Feeling: Sentiment Analysis on Twitter

Computer Science and Engineering

Bachelor's degree Thesis

Author: Sinclert Pérez Castaño

Directora: Lara Quijano Sánchez

Year: 2016 - 2017

Special thanks to:

Lara, my academic tutor for having patience with me and my irregular workflow.

My English-speaking friends for correcting my not very academic English grammar while being abroad in California.

My parents, for supporting me every single moment. The completion of this Bachelor's degree would not have been possible without them.

GENERAL INDEX:

Index of figures

Index of Equations

0. Abstract

Nowadays, social networks contain information that can be used to extract knowledge from it, however, the insights from that information are usually accessible to companies but not to individuals. The aim of this project is to provide an accessible web application that makes use of Machine Learning algorithms together with Twitter official's *API* to perform Sentiment Analysis over a set of tweets. Different natural language and feature engineering techniques have been applied to generate a range of trained models, including *Logistic Regression, Bernoulli Naïve Bayes, Support Vector Machines* and *Random Forest*. Finally, these models have been compared to one another and several conclusions were drawn: *Bernoulli Naïve Bayes* and *SVM* performed the best, even if resulting in a F-score larger than 0.9, without suffering overfitting, is extremely difficult.

Actualmente, las redes sociales contienen información que puede ser utilizada para extraer conocimiento, sin embargo, las conclusiones de esa información son normalmente accesibles solo a las empresas, y no a los individuos. El objetivo de este proyecto es presentar una aplicación web que haga uso de algoritmos de Aprendizaje Automático junto a la *API* oficial de Twitter para realizar análisis de sentimiento en conjuntos de *tweets*. Diferentes técnicas de lenguaje natural y de ingeniería de características han sido aplicadas para obtener un conjunto de modelos, incluyendo *Logistic Regression, Bernoulli Naïve Bayes, Support Vector Machines* y *Random Forest*. Finalmente, estos modelos han sido comparados entre ellos y varias conclusiones fueron obtenidas: *Bernoulli Naïve Bayes* y *SVM* son los modelos que mejor funcionan, considerando que obtener una *F-Score* superior a 0.9, sin sufrir sobreajuste, es extremadamente difícil.

1. Introduction

1.1. General overview

In the world we live, social networks are one of the main sources of personal information and interests across our daily lives. This is the reason why new sciences like "*Social Network Analysis*" and "*Sentiment analysis*" have become main fields of study and interest not only for the individual users but also for companies and governments.

To get an idea of the amount of the data social networks, such as Twitter, store every day, it is enough to consider that each second around 6,000 tweets are generated (according to *"Internet Live Stats"* [1]), which give us around 500 million tweets per day. All these data contain information about very different topics and nature, so differentiating or analyzing very specific pieces of information across all the available resources is a powerful tool.

Sentiment analysis is a technique that is widely used in areas of social analysis research such as marketing, social networking development, reviews and survey responses, and especially in customer service. It uses natural language processing and text analysis to determine how an individual or group of people feel towards a specific situation.

It can be also used to perform geolocation analysis such as political feelings (which political leader has better acceptance), marketing preferences (competitor's products opinion over population) or, in general, any statistical analysis within the specified parameter.

Some real-world examples could be the analysis of the *Brexit* sentiment specified U.K. cities, or the acceptance of the new president of the United States throughout the different states.

This type of analysis can be used in respect to the social network Twitter and its "*Application Programming Interface"* (API). Twitter's API is unique in the sense that it is easy to use, so anyone with basic programming knowledge can access large amounts of data in the form of tweets. Additionally, because of how people generally structure their tweets, most of them are solely comprised of plain text (unlike other social media platforms where users tend to include other media formats such as photos or videos), making them easier to analyze.

1.2. Project description

The main goal of this project is to provide a simple sentiment analysis tool that can be used jointly with the Twitter API to classify tweets as “*positive”, “negative”* or “*neutral”*.

The main differentiator factor with respect other Sentiment Analysis web application is that it is integrated with Twitter API, so automatic analysis over a certain account or a certain stream can be performed. Additionally, it is open to every individual user, not only to companies.

Although it was created relying on the previous introduced concepts, the project contains another important actor: Machine Learning (ML). ML includes a wide range of techniques that provides computers with the ability to learn without being explicitly programmed. Its use is not strictly necessary to perform a sentiment analysis task, because it could be done by simple human review, but for complex applications and tools in which large amounts of data are processed, it becomes the most common practice.

Machine learning plays a significant role in the data analysis processes because it helps to automate and generate a quicker response than the case of processing each piece of information (or tweet) individually by a person. In this project, ML techniques are used to classify tweets obtained through the Twitter API with one of the possible labels: *positive*, *negative* or *neutral*.

Obviously, the application of ML algorithms does not guarantee an error reduction with respect to the individual processing case, however, the advantages due to the time performance and effort improvements make it worth. In this project, several algorithms will be tested and compared with each other to understand why some models perform better than others and which features are the best ones to consider *(“feature engineering”).* Testing them will be one of the core sections of this memory because improving the ML predictions on unclassified data will suppose a general improvement in the behavior of the application.

Finally, as a way of making this project more accessible in case of using a web domain to host it, the project contains a web-application structure so it can be used thought any internet browser.

In conclusion, this bachelor’s thesis consists of a web application that uses trained ML models to perform a sentiment analysis over a set of Twitter data specified by the user. The trained models need to be generated before the classification task, using a series of data sets obtained from a Python package called *Natural Language Toolkit* (NLTK).

1.3. Memory structure

The memory document is structured as follows:

- Information about sentiment analysis papers and tutorials, why they are relevant, and some useful tools and ideas to apply in a project like this (point 2).
- A big overview of the main objective of the application (point 3).
- A detailed explanation about the component and technologies behind the application, why I have chosen those and how do they work (point 4).
- Testing and comparison of the different ML algorithm and cases with the obtained conclusions (points 5 and 6).
- A brief specification of the possible future work in case the work on this project is continued (point 7).

2. State of the Art

2.1. Sentiment Analysis

Sentiment analysis is a technique that determines the attitude of a person or group of people towards a certain product, situation or event. The premise is that all opinions can be classified in different groups (“*positive*”, “*negative*”, “*neutral*” …), each of them grouping several mood states. For example: sadness and anger belong to negative.

2.1.1. Type of problem

The main goal is to correctly classify the units of information (tweets) into one of the three aforementioned groups. It is considered a classification problem that can be approached using different methodologies, as this graph [2] shows:

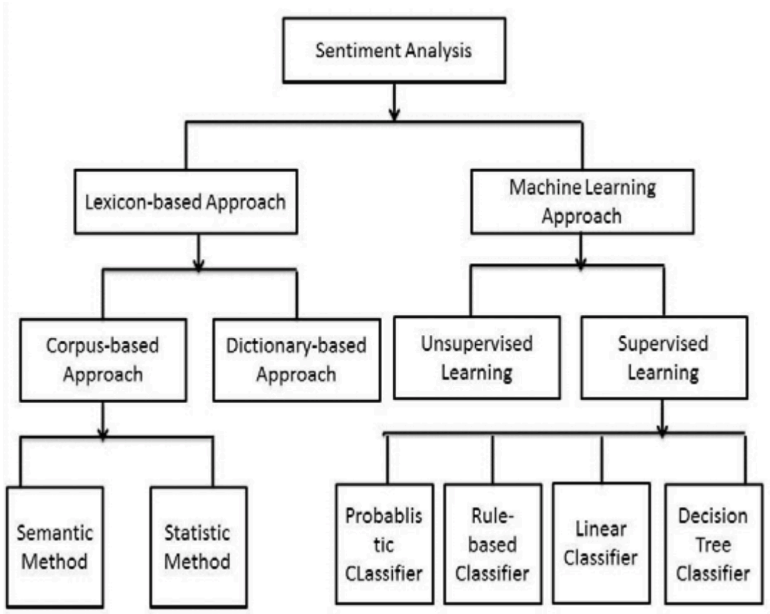

Figure 2.1: Sentiment Analysis approaches

This project focuses on the ML branch because it is the most commonly used and the most popular because of nowadays trend. Some of the most relevant ones are the *Naïve Bayes* (NB) and the *Support Vector Machine* (SVM) classifiers, under the “*Supervised Learning”* category, which will be explained and analyze later on.

This classification problem may also be included under the *“Unsupervised Learning”* category if there is no labeled data available. However, due to the amount of data not only in Twitter, but also in other social networks and web pages such as Amazon or eBay, it is not difficult to obtain datasets labeled with the *Likert scale* those e-commerce pages use. They can be found, for instance, in Kaggle.com.

The training data have been obtained from the NLTK corpus, where there are files containing neutral-polarized and negative-positive sentences. The downloaded files from the corpus are: *sentence_polarity* (containing 5,331 negative and 5,331 positive examples) and *subjectivity* (containing 5,000 neutral and 4,985 polarized examples). These examples are just sentences of different lengths that have been previously identified by their sentiment.

2.1.2. Main challenges

Probably the most difficult part of the classification process is to differentiate between subjective tweets, which can be classify as *positive* or *negative*, and objective ones, which are considered as *neutral*. The subjectivity usually depends on the context so the units of information should be understood in the context of a conversation or in a sequence of them if the talk about the same topic.

On top of that, the utilization of sarcasm while writing an opinion make extremely difficult for the classifiers to detect it. This would not be that important if the percentages of comments using sarcasm remain low, however, due to the Twitter length limitation to 140 characters, users usually write in more creative ways than in others social networks, increasing the use of it.

2.1.3. Accuracy issues

Due to the difficulties stated in the previous part, reaching an accuracy close to 100% is practically impossible. Even more, according to this sentiment analysis article [3], human judgement accuracy is around 79%. Because of this, if a specific software were right 100% of the times and we review the classified examples, we will still disagree with the program around 20% of the time.

These facts imply 2 things:

1. If a model, using any of the techniques of the *Figure 2.1*, achieve an accuracy rate higher than 79%, we could state that it is better than an average person trying to classify tweets into the polarity labels.

2. The complexity of achieving a high accuracy rate is extremely high, so the conclusions extracted from a sentiment analysis over a certain entity should always consider a margin of error.

2.2. Machine Learning algorithms

The algorithms used during the development of this project have been chosen due to their flexibility. These algorithms are:

1. Logistic Regression.
2. Bernoulli Naïve Bayes.
3. Linear Support Vector Machine.
4. Random Forest.

2.2.1. Logistic Regression

The idea behind this machine learning approach is to use a vector of weights called "θ" as long as the number of features. It could be described as a linear regression vector of weights, applying them into classification. Although this *discriminative* algorithm was designed to classification problems in which the number of possible classes is two, there exist an alternative for the multiclass problem *(Soft-Max)* [4][5].

The labels or classes are separated from each other using a hyperplane, which express the area of the features space with maximum confusion.

In order to obtain a classification result from the set of classes given a numeric result (multiplying the weights vector and each vector of features values), a *Logistic sigmoid* function is applied. This function computes the probability of a particular instance to be considered as one or the other class.

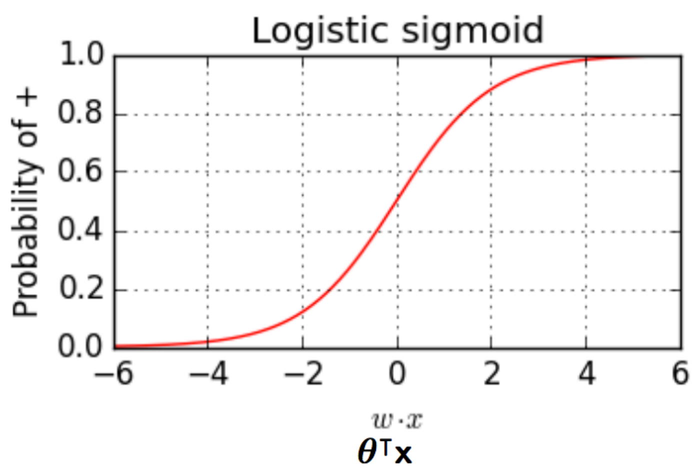

Figure 2.2: Logistic Regression

$$P(0 \mid \mathbf{x}; \boldsymbol{\theta}) = 1 - h_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{\exp(-\boldsymbol{\theta} \cdot \mathbf{x})}{1 + \exp(-\boldsymbol{\theta} \cdot \mathbf{x})} = \frac{1}{1 + \exp(\boldsymbol{\theta} \cdot \mathbf{x})}$$

Equation 2.1: Logistic Regression

2.2.2. Bernoulli Naïve Bayes

There exist a group of ML algorithms that rely on probabilities. All of them are possible thanks to the "*Bayes Rule"* (explained later), in which knowledge can be obtain from a set of available labeled instances.

