however, also look at the dangerous connotations and **problems** that come with the sharing of information on the internet. Due to the nature of the web itself, fake news can be propagated, bold claims can be made with no backing and go viral, and **there is no way to trace how information is being disseminated, from whom it originated and how it was shared**. Alongside this there is also the problem of networks like Twitter allowing for the co-existence of gated sub-cultures in the form of *private/protected* accounts which normal users or analysts cannot access without being granted permission.

It would be helpful if there was a tool that allowed analysts to easily access Twitter user's information, tweets and the graphical visualization of the connected network of users and tweets in order to ascertain the flow of information through the platform, what the main information origin focuses are, and so on. But this type of information would be impossible to collect. One would have to personally use a Twitter account to navigate the social network in order to infiltrate its communities and gather user info.

Our proposed **solution** to all of these problems lies in the usage of automated intelligent agents, or **bots**. Why spend countless hours mindlessly going through Twitter when you can employ a fleet of bots to automatically and effortlessly scour

the social network. These bots are tasked with simulating a normal user's behaviour, they wander the network of connections in search of users to follow, tweets to retweet and content to like, all while storing all information gained throughout this process. All that info is then displayed in an easy to read and easy to understand format for further analysis.

## 3.2 Requirement Analysis

### 3.2.1 Target Audience and Use Cases

This project was originally conceptualized as a complementary support tool to help both professional Data Analysts, but also network investigators and students, working under the field of *Network Analysis*, a multi-paradigm field which can encompasses concepts of Social Studies, Data Science and even Physics. Figure 3.1 showcases some of our service's use cases and actors.

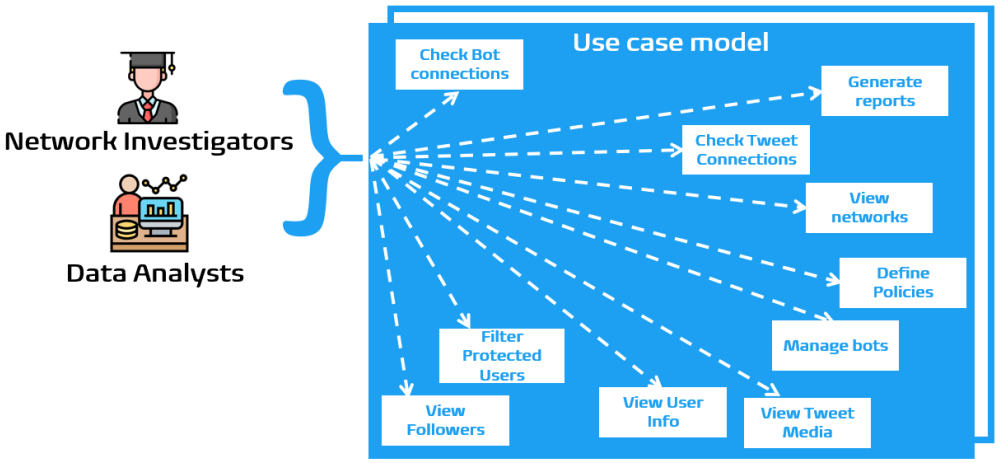


**Figure 3.1:** Our simplified Use Case diagram.

Our service is comprised of not only by a plethora of backend services and modules, but also by a user friendly frontend web application that serves as an interaction point between our core users and both the data our bots scour and the bots themselves. It should be noted that, from a use-case point of view, our users should be able to use our application to perform several actions, all in accordance to the core theme of network analysis and study.

### 3.2.2 Functional Requirements

In terms of what our system had to accomplish we started by inquiring Prof. Diogo Gomes about them. After several conversations and meetings, the following requirements were agreed upon by all parties:

- **Automatic and independent Bots capable of following users depending on content and produce retweets**.
- **Display of all information scoured by the Bots**.
- **Network Graph visualization and interaction**.
- **Graphs showcasing data growth and bot activity**.
- **Twitter User information display**.
- **Bot statistics visualization and management**.
- **Policy**[1] **Creation, Editing and Deletion**.
- **Report Generation**.
- **Network Querying using an accessible language for less tech-savvy users**.
- **Twitter User lookup and filtering by private accounts**.
- **Tweet content and media visualization**.
- **Display and allow the navigation to Twitter User's followers and friends**.
- **Automatic email system for Policy training status**.
- **Attempt to track down and follow *private/protected* accounts**.

### 3.2.3 Non Functional Requirements

Taking into account the scaleability of our system and the vast amount of data that both our databases have to store and that our modules have to process, the following backend's non functional requirements were kept in mind during the development:

- **The system should be able to handle high loads of data**.
- **The system should be able to handle high workloads**.

[1] Policies are a group of customizable keywords and tags assigned to bots to control the type of content they should focus on and attempt to follow. For example, a bot being assigned a policy with a tag like 'cars' should attempt to follow users that post content about cars. This is further expanded upon on further sections

- **The bots should showcase enough human-like behaviour to fool Twitter users into believing they're not bots.**
- **The message queue systems should be able to treat and handle a high amount of message processing and exchange between our modules.**
- **All modules should be low coupled and focused on their own individual tasks.**
- **All services should be deployed using the appropriate tools.**
- **All features should be properly tested prior to integration.**
- **Data Caching system to decrease data loading times.**
- **Message batch processing to decrease number of messages sent between each service.**
- **Bots should focus on trying to find and follow/be followed by private/protected accounts.**

It should be noted that our platform's actors and main users may not come from a background in Information Technology (IT) and, as such, we should keep in mind that they may not necessarily be the most Tech-Savvy. With this in mind, we set our web application's non functional requirements to focus, not only on speed and load performance (although these metrics are rather important), but also on its accessibility and capability to be used no matter the user's prior knowledge of network-based databases and querying languages. Our frontend should, therefore:

- **Be easy To Use.**
- **Be easy To Learn.**
- **Be highly accessible and responsive between several browsers and devices.**
- **Use *layman* terms.**
- **Offer helpful *ToolTips*.**
- **Offer a Clean and Minimalist design, devoid of unnecessary clutter.**
- **Load only the necessary/requested information.**
- **Showcase self-explanatory and clean graphs.**
- **Ease the interaction between the users and the more technical side of our backend.**

CHAPTER 4

# Procedure and implementation

In this chapter we will be discussing our technology stack and architectural choices, as well as exposing the development process that allowed for the implementation of our system. We will start off by diving into our system's architectural design, exposing its evolution throughout the course of development, as well as introducing each of the main technologies that were implemented into our product. Afterwards we will also describe each of our main modules individually in order to deepen the reader's understanding of Minerva's inner workings and how all cogs work in tandem to bring forth our final results.

## 4.1 Architectural Overview

As our project consisted in the continuation of an effort that had started a year prior we started off by analyzing the architecture that the previous developers had left us. There were some technology stack choices that we disagreed with, and as such decided to overall discard and replace. More specifically, we replaced **Flask** with **Django** as our middleman between our *frontend* and the remainder of our *backend* modules. This was done in order to allow us to create a more complex *REST API* as this framework offered us more flexibility, both in terms of features and due to prior experiences with the tool. Figure 4.1 shows our initial architectural design after these changes.

However, with the progress of the project, we saw ourselves compelled to change and add more elements to the architecture due to new features implemented, overall data growth, machine learning improvements, between other factors. Our final solution for the project's architecture is presented in figure 4.2. The remainder of this section will be utilized to introduce, and talk in more detail, about the major technologies present in our architecture.

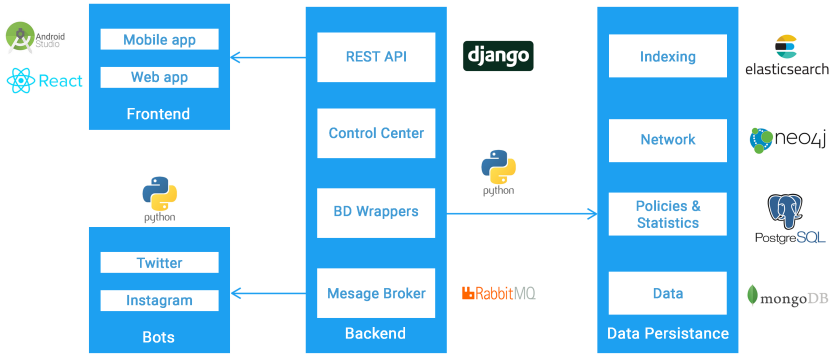


**Figure 4.1:** Initial system architecture. It differs from the work's prior edition by having replaced Flask for Django, and by having added Mobile App a module made in Android Studio.

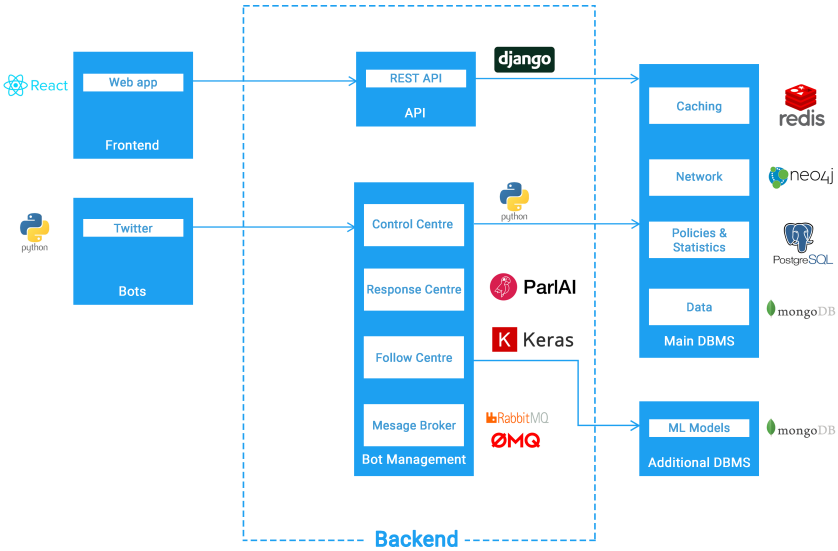


**Figure 4.2:** Final system architecture. New subsystems were integrated, the Mobile app module was removed, new technologies were added alongside new data stores and caching.

### 4.1.1 ReactJS

Starting off with our user's frontend, this module has been built from the ground up using the **React.JS** *javascript* framework [23]. This framework, originally built by Facebook, has been at the forefront of frontend development ever since its inception, beating out competitors such as *Angular.JS* and *Vue.JS* in terms of market usage and adoption. Due to how highly used it is, a lot of documentations has been made available for it, alongside several templates, tutorials and customize, almost plug-and-play modules for just about any needs a frontend developer might have. Indeed modularity is one of the key concepts of React, which, alongside its usage of the **npm** package manager (*Node.JS* Package Manager) and its other key feature of reusability of components allows for the fast conceptualization and creation of responsive web applications. This, alongside with our team's developers past experiences, lead to this choice being an easy one to make. We should also point out some of the major modules that were utilized in conjunction with React such as **Recharts** [24], used to plot and build our data graphs and **Vis** (or more precisely, Uber-Vis Force's adaptation of the Vis javascript library for React) [31] [32], used to construct our physics-enabled network graph visualization

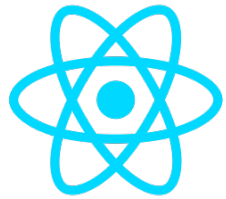

**Figure 4.3:** The React Framework logo.

Looking back at figures 4.1 and 4.2 one might notice that some rather important changes were made on the frontend block, noticeably, the removal of the mobile application. Indeed initially, and although it wasn't a particular requirement of the system, we sought out to create a companion mobile app that would allow for the management of the bots on-the-go. At first we wanted to build it using **Android Studio** as it was the only tool of mobile development our team was comfortable with. Over the course of development, however, we chose to replace it with **React Native** as this would allow us to build an app that would be transient over multiple smart devices, either running *iOS* or *Android* operative systems. Unfortunately, and due to time constraints, we decided that, since this was never a core requirement, it would be best to drop this feature in order to allow development focus to be put towards either finishing implementing or polishing other more vital components of our system.

### 4.1.2 Django

**Django** is a high-level python web framework that encourages rapid development and clean, pragmatic design. This framework has been growing in popularity and is used by some high profile sites like Instagram, Mozilla,National Geographic, just to name a few [4]. The main core of this framework is the fact the developers can focus on writing their app without reinventing the wheel by offering a variety of abstractions to make developer's life easier and helping with writing secure, safe, scalable and maintainable software. We chose this tool since some group members had already worked with Django, henceforth decreasing the development time. In addition to this Django provides easy external framework integration like cache mechanism and database services, excellent documentation, and helpful abstractions of database queries, all of which were useful for our endeavours.


**Figure 4.4:** The Django logo.

Although we had to re-made all features implemented in a previous version of this project (in the past REST-API was made using Flask), the time wasted on the first phase of re-build was compensated by the time that we save to implement new features since Django abstractions make development easy and maintainable.

### 4.1.3 ParlAI

**ParlAI** is an open source framework, written in *Python*, maintained by the *Facebook Research* team, which can be used to share, train and test artificial intelligence dialogue models [18].

We chose this tool because we needed a way to respond to interactions with real humans, i.e. we needed our bots to be able to reply to other user's tweets and retweets. At an initial stage we used an *ELIZA* style dialog simulation [6], but soon we came to the conclusion that the responses we were generating were too simple for our scenarios, with the main grievance being that our *ELIZA* bots were producing a lot of similar responses in a short period of time, which would arouse Twitter user's suspicions, hence running the risk of being flagged as bots. This fact, mixed in with how poor and simple the generated replies lead to our bots not being able to get nearly as much likes, retweets and controversy as we had hoped for. That was when we decided to swap over to **ParlAI**.

**Figure 4.5:** The ParlAI logo..

We chose this tool since it allowed us to utilize a pre-trained poly-encoder model [10], feeding it tweets we had been capturing and storing over time. By doing this, we can then give our model any input text and it will be able to select and transform one of the tweets we had previously fed it in order to generate a befitting response [7]. To obtain the responses from this tool, we also used ParlAI's **Remote Module**. This module allows for the creation of a simple server service which responds to interactions with the text selected by the described machine learning model.

### 4.1.4 Keras

**Keras** is one of the leading high-level neural networks APIs. It is written in Python and supports multiple back-end neural network computation engines. This framework was created to be user friendly, modular, and easy to extend. All types of functions and classes needed for Deep/Machine learning models are offered by this tool as a standalone module that you can combine [11]. We chose to work with this tool because of its user-friendly API, ease of learning, and ease of model building. Besides, it offers the advantages of broad adoption and integration with popular back-end engines such as Tensorflow and CNTK. To prove the quality of this tool, Keras is backed by some giant companies like Google, Microsoft, Amazon, Uber, and so on.


**Figure 4.6:** The Keras logo.

Our group had to use this tool to implement a Deep Learning algorithm that is able to classifies text to help our bots follow content that they should defend (defined by policies). More on this in further sections.

### 4.1.5 RabbitMQ

One of the two messaging systems we utilized to allow our multiple components and microservices to communicate with each other, **RabbitMQ** [22], is a lightweight message broker that can be used in a way such that the service receiving the messages, also known as the *consumer*, isn't forced to immediately consume the messages sent by other services, the *producers*. This allows our system's several modules to have **low coupling** and **high independence** communication-wise. Another important feature of **RabbitMQ** is that the message's delivery order is maintained as a queue making message trading and processing more easy to understand and implement.

**Figure 4.7:** The RabbitMQ logo.

We used **RabbitMQ** to make the *Control Centre* communicate with both the *Follow Centre* and our **Bots**. In these communications, all the modules were both producers of some *exchanges*[1] and consumers of other *exchanges*.

