Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Euro-Inf Bachelor awarded by EQANIE

Classifiers and text mining: application to a specific context

Estudante:	Roi Santos Ríos
Dirección:	Jesús Vilares Ferro
	Miguel Ángel Alonso Pardo

A Coruña, Setembro de 2022.

To the voiceless

Acknowledgements

First and foremost, I want to deeply thank my tutors Jesús and Miguel, who gave me the opportunity of working on this project and guided me throughout it's entirety. I also want to thank all the LyS group members, who allowed me to develop this project at CITIC and partake in NLP-related research activities, as well as giving me plenty of advice and sharing good stories over coffee.

Abstract

The constant growth of social networks has not only brought us new ways of interacting with each other, but has also given way to a severe increase in negative behaviors: hate speech, racism, gender harassment, cyberbullying, etc. Manually trying to detect this kind of behaviours in millions of daily social media posts is out of the question. The solution lies in developing intelligent systems to automate such detection tasks.

As the nature of these texts is completely subjective, this problem falls under the field of sentiment analysis, which aims to systematically identify and study affective states and subjective information in textual data using natural language processing techniques.

In particular, this project is focused on the research of different machine learning techniques related to natural language processing, in order to automate and perform a reliable detection and classification of sexist-related behaviours in social media texts. We will tackle the task of adequately processing the extracted data from social media, as well as researching various text classification techniques and models that we will use to develop and evaluate a variety of classifiers.

Keywords:

- Sentiment Analysis
- Natural Language Processing
- Machine Learning
- Text Mining
- Social Networks
- Sexist Language

Contents

List of Figures

List of Tables

Chapter 1
Introduction

As the world's digital population grows, so does the reach and usage of social media. Social media are now a part of our everyday life and continue to transform the way we interact with one another on a global scale. Social networking is one of the most popular online activities worldwide. The data provided in Figure 1.1 shows that, as of 2022, global social media audiences amounted to 4.59 billion users, and that number is expected to increase up to 5.85 billion in the coming 5 years.

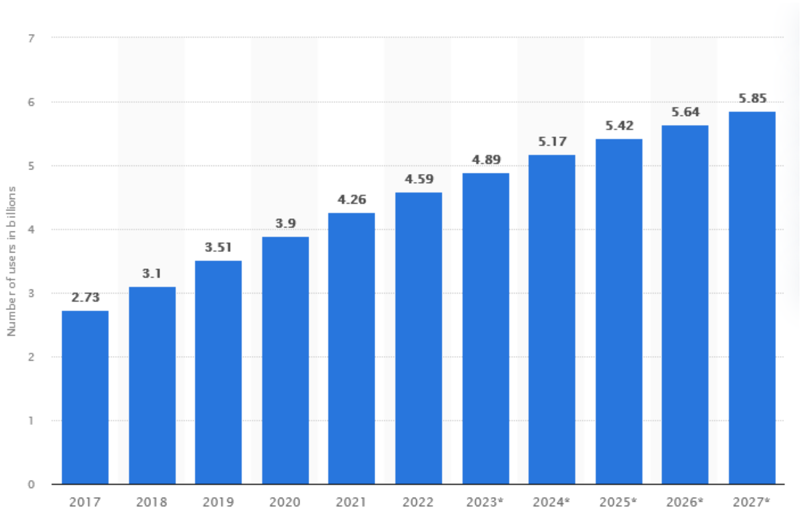


Figure 1.1: Evolution of social network usage [1]

The downside is that, as we interact more and more on social media, negative behaviors of regular social interactions increase their presence in these platforms: hate speech, racism, incitement to terrorism, gender harassment, cyberbullying, etc. Moreover, the feeling of anonymity these media provide encourages these type of actions. Between March 1st and April 31st, OBERAXE (Observatorio Español del Racismo y la Xenofobia) [4] has detected a 27% increase in hate speech in social media, compared to the previous time interval. This is a very worrying trend.

Manually evaluating millions of daily social media posts searching for those type of phenomena is out of the question. The solution lies in developing intelligent systems to automate such evaluation tasks. However, human language is riddled with ambiguities that make it incredibly difficult for a computer to determine the intended meaning of text accurately. That is where Text Mining (TM) comes in.

Text mining may be defined as the process of transforming unstructured natural language text into a structured format to identify meaningful patterns and new insights. By applying Natural Language Proccessing (NLP) techniques we can explore and discover hidden relationships within such unstructured text data.

NLP is the branch of Artificial Intelligence (AI) concerned with giving computers the ability to understand human language. In our case, by applying this type of techniques we can determine which social media posts contain this kind of harmful content. We also must consider that, by their own nature, these texts are heavily subjective, so to solve this problem we will have to apply NLP techniques that work with subjective information.

Sentiment Analysis (SA) [5] is a widely used text mining technique for obtaining useful and subjective information from text-based data. It's a key component of NLP, applied over text analysis and computational linguistics to systematically identify, extract, analyse, and study affective states or emotions from the textual data.

1.1 EXIST workshop

Instead of manually gathering and evaluating social media data, we opted to use the resources given by the research community at the Iberian Languages Evaluation Forum (IberLEF) [6]. This forum organized a variety of workshops, each centered around the use of NLP techniques to tackle a specific problem. One of these workshops was the sEXism Identification in Social neTworks (EXIST) [7], which was focused on implementing intelligent systems capable of detecting and classifying sexism in social media texts.

In this workshop, participants are asked to perform the following classification tasks:

1. A binary classification task where the system must decide whether a given text is sexist or not (i.e., it is sexist itself, describes a sexist situation or criticizes a sexist behavior).

2. A multiclass classification task where, once a text has been classified as sexist, the system categorizes it according to the type of sexism present (according to the categorization proposed by experts and that considers the different facets of women that are undermined).

For these two tasks, the participants are provided a dataset containing 6 977 texts extracted from Twitter[1] and Gabhttps://gab.com/. All these entries have been given two labels, one for each task. The first label dictates if the tweet/gab is sexist or non-sexist, and the second label dictates the type of sexism present. The proposed sexist categories are the following:

- **Ideological and inequality**: The text discredits the feminist movement, rejects inequality between men and women, or presents men as victims of gender-based oppression.
- **Stereotyping and dominance**: The text suggests that women are more suitable to fulfill certain roles (mother, wife, tender, submissive, etc.), or they are inappropriate for certain tasks (driving, hardwork, etc), or claims that men are superior to women.
- **Objectification**: Women are portrayed as objects, severed from their dignity and personal aspects, or the text assumes or describes physical qualities that women must have to fulfill traditional gender roles (hypersexualization of female attributes, compliance with beauty standards, women's bodies at the disposal of men, etc.).
- **Sexual violence**: The text contains requests for sexual favors, sexual suggestions or rape (or any sexual assault) threats.
- **Misogyny and non-sexual violence**: The text shows signs of hatred and violence towards women.

1.2 Main objective

The main objective of this work is the study and application of SA techniques for the detection and classification of sexist content in social media texts. For that purpose, we will implement, train and evaluate different classifiers, each one using a different SA model. The models we chose are detailed in Section 2.1.3.

We will tackle both tasks presented by the EXIST workshop in Section 1.1, so we will be implementing a binary and a multiclass variation of each model. We will also use the dataset they provide for the training and evaluation of our classifiers, in order to have a common ground to compare them.

Even though the project is aimed at this specific context, it is just an example of a field in which SA techniques can be applied to. The classifiers that we developed in this work will be easily adapted to other specific contexts regarding .

[1] https://twitter.com/

Chapter 2

State of the art

As the main objective of this work is to study current state of the art, regarding NLP is quite settled, as there are some language models that dominate over the rest.

2.1 Sentiment Analysis

As mentioned in Chapter 1, Sentiment Analysis can be used to identify the sentiment polarity of a text, in order to recognize if the text represents a positive, neutral or negative attitude. This is extremely useful for any kind of company that offers goods or services, as it can make use of SA to automate the task of evaluating the customers' opinions [8]. It's been used to analyze movie reviews [9], to make predictions on election opinions [10], to analyze airline reviews [11], to analyze amazon's customer reviews [12], etc. It is also extensively used to analyze textual data from social media [13] [14] [15], which aligns well with the aim of this project.

As shown in Figure 2.1, sentiment analysis is divided in four main steps: preprocessing, feature extraction, classification, and interpretation of results.

2.1.1 Preprocessing

Text Preprocessing is the first step in the NLP pipeline, and it has a lot of potential impact in its final process, being necessary in most cases. It involves. In our case, we will first take a look at the raw data, checking the label distribution and the general characteristics of the texts.

Text Preprocessing is an essential part of the NLP pipeline, it has a lot of impact on the other steps, as well as being necessary in most cases. It consists on analyzing and manipulating the dataset, bringing it into a form that is predictable and analyzable. Depending on the problem at hand, this manipulation can range from formatting the text, to correcting spelling mistakes, or even removing stopwords or other grammatical structures. In domains such as

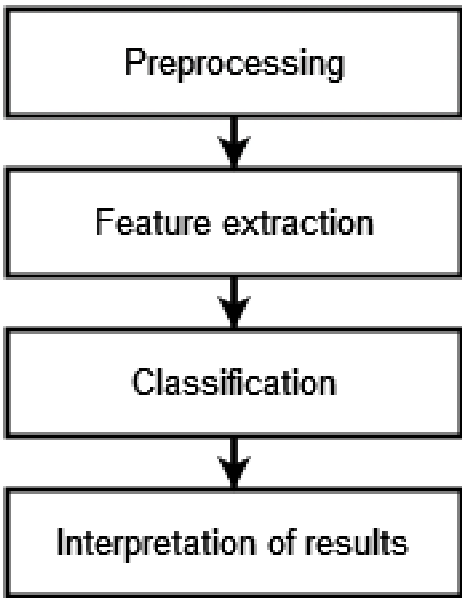


Figure 2.1: Diagram of the four main steps in sentiment analysis.

twitter data, removing links or mentions can be useful, whereas in other domains such as movie reviews there aren't any to remove. We will explore in detail the text preprocessing techniques we use in Chapter 6.

2.1.2 Feature extraction

Feature extraction is the process of preparing the textual data for the algorithms. It essentially consists on obtaining features from the text that can be used for the algorithms to learn, as they can't work directly with human language. These features range from word counts, punctuation counts, character length, the language of the text...as well as text-based features such as tokenzation, vectorization, stemming, etc [16]. There are two types of feature extraction methods:

- **Lexicon-based**: In lexicon-based methods, lists of positive and negative words are provided. These words are counted for each sentence, and the sentiment is decided by the frequency of the positive-oriented and negative-oriented words.
- **Machine learning-based**: In machine learning-based methods, the model tries to find patterns from the data provided. Machine learning methods are divided in supervised and unsupervised learning. In supervised techniques, the model is trained using labelled data, and then it is evaluated using unlabeled data. On the other hand, unsupervised techniques train the model using data that has not been labelled or categorized, enabling the model to function without the need for supervision.

The most commonly used machine learning techniques for feature extraction are Bag-of-Words [17] and n-grams. In these feature extraction techniques, the features are retrieved either using Count Vectorizer or Term Frequency - Inverse Document Frequency (TFIDF) [18]. These techniques use a one-hot word representation approach, in which the vocabulary size depends on the total number of words displayed in the document. This makes the feature space reach high dimensionality, thus raising scalability problems, and they also cannot capture the syntactical and semantic details of a sentence. In Section 5.1.1, we will further explain how the bag-of-words model works, as well as what n-grams are and explaining how the features are extracted with TFIDF.

In order to improve upon the constraints of the Bag-of-Words and N-gram techniques, word-embedding models were proposed. One of the most common word embedding model is Word2Vec, which we explain in detail in Section **??**. These circumvent these problems by extracting semantic and syntactic details from word representations. These embedding methods generate feature vectors only for words found in their vocabularies and are unable to cope with out of vocabulary words [19]. Another limitation of these models is that similar words from different sentences can have a different context [20], and opposite sentiment expression words such as ”good” and ”bad” can share very similar vectors [21].

One of the most advanced word embedding models as of today is BERT [3], which allows for bidirectional training (from left to right and from right to left), in order to learn deeper context from a given sentence. BERT's bidirectional encoding allows it to process the position of a each word in a sequence and incorporate that information into that word's embedding, while Word2Vec and previous models aren't able to account for word position. Thanks to this, it learns embeddings at a subword level, which allows BERT to generate embeddings for words outside of its vocabulary, as opposed to the previous models. We will explain in more detail how Transformers in Section 5.7, as well as detailing how BERT works in Section 5.8.

2.1.3 Classification

The classification part consists on feeding the previously obtained features into a classifier, in order to train it to predict the sentiment of a given text. Over the years, a variety of machine learning models have been used for SA classification [22], even specifically for twitter [23]. For our tasks at hand, we will mainly focus on machine learning techniques using supervised learning to train our classifiers, as our dataset is labeled. To implement the text classifiers of this work, we decided to use the following models:

- Multinomial Naive Bayes, as it is a simple model that can serve us to create a baseline classifier. We further expand on it in Section 5.1.2.

- FastText, as it is the evolution of the Word2Vec model, and can prove as a middle point between the other two models, regarding its complexity. We expand upon this model in Section 5.5.

- BERT, as it is one of the most powerful NLP models to this day. It will serve as a state-of-the art approach to the tasks at hand. In Section 5.8 we explain in detail this model.

2.1.4 Results

The final step is to interpret the results obtained from evaluating the previously trained classifiers. In Section 6.2 we will explore in detail how supervised classifiers' results are interpreted. We will be calculating metric scores from the classifiers, that will denote their performance regarding the different classification tasks.

Chapter 3

Methodology and planning

3.1 Chosen methodology

Choosing a software development methodology is a crucial task, as the whole project will be affected by this decision. A good model can provide us with a well structured way of understanding and planning our project.

As this work is mainly focused on the research aspect, we should choose a methodology that better suits it. Thus, an Agile methodology has been applied. Agile practices are based on adaptive planning, evolutionary development, continual improvement and flexible responses to changes in requirements, and these principles align perfectly with our interests. We will want to be able to produce working prototypes quickly to get a grasp on its general performance, in order to study how to further improve them.

Agile methodologies split up development in small increments that severely reduce the up-front planning and design. These small increments are called iterations or sprints, and each one of them lasts a short time frame, and work is done in all functions: planning, analysis, design, coding and testing. This way, at the end of each iteration we will have a working product that can be evaluated. This is in stark contrast with other development methodologies that only present the working software near the end of the development. This process also allows the software to adapt to changes or corrections in a faster way, which is something usual in research projects.

We decided to use the Iterative and Incremental development strategy. As shown in Figure 3.1, it is a combination of both iterative design and incremental method. For each increment, we will develop a different sentiment analysis model from start to finish, but we will do it in several iterations. In the first iteration we will design, code and evaluate a prototype using a given approach, which will be further improved throughout the following iterations.