Inside the family of pure probabilistic approaches, there are two main models: the *Gaussian* and the *Bernoulli*. The different between both of them is that the first one is considered a *generative* model, predicting a specific feature value (X_i) from the class that example belongs to (y); and the second one is a *discriminative* model, in which the class is predicted using the vector of features [4][5].

Considering the type of classification task in which the project is based on, the only useful approach is the discriminative one (*Bernoulli*), because we will try to predict the sentiment feeling (class) of a text.

The Bernoulli Naïve Bayes, as its own name indicates, makes the naïve assumption of supposing that each feature is independent to the rest. Additionally, it uses *Bayes rule* to compute the probability of each class:

$$P(y|x) = \frac{P(x|y) * P(y)}{P(x)}$$

Equation 2.2: Bayes rule

Where:

- P(y|x) = the probability of belonging to class *"y"* giving the features *"x".*
- P(x|y) = the probability of features *"x"* given *"y"*.
- P(y) = the total probability of the class *"y"*.
- P(x) = the probability of the vector *"x"* (ignorable because is a global constant).

2.2.3. Linear Support Vector Machine

The Support Vector Machines (SVM) are one of the most powerful ML algorithms due to their flexibility and good optimization. Additionally, thanks to the "kernel trick" technique in which more features are created almost free to expand the feature space, the classification errors are often reduced [4].

The basic idea is to create a linear boundary between examples belonging to different classes with the biggest possible gap (δ) between the closest examples to that line (quadratic problem). Those points are called "*support vectors"* and they give the name to the model.

As Logistic Regression, it also uses a vector of weights (*"w"*), following these formulas in the case of binary classification:

A) When the class is 1: $w * x_i + b \geq \delta$

Equation 2.3: SVM (A)

B) When the class is -1: $w * x_i + b \leq -\delta$

Equation 2.4: SVM (B)

Due to some mathematical optimizations, it is possible to define that vector of weights as a weighted sum of support vectors. In conclusion, the linear boundary only depends on the closest examples to the itself, which are the ones defining the supporting vectors.

Finally, the "Linear" nature of the SVM that I am describing is not because of the linear boundary, but because of how the *"kernel trick"* function is defined. This function is just a way of creating new features with the ones we already have and expand the features space so the boundary can easily differentiate among the classes [6].

Example: we want to classify the following examples:

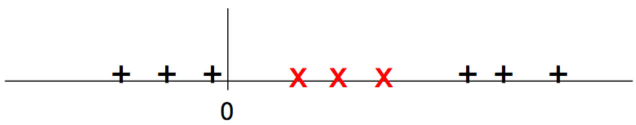

Figure 2.3: Examples distributed over space

However, there is no clear linear boundary, so a "*kernel trick*" should be used to add 1 more dimension to the features:

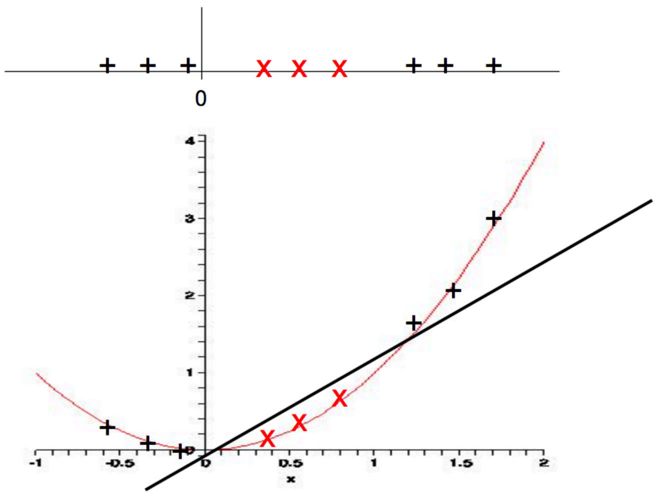

Figure 2.4: SVM kernel trick

Due to the new feature space, a linear boundary can be easily found.

2.2.4. Random Forest

The last considered ML algorithm in this project is based on decision trees. These trees are just connected nodes in which one of the features is evaluated, and depending on the result, different branches are taken. At the very end of the tree there are leaves, which are nodes containing the designated class [4][5].

The criteria to create the tree depends on the defined *impurity* function, which is going to determine when a specific node must be split into different ones and when that node is a leave. There exist several impurity functions for the binary classification problem:

Given ($p = n_+/n$), and ($q = n_-/n$), where "+" and "-" are the two classes:

- Gini index: $$2 \cdot p \cdot q$$

Equation 2.5: Gini index

- Entropy: $$(-p \cdot log\{p\}) - (q \cdot log\{q\})$$

Equation 2.6: Entropy

- Error rate: $$1 - \max[p, q]$$

Equation 2.7: Error rate

The main problem with decision trees is that, if no additional technique is applied when creating them, they tend to overfit. That is why popular approaches as pruning and trees combinations (Random Forest) started appearing.

A Random Forest is just a combination of decision trees, in which for each of them a small random features subset is used. Then the outputs of all these trees are combined to select the most popular class, a technique known as *bagging*.

2.3. Related papers, books and tutorials

From the set of public papers, books and sentiment analysis tutorials available on the subject, the ones that proved to be the most useful throughout the project are the following:

- Twitter Developer Documentation [7]:
  This is the official Twitter documentation about how their Application Programming Interfaces (APIs) work. There is information not only about how to make requests, but possible obtained data after each type of request.

  Some of the most important data fields composing each tweet are:

  - Coordinates
  - Creation date.
  - Entities (hashtags, URLs and user mentions).
  - Favorite counter.
  - Retweet counter.
  - ID.
  - Language.
  - Place.
  - Possible sensitivity.
  - Text.
  - User.

  However, there exist some statistics that can be obtained using the Twitter interface which cannot be obtained using its API: the number of views, and the number of engagements. Both would have been helpful to determine the importance of the retrieved tweets.

- *Twitter Sentiment Analysis: A Review* [3]:
  This paper contains a general overview about the use of sentiment analysis with Twitter using ML algorithms to predict the labels. It also contains a brief explanation of the feature selection process: case normalization, tokenize, *stop words* (those without any important significance), stemming…

- *Sentiment Analysis of Twitter Data* [8]:
  This one contains information about another sentiment analysis study in which they compare the use of different features and different algorithms to compare the results and extract conclusions. It is relevant because the use of Part of Speech (POS) tags is well explained.

- *The Importance of Neutral Examples for Learning Sentiment* [9]:
  Paper that contains a detailed explanation about why the *neutral* class cannot be ignored when doing a sentiment analysis.

- Machine learning and Text Mining slides [5]:
  Slides that contains information about ML algorithms and how they work, along with concepts descriptions. In terms of Text Mining, explanations and examples.

- Sentiment Analysis in Python [10]:
  Online tutorial that describe the process of training a ML model in the Python programming language, along with example of feature selection. It also contains some links to other resources and tutorials in which this one is based on, that were useful during the development of the project.

- Twitter Sentiment Analysis with NLTK [11]:
  Online tutorial that explains in a series of videos with supporting code examples the different possibilities of NLTK and the learning algorithms available inside the NLTK package. Although none of the NLTK algorithms were used in this project, the videos were helpful to understand some of the package capabilities.

2.4. Possible tools

There exist a large list of possible tools and technologies that are publicly available to perform a ML approach into the sentiment analysis of Twitter data. Each of these classification process stages is going to be considered independently in order to explain the different options in each of the cases.

2.4.1. Accessing Twitter API

In this case, there are not a lot of options because there exist only one official Twitter API. However, there exist several libraries and packages across different programming languages to make our communication with the API easier.

There exist libraries for several programming languages such as Java, Python, Objective-C, C++, Go, PHP and Ruby, although only Java and Python were considered at the end. The full list can be consulted in the Twitter developer's documentation, under the *Twitter libraries* category [12].

These libraries allow developers to access a wide list of information fields associated to each individual tweet, making possible to create statistic analytical tools as the one described here. Some of the information fields are: *coordinates, creation date, entities, number of favorites, number of retweets, language, place, possible sensitive, user…*

2.4.2. Analyzing data

The following pieces of software can help on two different processes: cleaning the tweets obtained from the API, and processing the training examples using Natural Language Processing (NLP) techniques to select the most informative features that the ML models will use to learn later.

- Python *Natural Language Toolkit (NLTK)* package.
- Java *Stanford CoreNLP* library.

2.4.3. Learning ML models

After the data analysis, a model must be trained to predict the classification label of future tweets. There is a big range of options, because even the NLTK contains its own classification module, but there exist more specialized libraries such as:

- Java *Weka* library.
- Java *General Architecture for Text Engineering (GATE)* library.
- Python *Scikit-learn* package.
- Software *Rapid-Miner*.

2.4.4. Web frameworks

Finally, with the aim of providing web characteristics, a framework to connect the Back-end consisting on the trained model and the Front-end was required. The libraries listed below were considered for their simplicity since the project's web behaviors does not required complex functionalities. They are all Python frameworks:

- Python Django
- Python Web2py
- Python Flask

2.4.5. Final decision

Due to the extended documentation of "*Tweepy" (3.5 version)*, "*NLTK" (3.2 version)* and "*Scikit-learn" (0.18 version)* packages, the Python programming language was used to implement the project combining the functionalities of these packages. Moreover, they have open source licenses so they can be used to academic projects but not commercial software.

Additionally, "*Flask" (0.12 version)* micro framework was used to create the web infrastructure. This framework is preferable over *Django* or *Web2Py* because it provides more direct control over the project structure and communication.

All these packages provide already implemented functions that make the development process easier, because they provide a solid base in which start building the logic that will shape the functionality of the software. Moreover, there are some of the *Scikit-learn* algorithms that have been incredibly optimized using *C* code, and a possible pure Python implementation will almost certainly be slower.

During the development process *PyCharm*, which is a very complete development environment, was used along with a *GitHub* repository, where all the changes have been submitted. Thanks to the version control in this last platform, the code can be diverged in different work flows called "branches", that will be necessary in the future (further explanation in section 4.1).

3. Project overview

This overview contains explanations about the project goals, how the project classification challenge has been approached and its general steps. There is a need for explaining the chosen approach because it does not follow the standard classification procedure.

3.1. Main objectives

The main objective is to build a software tool which performs sentiment analysis over pieces of information in the shape of tweets, and that it is accessible to everyone. Accessible meaning a free, and easy to use and web application (once the project is hosted) that any individual can use without previous knowledge.

A good example of its usage is to determine the feeling of citizens in a certain city over a specific political party, brand, law…

In order to achieve that functionality, a Twitter API text mining implementation for tweets extraction is required. The retrieved data can be used either to perform the analysis specified by the user or to expand the datasets used in the feature engineering process and in the ML models training.

After the data has been retrieved, the goal was to build a feature engineering function to get the most relevant features of each piece of information. This is a crucial step in the project because feature selection has a big effect in the final score of ML classifiers.

Additionally, obtaining an insight in the algorithms performance in the case of text classification was one of the goals. It is possible thanks to the use of ML algorithms such as *Logistic Regression*, *Naïve Bayes*, *Support Vector Machines,* and *Random Forest* as well as the manual modification of their parameters.

In terms of software, the project was designed to embody the following characteristics:

- Accessible: any individual with internet connection can use it.
- Effective: as much as possible due to the M.L. classification errors.
- Fast: it needed to be fast enough to later be transformed into a web application.
- Easy to use: no previous knowledge required.
- Scalable: if *user authentication* is implemented, API limitations can be avoided.

3.2. Classification description

When considering classification problems, the most popular technique to approach them is by discriminative models. These models predict the class (*y*) of a new instance (*x*). Some of the models use a probabilistic approach (*Naïve Bayes*) and some others just a linear boundary (*Logistic Regression* or *Support Vector Machines*), both of them have the same goal: to predict the correct class (or label).

In Sentiment Analysis, three possible classes exist: "*neutral*", "*negative*" and "*positive*", however, not all of them have the same relationship among each other. Although the labels "*negative*" and "positive" have an intrinsic relationship in which one is the opposite of the other, that is not the case with *"neutral"*. When discussing text classification, the opposite to *"neutral"* is *"polarized"*.