### 4.1.6 ZeroMQ

The other communication method we used in our project was **ZeroMQ** [34]. This *multi-language*, *lightweight* library provides *sockets* that support several transport types, such as *TCP* or *multicast*. It also supports multiple *message patterns* like *publisher/subscriber* and **request/reply**. All these features make this tool useful for many use cases.

**Figure 4.8:** The ZeroMQ logo.

More specifically, in Minerva this tool was used because **ParlAI**'s **Remote Module** implemented a **ZeroMQ** *reply server*. So, to decrease the complexity of the sys-

[1] **RabbitMQ Exchanges** are the **message routing agents**

tem, we made the *Control Centre* communicate with this service in particular using this messaging library.

### 4.1.7 Redis

**Redis** is an *open-source in-memory* data structure store used as a *key-value database*, *cache*, and *message broker* with support for different kinds of abstract data structures, such as string, list, maps, amongst others [25]. In addition to this, it is also possible to manage cached data through a programmatic *API* (Application Programming Interface), available for most popular programming languages. **Redis** looks rather impressive when you look at how much its position against other market leaders has improved over the last few years having become the de-facto, most popular key-value databases. Due to its quality, it is used by some very well-known companies like Twitter, *Github*, *SnapChat*, *StackOverflow*, just to name a few.

**Figure 4.9:** The Redis logo.

This tool is used on this project to store some costly database queries to speed up load times and to store some information about bots activities to prevent task repetitions.

### 4.1.8 Neo4j

One crucial component to our project is to keep track of the relations between users and tweets our bots find while scouring Twitter[15]. We have to keep track, not just of who follows whom, but also see who may propagate a user's tweets, via retweet or reply or anything along those lines. So we needed a database that could generate complex graphs and networks efficiently and without too much cost to the server.

To achieve these levels of performance, we have been using Neo4j, a graph-based database that's primarily focused on *Nodes* and the *Relations* they form between them. Due to its popularity, the database has drivers for a lot of different languages, including Python, facilitating the development of a wrapper that would insert the information found by the bots.

**Figure 4.10:** The Neo4j logo.

Neo4j also supports its own query language, **Cypher**. Since every connection between nodes is being stored, and not computed at the time of querying, Neo4j developed its own language to optimize the search and filter by these stored relations. It has a very simple and straightforward syntax that makes it easy to learn and understand, almost intuitive even for people who aren't used to SQL or NoSQL.

### 4.1.9 PostgreSQL

Due to our project's nature, we also needed to gather statistics on the users and keep track of our bots' activities [21]. In order to do this, we had to save all this information in a relational database, in order to ensure a decent query processing performance and ease of implementation of *aggregations functions*[2].

For these purposes, we chose **PostgreSQL**, an open source relational database system capable of running on any OS (Operative System) and with a plethora of drivers for a lot of different programming languages. Due to its high market usage, integrating it with Python was very easy, both due to the available drivers and available documentation. Another benefit that **PostgreSQL** brings to the table is the fact that it's *ACID* compliant [3] and highly scalable, which made it favorable for the problems of *Big Data* we're dealing with in this project.


**Figure 4.11:** The PostgreSQL logo.

[2] In Databases, an aggregate, or aggregation function is a function in which the values of multiple rows are grouped together to form a single value.

[3] ACID, which stands for Atomicity, Consistency, Isolation, Durability, is a set of properties certain database management systems guarantee intended to ensure data validity and integrity even in the event of errors, power failures, and other unexpected occurrences

While it's easy to use, as it employs an SQL-like language very similar to other database systems, **PostgreSQL**, unfortunately, doesn't have direct support for *Timestamp* data types, which are incredibly important considering statistics and logging of activities. To fix this we used the extension **TimescaleDB** which allowed our **PostgreSQL** database to perform time-oriented operations efficiently [28].

### 4.1.10 MongoDB

To save all the data that we are receiving from the bots, i.e. the complete Tweet and User object, we use **MongoDB**[14]. Being a document-based database, it is very easy to use and flexible in what we need to save, bypassing relational databases' restrictions like columns and table; **MongoDB** will simply let us save any document under a certain collection of documents, with no intrinsic restrictions.


**Figure 4.12:** The MongoDB logo.

Due to the lack of restrictions, we're able to save anything as a document in **MongoDB**, which meant not only the objects we receive from twitter, but also machine learning modules that we train for the other services. It's also very easy to use with Python, has it has a driver for the language, making the interaction between our modules and the database very intuitive and straightforward.

## 4.2 Bot Modules

### 4.2.1 Bots

With our project being all about **social network mining**, it shouldn't be of surprise that our **Bots** are at the core if our entire system. The bots we deployed were made using a *Python* library, **Tweepy** [29], that wraps **Twitter's API** [30] for developers, allowing us to programatically interact with the platform.

Each deployed bot is linked to a pre-made *twitter account* from which we extract the *authorization tokens* Twitter provides for developers [16]. With this authorization, we get enough access to do almost anything a normal user might do on the platform but through coding. As such, our bots are capable of recreating almost any *Twitter*

*interaction* there is. More concretely, the interactions we have integrated into our bots' behaviours are:

- **Follow specific users**.
- **Like specific tweets**.
- **Produce retweets to specific tweets**.
- **Reply to a specific tweet**.
- **Get a specific user's followers**.
- **Get a specific user's friends** (i.e, the users they follow).
- **Get a specific user's timeline** (i.e their latest actions, such as tweets and retweets).
- **Get a specific user's profile information**.
- **Get a specific tweet's information** (i.e who posted it, at what time, it's media content, and so on).

It is important to note that all of the *Get ...* interactions specified above can be automatically made by our bots, but all of the other interactions, which are said to *"change the state of the bot"* (for example, following a new user technically changes that bot's follower count, hence changing it's state, meanwhile simply getting a user's information produces no changes on the bot specifically), are only allowed to be made with the **Control Centre**'s endorsement. However, for the **Control Centre** to decide if the bot can follow through with any of these interactions, that bot has to produce a specific request and send it over to the **Control Centre**. For example, if some bot discovers a *new user that it does not follow*, it will ask the **Control Centre** whether it's allowed, or not, to *follow that user*. Then, the **Control Centre** will make a decision and if it accepts that interaction, he responds to the bot with a *green flag*, at which point the bot will then attempt to carry on said interaction.

As described on the section 4.1.5, all the communications between each **Bot** and the **Control Centre** are made using the **RabbitMQ** message broker. This communication uses two different **RabbitMQ Queues**:

- **Queue *API*** - this queue is where each bot publishes the messages it wants to send to the **Control Centre**.
- **Queue *bot-<bot id>***[4] - this queue is where the **Control Centre** publish its response to the **Bot** with id *bot id*. Then, the correspondent bot consumes this messages and makes the requested interactions.

[4]The *bot id* is the **Unique Identifier** (ID) Twitter attributes to each user. If, for example, the bot has $id = 123$, the **queue name** will be ***bot-123***.

All the messages that each bot publishes to the *message broker* are sent in blocks, or batches, containing multiple messages. For example, if some bot wants to send a *request to follow some user*, it does not send this message right away, but instead, aggregates this message with multiple others that it wants to send and, when the **bulk message** reaches some specified size, they get sent at the same time in the form of a single message. This is important to prevent the clogging of the *message queue* with a lot of messages alongside with improving the system's overall performance (for example, it's much faster to receive and acknowledge just a message instead of all messages it would be sent if we weren't using bulk messages).

Besides the actions associated with Twitter and Control Centre interactions, the bots also have some other types of behavior:

- They stop working time to time, for some fixed period, to mimic better how a human interacts with social networks.

- They do a new setup time to time, usually daily, to send to the Control Centre a updated list of the users who are following it and who are friend with it. This is important because sometimes users change their personal information over time, or make their account public or private, and the bot is not notified of this events.

### 4.2.2 Control Centre

The control centre is the main way for our bots to communicate with the other entities in the project. The bots themselves, by design, were purposely left as lightweight as possible: they can't interact with the databases, they can't make important decisions by themselves, and they can't process what they see. This is done so that the bot programs are kept as simple and as focused as possible on the tasks of scouring Twitter, with all actual data processing and decision making responsibilities being passed down to other modules.

That being said, whenever the bot has a new task to process, it will send it to our **Control Centre**. This is the entity that will communicate with the rest of the services in our project as the bots need them to. So, if the bot has found a new group of users, the control center will save or update all that information in all the databases, communicating with the database wrappers and sending them the proper objects. If the bot has requested to like or retweet a tweet, the control center will forward the request to the *PDP* service, who will decide if the bot should do it or not. If the bot has requested to follow a user, it will forward the request to the Follow Centre, who will analyse the user profile and give a proper response, i.e. if the bot should or shouldn't follow the user. Finally, if the bot has requested to reply to a tweet, it will go to the PDP service, again to know if the bot should do it, and then, if the response is positive, the text will be forwarded to the Response Centre, who will generate a response.

In this way, the control centre basically serves as the central entity, in the way that all requests will pass through the component so that they will be properly forwarded to the right components. That being said, there needs to be an important focus around efficiency and low coupling, as it can't take too long on processing each request, as it might result in too many messages being left on the queue to be processed; and it must be low coupled so as to ensure that launching a new Control Center won't cause troubles to the rest of the system.

For maximum efficiency, we have adopted a **bulk messaging approach**: the bots and control centre won't communicate with just one message, rather the bot will send a group of requests for the control centre to process, who will send back a list of messages and answers for the bot. This way, we're optimizing not just by the amount of messages in the queue of each entity, but also in the way we're introducing data to the databases, as this allows to do bulk insertions in the databases, optimizing the process.

Also, to boost up the efficiency, we developed the Control Centre using a concurrent computing approach. This was reached through the **asyncio** python library , which allows programmers to write concurrent code in a easy way [1]. With this, the Control Centre is capable of receiving and process three messages from the bots at the same time, making advantage of the machine's processor multiple cores.

To make sure that the Control Centre remains low coupled, all **dependencies** in the system are **well divided**, so that the entity doesn't do what he is not responsible for; the Control Centre himself won't save data in the databases, won't make any decisions, and will only forward the requests to the respective entity. This makes it possible to have more than one entity active at the same time.

### 4.2.3 Response Centre

This module was the unique part of this project we didn't need to do additional code, since as described on the section 4.1.3, this tool already brings a simple **response server** out of the box implementation. The only things we had to do where choose a machine learning model, feed him the tweets we already had collect during the project development and connect the **Control Centre** to this service using **ZeroMQ** (obviously, for this to happen, we also had to deploy the **ParlAI service**, as we will describe later on this report).

This module, which is in charge of generating an appropriate response to any given input text (i.e, it gets fed a tweet and generates an appropriate retweet, as seen in the figure 4.13) has been implemented using **ParlAI**, as described in section 4.1.3. As this tool already comes with its own *response server*, as an *out-of-the-box* module, all we had to do was pick the appropriate machine learning module, feed it tweets that we had been collecting throughout the course of the project and connect it to our **Control Centre** using **ZeroMQ**. Additionally we also had to deploy our **ParlAI** service, but this will be talked to more in length later on in this report.

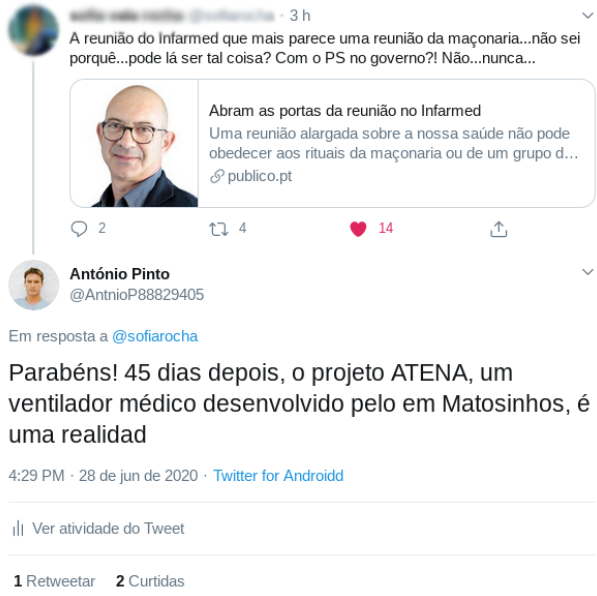


**Figure 4.13:** Example of a tweet reply made by one of our test bots.

### 4.2.4 Follow Centre

One of our bots possible core actions is the one of attempting to **follow a user**. Following users is at the cern of our whole scouring process as, after the follow request is accepted by the targeted user, the network of the bot that sent the request will increase, augmenting the data we have stored and increasing the possibilities of finding more users (public or, more importantly, protected) which may be part of the targeted user's followers and or friends network.

Each bot has a range of **policies** to establish their behavior defining what type of content it should be interested in and try to find more of. For example, a bot with a policy that defines tags correlated to a certain political party will try to scour twitter for users tweeting and talking about said political party, it will try to create retweets to tweets that talk about the content defined on the policy's tags and it will try to follow users who seem to be invested in publishing that type of content. With that being said, it is paramount for us to have a mechanism capable of correctly discriminating the content that bots must or mustn't follow so that its interactions are consistent.

In order to correctly classify texts in accordance to their content, we used a **deep learning algorithm** to categorize a given input text and then compare the topic obtained with the policies that the bot should follow. If the content is in line with any of the bot's policies, it'll then be interested in it, either retweeting, liking or attempting

to follow users related to the tweet the text originated from. For our algorithm to work correctly it was necessary to realize text pre-processing in order to convert the "raw" alphanumerical text into a set of numerical vectors. As expected, a model capable of categorizing texts is not enough in the context of this bot's interaction, so a micro-service (integrated with an email service) was implemented with the following responsibilities:

- **Train a Deep Learning model for each policy:**
  - There is a model per policy capable of transmitting the confidence rate of a text about the topic on which it was trained
  - For the training process, the policies' keywords are considered to perform a tweet search by topic (using search API from Twitter) of the last 7 days
- **Identify policy changes:**
  - If there is a change in policies' keywords the service must be able to re-train the model taking into account the new keywords
  - If a new policy is added, a new model training process should be triggered
  - If a policy is removed, the model must be erased
- **Decide whether a user should be followed or not:**
  - For the decision of following a user, the descriptions and, if the account is not protected, the most recent tweets are used as input
  - The bot's policies models are used to make predictions using the data collected in the previous step
    - If the confidence rate is higher than the defined threshold then the bot must follow the user
    - Otherwise should not follow
- **Notify Minerva's users about events:**
  - It is expected that policies will be added by our platform's users, so when this happens a new training process is started, however, this process can be time-consuming, and as such, we decided to implement an email system that would notify our users:
    - Confirming that the training process has started successfully, sent after a new policy is added or an old one is edited
    - When the training process for a given policy is finished

These features have not been implemented in the control center to avoid adding code complexity and most importantly due to performance reasons because the training process is very expensive which could cause a very high workload in the control center that would impact negatively our project.

Thus, the exchange of messages made to follow a user is the following:

- Bot asks to follow a user to control center, sending a message with the description and if available the most recent tweets
- The control center receives the message, adds extra information about bot's policies and forwards the message to the Follow service
- The Follow service takes texts previously provided and categorizes them, sending a message with the decision to follow or not a user
- The control center processes the message and forwards the respective decision to the bot

Although the decisions to follow a user are made by an external service, the bots do not realize that this decision is not made by the control center, and so, it was not necessary to change bot's code to implement this feature.

### 4.2.5 Messaging Systems

As we described in sections 4.1.5 and 4.1.6, we used **RabbitMQ** and **ZeroMQ** to implement all communications between our main **Control Centre** unit and all other modules it connects to i.e., to the **Response Centre**, **Follow Centre** and **Bots**. The importance of our messaging systems should not go understated as, to allow our system to be as low coupled and expandable as possible we decided to separate our code into multiple sub-services (kind of like what one might expect from a microservice architecture). Each of our centres is kept as lightweight as possible by being in charge of their own individual tasks and as such, for the whole system to work, appropriate communication and message handling between these services had to be implemented.