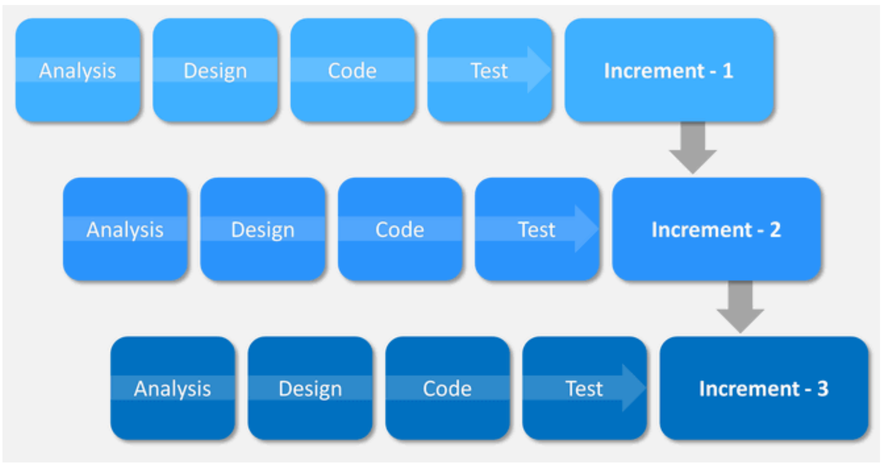
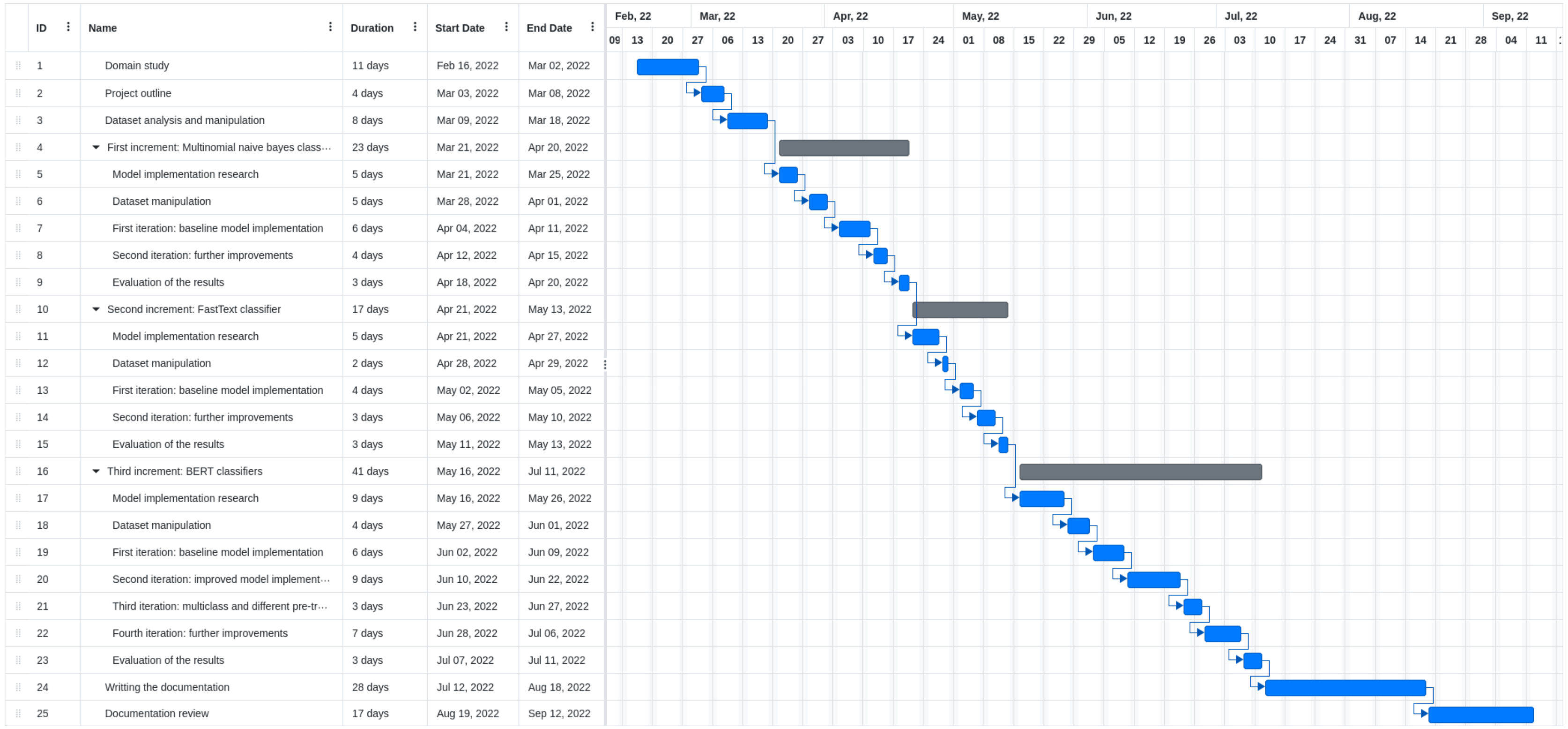


Figure 3.1: Incremental development model.



ID	Name	Duration	Start Date	End Date
1	Domain study	11 days	Feb 16, 2022	Mar 02, 2022
2	Project outline	4 days	Mar 03, 2022	Mar 08, 2022
3	Dataset analysis and manipulation	8 days	Mar 09, 2022	Mar 18, 2022
4	First increment: Multinomial naive bayes class···	23 days	Mar 21, 2022	Apr 20, 2022
5	Model implementation research	5 days	Mar 21, 2022	Mar 25, 2022
6	Dataset manipulation	5 days	Mar 28, 2022	Apr 01, 2022
7	First iteration: baseline model implementation	6 days	Apr 04, 2022	Apr 11, 2022
8	Second iteration: further improvements	4 days	Apr 12, 2022	Apr 15, 2022
9	Evaluation of the results	3 days	Apr 18, 2022	Apr 20, 2022
10	Second increment: FastText classifier	17 days	Apr 21, 2022	May 13, 2022
11	Model implementation research	5 days	Apr 21, 2022	Apr 27, 2022
12	Dataset manipulation	2 days	Apr 28, 2022	Apr 29, 2022
13	First iteration: baseline model implementation	4 days	May 02, 2022	May 05, 2022
14	Second iteration: further improvements	3 days	May 06, 2022	May 10, 2022
15	Evaluation of the results	3 days	May 11, 2022	May 13, 2022
16	Third increment: BERT classifiers	41 days	May 16, 2022	Jul 11, 2022
17	Model implementation research	9 days	May 16, 2022	May 26, 2022
18	Dataset manipulation	4 days	May 27, 2022	Jun 01, 2022
19	First iteration: baseline model implementation	6 days	Jun 02, 2022	Jun 09, 2022
20	Second iteration: improved model implement···	9 days	Jun 10, 2022	Jun 22, 2022
21	Third iteration: multiclass and different pre-tr···	3 days	Jun 23, 2022	Jun 27, 2022
22	Fourth iteration: further improvements	7 days	Jun 28, 2022	Jul 06, 2022
23	Evaluation of the results	3 days	Jul 07, 2022	Jul 11, 2022
24	Writting the documentation	28 days	Jul 12, 2022	Aug 18, 2022
25	Documentation review	17 days	Aug 19, 2022	Sep 12, 2022

Figure 3.2: Gantt diagram of our project.

3.2 Task planning

The Gantt diagram shown in Figure 3.2 represents the development time frame in a best case scenario. In order to be more precise with the time frame, we've made an educated guess based on other similar project's time frames.

- **Domain Study**: The tasks of to this group are oriented towards acquiring the necessary knowledge regarding NLP techniques and models, as well as getting acquainted with the tools that are used to develop this project.
- **Project outline**: Tasks that relate to the elaboration of the project planning, deciding what models to implement to tackle the main objectives of the project as well as a scheduling them in an approximate time frame.
- **Dataset manipulation**: These tasks correspond to the study of the given dataset, in order to analyze possible ways of reshaping it or even changing it depending on the model we are going to implement.
- **Writing the documentation**: Tasks corresponding to the documentation of this work, as well as the organization of the information and the synthesis of the conclusion of this work.

Every sentiment analysis model we implement will have its associated increment and as many iterations as it needs to be improved until a point we consider them finished enough. For the main design, development and evaluation of these models, we planned on the following structure:

- **First increment: Multinomial Naive Bayes classifier**
  - **First iteration**: We will develop and evaluate a baseline classifier using the MNB algorithm, that will serve as a first point of reference for the later models.
  - **Second iteration**: We will study and improve the previous model, in order to try to make it reach its fullest potential.
- **Second increment: FastText classifier**
  - **First iteration**: In this iteration we will use a different approach, a FastText classifier. We will also compare it to the previous one.
  - **Second iteration**: We aim to improve the model even further by considering some optimizations and improvements.
- **Third increment: BERT classifiers**

- **First iteration**: This time we will develop a baseline BERT model to tackle the binary classification task, as it takes considerably more time to code than the previous two models. We will test it and compare its performance with the others.
- **Second iteration**: We will redesign the previous model implementation, making changes in order to elevate the quality of the results.
- **Third iteration**: We will build BERT multiclass classifiers in order to address the second task of the project, as well as study and implement some different pre-trained BERT models that can prove to perform better.
- **Fourth iteration**: Finally, we will look into further enhancing the classifiers previously developed by trying hyperparameter optimization techniques. We will compare the results of the different implemented BERT models, as well as the previously developed MNB and FastText classifiers.

Chapter 4

Technological fundamentals

CHOOSING the adequate tools and technologies for a proper development is a crucial part of a project. In our case, as our work is mainly artificial intelligence, machine learning-based, we will choose a programming language that is tailored for that purpose.

4.1 Programming language

Python[1] is a general-purpose, high-level, multi-paradigm interpreted programming language. It supports imperative, object-oriented programming and, to a lesser extent, functional. Has a dynamic type system, allocates memory when needed and uses a garbage collector that checks the dereferenced memory to free it.

We decided to pick Python as our main programming language for this project, as it is one of the most popular languages in the data science field [24]. It provides a lot of libraries which feature many utilities and tools for developing these kind of projects, as we'll see in Section 4.3.

Thanks to this, it proves to be one of the best languages for the incremental and iterative and incremental strategy that we will use for this work, as detailed in Chapter 3. It will allow as to quickly prototype and implement the classifiers, which will let us focus on the evaluation and improvement of said classifiers.

4.2 Development tools

4.2.1 Integrated Development Environment

As for the development of this project, we decided to use Pycharm[2] as our IDE. It a multi-platform program, compatible with Linux, macOS, and Windows architectures, as well as

[1] https://docs.Python.org/3/faq/general.html
[2] https://www.jetbrains.com/pycharm

being one of the most used IDEs for Python development. Its quality is on par with the rest of the JetBrains catalogue, thanks to the powerful features it provides. These features range from an integrated debugger and test runner, a Python profiler, a built-in terminal, integration with major version control systems and even more, as well as providing ample customization via plugins for the user interface.

We chose this IDE thanks to its features and thanks to the fact that we were already familiarized with it, so we would have a smoother development

4.2.2 Version control

As a version control system, we decided to use Git[3]. Git is an open source distributed version control system designed to handle everything, from small to very large projects, with speed and efficiency. Specifically, we chose Github[4] as a Git platform, as we are also familiarized and used to it.

Having a strict version control allows for a more controlled and safe workflow, being able to properly track the evolution of a project are fundamental for developing a quality project.

4.3 Main libraries

4.3.1 Numpy

NumPy is a library that adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate them. It is an essential library for any machine learning project.

4.3.2 Pandas

Pandas[5] is a Numpy-based general purpose data analysis, cleaning and pre-processing open source library. It provides fast, flexible and high-level data structures to easily perform data analysis with labeled datasets. It also features powerful functionalities to combine, split, re-arrange or reorder sets, so that makes it the perfect tool for us to manipulate our dataset.

4.3.3 EasyNMT

EasyNMT[6] is a library that provides some functionalities to easily use state-of-the-art machine translation models for an enormous variety of languages. It automatically downloads the pre-

[3] https://git-scm.com/
[4] https://github.com/
[5] https://pandas.pydata.org/
[6] https://pypi.org/project/EasyNMT/

trained translation models, and can be used for both single sentences and whole documents. It even features threaded translation, in order to speed up the process.

4.3.4 Nlpaug

Nlpaug[7] is a library that features data augmentation techniques to expand datasets, specially useful with machine learning projects. It's specialized in both text and signal augmentation. Text augmentation can prove really useful in our project, taking into account that this library features a variety of pre-trained models that can insert, substitute, remove, swap words or even characters in sentences. We explain in detail data augmentation in Section 6.1.1.

4.3.5 Matplotlib and Seaborn

Both libraries provide data visualization functionalities. Matplotlib [8] can produce a variety of different plots and graphics, as well as an ample customization of color palettes and display settings.

Seaborn[9] is built on top of Matplotlib, and it can also compute metrics given the data by itself, and generate the corresponding graphics

4.3.6 Scikit-learn

Scikit-learn[10] is a Python machine learning module that includes a variety of functionalities, including classification, regression and clustering algorithms, as well as pre-processing techniques and model selection features, such as detailed metrics reports from the evaluation of supervised models.

4.3.7 NLTK

NLTK[11] is one of the most popular libraries for natural language processing. It provides a large set of functionalities for classification, tokenization, stemming, parsing and more, for a large variety of languages.

4.3.8 FastText

FastText[12] is an open-source, free, lightweight library that allows users to set up a variety of word representation models to learn text representations. It also provides functionalities to

[7] https://pypi.org/project/nlpaug/
[8] https://pypi.org/project/matplotlib/
[9] https://pypi.org/project/seaborn/
[10] https://scikit-learn.org
[11] https://www.nltk.org/
[12] https://fasttext.cc/

create different types of text classifiers in a quick and simple way

4.3.9 Pytorch

Pytorch[13] is a machine learning framework, that presents a variety of functionalities to develop a variety of applications such as computer vision, deep learning or natural language processing. It allows building complex neural network with ease thanks to the Tensors, optimized multi-dimensionalk arrays that can be computed with the GPU.

It's one of the most used deep learning tools in the research community as it is flexible, fast, easy to get prototypes up and running, and it boasts a lot of well documented examples and guides, which makes it even more approachable.

4.3.10 Transformers

Transformers[14] is a Python library that provides the user with already pre-trained, state-of-the-art models in a variety of artificial intelligence domains, including NLP, artificial vision, audio related tasks and even multimodal tasks that combine them. It also features integration with Pytorch, which will facilitate the implementation of our models.

4.3.11 Ray tune

Ray tune[15] is a library that features experiment execution and hyperparameter tuning at any scale, allowing for quicker executions thanks to its multi-threaded approach. It provides integrations with many machine learning frameworks, such as Pytorch, and also provides a lot of state-of-the-art algorithms to perform hyperparameter tuning.

[13] https://pytorch.org/
[14] https://huggingface.co/docs/transformers/index
[15] https://docs.ray.io/en/latest/tune/index.html

Chapter 5

Theoretical fundamentals

5.1 Multinomial Naive Bayes classifier

In probability theory and statistics, Bayes' theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

$$P(A|B) = P(A)\frac{P(B|A)}{P(B)} \tag{5.1}$$

Following its formula, we can describe it as the probability of A happening, given that B has occurred. In this example, B is the evidence and A is the hypothesis. The assumption made here is that the features are independent of each other. That is presence of one particular feature does not affect the other.

That leads us to the Naive Bayes Classifier. It is a supervised learning algorithm used for classification tasks, that uses features to make a prediction on a target variable. As it employs the Bayes' theorem, it assumes that features are independent of each other and there is no correlation between them. However, this is not the case in real life. This assumption of features being uncorrelated is the reason why this algorithm is called "naive".

It is computationally very efficient and easy to implement. There are two event models that are commonly used: the multivariate and the multinomial model. The latter one, frequently called Multinomial Naive Bayes (MNB), generally outperforms the multivariate one [25]. However, it is still inferior to support vector machine classifiers, in terms of accuracy [26], but as a baseline model, its performance will be enough for us.

5.1.1 Feature Extraction

In the bag-of-words approach a text is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity (the number of times each word occurs in the text). Each text has the words as their features, and each feature can take on an integer value counting the number of times the particular words occurs in said text.

We first have to determine the set of words (also called ”dictionary”) of our dataset by reading all the texts in it. Then, for each text, we record the number of times each of the words in the dictionary occurs in it, including those that did not occur by giving them a value zero.

We apply the Term Frequency - Inverse Document Frequency (TFIDF) [27] formula to all the texts in our dataset, as to then apply the MNB algorithm. The TFIDF is a numerical statistic that is intended to reflect how important a word is to a text in a dataset. The TFIDF value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general. Its formula is shown in Equation 5.2, where D represents the total number of texts, df is the number of documents containing the desired word and f represents the original word frequency.

$$TFIDF(word) = \log{(f+1)} \times \log{(\frac{D}{df})} \tag{5.2}$$

5.1.2 Classification

Now we will explain how MNB computes class probabilities for a given text. Let C be the set of classes, c be any class belonging to C and N the size of our vocabulary. The classifier assigns a test tweet/gab t_i to the class that has the highest probability P(c|t_i), which, applying Bayes' rule 5.1 is given by Equation 5.3:

$$P(c|t_i) = P(c)\frac{P(t_i|c)}{P(P(t_i)}, \qquad c \in C \tag{5.3}$$

The probability of the prior class P(c) can be estimated by dividing the number of texts belonging to the class c by the total number of texts. P(t_i|c) is the probability of obtaining a text like t_i in the class c, and it is calculated as denoted by Equation 5.4:

$$P(t_i|c) = (\sum_n f_{ni})! \prod_n \frac{P(w_i|c)^{f_{ni}}}{f_{ni}}, \tag{5.4}$$

where f_{ni} is the count of word n in our test text t_i and P(w_n|c) is the probability of word n given class c. The latter probability is estimated from the training text as seen in Equation 5.5,

$$\hat{P}(w_n|c) = \frac{1+F_{nc}}{N+\sum_{x=1} NF_{cx}}, \tag{5.5}$$

where F_{xc} is the count of word x in all the training documents belonging to class c, and the Laplace estimator is used to prime each word's count with one to avoid the zero-frequency problem [25]. The normalization factor P(t_i) in Equation 5.3 can be computed using Equation 5.6:

$$P(t_i) = \sum_{k=1} |C| P(k) P(t_i|k). \tag{5.6}$$

Note that the computationally expensive terms $\sum_n f_{ni})!$ and $\prod_n f_{ni})!$ in Equation 5.4 can be deleted without any change in the results, as neither depend on class c. This way, Equation 5.4 can be rewritten as shown in Equation 5.7, where α is a constant that drops out because of the normalization step.

$$P(t_i|c) = \alpha \prod_n P(w_n|c)^{f_{ni}} \tag{5.7}$$

5.2 Neural Networks

Neural networks are a part of machine learning and constitute the core of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another, and they started when Fran Rosenbflatt conceived the perceptron in the 1950s [28].