Due to the unbalanced relationship among the possible classes, the method to classify the tweets is not going to be a common classification over three labels. Instead, it will be a **hierarchical classification** over two possible labels (*"neutral"* and *"polarized"*), and depending on the outcome, between the other pair of them.

Therefore, the classification process is as follows:

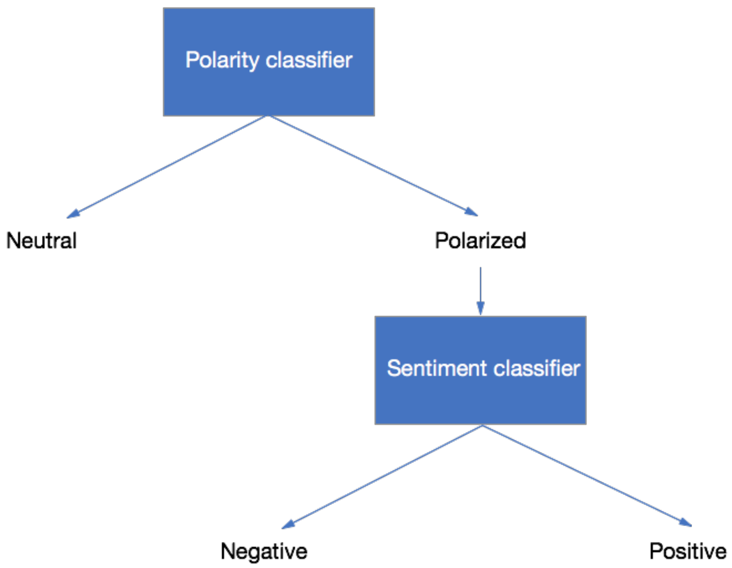

Figure 3.1: Hierarchical classification

This way, each label shares the same hierarchical level as its opposite and the classification process is more balanced. The main problem when including a hierarchical structure is that the errors from the first classifier affect the second one.

3.3. Classification scheme

In order to have a global overview of the classification scheme, the taken sequence of steps are shown:

1. Open the data files containing the raw sentences expressing sentiment. They have been obtained using the NLTK corpus, and the sentences are organized in 4 different text files: *Neutral, Polarized, Negative* and *Positive*.

Neutral example:	*“There has been an attack in London, according to CNN”*
Polarized example:	“*I believe they can win the next championship*”
Negative example:	*“Cleveland could have done more! so sad”*
Positive example:	*“Morning! today is a nice day in the bay!”*

2. Clear the sentences: tokenize them, converting all the words to lower case; remove the stop words (list of non-informative words obtained from the NLTK corpus to filter the sentences); and lemmatize the remaining ones. After doing this, we can obtain a list of words and a list of *bigrams* (sequences of 2 words).

Clean neutral:	[*“there”, “attack”, “London”, “according”, “CNN”*]
Clean polarized:	[*“believe”, “win”, “championship”]*
Clean negative:	[*“Cleveland, “more”, “sad”*]
Clean positive:	[*“morning”, “nice”, “day”, “bay”, “nice day”*]

3. Feature selection using NLTK functionalities. The best results have been obtained using the *Chi-Square* distribution to get the scores of words and bigrams, and selecting the best of them.

Informative words	[*… , “there”, “according”, “believe”, “sad”, “nice”, ...*]
Informative bigram:	[*... , “nice day”, ...*]

4. Transform the dictionary of features into a vector using *Scikit-learn* modules.

Neutral features:	{*… , “there”: 1, “according”: 1 , …*]
Polarized features:	{... , “believe”: 1, ... }
Negative features:	{... , “sad”: 1, ... }
Positive features:	{... , “nice”: 1, “nice day”: 1, ... }

5. Train the different models with pairs of training examples: Neutral – Polarize, and Negative – Positive. After comparing the results (in section 5.4), the model with higher combined accuracy is selected and stored in the *Models* folder.

6. On the web part, tweets are obtained using the Twitter API, and they are cleaned and vectorized, just as the data files sentences.

7. Finally, the best models are loaded into memory and the tweets are classified, both in the case of a user account and the streaming tweets.

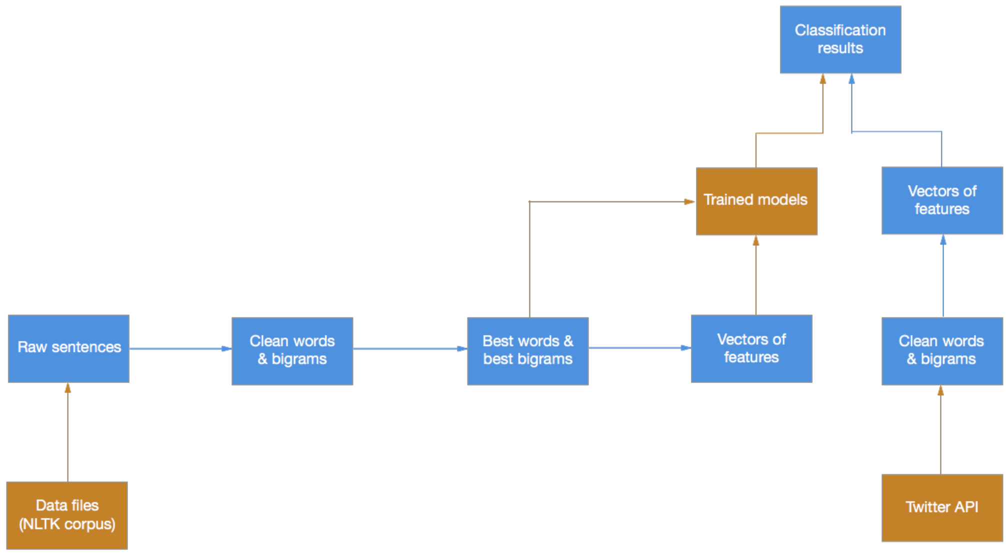

Figure 3.2: Classification scheme

4. Developed work

4.1. Introduction

The project was started as a local Python program in order to test the viability of the initial idea. Later, the code was separated into two different branches using *GitHub* to improve version control.

The first branch has a local scope, in the sense that it is thought to be executed in the local machine of the user that wants to operate with it, instead of using the software through the browser as a web application. The possible functionalities that are available in this branch are the following:

- Train models: L.R. / Bernoulli N.B. / Linear S.V.M. / R.F.
- Search for tweets to increase the size of the training datasets.
- Classify tweets of a user filtered by a word as neutral, positive or negative.
- Visualize a stream of tweets filtered by word and coordinates.

The second branch is a modification of the first one, in which other files such as the HTML, CSS and JavaScript have been included. Once the program is executed simulating a hosted web app, the functionalities are reduced to the following two:

- Classify tweets of a user filtered by a word as neutral, positive or negative.
- Visualize a stream of tweets filtered by word and location.

The reason why the first branch is relevant to the project and it has been kept instead of having just the second branch, is because it can be used to generate the models that can be moved to the second branch folder structure and use them. Additionally, almost all the debugging over the text processing, features selection, and model's accuracy comparison, have been done with that code.

From the final user perspective, only the second branch is useful once the best possible model has been selected and generated, because is the one containing the web functionalities, however both are important.

The project can be found in this *GitHub* URL: https://github.com/Sinclert/SentimentAI

4.2. Application architecture

4.2.1. Master branch

The main branch is structured into the following files and folders:

A) Files:

- Classifier: contains a class with the functions to train, obtain the features of a text, save and load the models.
- DataMiner: contains the class with the functions to obtain tweets from a specified user or in general, filtering with some parameters.
- GraphAnimator: contains a function to display the results of the stream.
- Keys: contains the app and administrator (me) keys to access the API.
- Parser: contains the functions to parse the user arguments and execute the indicated functionality.
- TwitterListener: contains a class with the methods to initiate the stream, handle the obtained tweets and close the stream.
- Utilities: contains several functions that are useful across different functionalities and files.

B) Folders:

- Datasets: contains the data files used to train the models.
- Models: is the folder where the trained models are going to be stored.
- Stopwords: contains the files with the *stop words* in different languages.

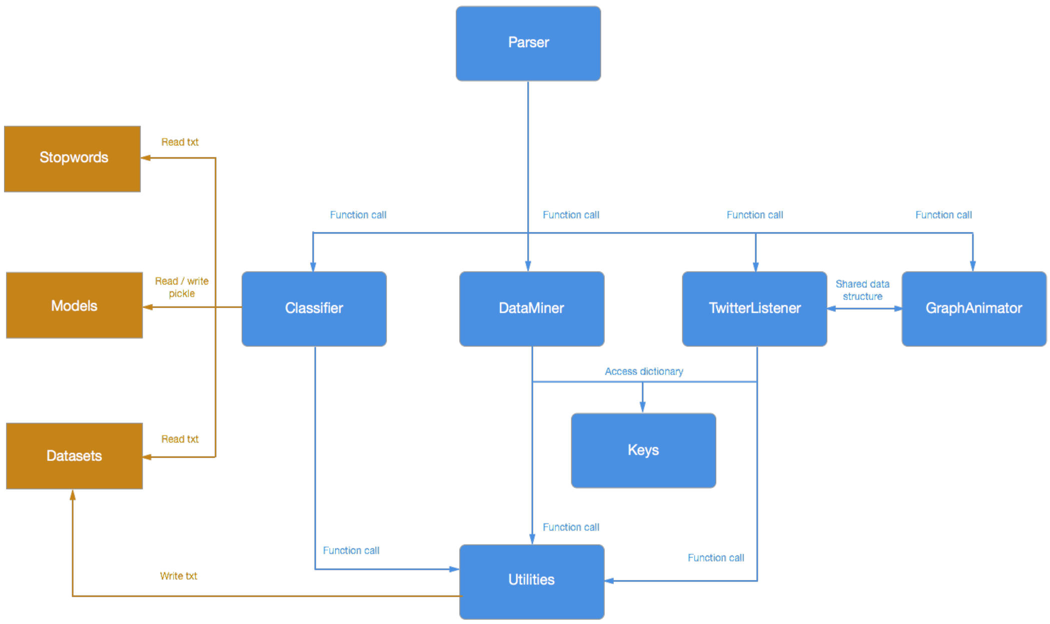

Figure 4.1: Main branch structure

4.2.2. Server branch

The server branch is structured into the following files and folders:

A) Files:

- Classifier: contains a class with the functions to train, obtain the features of a text, save and load the models.
- DataMiner: contains the class with the functions to obtain tweets from a specified user or in general, filtering with some parameters.
- Keys: contains the app and administrator (me) keys to access the API.
- Parser: contains the functions to parse the user arguments and execute the indicated functionality.
- Server: contains the associated functions to web directions.
- TwitterListener: contains a class with the methods to initiate the stream, handle the obtained tweets and close the stream.
- Utilities: contains several functions that are useful across different functionalities and files.

B) Folders:

- Datasets: contains the data files used to train the models.
- Models: is the folder where the trained models are going to be stored.
- Stopwords: contains the files with the *stop words* in different languages.
- Static: contains the static web files such as CSS, JavaScript, images…
- Templates: contains the HTML files.

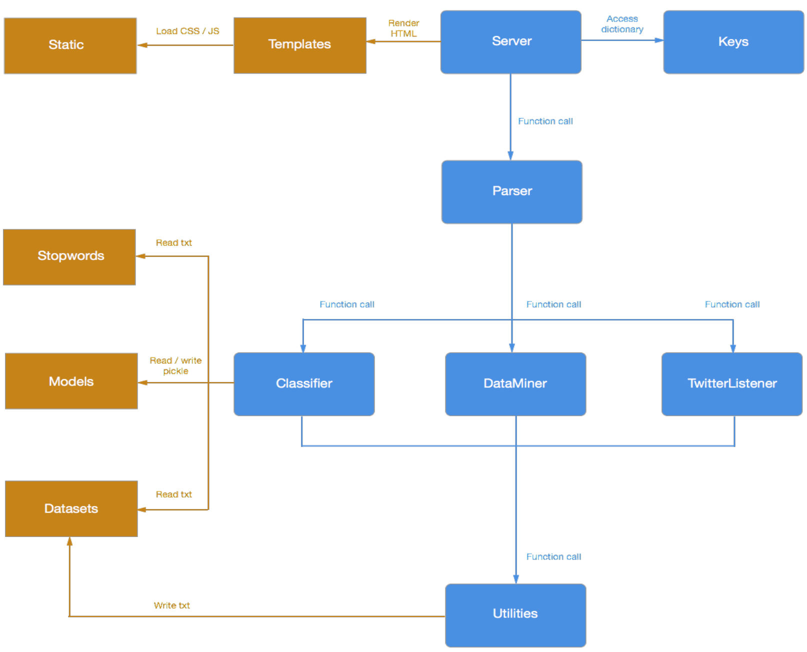

Figure 4.2: Server branch structure

4.3. Knowledge model

In this section, every function in the code is described. However, those that belong to a specific class are going to be covered under their class definition.