The message types sent over the **RabbitMQ message broker** are the following:

- From Control Centre to Bots:
  - **FOLLOW USERS** - used by the Control Centre to request Bots to follow some user.
  - **LIKE TWEETS** - used by the Control Centre to request Bots to like some of tweet.
  - **RETWEET TWEETS** - used by the Control Centre to request Bots to retweet some tweet.
  - **POST TWEET** - used by the Control Centre to request Bots to post some tweet.
  - **FIND FOLLOWERS** - used by the Control Centre to request Bots to find the followers of some user.
  - **KEYWORDS** - used by the Control Centre to request Bots search for tweets by keywords.

  - **GET TWEET BY ID** - used by the Control Centre to request Bots to search for some tweet by its id.
  - **GET USER BY ID** - used by the Control Centre to request Bots to search for some user by its id.
  - **FOLLOW FIRST TIME USERS** - used by the Control Centre to request Bots follow some list of users. This is sent when the Bot launches for the first time.

- From Bots to Control Centre:
  - **EVENT USER FOLLOWED** - used by the Bots to signal the Control Centre they followed someone.
  - **EVENT TWEET LIKED** - used by the Bots to signal the Control Centre they liked some tweet.
  - **EVENT TWEET RETWEETED** - used by the Bots to signal the Control Centre they retweeted some tweet.
  - **EVENT TWEET REPLIED** - used by the Bots to signal the Control Centre they replied to some tweet.
  - **QUERY TWEET LIKE** - used by the Bots to ask the Control Centre to like some tweet.
  - **QUERY TWEET RETWEET** - used by the Bots to ask the Control Centre to retweet some tweet.
  - **QUERY TWEET REPLY** - used by the Bots to ask the Control Centre to reply to some tweet.
  - **QUERY FOLLOW USER** - used by the Bots to ask the Control Centre to follow some user.
  - **SAVE USER** - used by the Bots to send the Control Centre a new user they have found.
  - **SAVE TWEET** - used by the Bots to send the Control Centre a new tweet they have found.
  - **SAVE FOLLOWERS** - used by the Bots to send the Control Centre the followers of some user.
  - **QUERY KEYWORDS** - used by the Bots to ask the Control Centre for keywords to search for tweets.
  - **EVENT ERROR** - used by the Bots to signal the Control Centre that some error happened during the execution.

- From Control Centre to Follow Centre:
  - **POLICIES KEYWORDS** - used by the Control Centre to send to the Follow Centre a new set of keywords associated to a Bot to train a new machine learning model with tweets having this keywords.

  - **REQUEST FOLLOW USER** - used by the Control Centre to ask the Follow Centre if some Bot can follow some user.

- From Follow Centre to Control Centre:
  - **REQUEST POLICIES** - used by the Follow Center, on the first connection, to the Control Center to ask for information about the policies to verify that all models are updated with the policy data.
  - **FOLLOW USER** - used by the Follow Centre to send feedback about the follow user request to Control Centre
  - **CHANGE EMAIL STATUS** - To change the notification status of an email to false, i.e this user's email doesn't need to be notified anymore because all emails had already been sents

## 4.3 Database Management Systems

### 4.3.1 Main User Data

The main focus of the databases is to save the objects that come from Twitter, i.e. users and tweets. As aforementioned, we have different database systems in our project, each with a specific function and in charge of saving different parts of the objects the *Tweepy API* returns.

In **MongoDB**, we have two collections: a **users** collection where we store our *User objects*, the information saved in the Tweepy API for the users; and the **tweets** collection where we save our *Status objects*, the object that represents Tweets in the Tweepy API. Since this database is *document-based*[5], we don't have any particular logic when inserting to the document, meaning we only insert the raw results that come directly from the Twitter API.

The same thing cannot be said about **Neo4J**. In this *graph-database*, all we really care about is how the different objects are **related** to each other: who follows whom, who wrote what, who replied to what, who propagated what. Keeping track of these relations is what will create the complex networks present in the social platform. However, saving the complete objects like we do in MongoDB is a complete waste of space, as we would be quite literally just duplicating data that we have already stored on our other databases. So, in this database, we have three types of nodes:

- Our **Bots**, where we save the Id, the screen name, and the username;
- The **users** our bots find, again saving the Id, the screen name, the username and a boolean value to know if it's protected or not;
- The **tweets** our bots find, only saving the Id, anything else would be pointless

[5]A document-oriented database, or document store, is a type of databases used for storing, retrieving and managing data stored in document-oriented formats.

Now, to keep track of the relations between these entities, we keep track of the following relations:

- **Follows** between two users or a bot and a user;
- **Wrote** between a tweet and its author;
- **Retweeted** between a tweet and who retweeted it;
- **Quoted** between the original tweet and the tweet who quotes it;
- **Replied** between a tweet and the tweet its replying to;

These relations helps us see how a tweet will be propagated in Twitter, from who wrote it, to everyone who has interacted with it.

Finally, on **PostgreSQL**, we are saving statistical data and the logs for our bots' activities. We save the amount of likes and retweets a status has, and the amount of follows and friends a user has, as well as a timestamp to identify when the row was inserted. This makes it possible to follow the progression of a user or a tweet throughout the a certain time period. We also insert the different logs to keep track of what our bots have been doing: what they requested, what were the responses, and general actions they have been doing.

To recap, we first use **Neo4j** to store the overall structure of our network and the different types of relations between our three main entities, users, tweets and bots. Each of our entities is represented in this database in the form of a node with an Id that identifies a certain document in our **MongoDB** collections. On those documents we store the actual information of our entities, to prevent overloading the Neo4J database with too much information. We then use **PostgreSQL** to store statistical data and logs that we then present in the form of graphs in our web application.

### 4.3.2 Indexation

Whether it be requests that come from the REST API, or internal logic in any of the algorithms on our modules, they may need to query the databases constantly. In order to ensure that our algorithms perform at adequate speeds and our workload is executed with decent performance, we require our queries to be efficient, decreasing the time it takes to get the high amount of information stored in our databases. For that, we have implemented indexes.

In the objects that we are storing the databases, there are values that we can trust to be unique, namely the Id's and the users' screen name. So in our MongoDB and our Neo4J databases, we created the index on these values, so that queries get maximum efficiency.

With this in mind, we have implemented the following indexes:

- **User ID** on Mongo, both in integer form and in string form

- **User's screen name** on Mongo
- **Tweet's ID** on Mongo, both in integer form and in string form
- **User's string ID** on Neo4J
- **User's username** on Neo4J
- **Tweet's string ID** on Neo4j

### 4.3.3 Database Wrappers

Each of our main databases have wrappers written in Python that integrate the commands needed to communicate with the databases and perform the CRUD operations[6].

This was done so that, to the Control Centre, it simply thinks that it's inserting and searching the databases via a single function, not having to worry about what exactly needs to be done for it to work, e.g. it doesn't need to worry about building the query itself to search the database, it doesn't need to worry about throwing the proper exceptions or writing to the right logs. All of this is properly handled through our wrappers, who will call the proper functions supported by the respective driver.

It's worth noting that the MongoDB wrapper actually performs **bulk insertion** instead of simply storing data row by row. This is not seen in the Control Centre, as it will just call the save function of each of the wrappers. This save function, however, is actually just **inserting or updating** the object in a **local list** saved under the wrapper. This data will only enter the database once the Control Centre finishes processing all requests from the bot, at which point the Control Centre calls for the wrapper to officially start the bulk insertion. This operation is actually very **time consuming**, but doesn't stop the progression of the program, so it may happen that the same object can be in two different bulks, and the wrapper may attempt to save the same object twice. This will of course cause an error.

Our way to fix this bug was simply add a small **cache** in Redis to our control centre, where we're saving the Id of every object we store in the databases for 60 seconds. This is such that, before we even try to save any new information, we check if the object Id is stored in Redis, which would mean the object is in the process of being added. If it is, we won't save again, if it isn't, we follow the process of saving. While this is working, it's important to note that if the bots notice the same Tweet or User twice and **something changed**, the changes won't be recorded in the databases.

[6]CRUD stands for create, read, update and delete, and comprise the basic operations for persistent data storage

### 4.3.4 Saving Users

We'll now discuss the operations the Control Centre must do whenever the bot requests him to save a User object. To better illustrate the algorithms, let's consider User X, with id 1.

The first thing we check is if an object with id 1 has recently been introduced. As said previously, this is done by checking the objects still available in the Redis-based cache system in the Control Centre. If so, we can ignore this object, promptly avoiding the rest the operations, and move to the next message. If not, then we have to check if it's already been inserted either in MongoDB or the Neo4J database, again looking for the id 1. Depending on the result, we have to either insert the User X or update the stored information with the new one.

Worth noting that these operations are not needed in PostgreSQL. Since the purpose of this database is also to save statistical data on the users the bot finds, then the information about User X will never be updated, only added with a more recent timestamp.

### 4.3.5 Saving Tweets

Saving a tweet is actually a more complex operation than simply saving a user. A request to save a new user only requires the user object to be properly saved in all databases, this is not the same as saving a Tweet. When saving a tweet, we must be careful about saving the Write relations to connect it to the author, as well as the Retweet, Quote or Reply relations if there is the need for it.

The process starts the same as saving a user, we first check if the Id was recently introduced in the Redis cache system, ignoring the object if a match was found. Then we check if we need to update the information or save as a new Tweet by checking the existence of the Id in the databases. If it's an update, then all we need to do is the update information, and change what is stored in the databases. Things get more complex, however, if it's a new tweet that needs to be stored.

If it's a new tweet, we need to check for four things: is it an original tweet, a reply in a thread, a quote or a retweet?

If it's an original tweet, all we need to do is save the Wrote relation between the tweet and its author. Fortunately, the complete information about the author of the tweets come with the Tweet Object, so we just need to save this author (using the previous algorithm) and forming the Wrote relation on Neo4J.

If it's a retweet or a quote, we need to save the original tweet and create the respective relation in the Neo4J database. This is done by checking the Retweeted and the Quoted field present in the object, only following this algorithm if it's set to true. It's worth noting that if it is set to true, then the original Tweet is stored in the object.

Finally, if it's a reply in a thread, the algorithm gets more complex, as in this case, the tweet it's replying to *does not* come as a field in the object, instead only having the original tweet's Id, and its author's id and username. This is enough to

create the relation in the Neo4J database, but not enough to create a new object in the MongoDB. So we start by checking if the original tweet is in the mongo database, and, if it's not, we save a stand-in. This replacement is a blank tweet, with all its non-important fields in a blank value. The control centre will then send a message to the Bot who sent him the reply to look for the original tweet object, at which point we update the stand in with the real tweet. The same logic will then apply to the original tweet's author

### 4.3.6 Caching

A *caching system* was necessary for this project due to performance issues related to our *Big Data* paradigm. This is where **Redis** comes in.

Firstly, the cache system was used to store the results of certain heavy queries made by our REST API to our databases to decrease the response time and improve the user's usability on the front-end, and so, a proxy has been implemented between the front-end and the REST API. If the request's data is not saved on cache then:

- **Queries are made to databases to obtain data**
- **The data is stored in the cache and returned to the user**
- **The cached data has no expiration time**

From now on, whenever the saved request is made, the data is returned directly from memory without the need to query the databases again, and when there are updates on database's data a callback is sent so that the cache updates the saved data.

Secondly, a cache system was integrated with bots to store some information about the bot's interactions as to avoid task repetitions. This repetitions can happen, for example, when bots are searching recursively for the users who interacted in the timeline of some other user. This can cause the respective bot to send multiple times in a row the same user to the Control Centre, overloading it. To avoid phenomenons like this, the bots always save the piece of information they are sending next to the Control Centre in Redis, with a pre-defined **time to live**, and when they want to send some new data to the Control Centre, they verify if they already send it previously.

### 4.3.7 Machine Learning Models Storage

As previously mentioned, a Mongo database is being used to store information related to twitter entities. But as it is visible in the architecture a second instance of this type of database is created for storing data related to deep learning models used in follow service, such as trained models, data from tokenizers, model's hyperparameters, etc.

We decided to create a new database instance for storing this type of data because the data is quite different and this data is private and so it is not supposed to have access to it (the Mongo main instance data is public).

## 4.4 Continous Integration

The complex nature of our project requires some way to make sure we remain organized as a team, and that the new additions to the code are, in fact, up to our initially defined standards.

### 4.4.1 GitHub Workflow

To organize ourselves as team, and appropriately work according to the size of our project, we decided to follow the *GitHub Guidelines* concerning **GitHub Workflow**[9]. When any of us want to work on a new feature regarding the project, we follow these basic steps:

1. Create a new branch from **master**
    - If it's a new feature, we follow the nomenclature **feature/<feature name>**
    - If it's a hotfix, we follow the nomenclature **hotfix/<bug name>**
    - If it's a branch that must be deployed, we follow the nomenclature **deploy/<branch name>**
    - Note there may be small variations to this, if it's features largely related to the one thing, the branch name may be grouped accordingly: REST features are called **feature/rest/<feature name>**, Web app features are called **feature/frontend/<feature name>**, etc.
2. Develop the needed functions for that branch, trying not to make it too different from the master and not including too many new functions and changes
3. Create a **Pull Request** when the branch is complete so that a second or even third person can look at the code developed, to make sure it makes sense and doesn't contain any obvious or unnoticed bugs; usually the code reviewers are people who are already familiar with the modules the branch is changing
    - If the reviewer finds problems with the code, he requests changes to the developer, who will have to appropriately fix them, push changed code and ask for a code review again, as depicted in Figure 4.14
    - If the reviewer thinks the code developed has no problems, he approves the changes, which allows the developer to go on to the next step, as depicted in Figure 4.15
4. Merge it to the master branch, once everything is done, making sure no conflict issues are caused when merging.

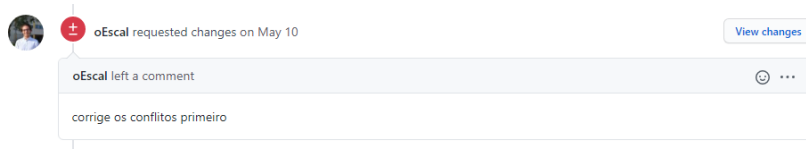


**Figure 4.14:** Example of someone requesting changes in a pull request.

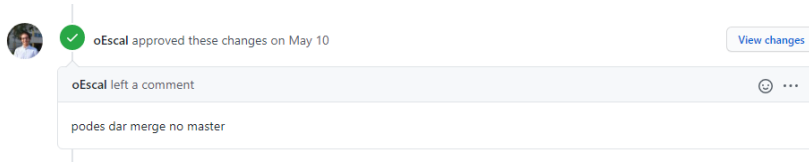


**Figure 4.15:** Example of someone accepting the branch, so that it can be merged to master.

### 4.4.2 Issues

Throughout the project, we may notice some bugs on the project or small, important, features associated with a branch recently merged to the master branch. In this case, we tell the developer responsible for the branch to implement the changes needed.

The problem arises when the developer is already busy with a different feature, so he can't interrupt his work to implement the changes. In this case, we have to add a new **Issue** and associate it with one of the teammates. This will then be saved under the Issues, that the developer can then check to see the details of the problem and work on the hotfix. Figure 4.16 showcases a list of opened issues that need to be worked on.

### 4.4.3 Tests

To guarantee that our REST API complies with the requirements and news changes don't mess up with work already implement, tests have been implemented that cover a majority of REST features.

To ensure good practices in CI development, these tests are integrated with our project's pipeline. Whenever a commit is made to the online repository the tests are run and feedback is returned 4.17.

These tests were implemented using Django libraries and mixer library [13] capable of generating objects compatible with Django models to simulate CRUD operations in the databases.