The perceptron is the mathematical representation of the workings of a human neuron, and thus it is the most basic neural network. It is composed by a vector of inputs $\vec{X} = [x_0, x_1, ..., x_n]$ that are associated to a set of features, alongside a weight vector $\vec{W} = [w_0, w_1, ..., w_n]$ that is initially randomized. The perceptron is tasked with combining these values as the sum of the multiplication of each input-weight pair $(x_0 \times w_0) + (x_1 \times w_1) + ... + (x_n \times w_n)$. Finally, a threshold function is applied to the sum, resulting in the following Equation 5.8.

$$f(x) = \begin{cases} 1 & if(\sum_{i=0}^{n} x_i \times w_i) > threshold \\ 0, & \text{otherwise} \end{cases} \tag{5.8}$$

It also features an error function $err(x) = ||yf(X)||$ (also known as loss function) that compares the output of the perceptron with the known output.The perceptron's architecture can be seen in Figure 5.1.

By combining many perceptrons in a layered structure, we can form a Multi Layer Perceptron (MLP). The input passes through the first layer, whose outputs are connected to the inputs of the second layer and so forth. But in order to make the neural network work, we need to train it.

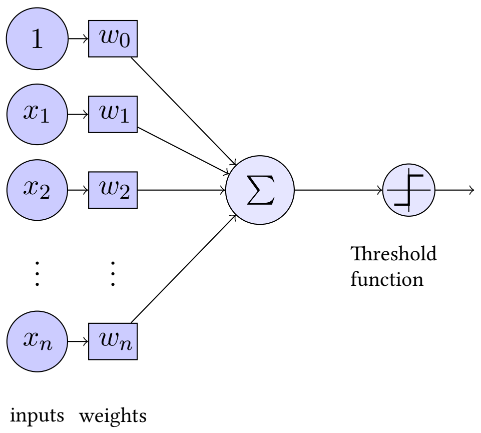


Figure 5.1: Representation of the perceptron's architecture

In the case of supervised learning, each data sample has a label that contains the information to which class it belongs to. The general idea is to make the neural network learn the patterns relative to each class, so when we input an unkown data sample it will be able to predict its class.

For this purpose, we will apply the backpropagation algorithm, which consists on recalculating the network's weights according to the error function, starting in the last neuron layer and proceeding towards the first input layer. The algorithm consists on the following steps:

- Initalize all weights with random values.
- Feed data into the network and calculate the value of the error function.
- Calculate the gradients of the error function with respect to each weight.
- Keep updating the weights in an iterative way until the error becomes lower than the stablished error threshold, or the maximun number of iterations is reached.

The general formula to update the weights is given as shown in Equation 5.9

$$w \leftarrow w - \alpha \frac{\partial Error}{\partial w}, \tag{5.9}$$

where the weight value at the current iteration is the value of the previous iteration minus a value proportional to the gradient. The parameter α represents the learning rate, which determines the amount of error correction that is applied to the weights. If the learning rate is too big, the corrections to the weights might be inefficient, so the error can increase, and if it is too small, the algorithm might get stuck without improving at all. The expression

$\frac{\partial Error}{\partial w}$ represents computing the partial derivatives of the error fucntion E with respect to each weight of the array w.

Regarding the training process, there exist diffferent methods to apply backpropagation. One of them is the stochastic gradient descent [29], where the weights of the network are updated after each training sample is processed. This approach is quite slow, as the weights are updated as many times as the amount of training samples that are used. In the other hand, batch learning is a faster method where a batch of samples are used to train the network, and then the weights are updated. This method is much less computationally expensive, but it can provide worse results due to the reduced weight update frequency.

Moreover, we can train the network in more than one epoch. An epoch is defined as an entire transit of the training data through the network. Increasing the number of epochs helps reduce the error further, but it can also adjust the weights too much in favor of the given inputs, so in this case the model would not be learning the characteristics of the inputs, but learning the inputs themselves. This phenomenon is known as overfitting, and it gives the illusion of good performance when in reality the model is just adapting to the training samples.

5.2.1 Neural Network Language Model

The neural network language model [30] is a probabilistic feedforward neural network specifically tailored for NLP tasks. Its architecture consists of input, projection, hidden and output layers. It becomes computationally complex between the projection and the hidden layer, as values in the projection layer are dense. For a common choice of 10 input neurons, the size of the projection layer might be 500 to 2000, while the hidden layer size is typically 500 to 1000 units. Moreover, the hidden layer is used to compute probability distribution over all the words in the vocabulary, resulting in an output layer with dimensionality as big as the number of the words present.

Several optimizations were proposed and applied to this model, such as using the Huffman tree based hierarchical softmax [31]. In the end, this model served as a base to be used to create more efficient models, like the CBOW, that will be described in Section 5.4.1.

5.3 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a neural network that has the following architecture, as shown in Equation 5.2.

It has an input layer, that takes in a sequence $x = (x_0, x_1, ..., x_T)$, a hidden layer h and an optional output layer y. At each time step t, the hidden state h_t of the RNN is updated by the Equation 5.10, where f is a non-linear activation function, like a sigmoid or a sotftmax

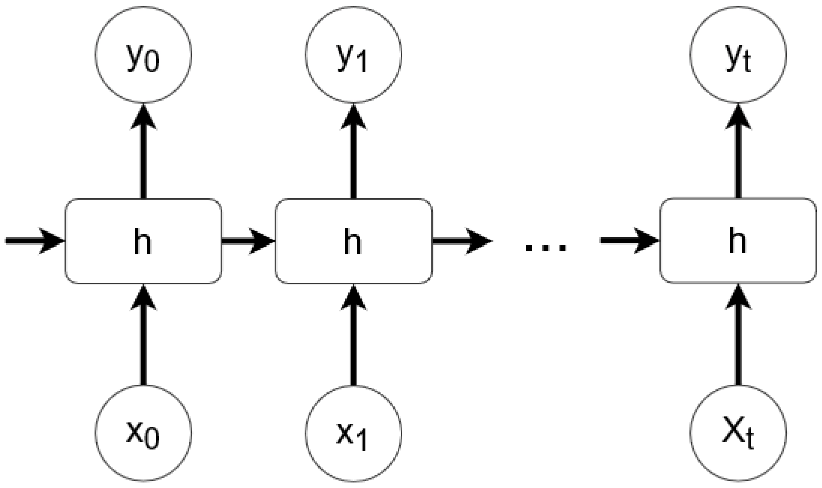


Figure 5.2: Recurrent Neural Networks architecture

function, for example.

$$h_t = f(h_{t-1}, x_t). \tag{5.10}$$

5.3.1 Recurrent Neural Network Language Model

Recurrent Neural Network Language Model (RNNLM) was proposed to overcome certain limitations of the NNLM model, (described Section 5.2.1) such as the need to specify the context length, and because RNNs can efficiently represent more complex patterns [32].

5.4 Word2Vec

Word2vec [33] is a group of related models, composed by shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. These vectors that are assigned to words receive the name of word embeddings, and they are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another. In Figure 5.3 we have a visualization of the vector space model.

There are two implementations of Word2Vec, as described by Mikolov et al. in their paper. Their main goal is to be efficient and simple, minimizing their computational complexity. These two architectures are based on previous works by Krbec and Mikolov et al., where they found that NNLM can be successfully trained in two steps: first, continuous word vectors are learned using simple model, and then the n-gram NNLM is trained on top of the distributed

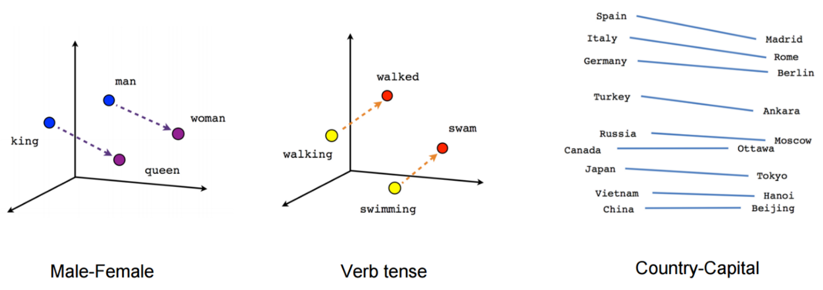


Figure 5.3: Example of the vector representation of the words' relationships

representations of words. For that purpose, they remove the non-linear hidden layer present in the NNLM and RNNLM models [33].

An n-gram is a contiguous sequence of n items from a given sample of text or speech. The items can be phonemes, letters, words, or base pairs according to the application. The n-grams are typically collected from a text or speech corpus. It typically consists of words, and they can be grouped in pairs to form bigrams, or in trios to form trigrams. These new groups can provide more context, as they can extract the meaning of compound verbs or other word formations.

5.4.1 Continuous bag of words model

CBOW model is derived from the NNLM, with the exception that it has no linear hidden layer, as mentioned before, and the projection layer is shared among all the words. The objective function of CBOW model is to predict the middle word when given a window size N, using the previous $N/2$ words and the following $N/2$ words. In the projection layer, the word vectors of the window words are averaged. There is no relevance of the position of the word in determining the word vector of the middle word, hence it is a bag-of-words model. We can see the model's architecture in Figure 5.4.

An averaged vector is passed to the output layer followed by hierarchical softmax to get distribution over V, which represents the vocabulary size and D is the size of each word representation. CBOW is a simple log-linear model where logarithm of the output of the model can be represented as the linear combination of its weights. Finally, the training complexity of this model can be defined as shown in Equation 5.11

$$Q = N \times D + D \times \log 2V. \tag{5.11}$$

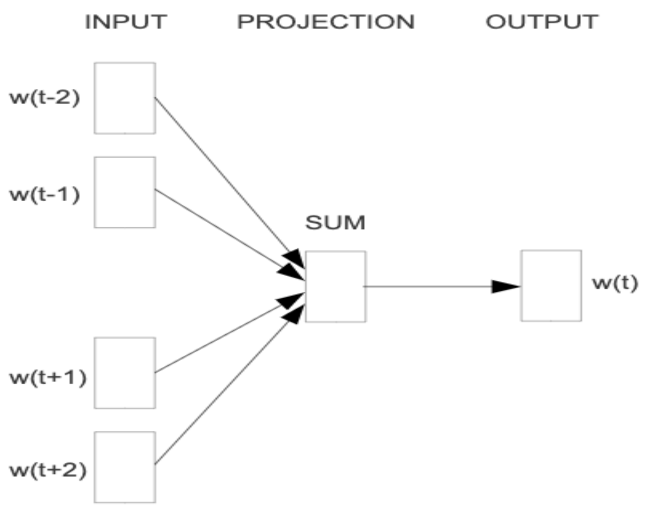


Figure 5.4: CBOW model architecture

5.4.2 Continuous skip-gram model

This model reverses an objective of CBOW model: given the current word, it tries to maximize the classification of the nearby previous and following context words, whereas CBOW tried to predict the current word by using the nearby context words. This model predicts the n-grams words, whith the exception of the current word, as it is used as the input for the model, as we can see in its architecture in Figure 5.5. As distant words are less related to the current word, they are sampled less compared to nearby words for generating output labels.

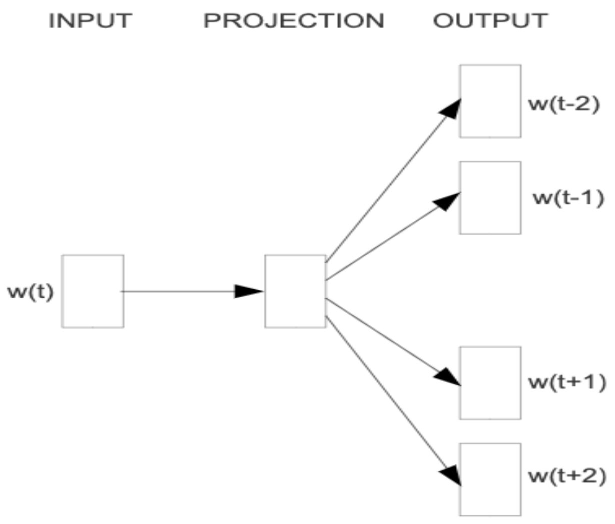


Figure 5.5: Skip-gram model architecture

The total complexity of the model can be calculated as the denoted by the Equation 5.12, where C is the maximum distance of the words, and D is the size of rach word representation. Notice, N also gets multiplied to D × log2(V) term as its not a single class classification problem compared to CBOW, rather N class classification problem. Hence overall complexity of skip gram model is greater than the CBOW model. Increasing the range of the window improves the quality of the resulting word vectors, but it also increases the computational complexity.

$$Q = C\times(D + D\times \log 2V). \tag{5.12}$$

5.5 FastText

FastText is one of the succesors of Word2Vec, primarily based on the skip-gram architecture. The key improvement that FastText achieved over Word2Vec, or even Glove [36], (another succesor of Word2Vec, that predates FastText by 2 years) is to incorporate sub-word information.

As previously shown in section 5.4.2, the skip-gram model has as a goal to learn a vectorial representation for each word of a vocabulary. Given a word vocabulary of size W , where a word is identified by its index $w \in 1, ..., W$, the goal is to learn a vectorial representation for each word w. Inspired by the distributional hypothesis [37], word representations are trained to predict well words that appear in its context.

FastText is able to achieve really good performance for word representations and sentence classification, specially in the case of rare words by making use of character level information. Each word is represented as a bag of character n-grams in addition to the word itself. Using an example, for the word matter, with n = 3, the representations for the character n-grams is <ma, mat, att, tte, ter, er>. < and > are added as boundary symbols to distinguish the n-gram of a word from a word itself, so for example, if the word mat is part of the vocabulary, it is represented as <mat>. This helps preserve the meaning of shorter words that may show up as n-grams of other words, and also allows us to capture meaning for suffixes/prefixes. The length of n-grams can be chosen for minimum and maximum number of characters to use respectively.

We can also turn n-gram embeddings completely off as well by setting them both to 0. This can be useful when the 'words' in our model are not words for a particular language, and character level n-grams would not make sense. The most common use case is when we're putting in ids as our words. During the model update, FastText learns weights for each of the n-grams as well as the entire word token.

The scoring function used for this model is Equation 5.13, where G is the size of a given dictionary of n-grams, w is a word, and $G_w \subset \{1, ..., G\}$ is the set of n-grams in the word w, we associate a vector representation z_g to each n-gram g. With this, we can represent a word by the sum of the vector representations of its n-grams.

$$s(w, c) = \sum_{g \in G_w} {z_g}^T v_c. \tag{5.13}$$

This model allows sharing the representations across words, thus allowing to learn reliable representation for rare words. For optimization purposes, fastText uses a hashing function that maps n-grams by using the $hashingtrick$ [38] with the same hashing function as described in Mikolov et al.. Ultimately, a word is represented by its index in the word dictionary and the set of hashed n-grams it contains.