4.3.1. Master branch

CLASSIFIER	
Type:	Python - Class
Description:	Represents a classification model.
Parameters:	• Tokenizer: object that separate words in a text. • Vectorizer: object that transform features into a vector. • Lemmatizer: object that obtain words lemma (root). • Stopwords: list of stop words of the given language. • Model: object containing the trained model. • Best_words: list of most informative words. • Best_bigrams: list of most informative bigrams.
Functions:	• getWords: obtains a list of filtered words in a sentence. • getWordsAndBigrams: obtains a list of all filtered words and bigrams in the specified file. • getFeatures: obtains a dict. of features given a sentence. • performTraining: trains the model and prints its accuracy. • train: obtain the training data after several function calls. • classify: classify a sentence after obtaining its features. • saveModel: save the model in a *pickle* file • loadModel: load a model from a *pickle* file

DATA MINER	
Type:	Python - Class
Description:	Mining object that extract data from Twitter API.
Parameters:	• API: object that contains the API connection to Twitter.
Functions:	• getUserTweets: gets user tweets, optionally filtered. • searchTweets: stores in a text file the tweets that fulfill the specified query conditions.

GRAPH ANIMATOR – ANIMATE PIE CHART	
Type:	Python - Function
Description:	Graph generator to visualize the results from the Twitter listener streaming, using *Matplotlib*.

PARSER	
Type:	Python - Executable file
Description:	Executable containing the argument parser for the different project functionalities.
Parameters	• Dataset folder: folder name where the data is stored. • Models folder: folder name where the models are stored. • Labels: labels among which classification is performed.
Functionalities:	After parsing the received argument, select one of the following: • Train: trains and saves a model. • Classify: prints the classifications results after loading a trained model. • Search: extracts tweets and stores them into a text file. • Stream: starts a streaming sharing a data structure with the *GraphAnimator* function.

TWITTER LISTENER	
Type:	Python - Class
Description:	Object taking control of each Twitter streaming API.
Parameters:	• API: object that contains the API connection to Twitter.
Functions:	• updateBuffers: updates a circular buffer that store the last "n" classifications results. • initStream: initiates the stream with the provided args. • closeStream: closest the stream. • on_status: process each received tweet (overrides). • on_error: handles any error during execution (overrides).

UTILITIES – GET STOP WORDS	
Type:	Python - Function
Description:	It gets all the stop words defined in the specified language text file inside the "*Stopwords*" folder, and return them as a list.

UTILITIES – GET BEST ELEMENTS	
Type:	Python - Function
Description:	It performs a statistical distribution analysis using the *Chi-square* test to select the most informative features given two sets of features in the form of Counters.

UTILITIES – GET CLEAN TWEET	
Type:	Python - Function
Description:	It obtains the text from a tweet object, without all the entities that are not considered text (user names, hashtags, links…)

UTILITIES – GET SENTENCES	
Type:	Python - Function
Description:	It split a given text into sentences returning them as a list.

UTILITIES – STORE TWEETS	
Type:	Python - Function
Description:	It stores the list of tweets received as argument in the specified text file.

4.3.2. Server branch

This section contains the description of the new functions introduced in this branch of the code. Some of them are not included because they are exactly the same as the ones in the main branch.

All the Back-End functionalities are inside the *Server* file. Additionally, there is a web structure composed by a HTML, a CSS and a JavaScript (JS) file.

GRAPH.JS	
Type:	JavaScript
Description:	Contains the JS functions used to animate the web page and communicate with the Python Back-end.
Parameters:	None.
Functions:	• ready: links JS functions to web page buttons. • toogleAccount: hides the "*stream*" lateral bar section, showing the "*account*" one. • toogleStream: hides the "*account*" lateral bar section, showing the "*stream*" one. • showAbout: shows a section in the middle of the web page containing information about the author. • hideAbout: hides the section in the middle of the page. • accountRequest: performs a HTTP request to Back-End, showing a loading wheel while waiting. • streamRequest: performs a HTTP request to Back-End to initiate the stream, and then one each 750 ms. • showLoading: shows the loading wheel. • hideLoading: hides the loading wheel. • getCoordinates: gets coordinates of the location the user has specified using Google geocode API. • finishStream: stops the stream sending a HTTP request. • accountHandler: updates the pie with the response. • streamHandler: updates the pie graph with the response. • updatePie: updates the pie graph with the new counters. • createPie: create a pie graph in case of not having one. • showError: shows and error in the center of the screen. • getSum: returns the sum of a number array.

SERVER	
Type:	Python – Back end
Description:	Contains the different Flask functions callable from the front end (through HTTP requests).
Parameters:	None.
Functions:	• index: loads the classifiers and renders the *index.html*. • classifyAccount: classifies the tweets contained the filtering word of the specified user, returning the counters as a *json*. • initStreaming: initiates a Twitter streaming. • endStreaming: finishes the Twitter streaming. • streaming: returns as *json* the results of the Twitter streaming buffer (classification counters). • getAPI: generates an API connection with Twitter. It can be application or user type.

4.4. Execution dynamics

All the previously detailed functions need to work together to provide some functionalities. All the following ones are available to the developer behind the project, but only two of them (*"classify"* and *"Streaming"*) are accessible to the final users through the web application.

4.4.1. Train functionality

In this functionality, the input strings are parsed using the *Parser.py* file, creating a Classifier object, and calling:

1. Method *train*:
    - Inputs:
        - Classifier name (string).
        - First dataset name (string).
        - Second dataset name (string).
        - Percentage of words to use as features (integer).
        - Percentage of bigrams to use as features (integer).
    - Outputs: -
    - Calls:

    1.1. Method *getWordsAndBigrams* (x2):
    - Inputs: dataset path (string).
    - Outputs:
        - Dataset sentences (list of strings).
        - Dataset words (counter of strings).
        - Dataset bigrams (counter of strings).
    - Calls:

        1.1.1. Method *getWords*:
        - Inputs: sentence (string).
        - Outputs: sentence words (list of strings).

    1.2. Method *getBestElements*:
    - Inputs:
        - First dataset elements (counter of strings).
        - Second dataset elements (counter of strings).
        - Percentage of them to keep (integer).
    - Outputs: final features (set of strings).

1.3. Method *getFeatures* (x2):

- Inputs: sentence (string).
- Outputs: features (dictionary of words and bigrams).
- Calls:

1.3.1. Method *getWords*:

- Inputs: sentence (string).
- Outputs: sentence words (list of strings).

1.4. Method *performTraining*:

- Inputs:
  - Classifier name (string).
  - Features (sparse numpy matrix).
  - Labels (numpy array).
- Outputs: -

2. Method *saveModel*:

- Inputs:
  - Models folder name (string).
  - Output model name (string).
- Outputs: saved "*pickle*" file.

4.4.2. Search functionality

In this functionality, the input strings are parsed using the *Parser.py* file, creating a DataMiner object, and calling:

1. Method *searchTweets*:

- Inputs:
  - Filter query (string).
  - Language (string).
  - Storing file path (string).
  - Depth (integer).
- Outputs: -
- Calls:

1.1. Method *getCleanTweet*:

- Inputs: tweet (object).
- Outputs: clean tweet text (string).

4.4.3. Classify functionality

In this functionality, the input strings are parsed using the *Parser.py* file, creating a DataMiner object, and calling:

1. Method *getAPI* (only in Server branch):
    - Inputs: connection type (string).
    - Outputs: Twitter API (object).
2. Method *getUserTweets*:
    - Inputs:
        - User name (string).
        - Filter word (string).
        - Depth (integer).
    - Outputs: tweets (list of strings).
    - Calls:

        1.1. Method getCleanTweet:
        - Inputs: tweet (object).
        - Outputs: clean tweet text (string).
3. Method *getSentences*:
    - Inputs:
        - Tweets (list of strings).
        - Filter word (string).
    - Outputs: sentences (list of strings).
    - Calls: *getSentences* (recursive).
4. Method *classify* (x2):
    - Inputs: sentence (string).
    - Outputs: predicted label (string).
    - Calls:

        3.1. Method *getFeatures* (x2):
        - Inputs: sentence (string).
        - Outputs: features (dictionary of words and bigrams).
        - Calls:

            3.1.1. Method *getWords*:
            - Inputs: sentence (string).
            - Outputs: sentence words (list of strings).

4.4.4. Streaming functionality

In this functionality, the input strings are parsed using the *Parser.py* file, creating two Classifier objects, and calling:

1. Method *getAPI* (only in Server branch):
   - Inputs: connection type (string).
   - Outputs: Twitter API (object).
2. Method *loadModel* (x2):
   - Inputs:
     - Models folder (string).
     - Model name (string).
   - Outputs: -
3. Method *initStream*:
   - Inputs:
     - Query (string).
     - Language (string).
     - Coordinates (string of four floats separated by commas).
   - Outputs: -
   - Set listeners:

     2.1. Method *on_status*:
     - Inputs: tweet (object)
     - Outputs: -
     - Calls:

       2.1.1. Method *getCleanTweet*:
       - Inputs: tweet (object).
       - Outputs: clean tweet text (string).

       2.1.2. Method *updateBuffers*:
       - Inputs: label (string).
       - Outputs: -

     2.2. Method *on_error*:
     - Inputs: tweet (object).
     - Outputs: -
4. Method *animatePieChart* (only in Master branch):
   - Inputs:
     - Labels (list of strings).
     - Tracks (string).
     - Shared dictionary (dictionary).
   - Outputs: -

4.5. Users guide

From the point of view of the user, the interaction with the project is going to be through a web browser. In order to make this possible, all the project files need to be hosted by a service such as, *Google Cloud* or *Heroku*. Even if the final deployment in a hosting service does not take place, this guide will still be useful once the project is deployed.

The main and single page of the web application is like this:

Figure 4.3: Web app interface

The interface is divided in the following sections:

1. Header: shows information about the tool. The title and a brief description of what the application does.
2. Options: a pair of buttons to select the functionality of the tool. Each of them requires different input fields to be fulfilled by the user.
   A) Account: classify the tweets from an account.
   B) Streaming: classify live tweets from a location.
3. Input fields: contains the fields the user needs to provide to start the analysis. They change depending on which functionality is selected.
4. Content: shows the pie chart once it is generated.
5. About: shows contact information.