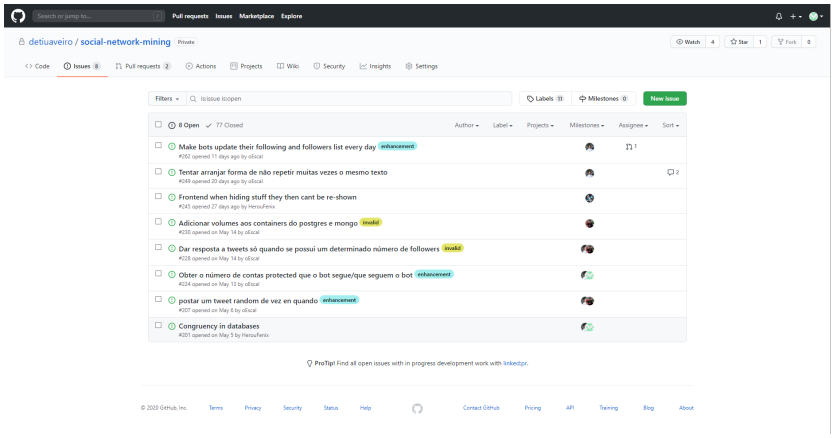


**Figure 4.16:** A list of Issues that can be present on Github.

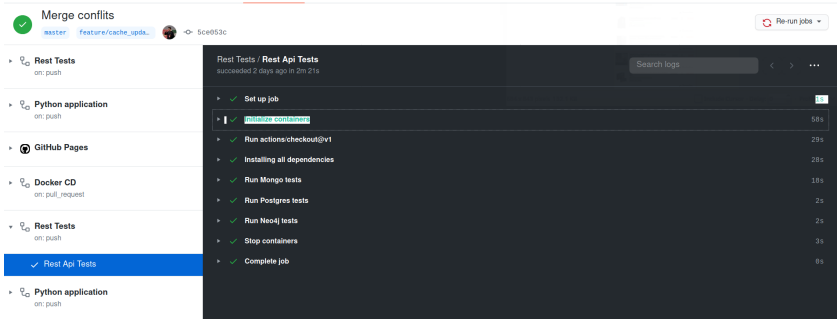


**Figure 4.17:** Rest API tests on GitHub jobs.

## 4.5 Deployment

Since our project had a lot of components working together, it was crucial to run everything on separate machines as to avoid creating a workload bottleneck by overloading a single one. Therefore, the **deployment** process was a crucial part of our project to ensure that all modules were properly working and verifying if everything was running as expected over a long period of time and to collect all the data.

### 4.5.1 Continuous Delivery

To facilitate the **deployment process**, we soon came to the conclusion that it would be extremely necessary to use a **Continuous Delivery philosophy**. This way, we used **GitHub Actions** [8] to create a **Continuous Delivery Pipeline**. We defined

that whenever a new **code push** was made to a branch under **Pull Request** with the **deploy tag**, as seen in the figure 4.18, a **Workflow** would create all the new **docker images** containing the new **modules versions**. With this, we always had the code ready to deploy, as is noticeable on the figures 4.19 and 4.20.

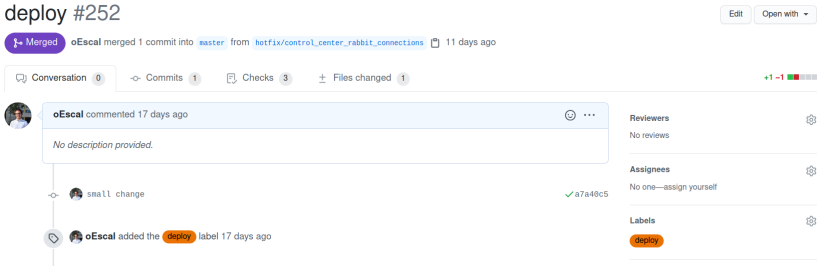


**Figure 4.18:** Example of a pull request with the tag deploy.

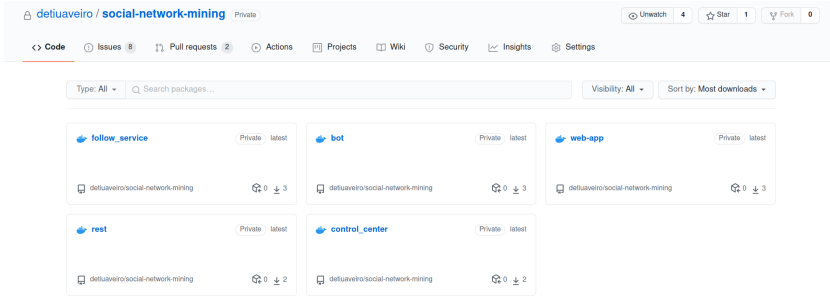


**Figure 4.19:** All the docker images ready to deploy, available on the project's GitHub repository.

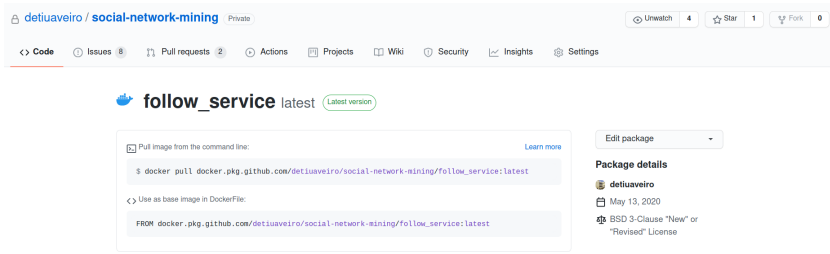


**Figure 4.20:** Example of the docker image of one of the project modules, available on the project's GitHub repository.

Through the implemented workflows, we can be certain that any important change to the code will be deployed to the server as soon as possible, without having to personally change anything in the servers themselves. The whole process of deploying the project became frictionless and easy without us, as the programmers, having to worry about the configurations in the server on each change.

### 4.5.2 Hardware

For our project to be able to run smoothly, we need a lot of hardware capable of executing our implemented services on a consistent basis. As was this was the case, for this project, we had access to three machines, kindly provided by our coordinator Prof. Diogo Gomes, with the following specifications:

- **Server 1**
  - **CPU:** Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
  - **RAM:** 16 GB
  - **ROM:** 160 GB
  - **OS:** Linux - Ubuntu 20.04 LTS
- **Server 2**
  - **CPU:** Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
  - **RAM:** 16 GB
  - **ROM:** 160 GB
  - **OS:** Linux - Ubuntu 20.04 LTS
- **Server 3**
  - **CPU:** Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
  - **RAM:** 8 GB
  - **ROM:** 175 GB
  - **OS:** Linux - Ubuntu 18.04 LTS

### 4.5.3 Tor

**Tor**, or **The Onion Router**, is a project that allows users to access internet anonymously [27]. To reach this goal, all the traffic moves across different **Tor Servers** before getting to the desired destination. If someone tries to intercept the traffic, they will perceive the sender as one of the **random nodes of the Tor Network** instead of the real sender [12].

We used Tor in order to access several Twitter accounts from the same machine. In order to achieve that goal, we configure as many **Tor proxies** as the number of

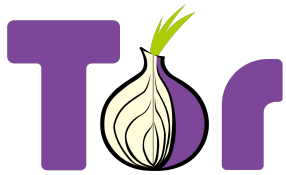

**Figure 4.21:** The Tor logo.

**Bots** we had running on the server. This way, each individual Bot was connected to Twitter using the same source IP address (the server IP address), but the receiver , i.e., Twitter, perceived them as **different IP addresses**. This method was very important, since Twitter blocks accounts that are connecting in a little period of time from the same machine, as it interpret this behavior as being from **Bots**.

### 4.5.4 Docker

**Docker** is a tool that provides the ability to easily create and deploy applications in containers [5]. Probably the main advantage of using **Docker** is that, because of the applications are running in containers, our code run as expected in any machine, with any specifications and operative system, as long as the machine have **Docker** installed.


**Figure 4.22:** The Docker logo.

We used this tool in our development and deployment process because it provides the versatility we described in the previous paragraph, as well as the ease of usage of this method. So, in the server side, any part of our project was deployed using **Docker containers**.

Since we had 3 machines at our disposal, we divided the services the following way:

- All the **main DBMS**, excluding the caching ones, where deployed in one of the machines.

- The **Follow Centre** and respective **MongoDB** database where deployed in one of the other machines. We isolated this module alone in one of the servers because it had an *heavy* **machine learning** model running.

- The **Web app**, **Rest API**, **Bots**, **Control Centre**, **Response Centre**, **RabbitMQ Message Broker** and **Caching Systems** where all deployed in the remaining machine.

It is important to notice that the last two machines listed above where grouped as a **Docker Swarm**, to facilitate the management [26]. This way, we were able to manage all the modules we created on a single machine and manage platform.

#### 4.5.4.1 Databases deployment

The databases deployment was done manually and at an early stage of the project.

All databases were deployed as containers using docker, which ended up streamline some configurations aspects due to the good documentation present in the docker HUB.

#### 4.5.4.2 Continuous deployment

As referred in the section 4.5.1, we add all individual components of the project always ready for production. To save time and to avoid repeat the same processes of deploying these parts over and over again always a new release was made, we set up a continuous deployment process to deploy the new releases automatically.

To reach this goal, we used the open source tool **Watchtower** [33]. We confederated it to, in intervals of 5 minutes, verify if there are new **Docker images** of the **deployed software** in the proper repository. Therefore, not only **Watchtower** was responsible to **deploy the new versions of our project components**, but also updated any software we had deployed using **Docker**, such as our databases systems, always there was a new version.

We also configured **Watchtower** to notify us in our **Slack project Workspace**, always there was a new deploy, as is shown in figure 4.23.

#### 4.5.4.3 Portainer

**Portainer** is a **lightweight management platform** that wrap some of the main **Docker tools** in a simple **GUI**, allowing an easy administration of all the containers we have deployed in some **Docker host** or **Docker Swarm cluster** [20].

This way, we used this software to more easily manage the **deployed Docker images and containers** and to verify **log files of our service's new versions**, using a single platform in our browser. In the figure 4.25 is possible to confer an example of one of the **Portainer Container Dashboards**.

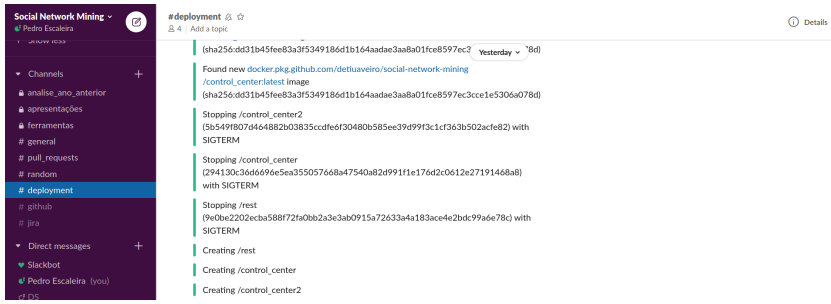


**Figure 4.23:** Slack Watchtower bot notifying new deploys.


**Figure 4.24:** The Portainer logo.

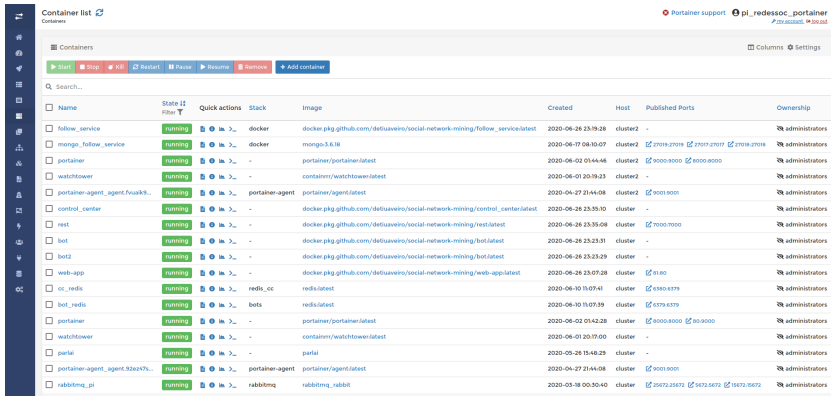


**Figure 4.25:** Portainer Container Dashboard.

## 4.6 Frontend & REST API

As aforementioned our frontend module constitutes in a web application built using **React JS** and some other modules. We will be talking at length about the web-app in chapter 6. In tandem with the construction of our frontend we also developed our REST API.

### 4.6.1 REST API

The success of a service is based on the target audience's satisfaction and therefore that must exist a bridge between the business logic of service and the target audience. To apply this bridge, it is necessary to create a mechanism capable of translating "raw" data present on service in clean and legible information for the target audience. In addition, this bridge is important to connect user interactions with service.

To meet these requirements in this project it was essential to create a REST API that allows us to connect our service to the outside world, more specifically to the web environment.

Our REST API was implemented using **django** and allows us to return all kinds of information related to the interactions of the bot, to do search queries using well-defined parameters, and add information related to the bot's policies. To get information about what endpointss were implemented just visit our <documentation> [7].

In addition to the basic features related to a REST service, our API is capable of:

- **Treat and merge data from two different sources (in this case two different databases) acting as a transaction**
- **Reports' generation**
- **Make aggregation queries for statistical purposes**
- **Improve the performance of heavy queries using a cache mechanism (further reading on chapter X)**
- **Notify the cache about data updates in a decoupled manner, implementing the observer standard using the "Django signal" tool**

### 4.6.2 Report Generation

Report Generation is a particularly complex operation in our system, as it has to communicate with two databases at the same time. Let's consider a simple example of wanting to know the number of friends and followers the users that follow our bots. The problem with this is that it need information from both **Neo4J**, to know who follows our bots, and **MongoDB**, to know those users' friends and followers count.

So the first thing to do is to extract the network from the requirements specified. From what the user inputted in the frontend, we form a *Cypher*[8] query and send it to our Neo4J wrapper, who will return a list of all nodes and relations that comprises the network requested. Following the previous example, this means getting a network that shows all users who follow our bots.

[7] http://192.168.85.60:7000/documentation/
[8] Neo4j's Query Language

Next, we extract all nodes and send the **ID**s to our MongoDB wrapper, along with what data that the client asked for, this means we're doing a search in bulk to the Mongo database. In the previous example, this would mean getting the friends and followers count from the users' ids we got from the network.

Now there are two different approaches you can take in this scenario: go relation by relation in the network, get the user node, search that user's information in MongoDB and append the results to a list of rows. While it can have some performance improvements, the idea of searching the information user by user is too slow for the service, or for our standards. A network with only thousands of relations would already take minutes, most of it due to these singular searches.

So instead, we decided to look for the information in bulk. This means we go through the entire network, extract all nodes, and do a search in bulk to the MongoDB database. The first major problem with this strategy is that we'd actually be losing the sense of the network after extracting the nodes: unless we go through the network a second time, we wouldn't know who's related to whom to make the report, which would cause another performance issue. So, to account for this, we have to keep track of the position each node has in the results list; we would then be able to simply add the MongoDB information to the list of results with a O(1) operation. This approach, while complex, massively improved the performance of our report generation, to the point where it could create reports with hundreds of thousands of rows under a minute.

CHAPTER 5

# Results and discussion

Throughout these past months, we have been working on bots that can scour the Twitter network and continuously store all users and tweets they found. This will help us then create complex networks around these entities, which can be further studied and analysed.

## 5.1 Data Present in the Databases

After a few weeks of scouring the platform, we have saved millions of new objects and relations between these, sizes akin to other big data projects. As we can see on Figure 5.1, our bots have gathered information about more than 700,000 tweets and around 350,000 users. Furthermore, figures 5.2 and 5.3 show how these numbers came to be over time. Overall, when looking at these graphs, we can see that the bots are more inclined to look for tweets than for users, following one of the main goals of the project, which was to be able to follow how a tweet is propagated through the network.