5.5.1 Model structure

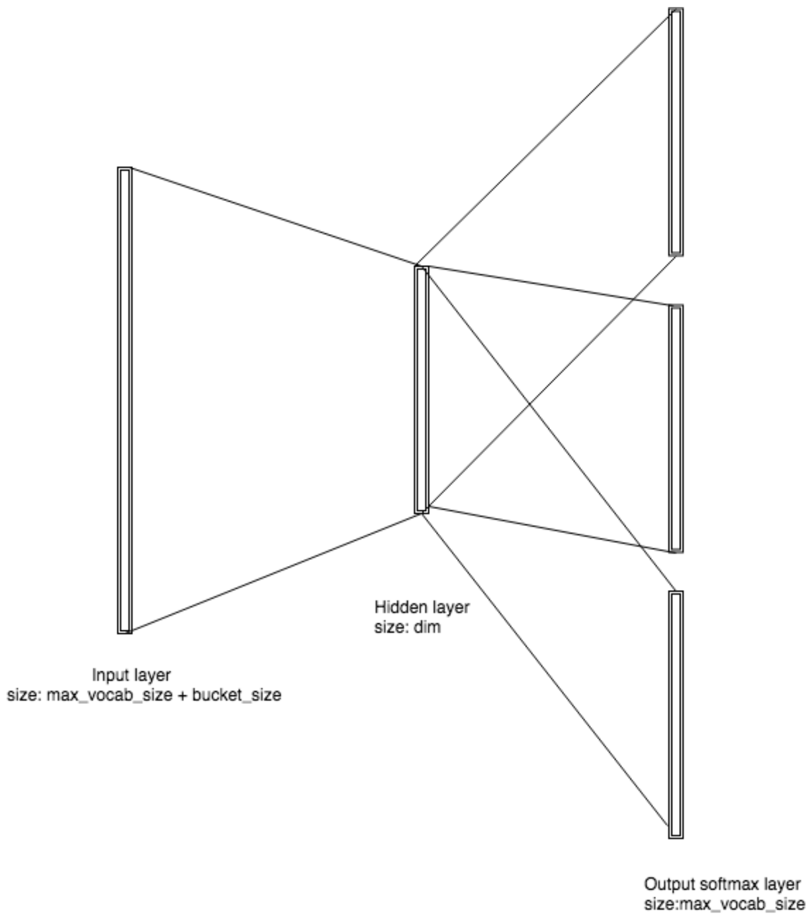


Figure 5.6: Topography of unsupervised Skip-gram fastText model

Figure 5.6 shows us the architecture that a fastText classifier has.

The input layer vectors, if unspecified, are initialized as a matrix of dimension M x N where M is the maximum vocabulary size added to the maximun bucket size, and N corresponds to the dimension both parameters can be setted by the user + $bucket\_size$ and N = dim. $bucket\_size$ corresponds to the total size of array allocated for all the n-gram tokens.

n-grams are initialized by a numerical hash (the same hashing function) of the n-gram text, and fitting the modulo of this hash number onto the initialized matrix at a position corresponding to MAX_VOCAB_SIZE + hash. There could be collisions in the n-grams space, whereas collisions are not possible for original words. This could affect model performance as well.

Dim represents the dimension of the hidden layer in training, and thus the dimension of the embeddings. The matrix is initialized with a uniform real distribution between 0 and 1/dim.

5.6 RNN Encoder-Decoder

The encoder-decoder [2] is an architecture that learns both to encode a variable-length sequence into a fixed-length vector representation and to do the opposite, which is given a fixed-length vector representation decode it back into a variable-length sequence.

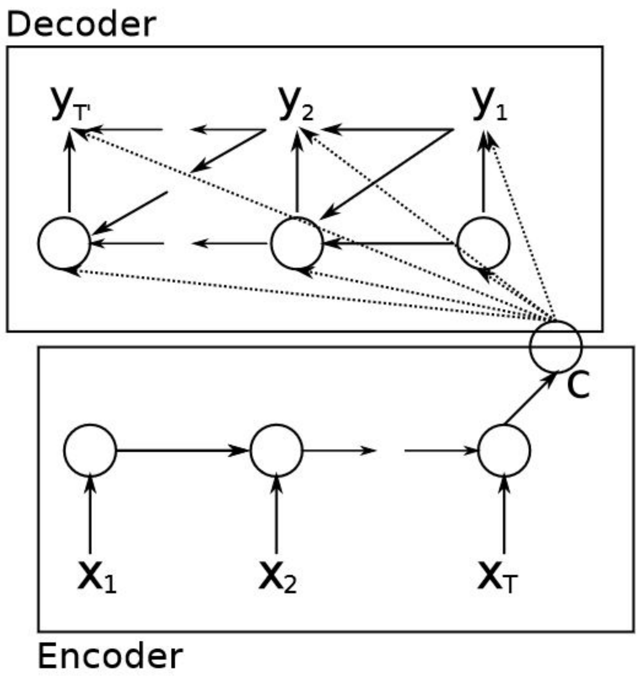


Figure 5.7: Encoder-Decoder architecture [2]

In Figure 5.7 we can see the Encoder-Decoder architecture representation. The encoder is a RNN that receives an input sequence and reads it symbol by symbol, and as it does it, its hidden state changes according to the function 5.10. Upon finishing reading the sequence, the hidden state of the RNN is a summary of the whole input sequence, denoted by c.

On the other hand, the decoder is another RNN that is trained to generate the output sequence by predicting each symbol using the encoder's hidden state h_t.

When the Encoder-Decoder finishes training, it has two possible uses: generating target sequences given any input sequences or score employing a probability model a pair of input and output sequences.

But this sequential approach forbids the usage of parallelization during training, which severely limits the size of the sequences that can be used, taking into account the memory constraints of the hardware. Also, long data sequences proved to be a problem for RNNs, due to the vanishing gradient problem [40]. For example, in machine translation problems, the RNNs have to find connections between long input and output sentences made of dozens of words.

A solution to the long sequences of data is the attention mechanism [41]. It was introduced as an extension to the encoder-decoder model, where instead of encoding a whole input sequence into a vector, it encodes the sequence into a series of smaller vectors and chooses them adaptively while decoding, choosing the most relevant information each time.

5.7 Transformer model

The Transformer model [42] was conceived in order to simplify the previous dominant architecture of complex neural networks that had an Encoder-Decoder model architecture, such as the previously mentioned RNNs 5.6, as well as to facilitate the parallelization of the training process to drastically reduce the time it consumes.

The architecture of the Transformer follows the same principle as the encoder-decoder, as depicted in Figure 5.8. The whole architecture is used when addressing translation, but for other tasks such as text classification only the encoder part is needed.

At its core lies the attention mechanism previously introduced. It is facilitated with the help of keys queries and values. Keys are labels for each word, used to distinguish them. Queries are an active request for specific information, i.e. a specific key from all the available ones. And finally, values are the information that a word contains. Keys and values always come in pairs, and when a query matches a key, it's value gets propagated further into the next layer of the network.

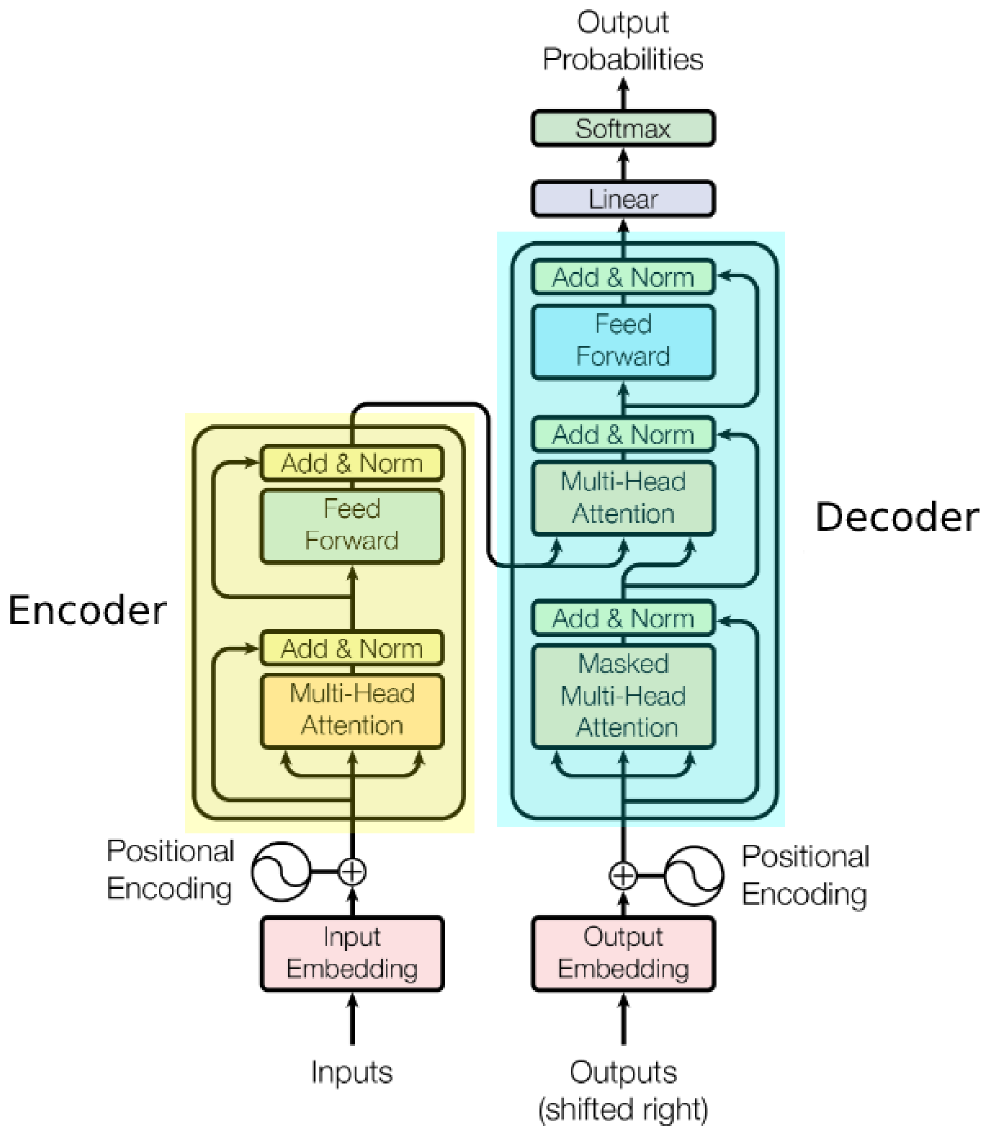


Figure 5.8: Transformer model architecture [2].

5.7.1 Scaled Dot-Product Attention

The transformer model uses what is called Scaled Dot-Product Attention. As shown in Figure 5.9, the process consists on computing the dot products using the query vector q and the key vector k, in order to obtain the vectors k with the most similarity. After this, a softmax function is applied to obtain the results in a [0,1] interval. Finally, we multiply the result by the values vector v, so that the more similar vectors will have higher attention. To speed up the process, the attention function is computed over a set of queries, keys and values packed into matrices Q, K and V respectively. This matrix attention function can be seen in Equation 5.14, where d_k represents the dimension of the keys matrix.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V. \tag{5.14}$$

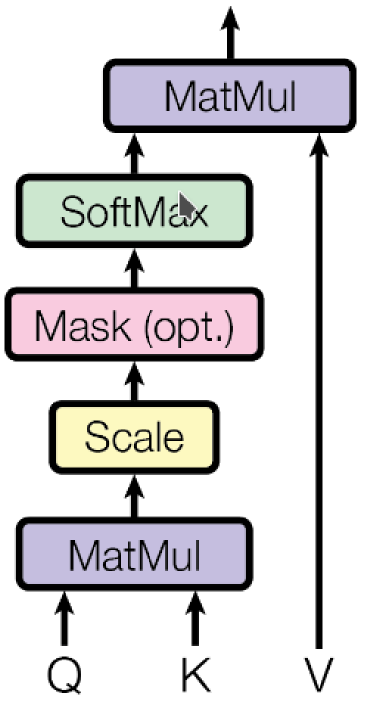


Figure 5.9: Scaled Dot-Product Attention calculation [2].

5.7.2 Multi-Head Attention

In order to efficiently use the attention mechanism, the Transformer model introduces the concept of Multi-Head Attention, where they employ parallelization to incorporate simultaneously various attention heads to the same input. Each attention head is capable of obtaining different relations between the input words where each head obtains slightly different relations, that are concatenated in the final layer, to obtain the results. In Figure 5.10 we have depicted this Multi-Head Attention mechanism

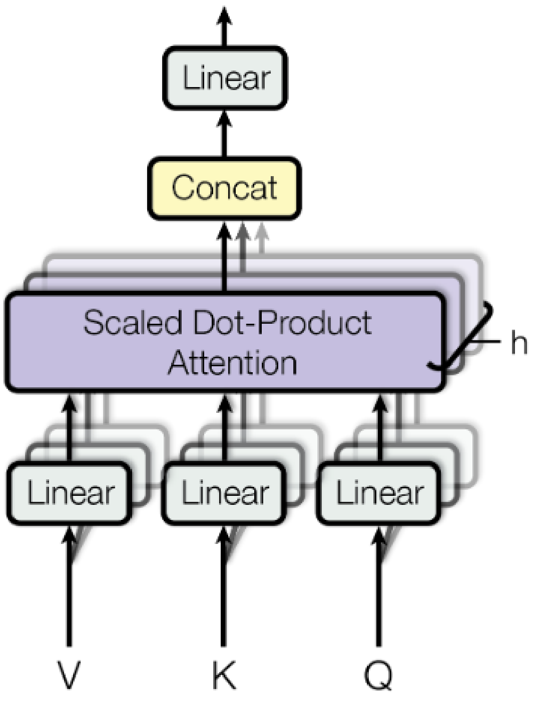


Figure 5.10: Multi-Head Attention diagram [2].

5.8 BERT

BERT [3] is one of the most advanced deep learning models as of today in NLP. Its framework is comprised of two tasks: pre-training and fine-tuning. During the first one, the model is trained on unlabeled data over different pre-training tasks, and for the latter task, the model is first initialized with the pre-trained parameters, and then all the parameters are tuned using the labeled data belonging to the tasks we want to tackle.

5.8.1 Architecture

One of the most distinctive feature that BERT exhibits is that it presents a unified architecture across different applications of the model. The pre-trained parameters are the same for all the tasks we apply BERT on, each one will have a unique fine-tuned model.

It's main architecture is a multi-layer bidirectional Transformer encoder, and as mentioned in [3]. In Figure 5.12, we can see the two pre-training and fine-tuning procedures of the BERT model.

The $BERT_{base}$ is composed of 12 Transformer blocks, with a hidden layer of 768 neurons and 12 self-attention heads. There is another model, the $BERT_{large}$, that features 24 Transformer blocks, 1024 hidden neurons and 16 self-attention heads. Both these two architectures are the base from which other more specialized BERT models emerged. We show such models in section 7.3.2, where we test their performance over our tasks.

5.8.2 Input and output

BERT uses as input token sequences, in which each token represents a word from a sentence, or even a pair of sentences. This input system allows BERT to be able to tackle an even wider array of tasks. In Figure 5.11, we can see a diagram depicting the structure that composes the input sequences.

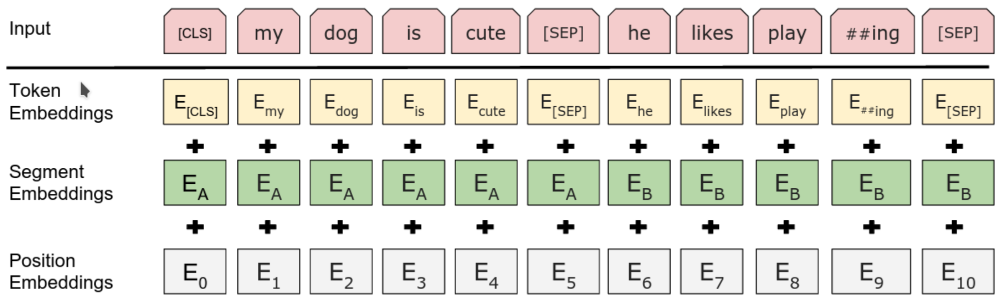

Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Figure 5.11: BERT input sequence diagram [3].

The first token of each sequence is always going to be the special classification token CLS, which is used as the aggregate representation for classification tasks. When multiple sentences are packed in the same sequence, the SEP token is introduced to differentiate them. Every token then receives a learned embedding, indicating the sentence to which it belongs. The input representation of each token is given by summing all its embeddings (token, segment and position).