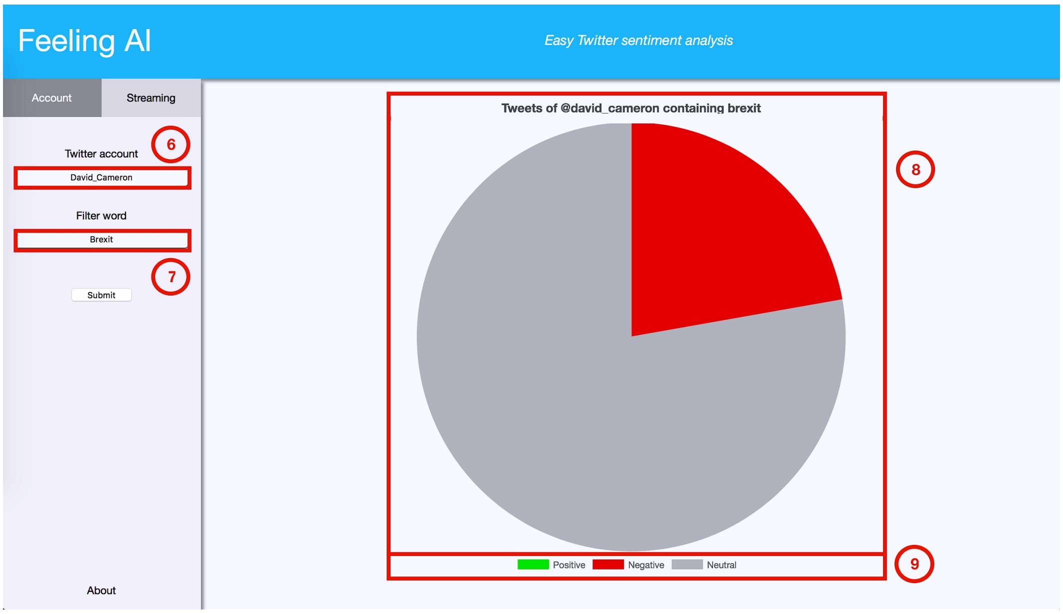

Figure 4.4: Account analysis interface

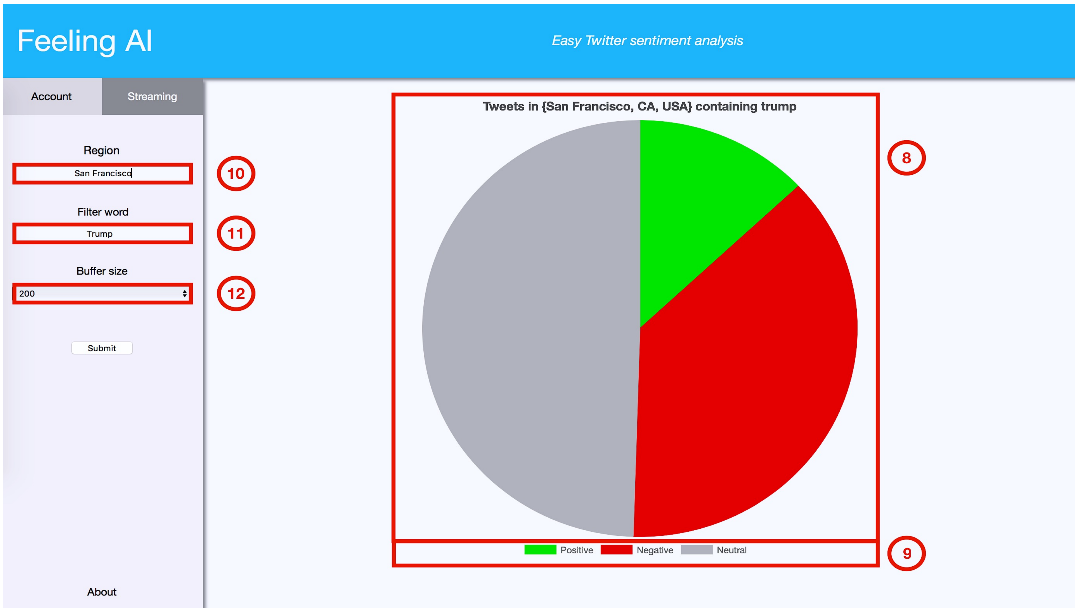

Figure 4.5: Stream analysis interface

Where:

6. Account field: where the Twitter account is specified (without the "@" symbol).
7. Filter field: where the word used to filter the tweets is specified.
8. Graph: it contains the classification categories represented as a pie chart.
9. Graph legend: interactive legend whose labels can be removed or added.
10. Region field: where the city or region to filter the stream of tweets is specified.
11. Filter field: where the word used to filter the tweets is specified.
12. Buffer size: where the number of most recent tweets to consider is specified.

4.6. Developer guide

This guide is designed for developers working with this project in the future. It contains information about how to execute both branches and system requirements.

4.6.1. Execution commands

The Master branch is designed to be executed locally to refined the feature selection and ML model. Additionally, it could be used to increase the training dataset by searching tweets, and check the functionalities of the web-application, which are: classify account tweets and classify tweets from a stream.

The way to execute this project branch is with the *"Parser.py"* file:

```
$> python Parser.py <Functionality> <List of arguments>
```

Where the list of functionalities with their group of arguments are:

1. Train a Machine Learning model:
   Functionality: "Train"
   Arguments:
   A) Classifier: *"Logistic-Regression"*, *"Naive-Bayes"*, *"Linear-SVM"* or *"Random-Forest"*.
   B) File 1: name of the first file inside *"Datasets"* folder used to train.
   C) File 2: name of the second file inside *"Datasets"* folder used to train.
   D) Words percentage: percentage of total words used as features.
   E) Bigram percentage: percentage of total bigrams used as features.
   F) Output: name of the output classifier to store in *"Models"*.

   Example:

```
$> … Train Logistic-Regression Positive.txt Negative.txt 5 1 Pos-Neg
```

2. Search for Tweets:
   Functionality: "Search"
   Arguments:
   A) Query: sequence of words to filter the search.
   B) Language: language in which the retrieved tweets are.
   C) Depth: number of tweets that we want to retrieve.
   D) Output: name of the output dataset to store in *"Datasets"*.

   Example:

```
$> … Search "#optimistic OR #happy" en 1000 PosTweets.txt
```

3. Classify tweets from an account:

   Functionality: "Classify"

   Arguments:

   A) Polarity model: name of the model to differentiate *neutral* and *polarized* sentences, stored in *"Models"* folder.
   B) Sentiment model: name of the model to differentiate Positive and Negative sentences, stored in *"Models"* folder.
   C) Account: Twitter account whose tweets are going to be classified.
   D) Filter word: word or sequence of words used to filter the tweets.

   Example:

```
$> … Classify Neu-Pol Pos-Neg David_Cameron Brexit
```

4. Classify tweets from a stream:

   Functionality: "Stream"

   Arguments:

   A) Polarity model: name of the model to differentiate *neutral* and *polarized* sentences, stored in *"Models"* folder.
   B) Sentiment model: name of the model to differentiate Positive and Negative sentences, stored in *"Models"* folder.
   C) Buffer size: number of most recent tweets to consider in the chart.
   D) Filter word: word or sequence of words used to filter the tweets.
   E) Language: language in which the tweets are filtered.
   F) Coordinates: sequence of 4 float numbers separated by commas that indicates the southwest and the northeast corners of the region delimiting where the tweets are retrieve.

   Example:

```
$> … Stream Neu-Pol Pos-Neg 500 Obama en -122,36,-121,38
```

Even if the provided implementation does not use all the possible arguments a Twitter Stream can take, the whole list of them can be checked in the official Twitter documentation [13].

As for the Server branch, the python file to run is called *"Server.py"* with no arguments. Note that it may take a few seconds before the server is operational.

The way to execute this project branch is with the *"Server.py"* file:

```
$> python Server.py
```

4.6.2. Requirements

The system requirements in this section are just software requirements. The project requires a specific version of the Python interpreter alongside a series of Python packages that have been used during the development.

Any package dependencies are state under their names:

- Python interpreter: Python 3 (v3.4 or superior)
- Packages:
    - *Tweepy* (v3.5 or superior)
        - *Six* (v1.10 or superior)
        - *Requests* (v2.14 or superior)
        - *Requests-oauth* (v0.8 or superior)
    - *NLTK* (v3.2 or superior)
        - *Six* (v1.10 or superior)
    - *Scikit-learn* (v0.18 or superior)
    - *Numpy* (v1.12 or superior)
    - *Flask* (v0.12 or superior)
        - *Werkzeug* (v0.12 or superior)
        - *Jinja2* (v2.9 or superior)
        - *Click* (v6.7 or superior)
        - *Itsdangerous* (v0.24 or superior)

4.7. Estimations and planning

4.7.1. Estimation of costs

In this section, the software development costs of this Bachelor's degree thesis are going to be estimated. Physical, indirect and human resources during the nine months of development are going to be detailed. All the costs have been calculated without taxes.

- Physical resources:

The physical resources associated with the software development are basically technical because of the nature of the project. The cost related with the future hosting have not been included.

Resource	Quantity	Cost
MacBook Pro 13''	1	1,445 €
PyCharm license	1	159 €
Microsoft Office 365 license	1	70 €

- Indirect resources:

All the costs related with the completion of the project but that cannot be considered key project development tools are considered indirect costs.

Resource	Cost per month	Months	Cost
Internet	30 €	9	270 €
Electricity	40 €	9	360 €

- Human resources

Given that the development has been done by one person and supposing a salary of 9€ per hour, the cost of human resources are as follows:

Phase	Salary per hour	Hours	Cost
Software project	9 € / h	350	4,050 €
Documentation	9 € / h	180	1,620 €

- Total costs:

Concept	Cost
Physical resources	1,675 €
Indirect resources	630 €
Human resources	5,670 €
TOTAL	**7,975 €**

4.7.2. Possible risks

There exist some risks related with the correct deployment and functionality of the software tool. Most of these risks are external circumstances to the software that is provided in the link of section 4.1.

- Incompatibilities among the different used packages:
  As the explained software project use several Python packages to provide a more complex functionality, the upgrade of those packages could produce incompatibilities among them, leading to an error when trying to execute any of the functionalities (*train a model, search for tweets, classify tweets of an account, classify tweets of a stream*).

- Web application hosting server down:
  In case of hosting the web application in a hosting service such as *Google Cloud* or *Heroku*, the software is not going to work if the hosting server goes down. The only recommendation to avoid this risk is to select a good hosting service that could guarantee a certain minimum of time up.

- Impossibility of increasing the datasets with new examples:
  When using the *Search* functionality to try to increase the training datasets, it is not always easy to find relevant and informative tweets for the desired sentiment. Moreover, as the project uses the Twitter official API, the retrieved tweets could suffer bias depending on how Twitter retrieve them for the search queries. The *search* functionality only make sense if we assume there is not a bias on the way those tweets are retrieved.

4.7.3. Project planning

The project can be split into two phases: *software development* and *documentation*, being the first one the one taking most of the time due to the complexity when integrating multiple Python packages. Both phrases can be split into smaller sub-phrases that are going to be explained in this section.

Project phrase	Project sub-phase	Time Period	Hours
Software development	Find information	08/2016 – 05/2017	50 h.
	Python packages comparison	08/2016 – 08/2016	10 h.
	Master: search	08/2016 – 09/2016	20 h.
	Master: train	08/2016 – 05/2017	60 h.
	Master: classify account	09/2016 – 04/2017	40 h.
	Master: classify stream	09/2016 – 05/2017	60 h.
	Server functionalities	02/2017 – 05/2017	70 h.
	Server: web	02/2017 – 05/2017	40 h.
Documentation	Introduction	02/2017 – 06/2017	10 h.
	State of the art	03/2017 – 06/2017	30 h.
	Project overview	03/2017 – 06/2017	20 h.
	Developed work	03/2017 – 06/2017	50 h.
	Results	04/2017 – 06/2017	35 h.
	Conclusions	05/2017 – 06/2017	10 h.
	Future work	05/2017 – 06/2017	10 h.
	References	06/2017– 06/2017	15 h.

5. Results

5.1. Evaluation procedure

There exist several different procedures of evaluating a ML classifier. It is enough to train the ML model we want to test with some part of the available data, and test over the unseeing one. However, this simple approach can lead to an undesirable effect called *overfitting* [14].

In case of *overfitting*, the model describes with very high accuracy the test set, making it excessively complex and violating *Occam's razor* principle, which can be summarized as "*the simpler the solution the better the generalization*". A model suffering this effect will provide bad predictive performance as it overreacts to minor changes in the data.

On the other hand, *underfitting* is the situation in which a trained model does not capture the general trend of the data. The predictive performance will be also bad because the model is too simple to describe the underlying relationships in the data.

A balance between these two phenomena is what creates a good classifier, and this problem is usually called *"the bias-variance tradeoff",* in which a model must neither be too complex so its predictions vary a lot from dataset to dataset (small bias but high variance), nor too simple because the model will do assumption to simplify the reality, lowering the predicting performance (low variance but high bias).

One of the most used approaches to test a complex classifier without falling into the overfitting problem is ***K-folds Cross-validation (CV)***. This technique considers only one dataset to be used for both training and testing. It divides the dataset in which what are called "folds", training with all of them but one, which is used for testing. The procedure is repeated until all folds have been used for testing once. Because the training and test sets were varied, a good score about how well our model is going to generalize can be obtained.

Before using CV, the number of folds (k) must be decided. This parameter is very important because if it is defined as a very low number or a very high one, the overfitting testing is not as high and the results must be considered less into account. Although the values that are most frequently used are 5 and 10, the evaluations were performed using just 10 folds.

5.2. Evaluation metrics

The comparison among the different classifiers (*Logistic Regression, Bernoulli NB, SVM* and *Random Forest*) needs to have a specific metric to evaluate the quality of them and to select the one that performs better.