## 5.2 Bots' Activity

For most of our project, we had only one bot active in the network, having only recently launched new bots to scour the social platform. Keeping this in mind, the following graphs show the bots' activity in the platform, based around the logs that are inserted in the PostgreSQL database.

Figure 5.4 and 5.5 show us how active the bots have been. In Fig. 5.4 especially, we see that the bots have a lot of ups and downs, mostly focused around new features being added and bugs being found that would cause the bots to be paused for the time being.

While we know how active the bots are, we also need to know what they have been doing on the platform itself. Figure 5.6 and 5.7 show us how they have interacted with Twitter: how many people they followed, how many tweets they replied, how many retweets they made, and how many tweets they liked.

Finally, we have to measure how well our bots are doing at infiltrating protected circles, we have to see how many protected users our bot was allowed to follow. As of writing this report, the bot has established contact with around **59 protected**

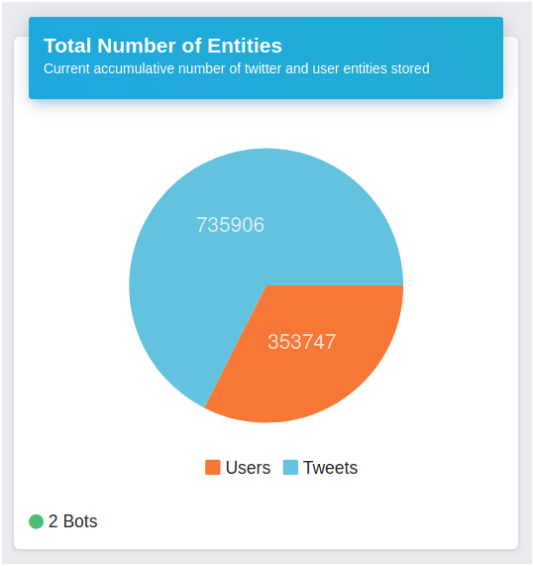


**Figure 5.1:** Graph representing all the entities, Tweets and Users, saved on the platform. Obtained from the project Statistics Dashboard.

**users**, out of 129 total protected accounts he has *tried* to follow, consisting of a success rate of around 45.7%.

It should be said that, despite our best efforts, we have not found a way to make process of saving to all databases a transactional operation. This means the databases may not be consistent between each other, so some information may be present in some of the databases, and missing in others, which can influence some results.

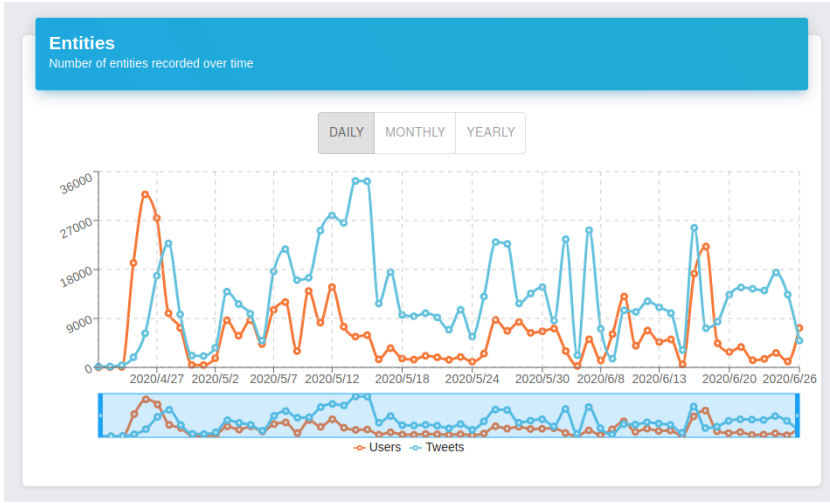


**Figure 5.2:** Graph representing the number of entities, Tweets and Users, saved on the platform over time. Obtained from the project Home Dashboard.

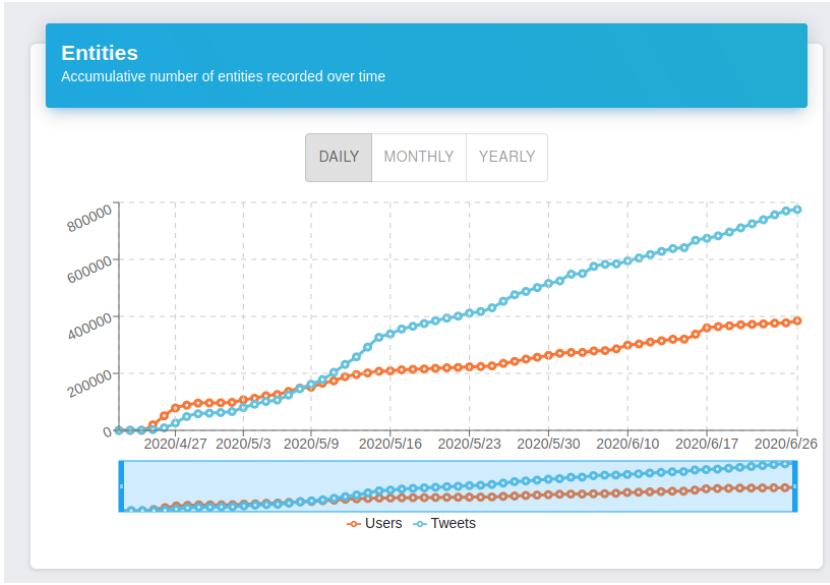


**Figure 5.3:** Graph representing the cumulative number of entities, Tweets and Users, saved on the platform over time. Obtained from the project Statistics Dashboard.

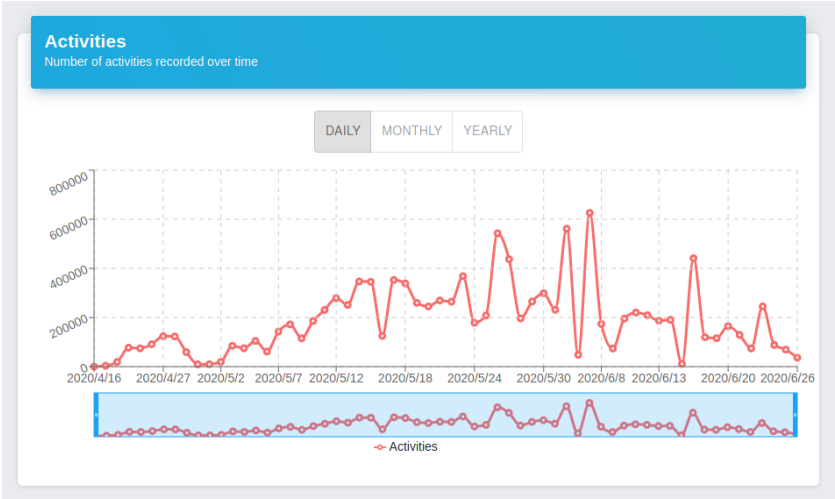


**Figure 5.4:** Graph representing the number of activities done by the bots, saved on the platform over time. Obtained from the project Home Dashboard.


**Figure 5.5:** Graph representing the cumulative number of activities done by the bots, saved on the platform over time. Obtained from the project Statistics Dashboard.

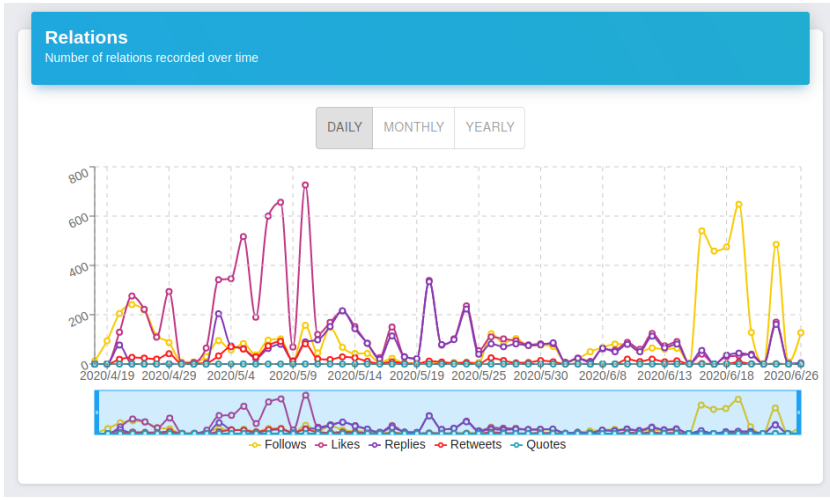


**Figure 5.6:** Graph representing the number of relations connected to the Bots, saved on the platform over time. Obtained from the project Home Dashboard.

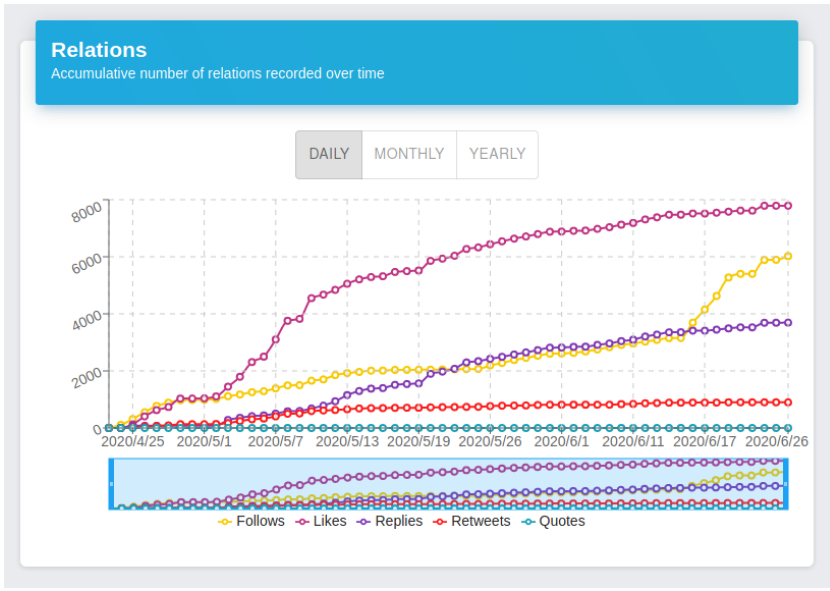


**Figure 5.7:** Graph representing the cumulative number of relations connected to the Bots, saved on the platform over time. Obtained from the project Statistics Dashboard.

# CHAPTER 6

# Web-App

In this chapter we will be going through the different features and aspects, as well as the overall design of our web service. This will be done both to ease in readers into the utilization of our app, but also to serve as a showcase of the various actions a normal user can perform on our platform.

## 6.1 Design

Let us first explain our design philosophy and the aesthetics of our web-app. We aimed to produce a *User Interface* (UI) that would be as clean and user friendly as possible. To achieve this, the word *minimalist* was always on our mind as we planned and designed all of our components and pages. A user should not require a tutorial to use our app, and as such all buttons' placements had to be in their most logical place, all labels should be sufficiently explanatory and, for less intuitive actions and features, non-intrusive tool tips should be provided. Furthermore, we had to consolidate being as thorough in the showcasing of information as possible, whilst avoiding cluttering our interface. In order to avoid visual nausea a very reduced color palette was picked, both for the overall pages but also for our graphs, with our primary color being a soft blue that resembles Twitter's own, hence creating a visual connection between our platform and the social network. Besides this, we cared to keep our text and terms as consistent as possible throughout all pages and to have them be thoroughly understandable, using as many layman terms as possible as to not discriminate or ostracize any less tech-savvy users who might come from a background of network analysis rather than one of information databases and computer systems.

## 6.2 Features and Pages

We will now be taking a look at our wide range of pages, going into some detail as to what can be accomplished in each of them, but also explaining their overall layout and structure. All of these pages are accessible through our side dashboard. For purposes of cleanliness and having the website be responsive and usable through any type of device, regardless of dimensions, this dashboard can both be minimized or hidden entirely. We will be tackling each page in the order (top-down) they're presented in

the dashboard. It should be noted that, to refrain from confusion, we will be using *user* to describe our platform's user and *Twitter User* or *(Twitter) User* to refer to Twitter user's whose information are in our databases.

### 6.2.1 Home

Starting off with the page a user gets redirected to upon entering our platform. Our *Home* page contains several statistics pertaining to **new** incoming data. Right on the top we have two graphs, one showing the new entities registered on the current day the user is accessing the platform on and the other showing to growth of new entities added to our databases. By default we start by showing the monthly growth since it allows for a faster load (since it is obviously going to show less data than if we were to see the statistics per day), but we allow a user to pick if they want the data shown by day, month or year. Below this are two similar graphs, this time showing the same statistics but for relations rather than entities. Lastly we show two tables on the bottom of the page, one for the latest 100 (by default) tweets registered and the other for the latest 100 (by default) activities our bots performed. These values can be altered, allowing the user to show any number of latest records. A small showcase of the Home page's appearance can be seen in figure 6.1.

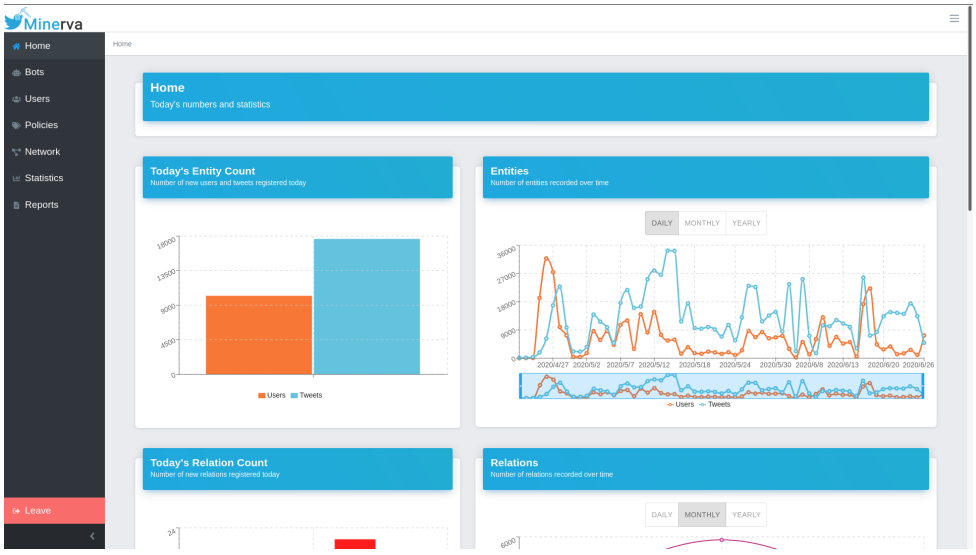


**Figure 6.1:** A snippet of what the Home page looks like.

### 6.2.2 Bots

Moving on we have the pages pertaining to our bots, of which there are two. These pages allow users to both visualize all the bots we have deployed, but also exert some control over them.

#### 6.2.2.1 Bots List

Firstly, clicking on the *Bots* tab in our side menu brings us to our *Bots List* page. In here users can find out how many bots are active, they can see their general info, such as name, *Twitter Tag* (the name that identifies them on Twitter), follower count, as well as pause their behavior, effectively deactivating them for the time being (the reverse operation of activating a deactivated bot is, obviously, also available). There is also a specific button that allows users to travel to any specific *Bot's Profile* page. Figure 6.2 show what the list of bots looks like.