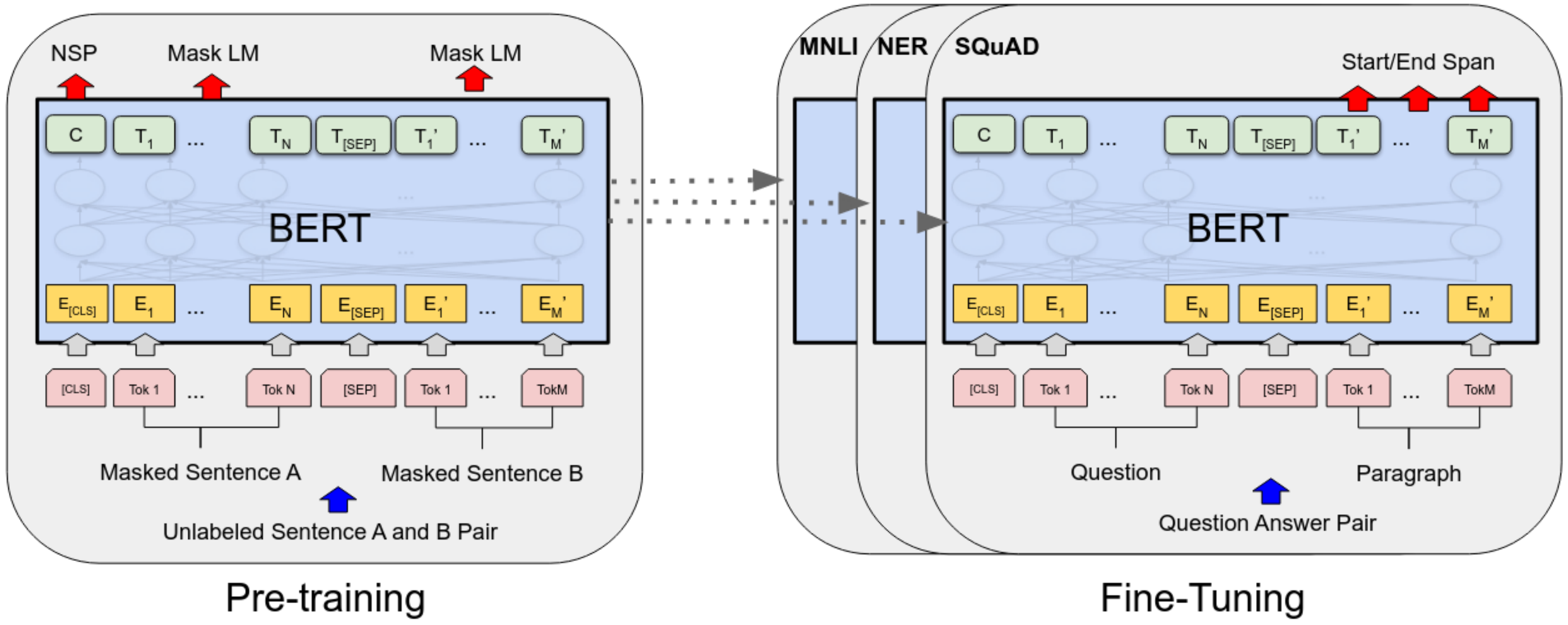


Figure 5.12: BERT architecture, depicting both pre-training and fine-tuning procedures [3].

5.8.3 Pre-training

BERT, being a bidirectional model, can't be pre-trained using left-to-right or right-to-left language models, so it is instead pre-trained using masked language modeling and next sentence prediction tasks.

Masked Language Modeling

For this task, a percentage of the input tokens is masked at random, replacing them with the $MASK$ token, and the network must then predict them. This token only appears during pre-training, and that creates a mismatch with fine-tuning, as that part of the model doesn't feature the $MASK$ token. To mitigate this, only a percentage of words get replaced with the $MASK$ token, while other words just get replaced with another random word.

Next Sentence Prediction

Next sentence prediction aims to understand the relationship between two sentences, which is the main focus of many applications of the model, and it's something that language modeling can't achieve properly. BERT is pre-trained with a binarized next sentence prediction task, due to being easy to generate from any text corpus.

One of the best advantages that BERT provides is that we can use any existing pre-trained model, and fine-tune it to our specific task. This greatly alleviates the time we have to spend in order to prepare a model, as the pre-training is the most resource-intensive process with BERT.

5.8.4 Fine-tuning

Fine-tuning BERT is a really straightforward task, all thanks to the attention mechanism present in the Transformer. This allows to prepare the model for any given application of it, be it single sentence or sentence pair based. For sentence pairs, BERT uses the self-attention mechanism to both encode and apply bidirectional cross attention at the same time. Fine-tuning is computationally less expensive than pre-training, thus it can be achieved a lot faster.

For any given task, we just put in the inputs and outputs into BERT and it will fine-tune all the parameters by itself. At the input, we have our given sentences, outputted from pre-training. The output will be different according to the application of the model. For sequence tagging or question answering tasks, we would get an output layer made of t outputted token representations, while for sentiment analysis applications, the CLS token representation would be our output.

Chapter 6

Materials and methods

BEFORE starting with the main development of the classifiers, we will first have an in depth look at the dataset, analyzing what preprocessing techniques can be applied to it in order to prepare it for the coming tasks. We will also have a look at how to train our classifier models, as well as what metrics to use for their evaluation.

6.1 Dataset

Our dataset consists of a Tab-Separated Values (TSV) file, which is a simple text format for storing data in a tabular structure: each record in the table is one line of the text file, and each field value of a record is separated from the next by a tab character. This is a variation of the well-known Comma-Separated Values (CSV) format. Table 6.1 shows the general structure of the file.

test_case	id	source	language	text	task1	task2
EXIST2021	000001	twitter	en	...	sexist	ideological-inequality
EXIST2021	000002	twitter	en	...	non-sexist	non-sexist
EXIST2021	000003	twitter	en	...	sexist	objectification
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
EXIST2021	006975	twitter	es	...	non-sexist	non-sexist
EXIST2021	006976	twitter	es	...	sexist	objectification
EXIST2021	006977	twitter	es	...	sexist	stereotyping-dominance

Table 6.1: Structure of the dataset's content. The text column contains the unprocessed tweet.

As previously mentioned in Section 1.1, we are given a total of 6 977 tweets and gabs, of which 3 436 are in English and 3 541 in Spanish. Since we will deal with both classifying English and Spanish texts, we can take advantage of this and extend our dataset by translating the English text to Spanish and vice versa, essentially doubling our training samples. We have used the EasyNMT library to perform both translations. The resulting doubled dataset, whose structure is shown in Table 6.2 contains both English and Spanish versions of eath tweet.

Further preprocessing will be applied in each model, as they all have their differences. They will be expanded upon in sections 7.1.1, 7.2.1 and 7.3.1.

test_case	id	source	language	text	task1	task2	English	Spanish
EXIST2021	000001	twitter	en	...	sexist	ideological-inequality	...	...
EXIST2021	000002	twitter	en	...	non-sexist	non-sexist	...	...
EXIST2021	000003	twitter	en	...	sexist	objectification	...	...
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
EXIST2021	006975	twitter	es	...	non-sexist	non-sexist	...	...
EXIST2021	006976	twitter	es	...	sexist	objectification	...	...
EXIST2021	006977	twitter	es	...	sexist	stereotyping-dominance	...	...

Table 6.2: Structure of the doubled dataset, with English and Spanish columns containing their respective translation of the text column.

Now, with our doubled dataset, we are going to check the label distribution. In Figure 6.1 we can see that the binary labels are quite balanced. But on the other hand, if we check the multi-class label distribution in Figure 6.2, we can see a considerable class imbalance between the labels, as the sexist class was divided unevenly between 5 other classes, each one having roughly 1/6 of the representation that the non-sexist class gets.

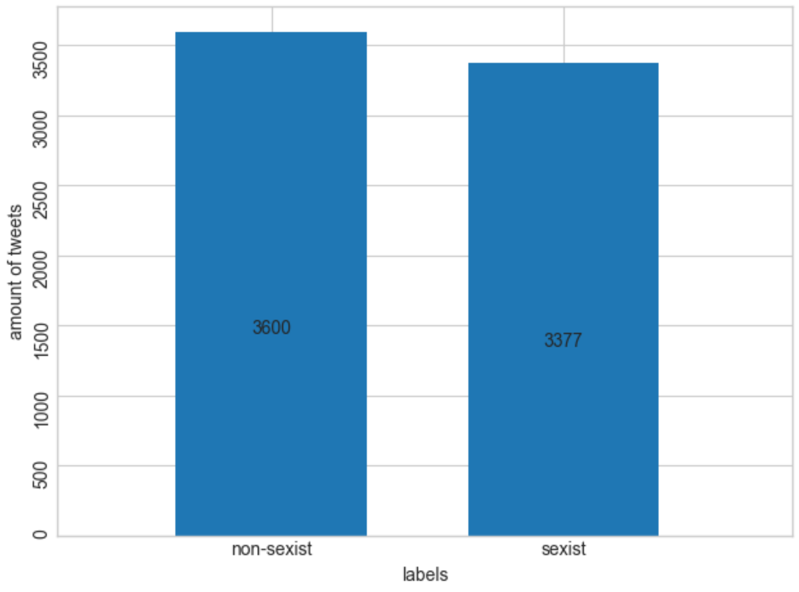


Figure 6.1: Label distribution for the binary classification.

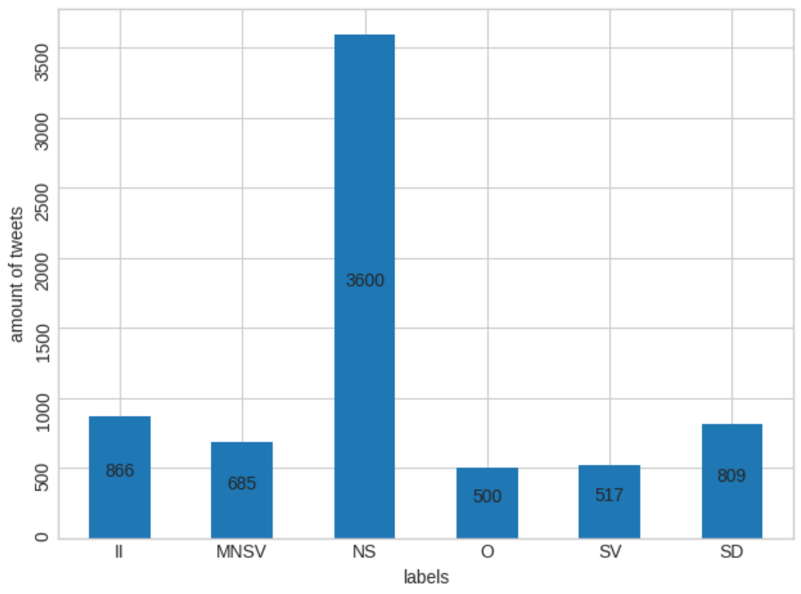


Figure 6.2: Label distribution for multiclass classification. NS: non-sexist. II: ideological inequality. SD: stereotyping dominance. MNSV: misogyny non sexual violence. SS: stereotyping dominance. SV: sexual violence.

6.1.1 Class imbalance

The imbalanced nature of the dataset, as seen in Figure 6.2, can prove to be a problem for the classifiers. This phenomenon can make them unable to properly learn the features of the minority classes, and thus make lots of wrong classifications. There exist a variety of methods and approaches to tackle this problem.

There are two main techniques to balance datasets:

- **Undersampling**: These methods work by reducing the amount of samples from the majority class. This reduction can be done randomly or it can be done by using some statistical knowledge, in which case it's called informed undersampling. Some of the latter methods and iteration methods also apply data cleaning techniques to further refine the majority class samples.

- **Oversampling**: In these methods new samples are added to the minority class (or classes) in order to provide balance. These methods can be categorized into random oversampling and synthetic oversampling. In random oversampling, existing minority samples are replicated in order to increase the size of a given class. On the other hand, in synthetic oversampling, artificial samples are generated for the minority class, ranging from textual augmentation, synonym replacement, back translation, etc.

In our case, we will be using oversampling techniques, as our minority classes are too imbalanced, and our dataset is not very large. We will use the Python library Nlpaug to augment our dataset. Data augmentation consists on adding synthetic samples to our dataset by generating them in various ways. It can be as simple as duplicating the samples of a class, or as complex as using pre-trained language models to generate whole new sentences to add them as samples.

We decided to expand the dataset by duplicating the entirety of the minority classes but applying synonym replacement to them, in order to keep them slightly different to the original samples. We tried back-translation as well, a process in which we translate a sentence into a language and then back to its original one, as to try to obtain a similar meaning but with different wording. In our case, the resulting text entries from back-translation were almost all the same, so we decided to stick only to synonym replacement. In Figure 6.3 we can see the resulting plot of the label distribution.

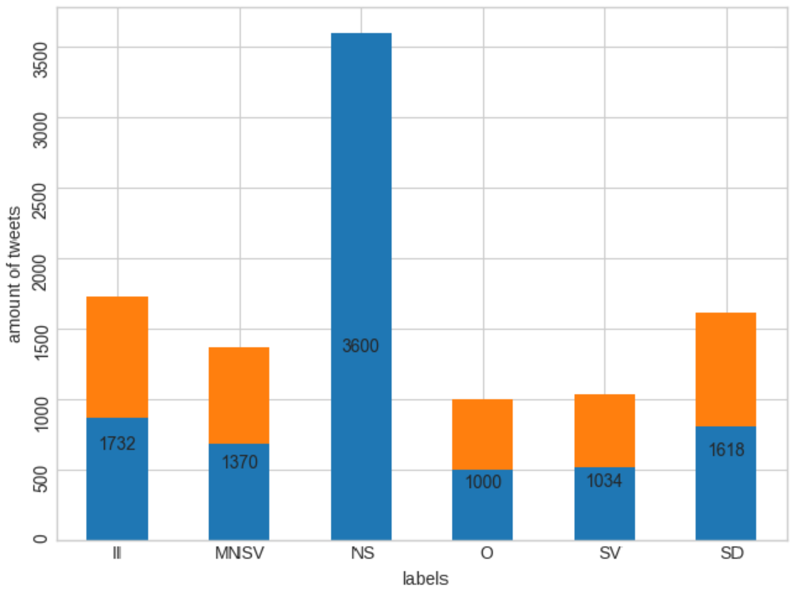


Figure 6.3: Label distribution for multiclass classification, after applying data oversampling with data augmentation and synonym replacement. NS: non-sexist. II: ideological inequality. SD: stereotyping dominance. MNSV: misogyny non sexual violence. SS: stereotyping dominance. SV: sexual violence.

Even though non-sexist tweets still compose the majority of the data, part of the imbalance has been alleviated. This could be further improved with better over-sampling methods, such as using text generation techniques, but we wanted to keep it quick and simple. We won't train any of the binary classifiers with the augmented dataset, as the test with the multiclass classifiers was enough to prove that over-sampling can improve our results.

6.2 Training and evaluation

In order to make our classifying algorithms properly learn the patterns of the text features and evaluate the performance of the classifications, we will divide the execution of each model in two steps: training and evaluation.

All our classifiers are going to be subjected to supervised learning, as we mentioned in Section 2.1.3, so the training step is going to be composed of a series of learning cycles where at the end of each of them, we will evaluate the state of the classifier.

The evaluation step is going to test the model over a subset of the dataset that hasn't been fed to the classifiers before. This test subset is more suitable for applying the evaluation metrics and to reach conclusions about the classifier's performance.

Evaluation metrics

For supervised learning models, there are many metrics that can measure different aspects of the model. In our case, as we are developing classifiers, the following metrics will be used to evaluate performance:

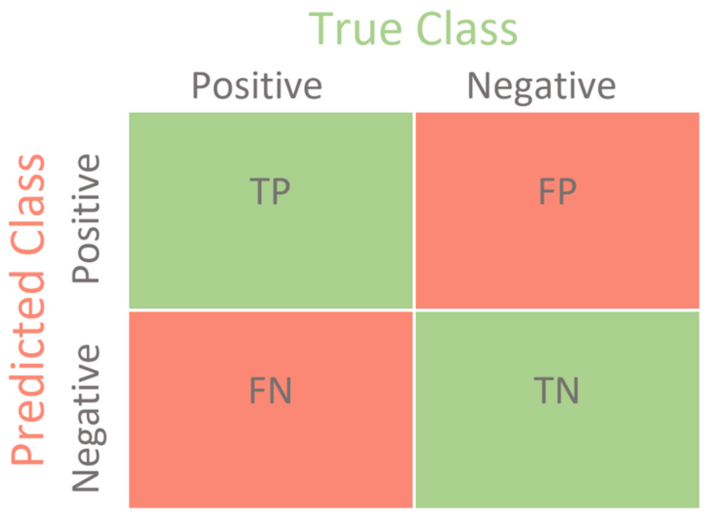


Figure 6.4: Confusion matrix representation.