In any ML classification problem, one of the first steps to determine the comparison metric is to decide if the classification errors regarding one of the classes are more important than the ones in the other classes. Considering the classification scheme in *Figure 3.1*, there exist 2 classifiers (*neutral-polarized* and *positive-negative*), without any classification error unbalance towards any class with respect to the other.

However, as the classification follows a hierarchical model, the errors in the first classifier (the one that discriminate between neutral and polarized tweets), have a bigger impact into the final assigned label than the second one, which relies in a second hierarchical level.

In terms of which metric has been used to compare the classifiers quality, ***F1 score*** **[15]** was the chosen one. This metric is a combination of *recall* (false negative rate) and *precision* (false positive rate), following this formula:

$$F1\ score = 2 * \frac{Precission * Recall}{(Precission + Recall)}$$

Equation 5.1: F1 Score

The main reason to choose *F1 score* over the commonly used *accuracy* (fraction of well classified instances over the total number of them) is that, in case the number of examples over one class is way different from the number representing the other class, the accuracy rate can hide a wrong classification of those in the less represented class. A good example of this can be found in the "*R*" programming language blogs [16].

Once the metric has been chosen, it is interesting to figure out which algorithm performs better in case of each classifier (*neutral-polarized* and *positive-negative*). Moreover, comparing the F1 scores in case of using unigrams or unigrams + bigrams can give us an idea of the importance of those structures inside the training dataset.

N-grams of more than 2 tokens have been not considered because they do not appear as frequent in the dataset and therefore they are not likely to provide an error decrease, while increasing the number of features. This decision is shared by other academic papers such as the one by *Pang et al.* [17].

5.3. Feature analysis

Because the number of features used has a direct impact on the classification error of the models, a comparison between different configurations of them yields useful results.

Thanks to the implementation of the feature extraction function, the number of features can be varied, and it needs to be expressed as a percentage of the total. Two percentages must be specified: one for unigrams, and another for bigrams. It is usually sensible for the proportion of considered unigrams to be bigger than the one for bigrams because if there are '*n*' different unigrams, there could be up to (n^2) bigrams.

The evaluation is performed by the shell script called *"Evaluate.sh"*, which generates four files in a folder called *"Evaluations"*, one file for each ML algorithm, using 10-folds cross validation. The comparison performed separately for each model (*polarity* and *sentiment*).

5.3.1. Polarity comparison

In this case the comparison takes place between the models generated with the dataset *Polarized.txt* and *Neutral.txt*. The conclusions are stated in section 5.3.3.

- Considering 1% of unigrams in relation to the bigrams:

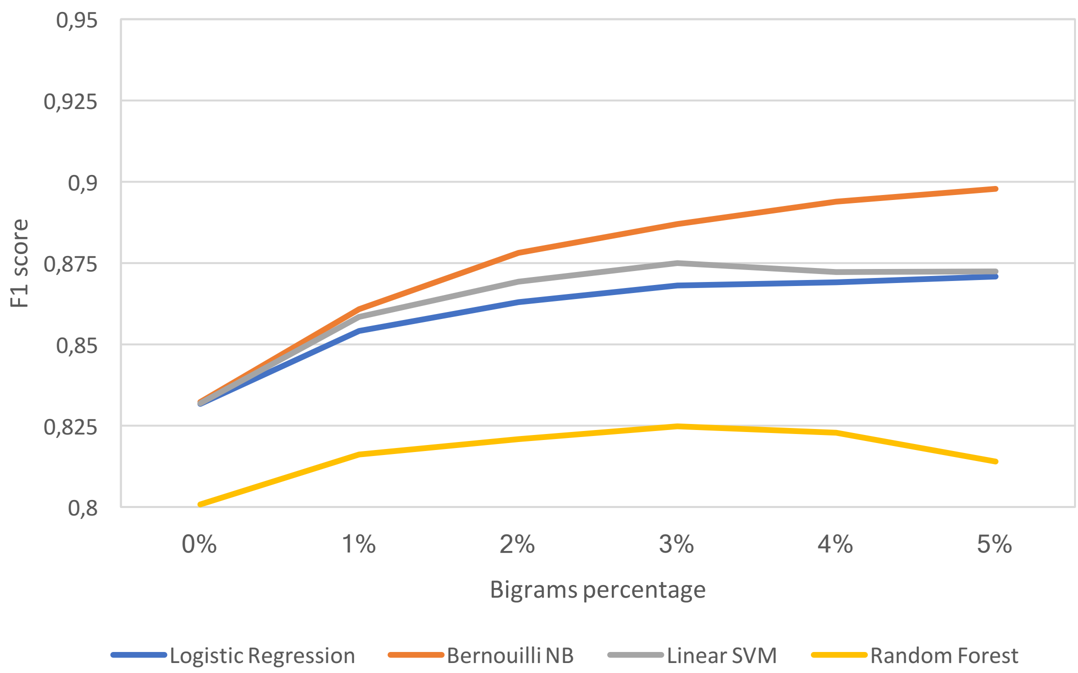

Figure 5.1: Polarity model with 1% unigrams

- Considering 2% of unigrams in relation to the bigrams:

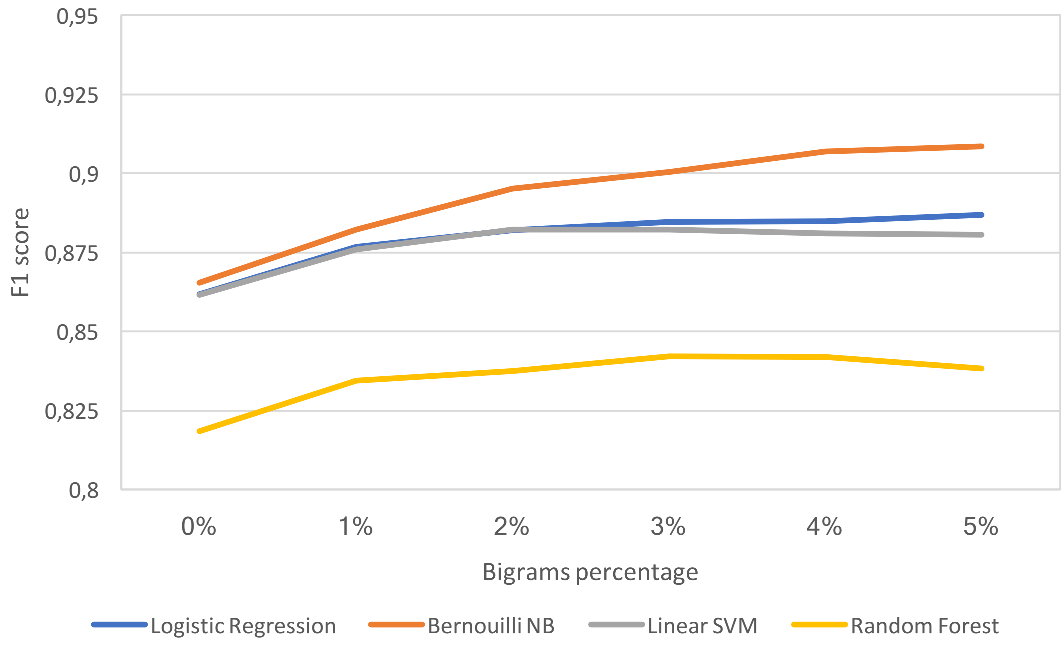

Figure 5.2: Polarity model with 2% unigrams

- Considering 3% of unigrams in relation to the bigrams:

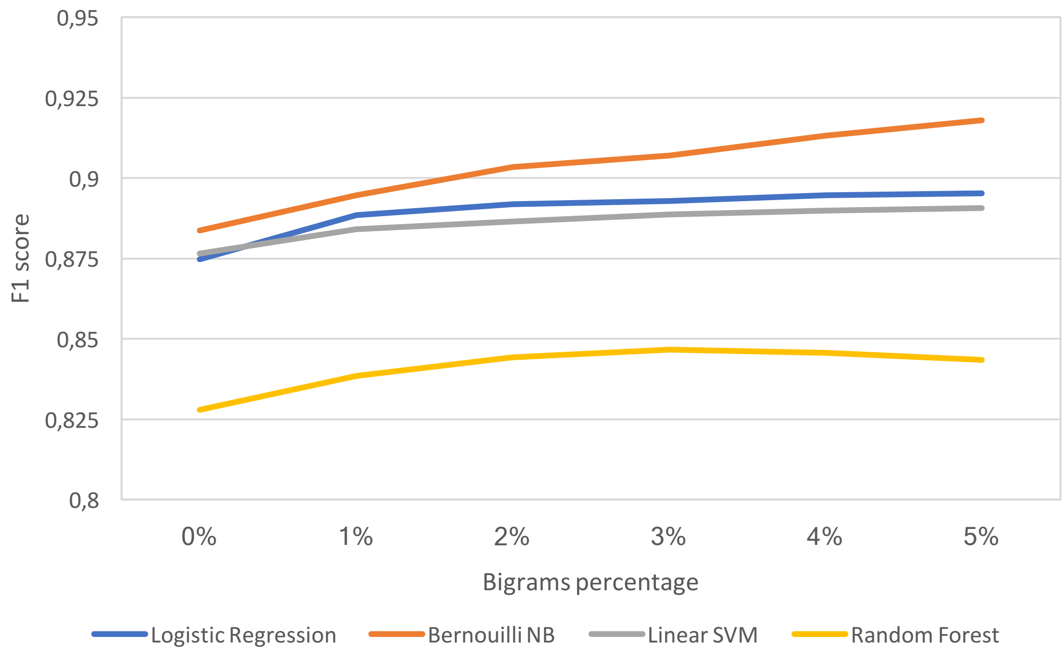

Figure 5.3: Polarity model with 3% unigrams

- Considering 4% of unigrams in relation to the bigrams:

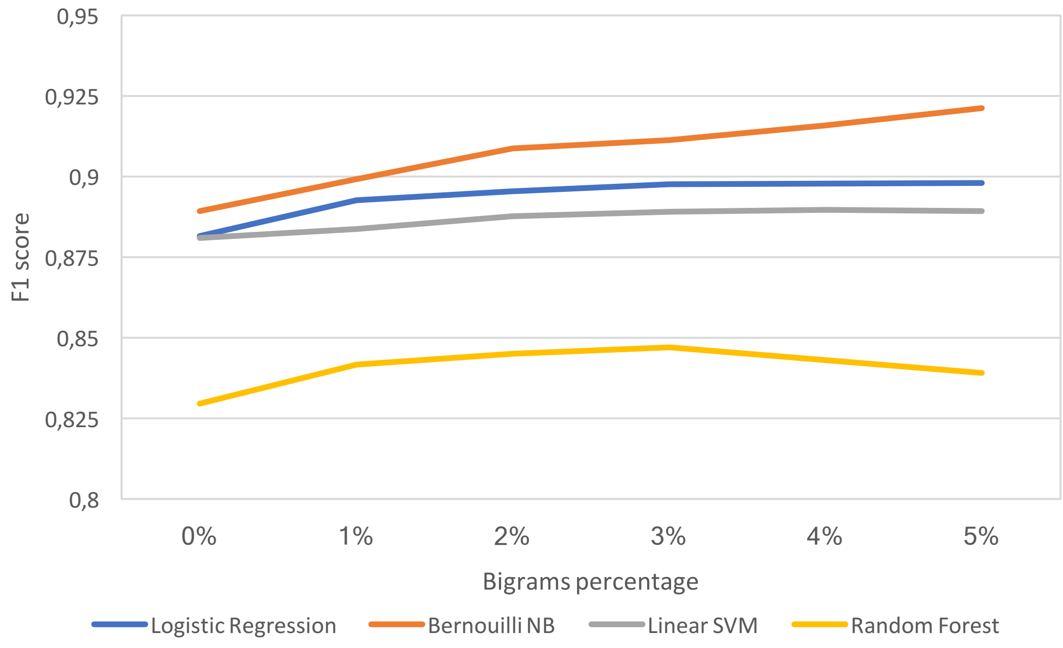

Figure 5.4: Polarity model with 4% unigrams

- Considering 5% of unigrams in relation to the bigrams:

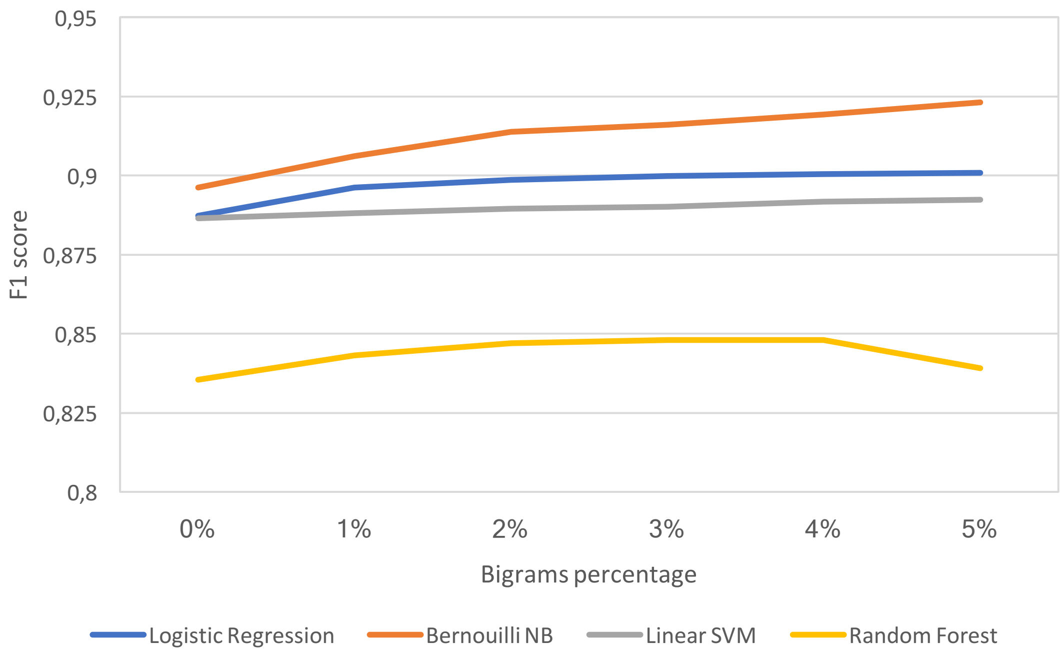

Figure 5.5: Polarity model with 5% unigrams

5.3.2. Sentiment comparison

In this case the comparison takes place between the models generated with the dataset *Positive.txt* and *Negative.txt*. The conclusions are stated in section 5.3.3

- Considering 1% of unigrams in relation to the bigrams:

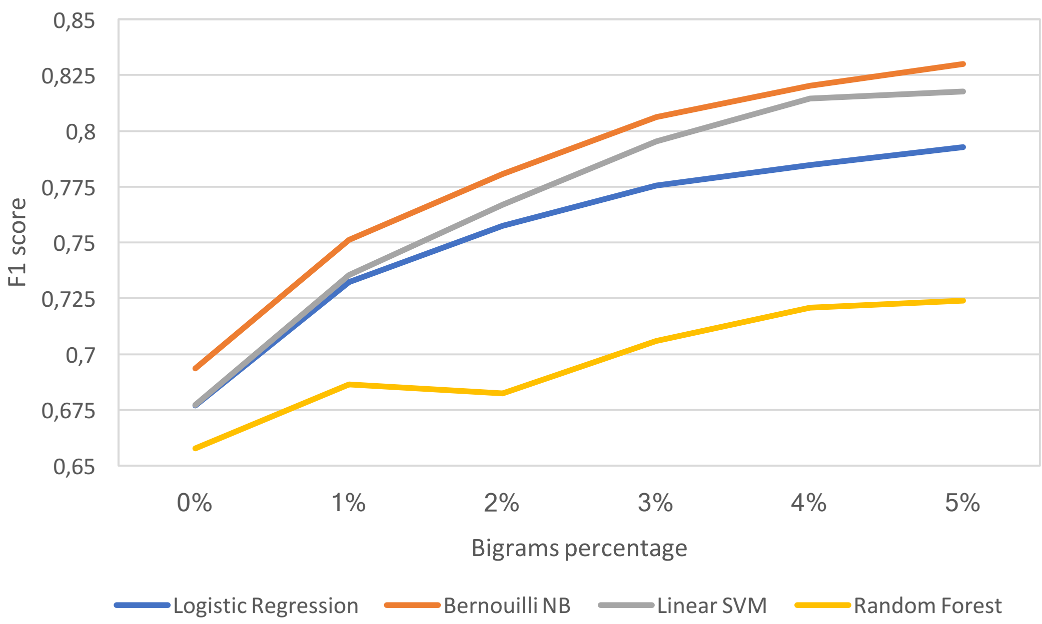

Figure 5.6: Sentiment model with 1% unigrams

- Considering 2% of unigrams in relation to the bigrams:

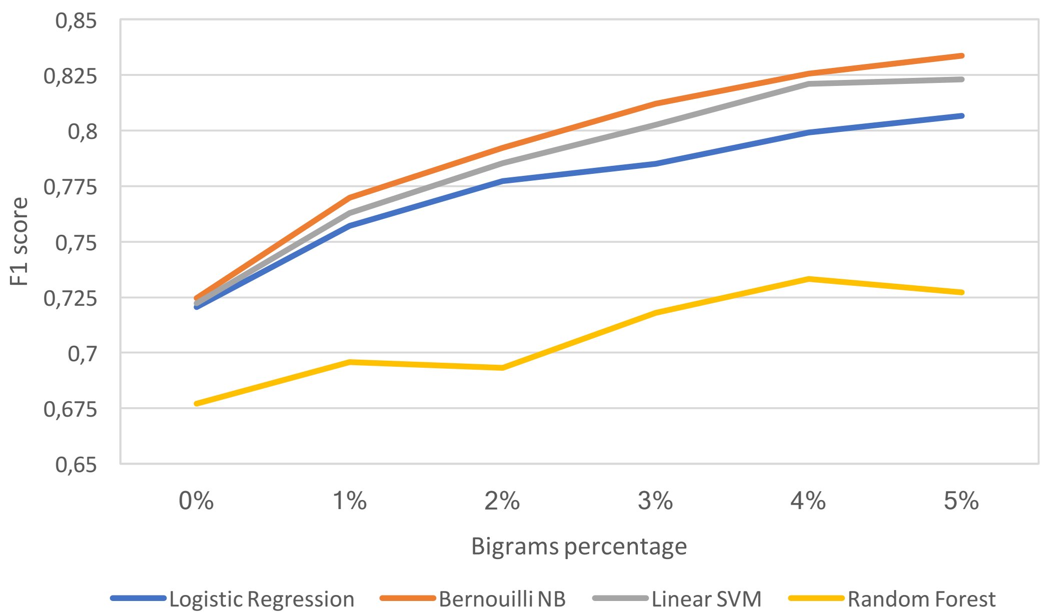

Figure 5.7: Sentiment model with 2% unigrams

- Considering 3% of unigrams in relation to the bigrams:

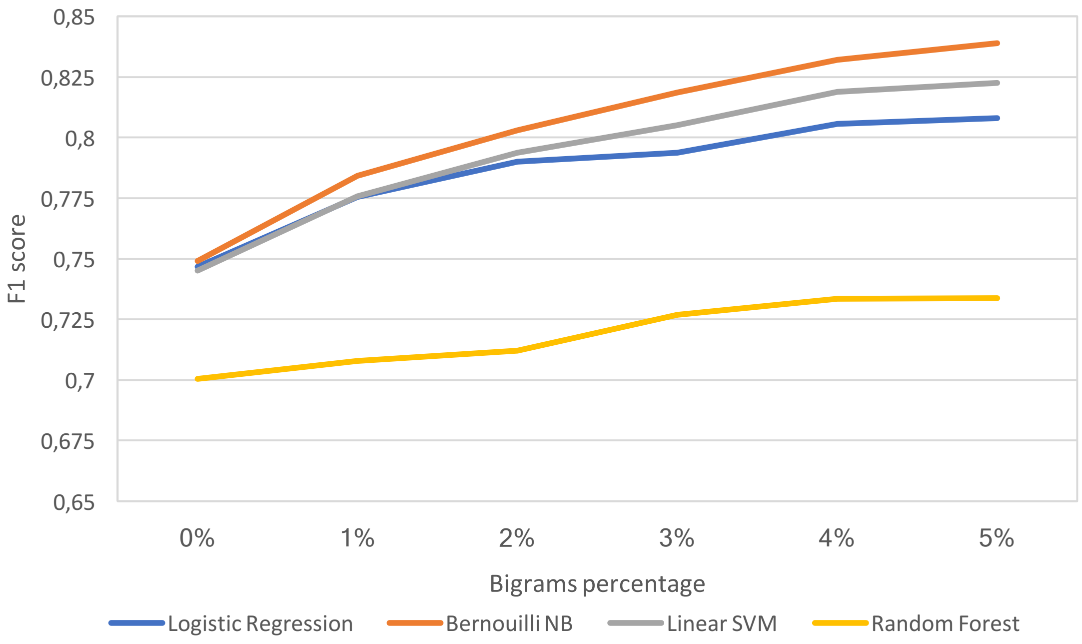

Figure 5.8: Sentiment model with 3% unigrams

- Considering 4% of unigrams in relation to the bigrams:

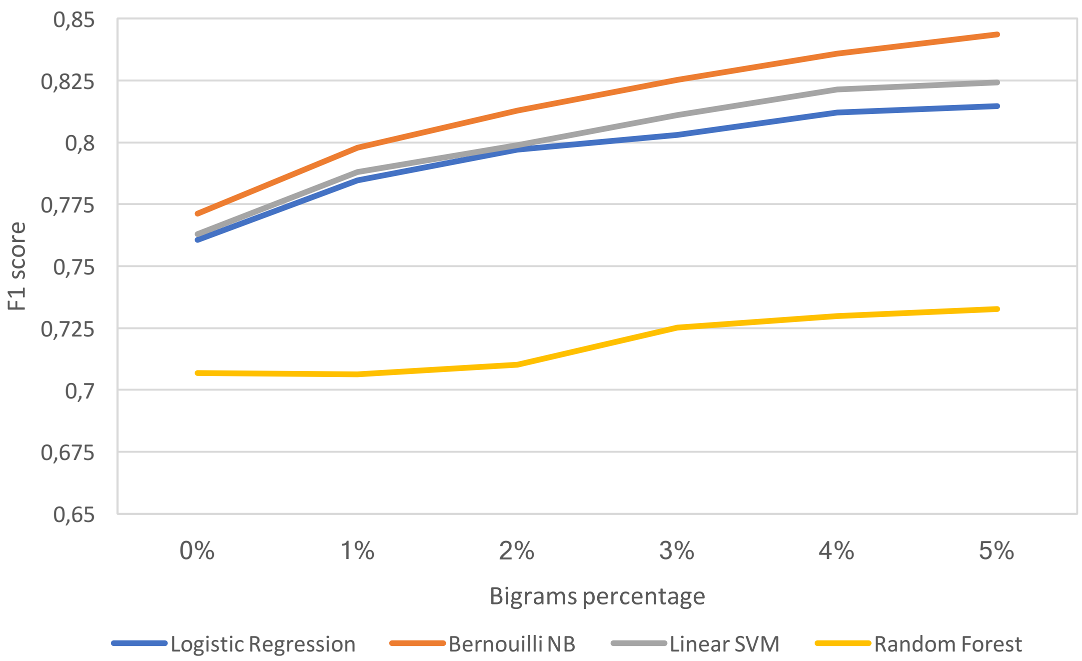

Figure 5.9: Sentiment model with 4% unigrams

- Considering 5% of unigrams in relation to the bigrams:

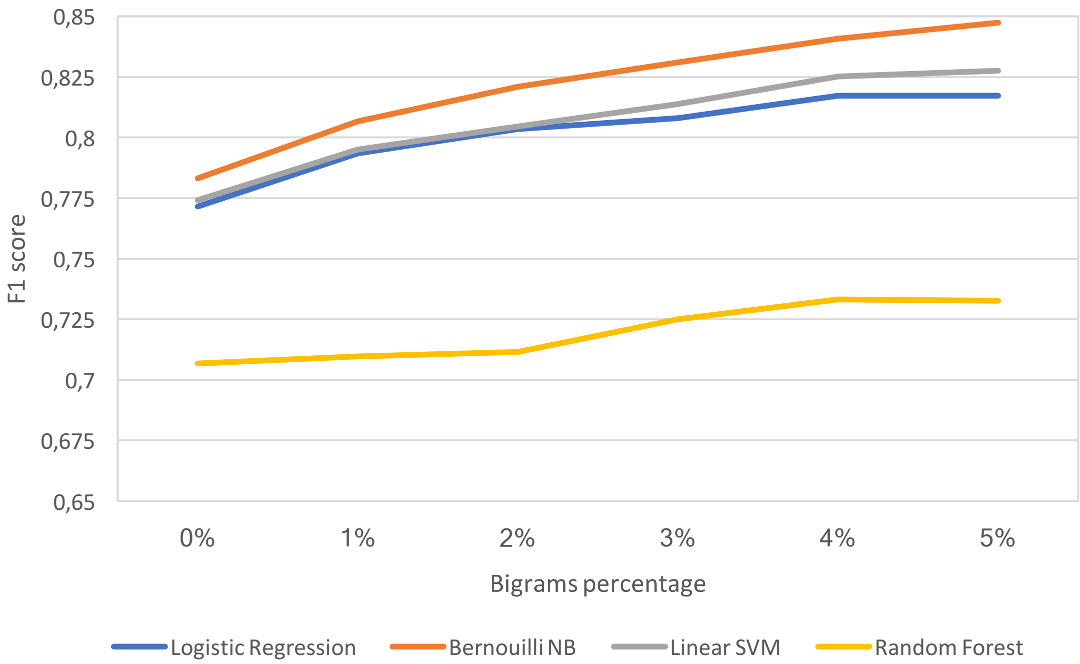

Figure 5.10: Sentiment model with 5% unigrams

5.3.3. Comparisons conclusions

After generating the evaluation curves for both classifiers and for all the considered ML algorithms, depending on the classifier, the following conclusions were drawn:

In the case of the polarity classifiers (*Figure 5.1* to *Figure 5.5*), the score curves increase less with each new bigram percentage, and are softer with each new unigram percentage. Once the most informative features have been selected, the rest of them are not that useful, and they can produce overfitting (the scores decrease) in SVM and Random Forest when more than 3% of bigrams are selected. In terms of the algorithms, Naïve Bayes performs the best, followed by Logistic Regression and SVM, leaving Random Forest with the lowest scores.