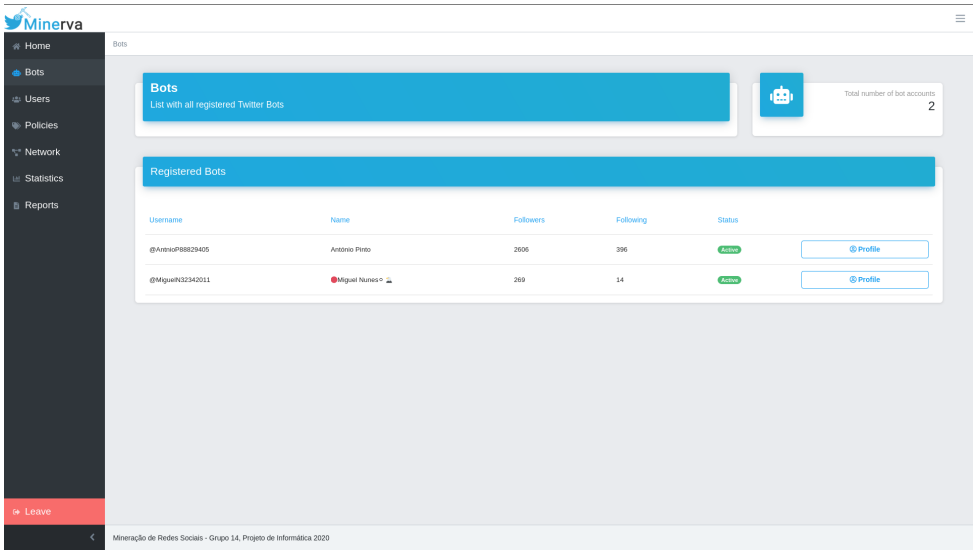


**Figure 6.2:** Our bot list with two currently active bots..

#### 6.2.2.2 Bot's Profile

The *Bot's Profile* page is a bit more complex than the latter one described. First and foremost it contains that specific bot's Twitter profile information - name, Twitter Tag, number of followers, number of people they're following, country and description. Besides this it also contains some statistics on the bot, such as their overtime growth of followers and people their following. This is, once again, presented by default on a monthly scale, but can be changed for a daily or yearly. Next we have a list of all the

bot's *retweets* and *quotes* sorted from latest to oldest, as well as their latest *retweet* or *quote* showcased on the left. We also allow our users to visualize the specific tweets, both their text content but also their media (photos, videos or GIFs) by clicking on the corresponding button. All of the features so far can be seen in figure 6.3. Under this section we have both the bot's active policies and a paginated listing of all their actions, sorted by the time they were performed, from latest to oldest. On the policies listing we also included the option to easily remove a policy from a bot as well as a button that, when clicked, will redirect the user to a form, allowing them to add a new policy to that specific bot, either by picking from the existing list of policies (in which the bot is not registered), or by creating a new one, as exemplified in figure 6.5. Upon creation, the user will be redirected back to the profile page. Finally, on the bottom of the page are two tables, one with a listing of all of the bot's followers and another with all of the people their following. Each of these rows point to a user and clicking them will redirect the user to that Twitter User's specific profile in our platform. This can be seen in figure 6.4

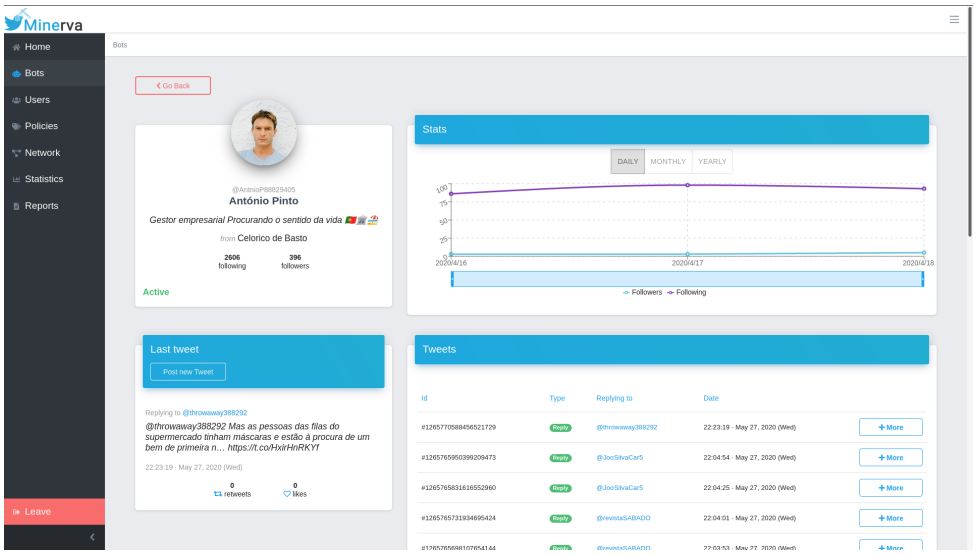


**Figure 6.3:** The top of the bot's profile page, showcasing the bot's account information and follower/friend count growth, latest retweet and its latest interactions with the Twitter platform.

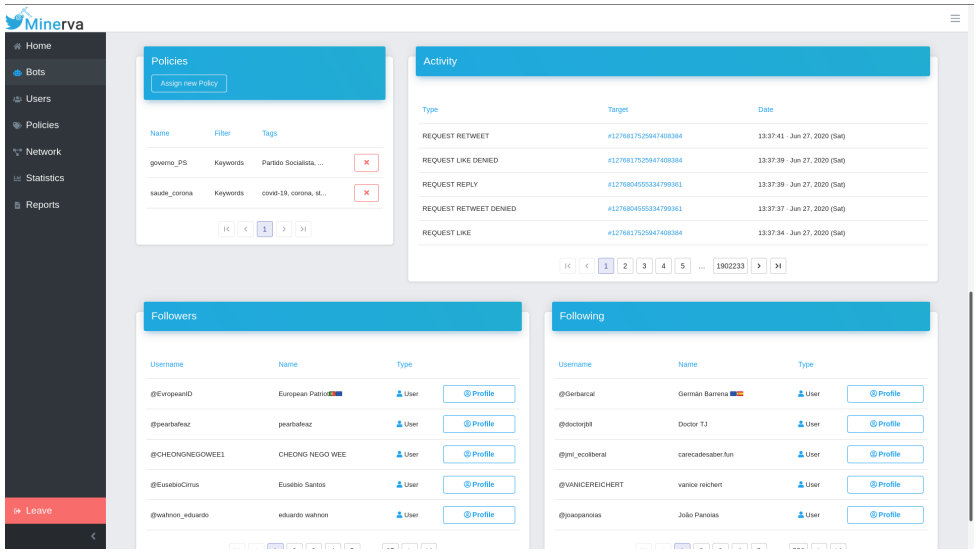


**Figure 6.4:** The bottom of the bot's profile page, showcasing a list of the bot's activity, policies, followers and followings.

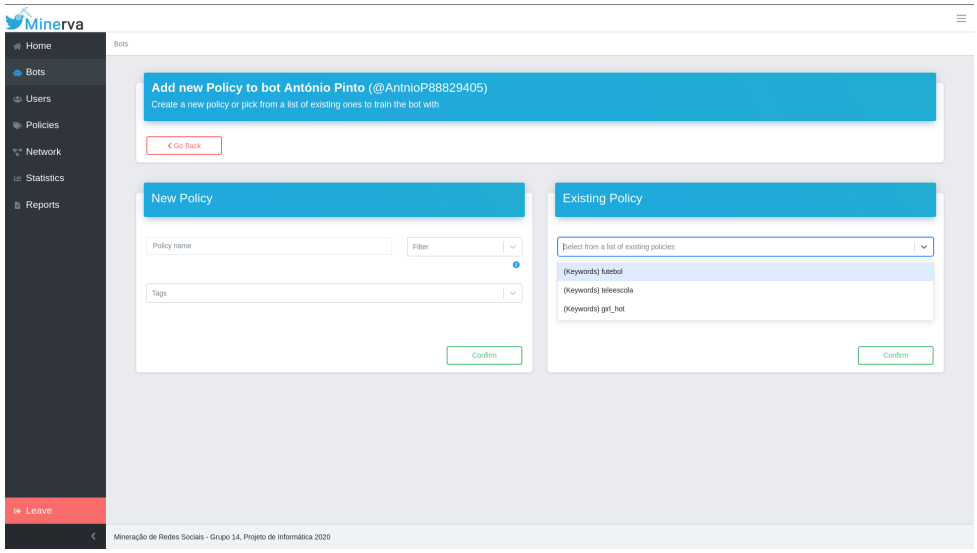


**Figure 6.5:** The form that allows users to directly add a new or existing policy to that specific bot.

### 6.2.3 Users

Next we have a very similar set of pages to the ones previously described that, instead of referring to bots, aim at showcasing the Twitter Users and their information, scoured by our bots.

#### 6.2.3.1 Users List

Our (Twitter) *Users List* consists on a similar, but more advanced, page to the *Bots List* page. Accessible by clicking the *Users* tab on our sidebar, this page has a counter for how many users are in our database as well as a paginated list containing all of them. Figure 6.6 illustrates this page's appearence. Alongside it there is also a search input that allows users to search for a specific Twitter User either by name or *Twitter Tag*, as seen in figure 6.7. We also permit the filtering for protected (or private) accounts only, as seen in image 6.8. Each row on the table is similar to the ones in the *Bots List*, with each specific Twitter User's name, tag, follower counts, and most importantly, a button that allows users to travel to that individual Twitter *User's Profile Page*.

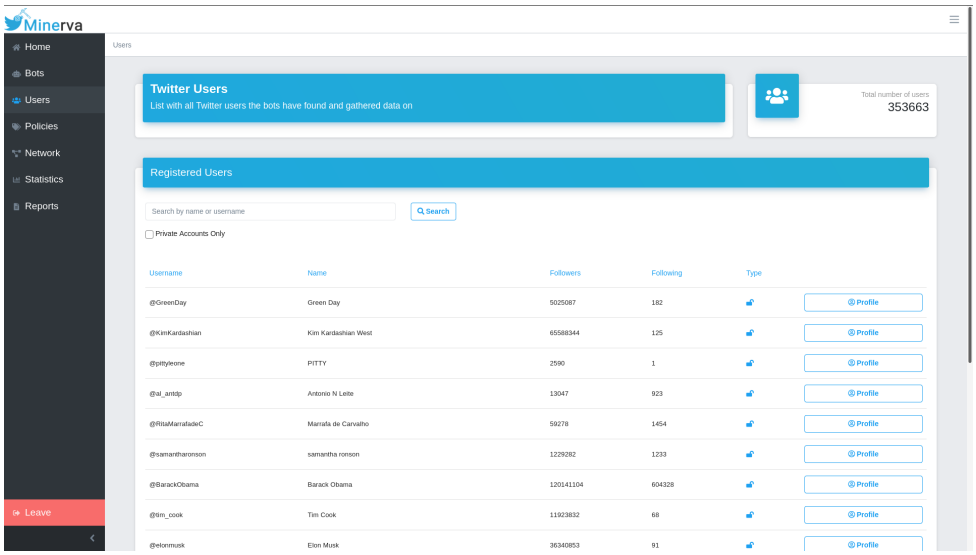


**Figure 6.6:** Our users list.

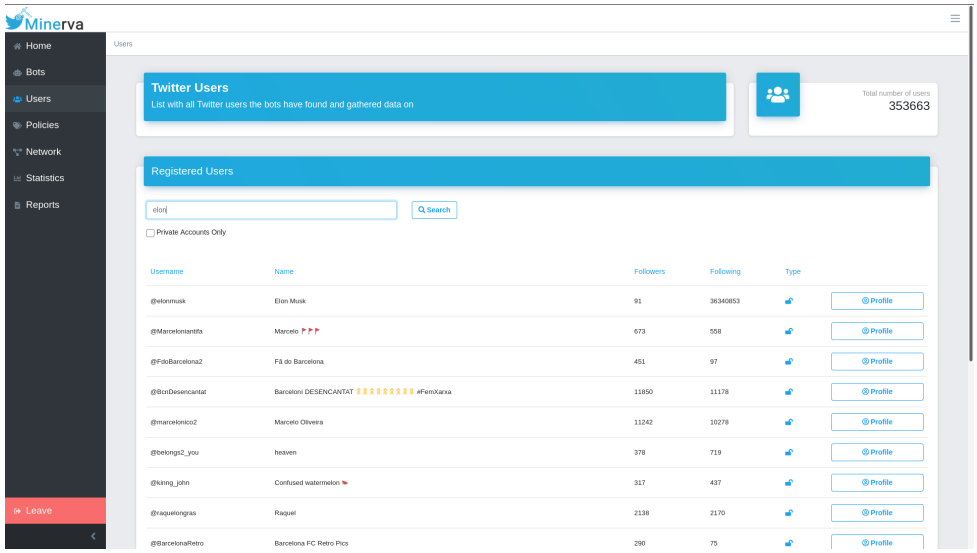


**Figure 6.7:** Our Minerva user's can look up certain Twitter Users by searching by name or tag.

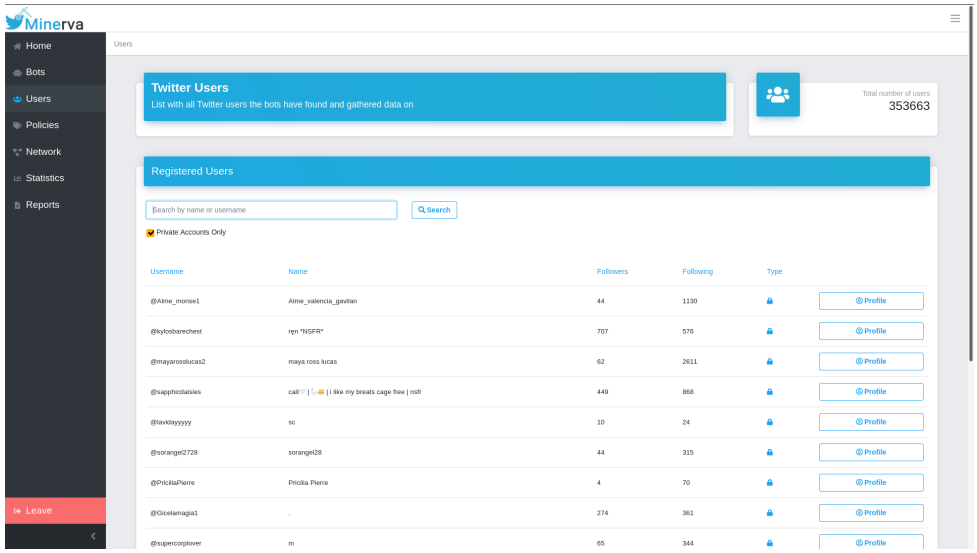


**Figure 6.8:** We allow the option to display only private accounts.

#### 6.2.3.2 User's Profile

Following we have the (Twitter) *User's Profile*. This page is, on a layout and content perspective, identical to the *Bot's Profile* page. The difference being that we no longer have the listing of policies, nor the listing of activities. All other features are there though, such as the general Twitter information, followers and followings tables and tweet listings (alongside the latest tweet showcase). Figures 6.9 shows the general appearance of this page, whilst figure 6.10 shows how we display tweets on our platform.