- **Confusion matrix**: This matrix shows when a class has been wrongly classified as other one, comparing the predicted values with the true ones. The main diagonal shows the cases in which the algorithm has correctly predicted the class, and outside of the cases where it failed are shown. In Figure 6.4, we can see an example of a confusion matrix, where the letters represent the following:

  - **positive (P)**: the model predicted that the class is present.
  - **negative (N)**: the model predicted that the class is not present.
  - **true (T)**: the model predicted correctly.
  - **false (F)**: the model failed to predict correctly.

- **Precision**: It's a metric that represents the amount of true positive class predictions over the total of positive predictions. In the case of minimizing false positives, this metric should be prioritized. It is calculated as shown in Equation 6.1.

$$Precision = \frac{\text{True positive}}{\text{True positive} + \text{False positive}} \tag{6.1}$$

- **Recall**: Also known as sensitivity, it represents the amount of true positive class predictions over the total amount of true positives. It's a relevant metric in any field that aims to minimize the chances of missing positive predictions, and its formula is shown in Equation 6.2

$$Recall = \frac{\text{True positive}}{\text{True positive} + \text{False negative}} \tag{6.2}$$

- **Specificity**: This metric represents the amount of true negative class predictions over the total amount of true negatives. We can see its formula in Equation 6.3.

$$Specificity = \frac{\text{True negative}}{\text{True negative} + \text{False positive}} \tag{6.3}$$

- **F1-score**: This metric combines precision and recall by performing the harmonic mean of both. It works really well for imbalanced datasets, as it requires a good balance between both metrics to produce good results. It's calculated as seen in Equation 6.4

$$\text{F1-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{6.4}$$

- **Receiver Operating Characteristic (ROC) curve**: As shown in Figure 6.5, this graphical plot illustrates the classification ability of a binary classifier system. It is created by plotting the true positive rate (TPR) and false positive rate (FPR). The TPR is the same as the recall, and the FPR can be computed as $(1 - specificity)$. The steeper the curve, means that the classifier is more capable of properly differentiating between the classes.

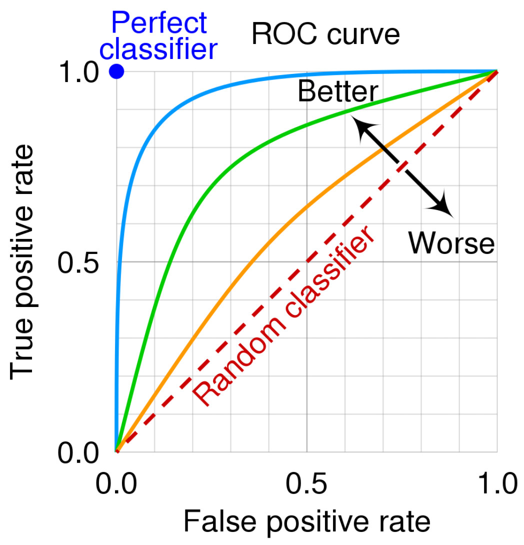


Figure 6.5: Example of a ROC curve.

- **Area Under the Curve (AUC) score**: The AUC score represents the area under the ROC curve, and it's used to summarize the performance of a classifier in a single meassure. It is equivalent to the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative instance, and ranges from 0.5 up to 1, being 0.5 a completely random classifier and 1 a perfect classifier.

Chapter 7

Development and experimental results

7.1 First increment: Multinomial Naive Bayes Model

7.1.1 First iteration

We decided to start off implementing a baseline model, using the MNB algorithm, as previously explained in Section 2.1.3. As we are tasked with classifying text either in Spanish and English, we decided to create one classifier for each language. In principle, this means that each classifier would be trained with only half the dataset. That's why we decided to first extend the dataset by translating all tweets to both languages, as described in Section 6.1.

Regarding the two main classification tasks, we will also create two independent classifiers, one for the binary classification task and the other for the multiclass classification task. This makes up for a total of four classifiers, each one specialized in their respective task and language. The architecture of the resulting classifiers can be seen in Figure 7.1, and it's the same for all of them.

After the initial preprocessing mentioned in section 6.1, we proceed to further preprocess the data in order to properly adapt it. The MNB classifier, being a bag of words model, doesn't require any punctuation marks or grammar whatsoever; so, we will perform the following actions:

- Lowercase the input text.
- Replace ”'t” by ”not”.
- Remove Twitter mentions.
- Isolate and remove punctuation marks, except ”?”.

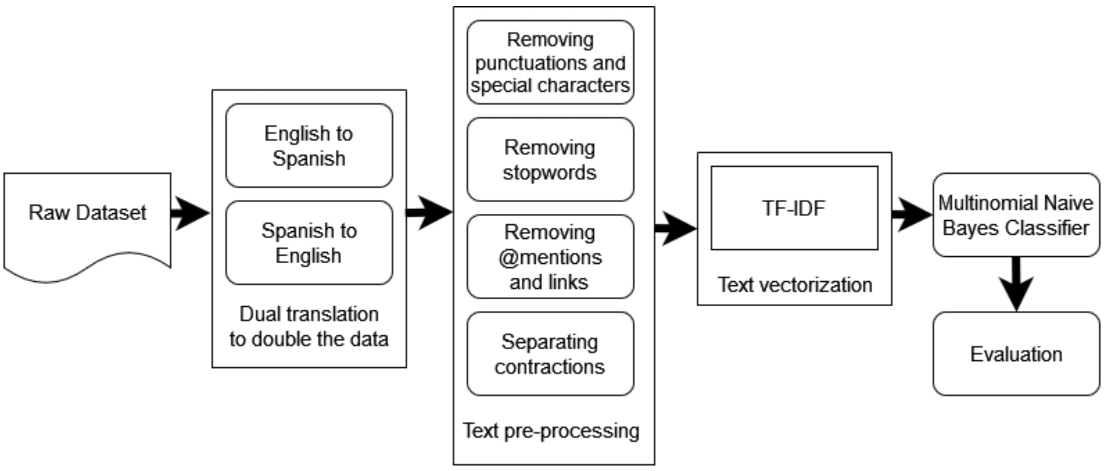

Raw Dataset
English to Spanish
Spanish to English
Dual translation to double the data
Removing punctuations and special characters
Removing stopwords
Removing @mentions and links
Separating contractions
Text pre-processing
TF-IDF
Text vectorization
Multinomial Naive Bayes Classifier
Evaluation

Figure 7.1: Architecture of our Multinomial Naive Bayes classifier.

- Remove other special characters.
- Remove stopwords except ”not” and ”can” (in the case of English texts).

After having preprocessed the tweet content, the dataset is splitted into training and test sets, with the following proportions: 90% training and 10% test.

Next, the input text is vectorized with TF-IDF, using both unigrams, bigrams and trigams (n-grams of size 1, 2 and 3, respectively). This allows to capture the meaning of multi-word expressions (such as compound verbs), that otherwise would be meaningless.

After this, the MNB classifier is initialized with an alpha of 1. This alpha is the hyperparameter of the model (as mentioned in Section 5.1.2) to be fine-tuned in the following iteration.

To evaluate and compare the classifiers, Precision, Recall and F1-score metrics are used, as shown in Table 7.1, with AUC score being used with the binary classifiers as well, as seen in Figure 7.2.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
MNB	Precision	71.85	69.52	41.07	52.87
	Recall	71.85	69.52	41.07	40.94
	F1-score	71.85	69.52	41.07	40.94

Table 7.1: Precision, Recal and F1-score values for the Multinomial Naive Bayes classifier.

The binary English classifier reached an AUC score of 0.7767 and the Spanish one reached 0.7802. These scores indicate that their performance is good, but there is still room for improvement. Regarding the Precision, Recall and F1-score, the binary models performed well

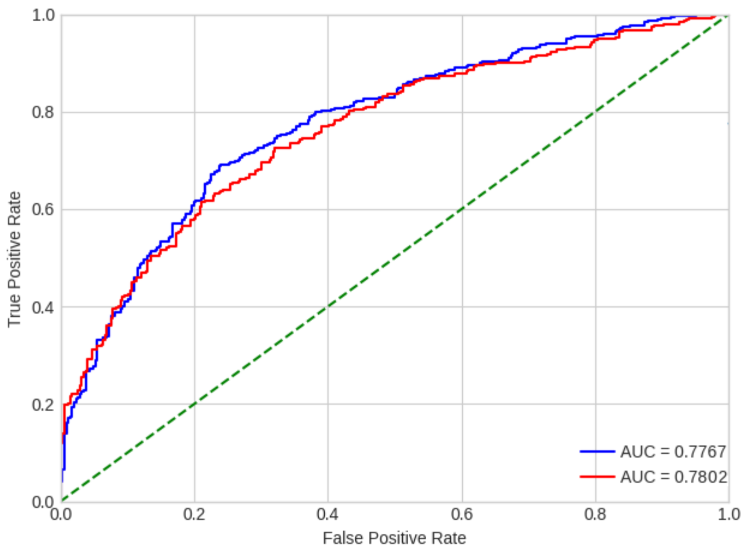


Figure 7.2: AUC plot of the MNB binary classifiers. Red: Spanish. Blue: English.

enough considering this is just a baseline model, but the multiclass ones are quite lacking. In Table 7.2 we can see that they reach very low F1-scores with all sexist classes due to the severe class imbalance present in the dataset, as previously shown in Section 6.1.

Label	English	Spanish	Support
NS	67.73	68.06	346
II	09.26	05.83	99
O	06.90	06.78	55
SV	14.81	11.54	48
SD	25.00	26.67	90
MNSV	14.71	20.29	60

Table 7.2: F1-score of each label of the MNB multiclass classifiers, alongside the number of test samples for each label. NS: non-sexist. II: ideological inequality. SD: stereotyping dominance. MNSV: misogyny non sexual violence. SS: stereotyping dominance. SV: sexual violence.

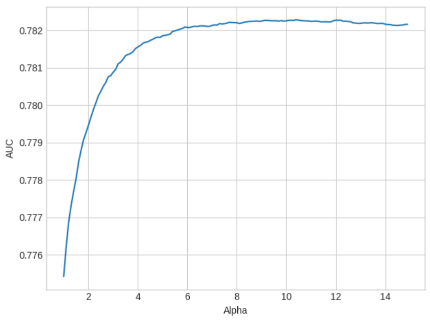
7.1.2 Second iteration

In this iteration we will focus on improving the previous classifiers by fine-tuning them, as well as tackling the dataset imbalance present for the multiclass task.

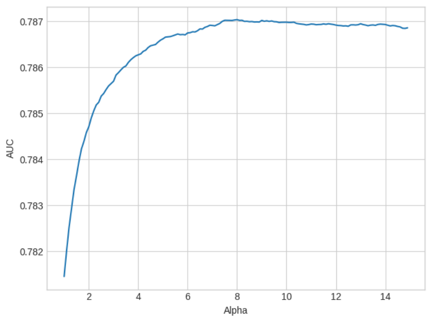
Regarding hyperparameter tuning, there are several techniques to perform it. However, since we just have a single hyperparameter (alpha), we can train and evaluate the models with a different alpha value each time and keep the one that performed the best. In this case we will look for the alpha that yields the best AUC score for the binary classifiers and the best F1-score for the multiclass ones. We can automate this process easily by using cross-validation to shuffle the data and then evaluate each model with a different alpha value.

To tackle the class imbalance problem, as previously explained in Section 6.1.1, we will use the synonym-augmented dataset to train the multiclass classifiers, looking for improvement.

For the binary classifiers, we sampled alphas between 1 and 10 with a step of 0.1, and in Figure 7.3 we can see the the evolution of the AUC over each alpha value. For the multi-class classifiers, we sampled alphas between 0 and 1 with a step of 0.001, and their F1-score evolution can be seen in Figure 7.4.

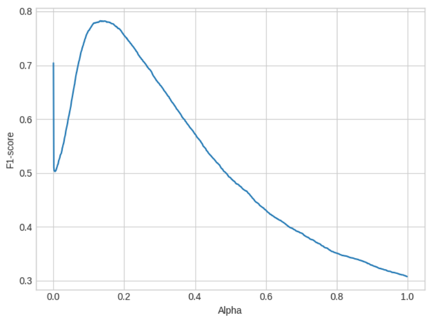


(a) Evolution of the AUC score over alpha in the binary English classifier.

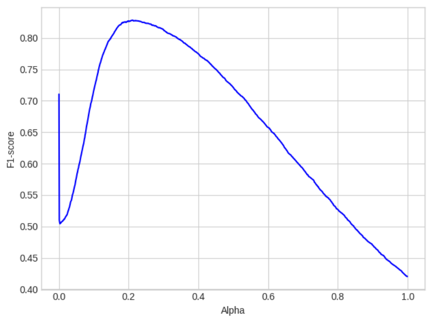


(b) Evolution of the AUC score over alpha in the binary Spanish classifier.

Figure 7.3: Evolution of the AUC score over alpha in the MNB binary classifiers



(a) Evolution of the F1-score score over alpha in the multiclass English classifier.



(b) Evolution of the F1-score score over alpha in the multiclass Spanish classifier.

Figure 7.4: Evolution of the F1-score score over alpha in the MNB multiclass classifiers

After optimizing alpha, we can see in Figure 7.5 that the AUC score of the binary classifiers improved by 0.1. Regarding Precision, Recall and F1-score, we can see in table 7.3 that the binary classifiers performed almost the same, but the multiclass classifiers reached a Precision of 70%, improving by 30% and 20% in the English and Spanish classifications, respectively. The Recall and F1-score of the Spanish classifier improved a 10%, but for the English one they remained the same. Both multiclass classifiers were able to reach more balanced F1-scores for each class, as shown in Table 7.4. This shows that increasing the dataset can improve the classifiers' ability to distinguish the different classes.

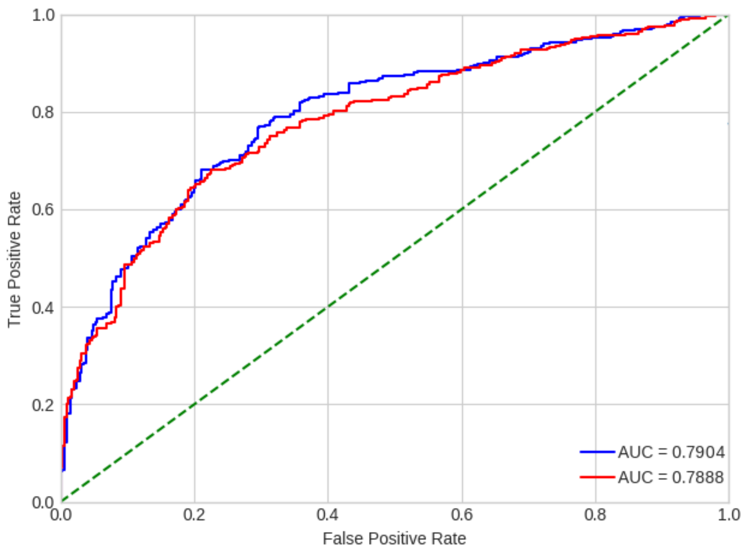


Figure 7.5: AUC plot of both English and Spanish MNB classifiers. Red: Spanish. Blue: English.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
MNB	Precision	73.36	71.94	70.84	77.26
	Recall	69.63	69.48	45.37	56.66
	F1-score	68.45	68.68	37.77	52.42

Table 7.3: Scoring metrics of each Multinomial Naive Bayes classifier after training the multiclass with the augmented dataset and applying hyperparameter tuning.

Since fine tuning alpha didn't bring much of an improvement to our model, we will now focus on implementing more powerful models.

Label	English	Spanish	Support
NS	55.78	60.87	346
II	41.35	76.66	175
O	20.34	24.59	104
SV	17.65	19.42	93
SD	41.05	63.16	179
MNSV	10.74	29.94	139

Table 7.4: F1-score of each label of the MNB multiclass classifiers, alongside the number of test samples for each label, after training with the augmented dataset. NS: non-sexist. II: ideological inequality. SD: stereotyping dominance. MNSV: misogyny non sexual violence. SS: stereotyping dominance. SV: sexual violence.

7.2 Second Increment: FastText Model

The MNB classifier was a good baseline model that performed relatively fast and provided good enough results. Now we will take it a step further and implement a classifier using FastText, so we can study whether neural networks can provide us with better results.

7.2.1 First iteration

For the first iteration, we decided to approach both binary and multiclass tasks, as well as both languages separately. This decision was made due to the ease of implementation that FastText provides. The architecture of the resulting classifiers can be seen in Figure 7.6. It is very reminiscent of that of the MNB, as they both are bag-of-words models.