In case of the sentiment classifiers (*Figure 5.6* to *Figure 5.10*), the score curves also increase less with each new bigram percentage and with each new unigram percentage. However, this time the overfitting effect can be seen in Logistic Regression and Random Forest in the last figure from 4% of bigrams. This results in SVM scores much closer to Naïve Bayes than were previously recorded, followed by Logistic Regression and Random Forest.

Finally, results indicate that the various models react differently to overfitting, such that the Random Forest algorithm is the most sensitive to it.

5.4. Algorithms analysis

In this section, all the feature evaluation results of the different classifiers are compared. All the ML algorithms were evaluated using the default *Scikit-learn* parameters, but Random Forest. In this last case, the parameter indicating the number of trees was increased to 100 in order to obtain comparable results.

Analysis the evaluation curves from section 5.3, it is clear that the higher the percentage of unigrams is, the softer the curve becomes while increasing the bigrams. This effect occurs because the higher unigrams are considered, the less improvement can be achieved when including more of them.

Although overfitting cannot be seeing while increasing the percentage of considered unigrams up to 5% (the curves always start higher than the last percentage), it takes place when the percentage of bigrams treated as features is larger than 3% (polarity classifier) and 4% (sentiment classifier) in SVM and Random Forest. Even if this effect does not seem to affect neither Bernoulli Naïve Bayes nor Logistic Regression, it is an indicator that low number of bigrams can produce overfitting.

Finally, with the aim of selecting the best model, only those with 5% unigrams, and either 3% bigrams (polarity classifiers) or 4% (sentiment classifiers) are compared:

Classifier	N-grams	Algorithm	F1 Score (K = 10)
Polarity	5% Unigrams + 3% Bigrams	Logistic Regression	0.8998
		Bernoulli NB	**0.9161**
		Linear SVM	0.8901
		Random Forest (n = 100)	0.8480
Sentiment	5% Unigrams + 4% Bigrams	Logistic Regression	0.8173
		Bernoulli NB	**0.8408**
		Linear SVM	0.8251
		Random Forest (n = 100)	0.7333

Bernoulli Naïve Bayes performs the best, followed by Linear SVM and Logistic Regression with similar scores, leaving Random Forest as the worst model.

6. Conclusions

After the complete development of the project and the evaluation process (classifiers comparison and number of features analysis), there are several conclusions that have been reached and that could be useful to future projects or work:

1. It is recommended to use libraries for accessing Twitter API:
   The original Twitter API is not as intuitive as the third-party libraries. Moreover, they simplify the communication process with it, saving developers time to focus in the real goal of the project. Taking a look to the internal code of those libraries, for instance *Tweepy,* we can realize the number of cases that we should consider when communicating with Twitter.

2. The datasets bias is adopted when learning:
   As the examples used for the learning process come from the training datasets, any bias that those examples / files have is going to be reflected in the way models are trained. In case of the neutral, polarized, negative and positive datasets provided by NLTK, they contain movie reviews, so although any text with any of those sentiment should be recognizable because of its words, people do not express themselves in the same way rating a movie than writing a tweet.

3. Scikit-learn was advantageous over other implementations:
   Initially, my own cross validation (CV) implementation was tried, and although it took advantage of all CPU possible threads, the implementation provided by Scikit-learn ran about 10 times faster while only using one CPU core. Furthermore, Scikit-learn's "Vectorizer" functionality transforms example features to sparse arrays which drastically reduces the memory usage.

4. Including more than 3-4% of bigrams as features produce overfitting:
   As shown in section 5.3, when the polarity classifiers (which discriminate between *neutral* and *polarized* classes) have a proportion of bigrams higher than 3%, the F-scores of Linear SVM and Random Forest algorithms decrease. The same effect can be seen in the sentiment models (which discriminate between *negative* and *positive* classes) such that the F-score decreases when the bigrams are increased to 4%. Even though the effect is not visible in every algorithm, it is a clear indicator of how irrelevant features can negatively impact results.

5. Naïve Bayes and SVM seem to return the best results:

   Among the compared algorithms, those are the ones with the best results. The reason of this is that due to their characteristics they perform quite well in the case of big number of features, the first one (NB) using a probabilistic approach and the other one (SVM) using support vectors and kernel functions as a way of creating the most flexible model without falling into overfitting.

6. Difficulty of achieving very good (> 0.9 F1-score) models:

   Considering the obtained results, it is easy to realize that training very good text classification models are taught to get because of all the corner cases that cannot be easily recognized (sarcasm, irony, text typos, weird abbreviations…). Models recognizing those concepts need to be complex in so many ways without falling into overfitting, that it will require not only more human resources, but also more linguistics knowledge.

7. Most the tweets nowadays are neutral:

   Finally, after the algorithms have been compared and the number of features selected, some streams of data have been analyzed and filtered by words that do not contain any inherent positive or negative feeling. It was observed that the majority of the tweets (50-60%) were neutral, through this we can infer that the users are not providing any opinion (at least in the positive-negative spectrum). Although this depends a lot with the specific word and its context, the results allude to a trend of using Twitter more like an information social network than a personal opinion one.

8. The usefulness of debatable trending topics analysis:

   Thanks to the provided web application, the sentiment analysis of trending topics and hashtags can be useful when performing statistical studies on social networks, define possible marketing targets and analyze how global events produce different feelings depending on the country, region, or city. Moreover, it could be used by companies to track product acceptance in different locations.

7. Future work

Even if the project has been finished and most of the main goals fulfilled, there is always space for improvements and new functionalities.

There are some ideas that will fit very well with the code base already available and that will create a richer tool for Sentiment analysis. They are going to be split into 2 categories: Back-End improvements and Front-End improvements.

7.1. Back-End improvements

- Introduce more complex techniques:

  During the training data processing and learning, all of the most common techniques for text mining has been applied (transform all words into lower case, tokenize them, remove the stop words and lemmatize the remaining ones). However, there are more complex techniques such as the use of *Part of Speech* (POS) tags, *chunking* using a parser and *word sense disambiguation* (WSD).

  One possible problem of introducing these capabilities is the performance decrease when analyzing streams of tweets. These streams, depending on the filtered words can go very fast, and there could be an increasing delay between the speed in which the tweets are provided and the classification results.

- Obtain better data sets using the search tool:

  Another possible aspect in which the project can improve is on the ML training part. By default, the datasets of neutral, polarized, positive and negative sentences from the NLTK corpus are used, but these datasets are not perfect and do not contain every possible word or bigram that someone can use to express a sentiment. Moreover, reading the documentation about the NLTK corpus datasets, it is clear that they have been created using movie reviews from a webpage. Using these datasets to train a model that will predict sentiment over unseen tweets is not ideal, supposing that users express themselves in the same way rating a movie and writing a tweet, is a dangerous assumption.

  For this reason, the “search” functionality that allows us to create our own dataset retrieving tweets from the Twitter API was implemented. The main problem comes when we must decide the set of words to filter the tweets in order to build our training data. It is more or less easy in the case of positive and negative data, but it is not when we consider neutrality.

7.2. Front-End improvements

- Implement a login system to manage tokens:

  The most important improvement over the web-application infrastructure is to create a login web-page before accessing the tool. If this login is implemented, each new user can have their own Twitter API token, and there will be less query limitations while trying to operate with the web-application. The change will be huge, because the application will be able of performing 180 queries per user instead of 240 for every user, each 15 minutes (current state) [18].

  The main reason why this improvement has not been included in the project is just a logistic problem: it was going to take a long time to implement, without providing any improvement over a future presentation of the project. The improvements will only be noticed if the project is publicly used by a large group of users.

8. References

1. "Internet Live Stats – Internet Usage & Social Media Statistics." *Internet Live Stats*, Real Time Statistics Project, www.internetlivestats.com.

2. Kishori K. Pawar, Pukhraj P Shrishrimal, R. R. Deshmukh. "Twitter Sentiment Analysis: A Review." *Twitter Sentiment Analysis: A Review*, volume 6, issue 4, 4 April 2015, http://www.ijser.org/researchpaper%5CTwitter-Sentiment-Analysis-A-Review.pdf.

3. "Sentiment Analysis." *Wikipedia*, Wikimedia Foundation, 13 June 2017, https://en.wikipedia.org/wiki/Sentiment_analysis.

4. Helmbold, David. "Machine Learning and Data Mining". University of California (SC).

5. Quijano Sánchez, Lara. "Machine learning and Text Mining". UC3M.

6. Kowalczyk, Alexandre. "Linear Kernel: Why is it Recommended for Text Classification?" *SVM tutorials*, 12 April 2017, www.svm-tutorial.com/2014/10/svm-linear-kernel-good-text-classification.

7. "Twitter Developer Documentation." *Twitter*, Twitter, https://dev.twitter.com/docs

8. Agarwal, Apoorv, et al. "Sentiment Analysis of Twitter Data." Columbia University, www.cs.columbia.edu/~julia/papers/Agarwaletal11.pdf.

9. Koppel, Moshe, and Jonathan Schler. "The Importance of Neutral Examples for Learning Sentiment." Bar-Ilan university, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.9735&rep=rep1&type=pdf.

10. Bromberg, Andy. "Second Try: Sentiment Analysis in Python." *Andy Bromberg RSS*, http://andybromberg.com/sentiment-analysis-python/ .

11. Kinsley, Harrison. "Twitter Sentiment Analysis with NLTK." *Python Programming Tutorials*, https://pythonprogramming.net/twitter-sentiment-analysis-nltk-tutorial/

12. "Twitter Libraries." *Twitter Developer Documentation*, Twitter, https://dev.twitter.com/resources/twitter-libraries .

13. "Streaming API Request Parameters." *Twitter Developer Documentation*, Twitter, https://dev.twitter.com/streaming/overview/request-parameters#locations.

14. "Overfitting." *Wikipedia*, Wikimedia Foundation, 9 June 2017, https://en.wikipedia.org/wiki/Overfitting .

15. “F1 Score.” *Wikipedia*, Wikimedia Foundation, 20 May 2017, https://en.wikipedia.org/wiki/F1_score .

16. Cisneros, Benjamin Tovar. “Accuracy versus F Score: Machine Learning for the RNA Polymerases.” *R-Bloggers*, 16 August 2013, www.r-bloggers.com/accuracy-versus-f-score-machine-learning-for-the-rna-polymerases .

17. Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. “Thumbs up? Sentiment classification using machine learning techniques.” *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 79–86, 2002. https://www.cs.cornell.edu/home/llee/papers/sentiment.pdf .

18. “Rate Limits: Chart.” *Twitter Developer Documentation*, Twitter, https://dev.twitter.com/rest/public/rate-limits .