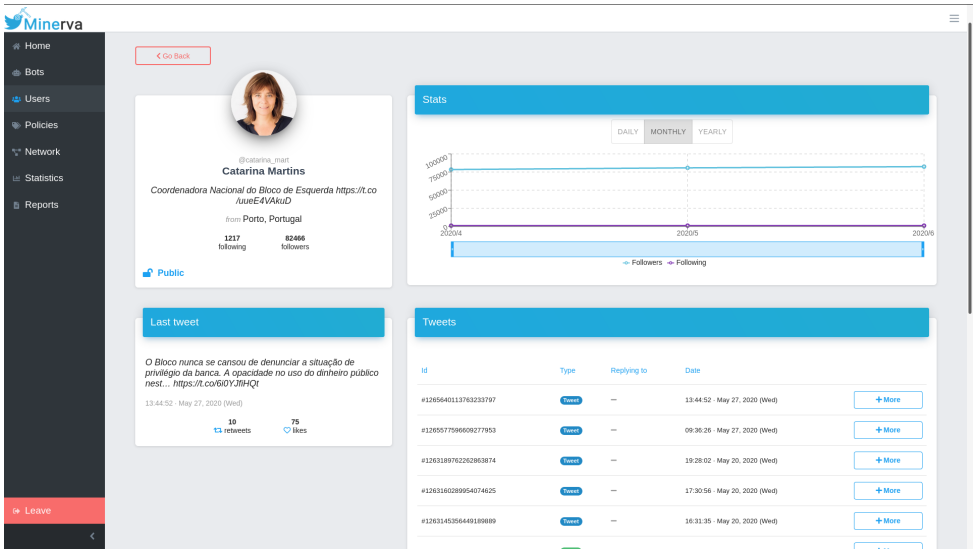


**Figure 6.9:** The Twitter User's profile page.

### 6.2.4 Policies

The policies page, which one could enter by clicking the *Policies* tab on the side menu, acts as a center for all the currently registered policies that are controlling our bots behaviours. A user can use this page to both visualize as well as edit all existing policies, changing their assigned bots, name, tags and types. They can also see the current state of the policy (whether it's in training or already active). This can all be seen in figure 6.11. A button allows users to register their email in order to receive a notification when a new policy has been put into training or when they're done training. The appearance of these emails can be seen in figure 6.13. Besides all this a button also allows the user to navigate into a new form page, allowing them to create and define an entirely new policy, as illustrated in figure 6.12. Upon completion of this form the user will be redirected back into the policies page.

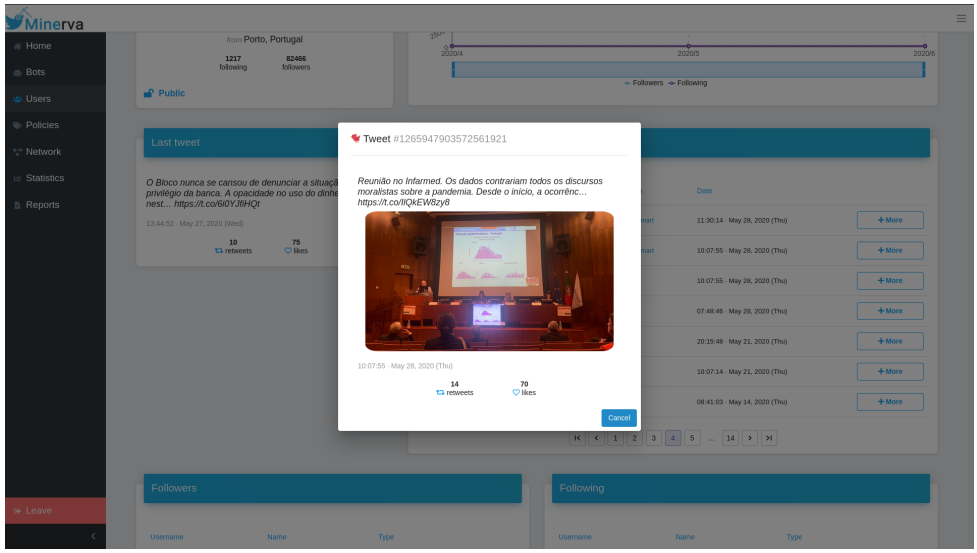


**Figure 6.10:** What the visualization of tweets looks like.

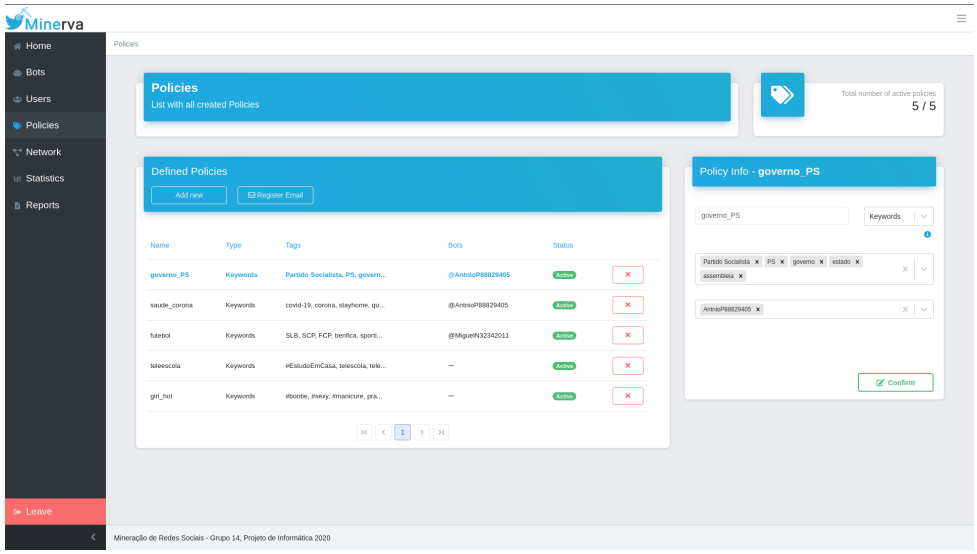


**Figure 6.11:** The Policy page lists all existing policies, allows their editing and deletion..

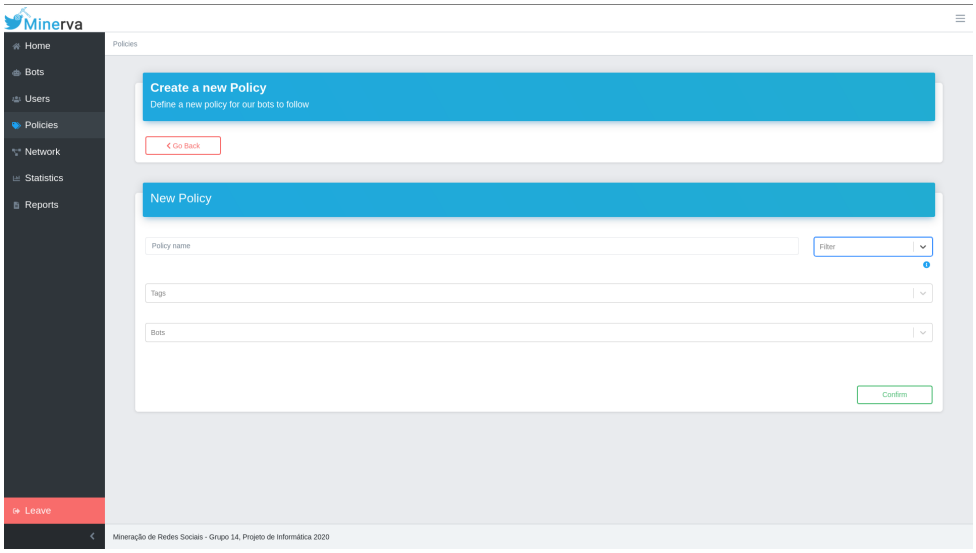


**Figure 6.12:** The form that allows the registering of a new policy.

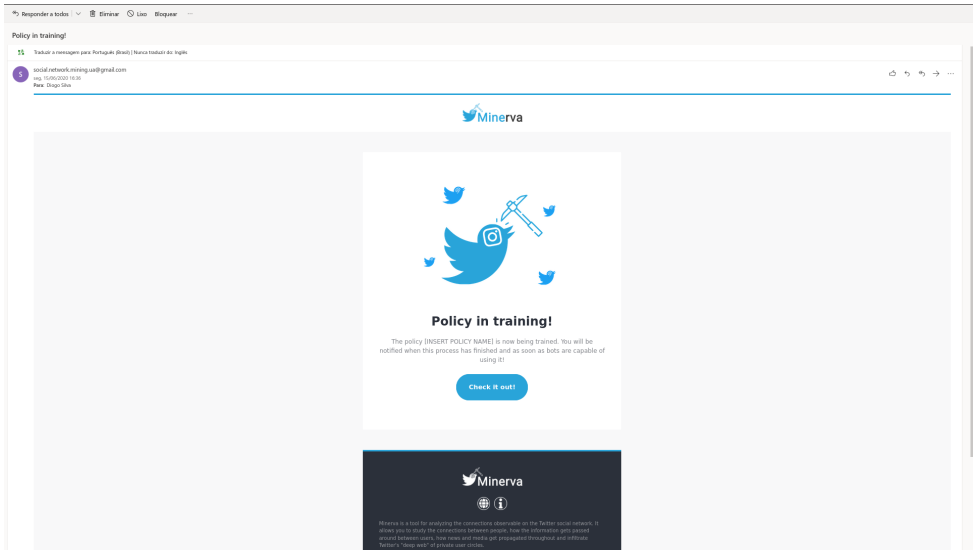


**Figure 6.13:** An email sent by our system notifying that a new policy has started training.

### 6.2.5 Network

The next two pages are relevant to the requirement of being able to visualize and travel the network of connections between users, bots and tweets for the purpose of getting a feel and analyzing the flow of information. As such, we needed both a graphical display for this network, as well as a way to query it that would be intuitive for users.

#### 6.2.5.1 Network Visualization

Clicking on the *Network* tab on the side menu brings a user to the *Network* page. In here we can find a window which show, by default a sub section of our network, in the form of a graph, as seen in figure 6.14. This graph is fully intractable, movable, zoom-able, clickable and physics enabled but this also means it can cause a heavy-load on the browser. As such we recommend users to never try to show more than around 2000 nodes at a time. On the right side we have a set of controls which allow us to find a specific node in the current network, focusing and zooming in on it, as well as filter out bot, (twitter) user or tweet nodes, or even hide the relations. The searching for nodes in the current network is exemplified in figure 6.15. Clicking a node will also automatically zoom in on it whilst double clicking a bot or user node will redirect the user to the corresponding profile page.

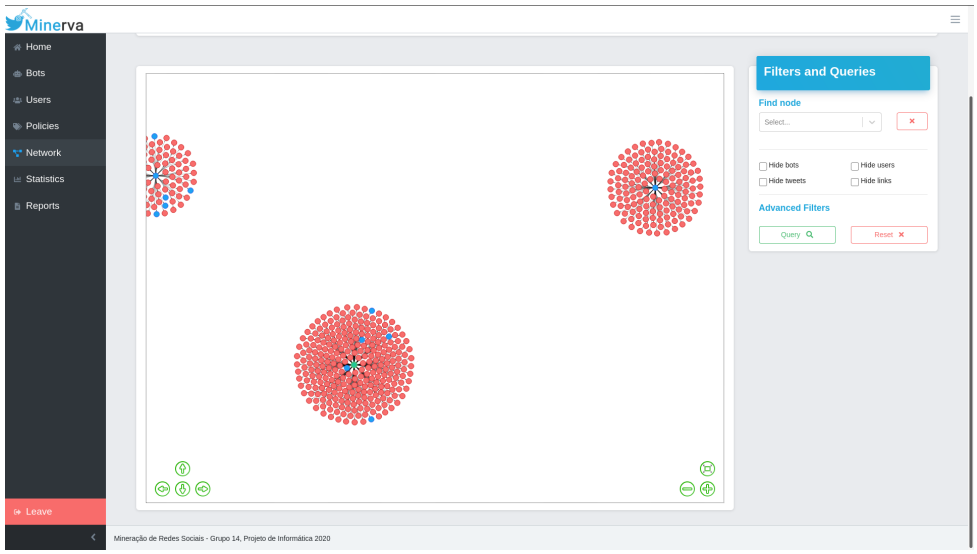


**Figure 6.14:** What the Network Visualization page looks like.

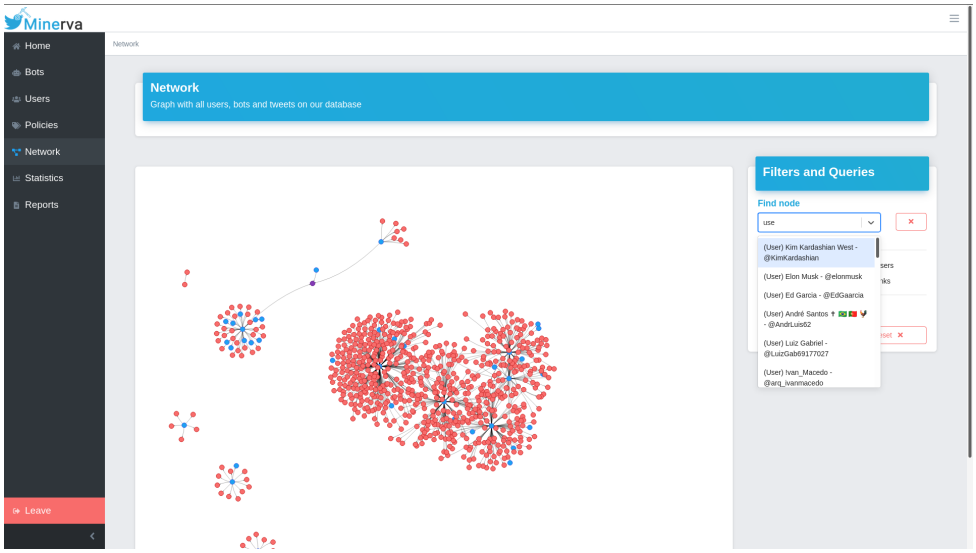


**Figure 6.15:** Minerva Users can look up for specific nodes present in the currently displayed network, hide certain types of nodes or even choose whether to display or not relations.

#### 6.2.5.2 Network Query Form

By clicking the corresponding button on the *Network Visualization* page, the user gets sent into this *Network Query Form* page, as is displayed in figure 6.16. In here any user, no matter the amount of prior knowledge on graphs or graph databases, is capable of making advanced queries on our network in order to visualize a specific sub-network of particular interest to them. Included is also a small tutorial which explains how the page operates, as well as tool tips which describe what each field is and what their effects are. After confirming the query the user is the redirected back to the Network Visualization page which will load the network correspondent to the inputted parameters on the form.

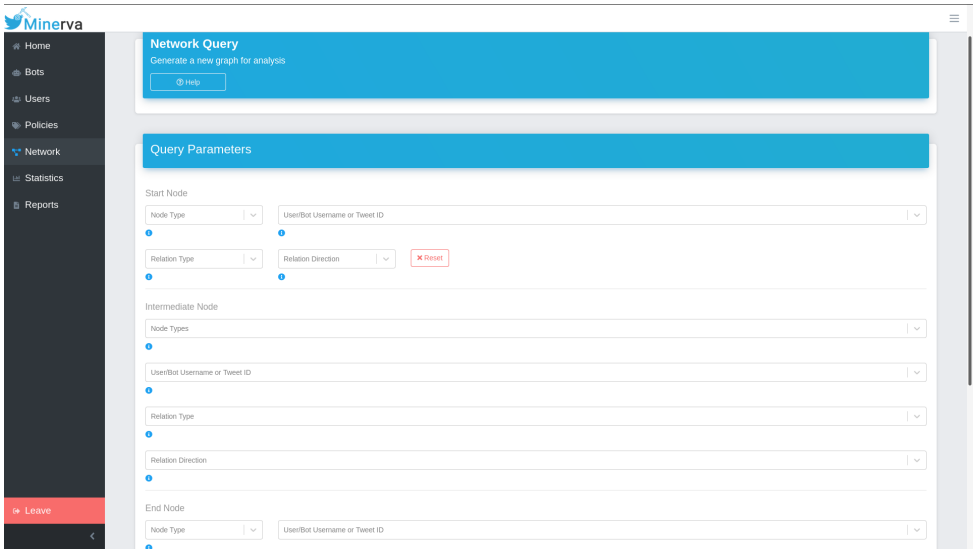


**Figure 6.16:** The advanced network querying form.

### 6.2.6 Statistics

The *Statistics* page, available through the *Statistics* tab on the side menu, has a similar layout to that of the *Home* page (which was done in purpose in order to preserve consistency between the web-app's pages). Its main function is to cleanly showcase accumulated data for analysis such as the growth in the number of activities over time, entities and relations. Some of the graphs this page has to offer are show in figures 6.17 and 6.18.