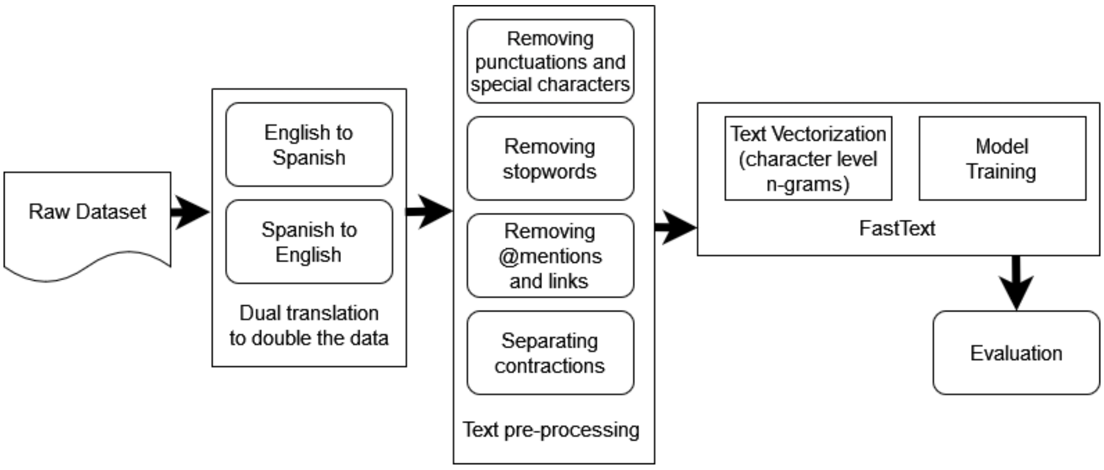

Raw Dataset
English to Spanish
Spanish to English
Dual translation to double the data
Removing punctuations and special characters
Removing stopwords
Removing @mentions and links
Separating contractions
Text pre-processing
Text Vectorization (character level n-grams)
Model Training
FastText
Evaluation

Figure 7.6: Architecture of our FastText classifier.

We started by taking the dataset, and applying to it the same preprocessing as with the MNB classifier but, this time, we replaced existing links with ”link” and hashtags with ”tag”. This is done because not removing this information could be beneficial to the classifier, as it provides the classifier with a little more context.

Next, we splitted the dataset into both training (90%) training and test (10%) sets, same as before. We chose to use the default hyperparameters that the training function provides, as we are yet to establish a baseline. The results shown in Table 7.5 depict all four classifiers and their performance.

We've got the same problem as before regarding the performance of the multiclass classifiers. Table 7.6 presents the F1-score values corresponding to each label, as well as the number of training samples. At this point, without applying further improvements, the binary classifiers performed the same as the MNB ones, while multiclass performed a 15% better. The baseline FastText model already performed better than the previous model, and it has the ad-

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
FastText	Precision	70.49	72.22	57.02	58.06
	Recall	70.49	72.21	57.88	58.60
	F1-score	70.48	72.20	56.87	58.08

Table 7.5: Scoring metrics of the first implementation of FastText classifiers.

vantage of being really simple and quick to set up, train and evaluate. We conclude that this is an overall better model.

Label	English	Spanish	Support
NS	69.31	71.71	346
II	50.00	62.07	99
O	47.83	24.44	55
SV	46.67	39.02	48
SD	46.75	43.48	90
MNSV	29.85	40.88	60

Table 7.6: F1-score of each label of the FastText multiclass classifiers, alongside the number of test samples for each label. NS: non-sexist. II: ideological inequality. SD: stereotyping dominance. MNSV: misogyny non sexual violence. SS: stereotyping dominance. SV: sexual violence.

7.2.2 Second iteration

Now, to improve our previous results, we are going to take a look at tuning the hyperparameters of the FastText classifiers. The library itself already provides us with an easy way of implementing this optimization, with the only downside that it takes considerably longer to train, as it is expected. For each classifier, FastText performs over 54 trials, testing different combinatios of hyperparameters in order to find those ones yielding the best F1-score.

The results obtained are shown in Table 7.7, containing improvements between 3 - 5%, which is a good starting point. Next we trained the multiclass classifiers with the augmented dataset, as we previously did with the MNB classifiers, looking for improvements. Table 7.8 and 7.9 show us the results obtained with the newly trained classifiers.

This time multiclass classifiers improved drastically, reaching a 25% improvement in total.

		Binary	Binary	Multi	Multi
Model	Metric	English	Spanish	English	Spanish
FastText	Precision	75.80	75.80	61.24	61.29
	Recall	75.64	75.79	62.46	62.61
	F1-score	75.62	75.79	59.83	61.25

Table 7.7: Scoring metrics of each FastText classifier after applying hyperparameter tuning.

		Binary	Binary	Multi	Multi
Model	Metric	English	Spanish	English	Spanish
FastText	Precision	75.80	75.80	80.16	88.40
	Recall	75.64	75.79	81.47	88.32
	F1-score	75.62	75.79	81.56	88.33

Table 7.8: Scoring metrics of each Multinomial Naive Bayes classifier after training the multiclass with the augmented dataset and applying hyperparameter tuning.

Label	English	Spanish	Support
NS	84.91	85.31	346
II	89.08	92.09	175
O	82.35	93.07	104
SV	83.23	88.42	93
SD	91.18	90.65	179
MNSV	84.25	84.56	139

Table 7.9: F1-score of each label of the FastText multiclass classifiers, alongside the number of test samples for each label, after training with the augmented dataset. NS: non-sexist. II: ideological inequality. SD: stereotyping dominance. MNSV: misogyny non sexual violence. SS: stereotyping dominance. SV: sexual violence.

As we see in the Table 7.9, even though the non-sexism class has double the amount of samples compared with the rest, the F1-score of the other labels surpasses it. But these results have to be taken with a grain of salt, as the synonym replacement method leaves the samples too similar between them. Having another test dataset, we could compare more precisely these results, checking if they still hold up with newer samples.

7.3 Third increment: BERT Model

At this point we feel comfortable with the obtained results thus far, but we will now focus on the implementing the last model of classifiers using BERT to see if they can improve even further.

7.3.1 First iteration: Baseline BERT

For the first iteration, we established a baseline BERT classifier, but this time it's not as simple to set up as the previous two models. In this case, we have to get acquainted with the Pytorch library, as we need a machine learning framework to properly set up a classifier such as BERT.

We started only with the binary English classification, in order to check if the foundation we create now is strong enough to be used for other BERT pre-trained models as well as for tackling the multiclass classification task.

First off, we applied some further preprocessing to the dataset, in order to prepare it to feed it to the network. In a separate file that is executed before the main program, we read the dataset into a pandas' dataframe. We take the labels of the "LabelTask1" column, as previously seen in Section 6.1 and convert them to 0 and 1, as the classifier needs integer-based labels. This way, we can reconvert them later to a string in order to show the results.

For the English part of the task, we reindex the dataframe to only contain the id of the tweet, the text in English (this means the whole dataset, as we mentioned before in Section 6.1, we have translated all tweets to both English and Spanish to duplicate our samples) and finally the label of the tweet, being a 1 if it sexist and a 0 if it's non-sexist. We can see the shape of the new dataframe in the following Table 7.10.

id	tweet	LabelTask1
000001	...	sexist
000002	...	non-sexist
...	...	...
006976	...	sexist
006977	...	sexist

Table 7.10: Structure of the dataframe used to train and test the BERT classifiers.

Now, we separate the dataframe into an 90% - 10% split between training and test sets, and we also split the remaining training into a 80% training and a 20% validation.

In the main program, we prepare the data by using the "base-bert-uncased" tokenizer.

This tokenizer will be tasked to encode the input text and labels to pass them into the BERT model. We set up the BERT classifier using the pre-trained "bert-base-uncased" model by initializing a Pytorch module, and defining custom batch training and evaluation functions that make use of the GPU in order to accelerate the process. We input them into the training function we defined, along with an Adam optimizer [43] of a fixed learning rate of 2×10^{-5}, and a set batch size of 16.

Finally, we evaluate the model based on the aforementioned metrics, obtaining the following results, as shown in Table 7.11:

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base	Precision	61.40	-	-	-
	Recall	61.23	-	-	-
	F1-score	61.21	-	-	-

Table 7.11: Scoring metrics of the first BERT binary English classifier.

As we can see, this basic BERT model underperformed compared to the previous FastText and MNB classifiers, where they had no problem reaching near 70% F1-score with the English binary classification. This probably has to do with the way we preprocessed and divided the data, as this time we divided the dataset in training, validation and test, as well as the lack of hyperparameter fine-tuning. We decided that the next iteration needed to improve the implementation overall, so we didn't implement the remaining tasks this iteration.

7.3.2 Second iteration:

The second iteration focuses on improving the more lacking aspects of the previous iteration, by rebuilding the previous BERT classifier. The architecture that we implemented can be seen in Figure 7.7

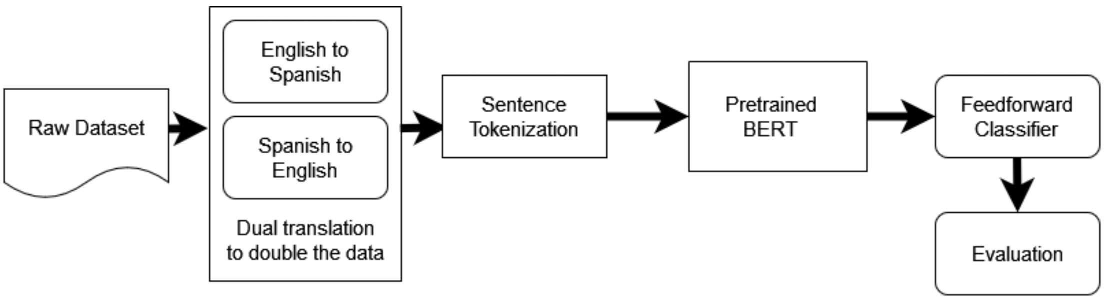


Figure 7.7: Architecture of our BERT classifiers.

Some of the more notable issues with the previous iteration were the lack of a better text preprocessing, so we are going to remove Twitter mentions and links. We don't need to remove punctuation marks or stopwords, as we had to do with the previous classifers. This is thanks to BERT being a model able to recognize grammar and the word order and structure of the sentences, as opposed to the NMB's bag of words model, where they are disregarded.

The pretrained BERT model that we use for the tokenization and classification is the same as before, the ”bert-base-uncased”. We changed the optimizer from Adam to AdamW [44], and also took a look at the hyperparameters that the huggingface team used during the pre-training of the model, and changed them as follows:

- Epochs: 5 -> 2
- Batch size: 16 -> 32
- Learning Rate: 2e-5 -> 5e-5

These small changes brought us far better results than the previous iteration, as seen in the Table 7.13, but there is still room for improvement. For now, we will shift our focus to the Spanish binary classification task.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base	Precision	79.35	-	-	-
	Recall	78.47	-	-	-
	F1-score	79.03	-	-	-

Table 7.12: Scoring metrics of the second BERT binary English classifier.

Even though the ”bert-base-uncased” pre-trained model we are using was pre-trained with an English corpus, we want to see how it performs by classifying the Spanish tweets. We took the English classifier as a base, and changed the preprocessing of the data so it extracts Spanish tweets from the dataset, reshaping it the same way as previously shown in Table 7.10. The rest of the architecture stayed the same, and the classifier performed as shown in 7.13. The results aren't as good as with the English tweets, but nevertheless it performed decently.

To improve the Spanish classification results, we took a look at other possible BERT models that could perform better in said language. One of the more prominent ones was SpanBERTa, a model based on RoBERTa.

RoBERTa [45] is another pre-trained transformers model that has been pre-trained in an even larger English corpus than the base BERT, making use of the $BERT_{large}$ architecture

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base	Precision	79.35	71.87	-	-
	Recall	78.87	71.78	-	-
	F1-score	79.03	71.73	-	-

Table 7.13: Scoring metrics of both Spanish and English second BERT binary classifiers.

5.8.1. But unlike the base model, RoBERTa dropped next sentence prediction from its pre-training phase, and relies only on being trained by using Masked Language Modeling. This makes it really effective at text classification or question answering, and that's a feature perfectly suited for our tasks.

SpanBERTa is a RoBERTa based model, that has been pre-trained fully in Spanish. It was created to improve the performance when approaching NLP tasks in Spanish, due to most BERT models being pre-trained with English texts. It performed better than the regular BERT did regarding the Spanish tweets, as we can see in Table 7.14.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
SpanBERTa	Precision	-	77.90	-	-
	Recall	-	77.79	-	-
	F1-score	-	77.76	-	-

Table 7.14: Scoring metrics of the SpanBERTa binary classifier.

After testing the classifier with both languages independently, we also tested it over both languages at the same time. This means that we trained the "bert-base-uncased" classifier using the whole dataset translated to English and Spanish, amounting to a total of 13954 tweets. Table 7.15 shows that even though it performed a bit worse regarding the purely English-trained classifier, it improved the results for the Spanish task, even taking into account that the pre-trained model was done so exclusively with an extensive English corpus. This indicates that the amount of data present in the dataset has great impact on the performance of the classifier. Overall, it's a good compromise between the two languages, nevertheless the optimal classification for now would be using the base BERT for English only, and SpanBERTa for Spanish only.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base (both)	Precision	76.46	76.46	-	-
	Recall	76.29	76.29	-	-
	F1-score	76.32	76.32	-	-

Table 7.15: Scoring metrics of the base BERT binary classifier, trained and evaluated with both English and Spanish tweets at the same time.

7.3.3 Third iteration: Multiclass classification and new BERT models

Once that we felt comfortable with the state of the binary classifiers, we started preparing the multiclass ones, in order to tackle the second task of the project.

We started by using the base BERT English binary classifier as a base, and changed the composition of the Feed Forward classifier layer in order to adapt the architecture for the multiclass classification. We changed the output from 2 to 6 classes, 1 for non-sexist tweets and the other 5 for their corresponding type of sexism. We also had to change the way we preprocess the labels from the dataset, as they are no longer binary, we decided to use a dictionary that translates the label text into an integer ranging from 0 to 5, so then we are able to translate them back to a string. After these adaptations, we trained and evaluated the multiclass classifiers, obtaining the metrics shown in Table 7.16

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base	Precision	79.35	71.87	67.36	58.08
	Recall	78.87	71.78	67.48	59.46
	F1-score	79.03	71.73	67.26	58.44
SpanBERTa	Precision	-	77.90	-	59.91
	Recall	-	77.79	-	61.75
	F1-score	-	77.76	-	59.83

Table 7.16: Scoring metrics of BERT base trained and evaluated separately in Spanish and English, as well as SpanBERTa's metrics upon the Spanish tasks.

BERT base performed surprisingly well against the English multiclass classification, but it didn't perform as well in the Spanish task. SpanBERTa proved to fail in the same regard, as its results are pretty similar with the BERT base. Finally, we tried the BERT base trained and evaluated with both languages at the same time, and as seen in Table 7.17 it performed as good as the English only model overall. This reinforces the previous conclusion, that the size of the training dataset still holds the most weight over the models' performance.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base (both)	Precision	76.46	76.46	67.48	67.48
	Recall	76.29	76.29	67.41	67.41
	F1-score	76.32	76.32	67.58	67.58

Table 7.17: Scoring metrics of the BERT base trained and evaluated with both English and Spanish tweets at the same time.

7.3.4 Fourth iteration: Tweet-oriented BERT models

After finishing the basic multiclass models, we are going to implement some more BERT models that are perfectly suited for our tweet-based sentiment analysis tasks, both in English and Spanish, as well as testing the multiclass classifiers against the augmented dataset.

BERTweet [46] is another model based on $BERT_{base}$. Its main feature is that it was pre-trained using a huge corpus of English tweets, so it should fit perfectly our classification tasks. This model is far more suited for the current shape of our text data, and it makes use of special tokens $@USER$ and $HTTPURL$ to interpret mentions and links, respectively. It also needs to separate certain word contractions, such as ”can't” into ”can n't” or ”you'll” into ”you 'll”, so we changed the text preprocessing accordingly.

On the other hand we have RoBERTuito [47], a model that could be considered an equivalent to BERTweet, but trained with Spanish tweets. It's relevant to note that this model is based on the RoBERTA architecture, so it's more focused on text classification or sentiment analysis. Even thought it's been trained on a Spanish corpus, it can perform really well with English tasks [47]. For this model it doesn't matter if we preprocess the text to accommodate for grammatical or spelling errors, as there is no significant improvement over just changing mentions for ”@usuario” and hashtags for ”hashtag” [47].