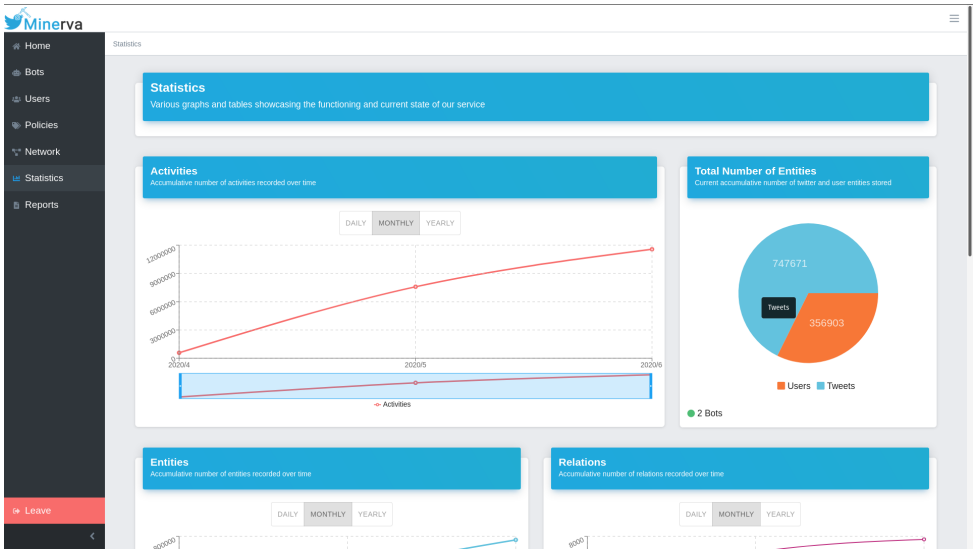


**Figure 6.17:** The top of the statistics page showing a pie graph with the total number of each entity we have registered, and a graph of the total number of activities over time.

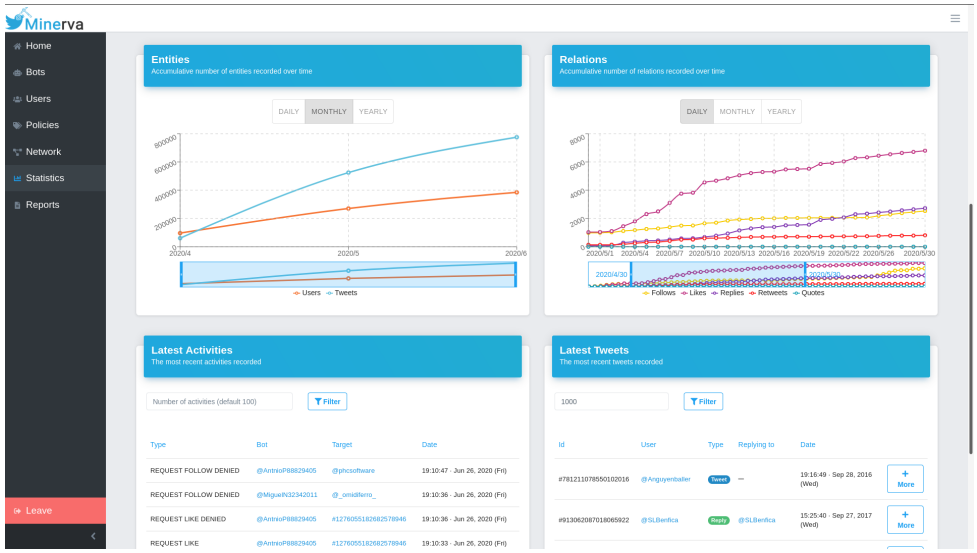


**Figure 6.18:** The bottom of the statistics page listing the latest activities and tweets registered, and graphs showing the growth in the number of entities and relations.

### 6.2.7 Reports

At last we have the *Reports* page. This page has a form very similar to that of the *Network Query Form* page, but instead of serving as a way to query our network in order to visualize a sub portion of it, it works to generate either a CSV or JSON format report with the information asked for on the inputs. Users can even pick what fields of information, relative to the nodes, they would like to include or exclude from the report. This page is exemplified in figure 6.19 alongside a possible query and its result in figure 6.20.

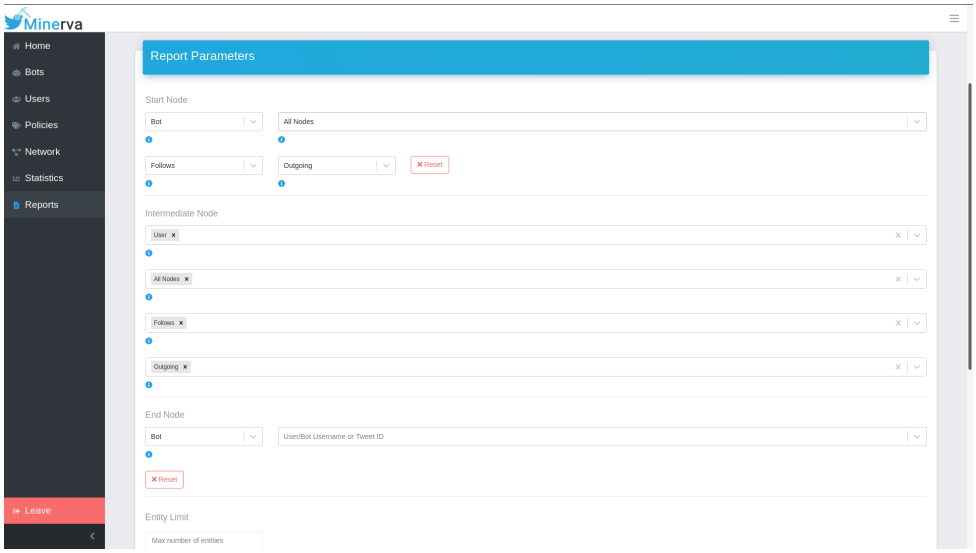


**Figure 6.19:** The reports form page and an example of a query that returns all users that any of our bots follow that also follow any of our bots back.

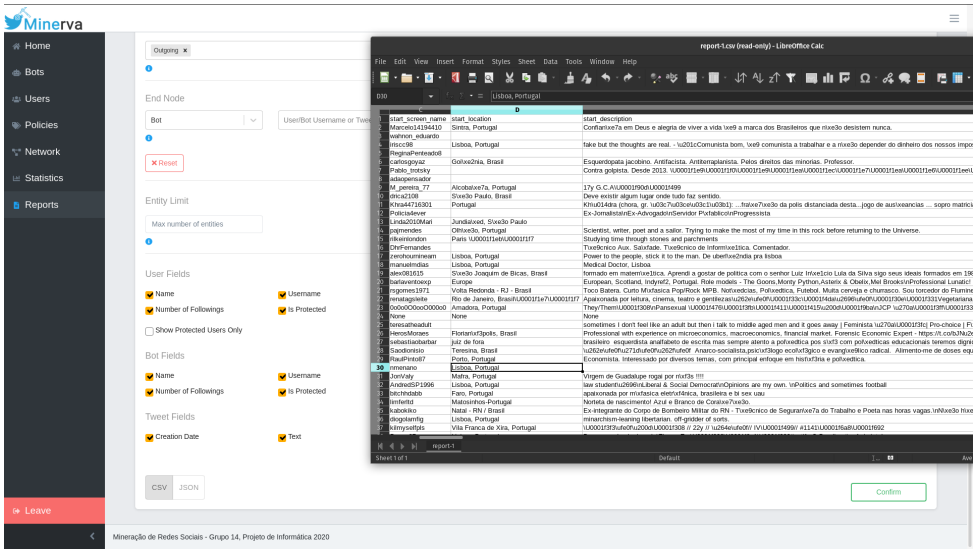


**Figure 6.20:** The CSV file that is generated by our reports page using the query specified in figure 6.19.

# CHAPTER 7

# Conclusion

## 7.1 Final Thoughts

**Minerva - Social Network Mining** is a system developed for the scouring of information on the **Twitter** social network through the usage of bots that penetrate this platform. These bots aim to imitate human behaviour as much as possible in order to fool **Twitter Users** into liking their generated content and even following and accepting to be followed by them. Through this process, the bots are able to gather more and more information since for each new follower/followee that they gain their network of connections grows larger. This process is specifically vital to allow the infiltration and information gathering of tweets passed around closed off sub-networks of protected accounts on the Twitter platform. A protected user is one whose information is only publicly available to the followers they allow. By having our bots be accepted by them, they'll be, inadvertently, gaining access to information that most other users don't get access to. By doing this, our system is then able to create a compendium of all of this information and connections observable on the network into an easy to digest, analyze and study format. Minerva gives its users the possibility to study the ever-growing growing network of Users and Tweets, as well as the relations between them. With this, we allow the possibility to follow and further study the way information flows and is disseminated through Twitter, one of nowadays main sources of information and media, consumed by millions of users worldwide at each ticking second.

We leave this project as a finished and polished product, ready to use. Our bots are deployed and actively scouring for meaningful information to store through the usage of deep machine learning models and natural language comprehension and processing algorithms. The web application makes the platform easy to use, allowing for the clean study of data through graphs and tables, the visualization and querying of the resulting network of connections in the form of a directed graph and through the usage of a robust query tool that allows even the less tech-savvy to create the most complex of queries, the straightforward management of each bots' behaviours through the definition of customizable policies and the generation of reports.

## 7.2 Future Work

Although the product is already ready to use and overall rather stable and robust, there is, of course, some room for improvement. One of our suggestions would be to create **Bots** who can perform more actions to mimic the human behavior even more realistically such as adding the capability to **answer private messages** or **create tweets** (rather than being limited to retweets and replies). For these efforts, more **machine learning** models would be required besides the ones we used.

It would also be a fortuitous to add the ability to create, deploy, pause and remove **Bots** automatically, since it would grant Minerva's end users more control over the bots' management.

There is also some room for improvement in terms of performance. Something we would have liked to have done would be migrating the statistical data to a system like **Cassandra**, which would make aggregation functions faster, and improve the overall performance of the statistical queries we have in our REST API.

In terms of frontend, although our form-based solution to allow less tech-savvy users to generate reports and query our network with as much control and freedom as possible, we have to admit that there is still a bit of a learning curve required to use these two features. Although not steep, and certainly helped by the presence of tooltips and a mini tutorial imbued in the corresponding pages themselves, a more graphical way of realizing queries would probably be something to consider to implement in the future. Something akin to the concepts of *Visual Programming*, if implemented in the right way, would retain the same complexity that we already offer, but grant a more appealing and user-friendly way to accomplish the same goals. Furthermore, some improvements could be made to the Network Visualization page, mostly in terms of performance. A way to quickly solve this would be to simply disable all physics. Although we, as a team, considered this options, we felt like it would lead to a poorer execution of the feature, with the removal of physics leading to the loss of a certain level of user interaction and visual stimuli.

Finally, there's the mobile application. In the end, due to time constraints and focus of our team's efforts on other parts of our system, we decided to cut this feature as it wasn't vital to our project. That's not to say that giving our users the ability to manage our bots or even check our tables of data on-the-go on a mobile application format wouldn't be interesting and something to consider in the future. It should, however, be stated that to counteract the lack of this companion app, we attempted to make our web application as responsive as possible for it to be easily utilized on any type of devices. Something else one might consider would be the usage of a PWA (Progressive Web App) as a quick way to convert the already existing web application into a downloadable mobile app.

While in this project we only created a way to mine and scour **Twitter data**, with some few tweaks and with the addition of some new features, it would be possible to adapt the work done to dig into some other social networks with the same degree of polish and quality, e.g. adapting this project to another, very famous, social platform like Instagram and follow the way posts are spread, and who it reaches.

# Bibliography


[1] *asyncio - Asynchronous I/O*. URL: https://docs.python.org/3/library/asyncio.html (cited on page 22).

[2] *Bot Sentinel - Dashboard*. URL: https://botsentinel.com/ (cited on page 4).

[3] *Data archive*. URL: https://about.twitter.com/en_us/advocacy/elections-integrity.html#data (cited on page 4).

[4] *Django introduction*. URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction (cited on page 14).

[5] *Docker*. URL: https://www.docker.com/ (cited on page 37).

[6] *ELIZA Chatbots*. URL: https://en.wikipedia.org/wiki/ELIZA (cited on page 14).

[7] Facebookresearch. *facebookresearch/ParlAI*. URL: https://github.com/facebookresearch/ParlAI/tree/master/projects/polyencoder// (cited on page 15).

[8] *GitHub Actions*. URL: https://github.com/features/actions (cited on page 34).

[9] *GitHub Workflow*. URL: https://guides.github.com/introduction/flow/ (cited on page 32).

[10] Samuel Humeau et al. "Poly-encoders:architectures and pre-trainingstrategies for fast and accurate multi-sentence scoring". In: Facebook AI Research. ICLR, March 2020. URL: https://arxiv.org/pdf/1905.01969.pdf (cited on page 15).

[11] *Keras*. URL: https://keras.io/ (cited on page 15).

[12] Thorin Klosowski. *What Is Tor and Should I Use It?* February 2014. URL: https://lifehacker.com/what-is-tor-and-should-i-use-it-1527891029 (cited on page 36).

[13] *Mixer's repository*. URL: https://github.com/klen/mixer (cited on page 33).

[14] *MongoDB*. URL: https://www.mongodb.com (cited on page 19).

[15] *Neo4J*. URL: https://neo4j.com (cited on page 17).

[16] *OAuth 1.0a - Twitter Developers*. URL: https://developer.twitter.com/en/docs/basics/authentication/oauth-1-0a (cited on page 19).

[17] *Olivia Taters, Robot Teenager*. URL: https://www.wnyc.org/story/29-olivia-taters-robot-teenager/ (cited on page 3).

[18] *ParlAI*. URL: https://parl.ai/about/ (cited on page 14).

[19] Alexa Pavliuc. *Watch six decade-long disinformation operations unfold in six minutes*. January 2020. URL: https://medium.com/swlh/watch-six-decade-long-disinformation-operations-unfold-in-six-minutes-5f69a7e75fb3 (cited on page 4).

[20] *Portainer*. URL: https://www.portainer.io/ (cited on page 38).

[21] *PostgreSQL*. URL: https://www.postgresql.org (cited on page 18).

[22] *RabbitMQ*. URL: https://www.rabbitmq.com/ (cited on page 16).

[23] *ReactJS Framework*. URL: https://reactjs.org/ (cited on page 13).

[24] *Recharts Library*. URL: https://recharts.org/en-US/ (cited on page 13).

[25] *redis*. URL: https://redis.io/ (cited on page 17).

[26] *Swarm mode overview*. June 2020. URL: https://docs.docker.com/engine/swarm/ (cited on page 38).

[27] *The Tor Project: Privacy & Freedom Online*. URL: https://www.torproject.org/ (cited on page 36).

[28] *TimescaleDB*. URL: https://www.timescale.com (cited on page 19).

[29] *Tweepy*. URL: https://www.tweepy.org/ (cited on page 19).

[30] *Twitter API*. URL: https://developer.twitter.com/en (cited on page 19).

[31] *Uber's React Vis Force*. URL: https://github.com/uber/react-vis-force (cited on page 13).

[32] *VisJS Network Documentation*. URL: https://visjs.github.io/vis-network/docs/network/ (cited on page 13).

[33] *Watchtower*. URL: https://containrrr.dev/watchtower/ (cited on page 38).

[34] *ZeroMQ*. URL: https://zeromq.org/ (cited on page 16).