After implementing the classifiers and testing them on both binary and multiclass tasks, we got the following results, as shown in Table 7.19. BERTweet performed extremely well with the binary task, and RoBERTuito managed to achieve the same results as SpanBERTa. Regarding the multiclass classification, all BERT classifiers up to this point achieved very similar results, so we will train them with the augmented dataset to check for improvements. In Table 7.19 we can observe that the F1-score did improve around 5% and 7% for all the different models, which is an overall good result.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERTweet	Precision	82.58	-	68.53	-
	Recall	82.52	-	69.20	-
	F1-score	82.51	-	68.73	-
RoBERTuito	Precision	-	76.65	-	65.04
	Recall	-	76.65	-	65.76
	F1-score	-	76.65	-	65.03

Table 7.18: Scoring metrics of both BERTweet and RoBERTuito in both calssification tasks.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base	Precision	79.35	71.87	72.45	64.32
	Recall	78.87	71.78	72.86	64.29
	F1-score	79.03	71.73	72.64	63.97
BERT base (both)	Precision	76.46	76.46	72.34	72.34
	Recall	76.29	76.29	73.98	73.98
	F1-score	76.32	76.32	73.45	73.45
SpanBERTa	Precision	-	77.90	-	64.46
	Recall	-	77.79	-	64.86
	F1-score	-	77.76	-	64.34
BERTweet	Precision	82.58	-	70.09	-
	Recall	82.52	-	70.01	-
	F1-score	82.51	-	70.04	-
RoBERTuito	Precision	-	75.34	-	72.15
	Recall	-	75.67	-	71.92
	F1-score	-	75.49	-	72.04

Table 7.19: Scoring metrics all multiclass BERT classifiers using the augmented dataset.

7.3.5 Fifth iteration: Hyper-parameter tunning

After having implemented all these classifiers, we still have one important improvement left to make: hyperparameter tuning. It may have a small impact over the performance metrics of the classifiers, but at this point that's the only approach left to try. As the BERT model architecture is more complex than the previous classifiers, we have a lot of hyperparameters to fine tune:

- Optimizer:
  - Learning rate
  - Betas
  - Epsilon
- Batch size
- Epochs

Training and evaluating every BERT classifier with different combinations of each of these parameters is a heavily time-consuming task that would be near impossible to do by hand. Thus, we decided to automate the process, and for that we tried two approaches: We first decided to implement a script that executes the training of each classifier, changing each time the hyperparameters from a list and checking if the metrics improve, but then we found a faster, more optimized way of doing this.

The Ray-tune Python library allows us to execute in various GPUs each training of the model, each time with different hyperparameters chosen randomly from a defined list. For each model, we chose a range of hyperparameters around their respective pre-training hyperparameters, which are shown in their huggingface documentation page [48].

After training and evaluating every BERT model that we've implemented, the final results are shown in the following Table 7.20. Each classifier was trained 10 times with a different selection of hyperparameters, to find the ones that yielded the best score metrics. Also, the multiclass classifiers have been trained with the augmented dataset.

Almost every classifier benefited a bit from the tuning, as their performances increased around 1-2%. Theoretically, we should train them many more times as to cover for all the possible combinations of parameters that we inputted for each model, but that would take an enormous amount of time and resources that we don't have available. If we were to get access to Google's TPUs or other high-end computational devices, we could test every combination possible and reach the best results possible. Nevertheless, as a proof of concept this method of hyperparameter tuning proved to be realiable and efficient, as well as showing potential for further improvement of the classifiers.

Model	Metric	Binary English	Binary Spanish	Multi English	Multi Spanish
BERT base	Precision	79.96	74.03	72.64	65.46
	Recall	79.94	73.78	72.78	65.78
	F1-score	79.94	73.70	72.59	65.77
BERT base (both)	Precision	77.46	77.46	74.02	74.02
	Recall	77.29	77.29	74.78	74.78
	F1-score	77.32	77.32	74.32	74.32
SpanBERTa	Precision	-	77.90	-	64.46
	Recall	-	77.79	-	64.86
	F1-score	-	77.76	-	64.34
BERTweet	Precision	83.48	-	72.23	-
	Recall	84.12	-	72.12	-
	F1-score	83.72	-	72.17	-
RoBERTuito	Precision	-	76.65	-	74.03
	Recall	-	76.65	-	73.94
	F1-score	-	76.65	-	73.87

Table 7.20: Scoring metrics all BERT models after tuning their respective hyperparametrs.

Chapter 8

Conclusions

IMPLEMENTING three different SA models is a complex task in and of itself, since each one requires to be design, studied and approached separately. Despite this, we have managed to create quite efficient classifiers with the material we have used, suitable for performing the tasks presented by the EXIST workshop.

Some of the topics studied in this project were already partially covered in the degree, thanks to the studied computing mention, so they served as a base to start from. All the research that went into making this project is valuable knowledge that we gained over the whole field of NLP and specifically, SA. We studied and implemented a plethora of techniques and methods from text preprocessing, feature extraction, text classification and supervised learning, etc. This knowledge will serve us in the future, as we further develop our career in field of computational linguistics.

8.1 Results overview

We will make an overview of all the previously obtained metrics from each classifier, as shown in Table 8.1, in order to evaluate and compare their performance.

The MNB classifier is not a bad choice for a more lightweight model that performs decently, but it is really outclassed by the other two models. It served its purpose as an introduction to the bag-of-words model, and thanks to the Scikit-learn Python library it was fairly quick to set up.

On the other hand, FastText is really easy to set up and to fine tune, as the python library already provides functions for each task. The only workload comes from preprocessing and formatting the data properly. Performance-wise, it does a pretty good job, even rivaling BERT in the multiclass classification. Overall it's a really versatile model that can be quickly prototyped and executed to produce good results, compared to the state-of-the-art BERT models.

Finally we have the various BERT pre-trained models that we tested, including BERTweet

		Binary	Binary	Multi	Multi
Model	Metric	English	Spanish	English	Spanish
MNB	Precision	73.36	71.94	70.84	77.26
	Recall	69.63	69.48	45.37	56.66
	F1-score	68.45	68.68	37.77	52.42
FastText	Precision	75.80	75.80	80.16	88.40
	Recall	75.64	75.79	81.47	88.32
	F1-score	75.62	75.79	81.56	88.33
BERT base	Precision	79.96	74.03	72.64	65.46
	Recall	79.94	73.78	72.78	65.78
	F1-score	79.94	73.70	72.59	65.77
BERT base (both)	Precision	77.46	77.46	74.02	74.02
	Recall	77.29	77.29	74.78	74.78
	F1-score	77.32	77.32	74.32	74.32
SpanBERTa	Precision	-	77.90	-	64.46
	Recall	-	77.79	-	64.86
	F1-score	-	77.76	-	64.34
BERTweet	Precision	83.48	-	72.23	-
	Recall	84.12	-	72.12	-
	F1-score	83.72	-	72.17	-
RoBERTuito	Precision	-	76.65	-	74.03
	Recall	-	76.65	-	73.94
	F1-score	-	76.65	-	73.87

Table 8.1: Final scoring metrics of all classifiers.

and RoBERTuito that were pre-trained with twitter data. BERT base multilingual (as well as the extended dataset version) was trained and evaluated with both English and Spanish tweets at the same time, while the other models have been trained and evaluated for each language separately, hence the different F1-score between languages. This multilingual model performed really well, improving upon the Spanish classification and being only 2% shy of the only English classifier.

The hyperparameter optimization techniques we implemented could be improved by using higher end computers or other computational devices, as mentioned in Section 7.3.5 to really evaluate each possible configuration of hyperparameters.

The best classifiers for participating in the EXIST workshop would be BERTweet for the English binary classification, SpanBERTa or even the base multilingual BERT for the Spanish binary classification, and finally for the multiclass classification we would use the multilingual base BERT. We wouldn't use FastText for the second task, even though it achieved a higher F1-score, as we mentioned in Section 7.2.2 this could be the result of overfitting the model. In that case, we would be safer using the multilingual BERT model instead, as it could prove to achieve better results testing it against a different dataset.

8.2 Future work

After adecquately preprocessing the data and fine-tunning the models, as shown in Sections 7.3.2 and 7.3.3, the best improvement that can be done is increasing the entries in the dataset. Hyperparameter tunning and model optimizations can surely improve the results of the models, but they reach a point where it's only a mater of decimals.

Expanding the dataset by acquiring more sexist and non-sexist tweets and properly labeling them is a time consuming process, and it has to be done by a human, so that's out of the question. But surely, employing advanced over-sampling techniques such as text generation could prove to be the best way to improve these classifiers.

Apart from augmenting the dataset, other BERT models could be used to tackle the EXIST tasks, as there are more powerful multilingual models that can benefit from being trained with the whole dataset. This problem could also be expanded to other languages as well, as there are specific BERT models for the majority of them.

Appendices

List of Acronyms

AI	Artificial Intelligence. 2
AUC	Area Under the Curve. v, 43–46
CBOW	Continuous Bag-Of-Words. iv, 21, 23–25
CSV	Comma-Separated Values. 35
EXIST	sEXism Identification in Social neTworks. 2, 3, 62
IberLEF	Iberian Languages Evaluation Forum. 2
MNB	Multinomial Naive Bayes. v, vi, 6, 11, 12, 17, 18, 42–44, 46–49, 52, 60
NLP	Natural Language Proccessing. 2, 4, 7, 11, 16, 21, 31, 60
NNLM	Neural Network Language Model. 22, 23
RNN	Recurrent Neural Network. ii, 21, 22, 27, 28
RNNLM	Recurrent Neural Network Language Model. 22, 23
SA	Sentiment Analysis. 2–4, 6, 60
TFIDF	Term Frequency - Inverse Document Frequency. 6, 18
TM	Text Mining. 2
TSV	Tab-Separated Values. 35

Bibliography

[1] "EStimated number of worldwide social network users up to 2027, Statista," Sep. 2022. [Online]. Avaliable at: https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users

[2] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[4] S. S. y. M. Ministerio de Inclusión, "OBERAXE - Observatorio Español del Racismo y la Xenofobia. Inicio," Sep. 2022. [Online]. Avaliable at: https://www.inclusion.gob.es/oberaxe/es/index.htm

[5] B. Liu, "Sentiment analysis and opinion mining," *Synthesis lectures on human language technologies*, vol. 5, no. 1, pp. 1–167, 2012.

[6] "Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2021)," Sep. 2021. [Online]. Avaliable at: http://ceur-ws.org/Vol-2943

[7] "EXIST: sEXism Identification in Social neTworks," Oct. 2021. [Online]. Avaliable at: http://nlp.uned.es/exist2021

[8] G. Xu, Y. Meng, X. Qiu, Z. Yu, and X. Wu, "Sentiment analysis of comment texts based on bilstm," *Ieee Access*, vol. 7, pp. 51 522–51 532, 2019.

[9] K. Machová, M. Mikula, X. Gao, and M. Mach, "Lexicon-based sentiment analysis using the particle swarm optimization," *Electronics*, vol. 9, no. 8, p. 1317, 2020.

[10] P. Chauhan, N. Sharma, and G. Sikka, "The emergence of social media data and sentiment analysis in election prediction," *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 2, pp. 2601–2627, 2021.

[11] A. I. Saad, “Opinion mining on us airline twitter data using machine learning techniques,” in *2020 16th International Computer Engineering Conference (ICENCO).* IEEE, 2020, pp. 59–63.

[12] P. Pandey, N. Soni *et al.*, “Sentiment analysis on customer feedback data: Amazon product reviews,” in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon).* IEEE, 2019, pp. 320–322.

[13] K. Ahmed, N. El Tazi, and A. H. Hossny, “Sentiment analysis over social networks: an overview,” in *2015 IEEE international conference on systems, man, and cybernetics.* IEEE, 2015, pp. 2174–2179.

[14] C. Tan, L. Lee, J. Tang, L. Jiang, M. Zhou, and P. Li, “User-level sentiment analysis incorporating social networks,” in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1397–1405.

[15] F. Pozzi, E. Fersini, E. Messina, and B. Liu, *Sentiment analysis in social networks.* Morgan Kaufmann, 2016.

[16] A. Kulkarni and S. Mhaske, “Tweet sentiment analysis and study and comparison of various approaches and classification algorithms used,” *IRJET*, 2020.

[17] Z. Jianqiang, G. Xiaolin, and Z. Xuejun, “Deep convolution neural networks for twitter sentiment analysis,” *IEEE access*, vol. 6, pp. 23 253–23 260, 2018.

[18] G. M. Raza, Z. S. Butt, S. Latif, and A. Wahid, “Sentiment analysis on covid tweets: An experimental analysis on the impact of count vectorizer and tf-idf on sentiment predictions using deep learning models,” in *2021 International Conference on Digital Futures and Transformative Technologies (ICoDT2).* IEEE, 2021, pp. 1–6.

[19] S. Wang, W. Zhou, and C. Jiang, “A survey of word embeddings based on deep learning,” *Computing*, vol. 102, no. 3, pp. 717–740, 2020.

[20] C. Colón-Ruiz and I. Segura-Bedmar, “Comparing deep learning architectures for sentiment analysis on drug reviews,” *Journal of Biomedical Informatics*, vol. 110, p. 103539, 2020.

[21] M. Kasri, M. Birjali, and A. Beni-Hssane, “Word2sent: A new learning sentiment-embedding model with low dimension for sentence level sentiment classification,” *Concurrency and Computation: Practice and Experience*, vol. 33, no. 9, p. e6149, 2021.

[22] M. Ahmad, S. Aftab, S. S. Muhammad, and S. Ahmad, “Machine learning techniques for sentiment analysis: A review,” *Int. J. Multidiscip. Sci. Eng*, vol. 8, no. 3, p. 27, 2017.

[23] M. Neethu and R. Rajasree, “Sentiment analysis in twitter using machine learning techniques,” in *2013 fourth international conference on computing, communications and networking technologies (ICCCNT).* IEEE, 2013, pp. 1–5.

[24] J. C. Luna, “Top programming languages for data scientists in 2022,” *DataCamp*, Jan. 2022. [Online]. Avaliable at: https://www.datacamp.com/blog/top-programming-languages-for-data-scientists-in-2022

[25] A. McCallum, K. Nigam *et al.*, “A comparison of event models for naive bayes text classification,” in *AAAI-98 workshop on learning for text categorization*, vol. 752, no. 1. Citeseer, 1998, pp. 41–48.

[26] T. Joachims, “Text categorization with support vector machines: Learning with many relevant features,” in *European conference on machine learning.* Springer, 1998, pp. 137–142.

[27] A. M. Kibriya, E. Frank, B. Pfahringer, and G. Holmes, “Multinomial naive bayes for text categorization revisited,” in *Australasian Joint Conference on Artificial Intelligence.* Springer, 2004, pp. 488–499.

[28] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[29] S.-i. Amari, “Backpropagation and stochastic gradient descent method,” *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[30] Y. Bengio, R. Ducharme, and P. Vincent, “A neural probabilistic language model,” *Advances in neural information processing systems*, vol. 13, 2000.

[31] A. Mnih and G. E. Hinton, “A scalable hierarchical distributed language model,” *Advances in neural information processing systems*, vol. 21, 2008.

[32] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, “Recurrent neural network based language model.” in *Interspeech*, vol. 2, no. 3. Makuhari, 2010, pp. 1045–1048.

[33] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[34] P. Krbec, “Language modeling for speech recognition of czech,” 2006.

[35] T. Mikolov, J. Kopecky, L. Burget, O. Glembek *et al.*, “Neural network based language models for highly inflective languages,” in *2009 IEEE international conference on acoustics, speech and signal processing.* IEEE, 2009, pp. 4725–4728.

[36] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[37] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[38] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1113–1120.

[39] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černockỳ, "Strategies for training large scale neural network language models," in *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*. IEEE, 2011, pp. 196–201.

[40] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[41] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[44] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[45] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[46] D. Q. Nguyen, T. Vu, and A. T. Nguyen, "Bertweet: A pre-trained language model for english tweets," *arXiv preprint arXiv:2005.10200*, 2020.

[47] J. M. Pérez, D. A. Furman, L. A. Alemany, and F. Luque, "Robertuito: a pre-trained language model for social media text in spanish," *arXiv preprint arXiv:2111.09453*, 2021.

[48] "Models - Hugging Face," Sep. 2021. [Online]. Avaliable at: https://huggingface.co/models?filter=bert