ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA

DEPARTMENT OF ECONOMICS

Second-cycle/Master's Degree

in

ECONOMICS

Sentiment Analysis and Emotion Detection of Italian Tweets on Gender-Based Violence: A Comparative Study of Approaches

DEFENDED BY

Federico Bertoia

0001030753

SUPERVISOR

Prof. Sergio Pastorello

Graduation session of December

Academic year 2022/2023

Abstract

This thesis compares the effectiveness of various approaches to sentiment analysis and emotion detection on a sample of Italian tweets about gender-based violence. Despite numerous comparative studies in this field, the literature about the Italian language lacks clear results on which approach is the most promising. The first part of the thesis analyzes the tweets in the reference datasets, the main techniques of data preprocessing and feature extraction, and the evaluation metrics used to assess model performance. The second part delves into the details of the different models employed, including VADER for the lexicon-based approach, Naive Bayes classifier and Support Vector Machines for the machine learning approach, and various models based on BERT for the Transformers-based approach. In the concluding section, the results are discussed, highlighting the superiority of models based on the Transformers architecture. Furthermore, the importance of the quantity and quality of data used for pre-training these models is emphasized as a crucial factor in achieving state-of-the-art results.

Keywords: Sentiment analysis, Emotion detection, Italian tweets, VADER, Naïve Bayes classifier, Support Vector Machines, BERT

Contents

Chapter 1

Introduction

In this first chapter we will illustrate the importance of sentiment analysis and emotion detection. Then, we will explain which is the research question and we will give a brief background of the field.

1.1 Motivation of Sentiment Analysis and Emotion Detection

In recent years, the volume of data containing sentiments and emotions has grown exponentially, primarily due to the diverse ways of communicating and expressing opinions on social networks and on the internet in general. Examples of this trend can be observed on platforms like Twitter, Instagram, or Reddit, which enable individuals to freely express their opinions on a wide range of topics. Furthermore, there are numerous web platforms where users can leave reviews, including massive online stores like Amazon, movies and TV series catalogs like IMDB, restaurant reviews on Google, and so on. Opinions serve as valuable tools for companies, allowing them to gain insights into customer perceptions of their products or services, and this feedback is instrumental in enhancing their business operations. Similarly, government entities, institutions, and public figures seek to gain insights into the public's sentiments and understand how they are perceived. From a customer's point of view, opinions and sentiments about products provide a reference point for decision-making before purchasing specific items.

In this thesis, our focus is directed towards the analysis of Italian tweets related to gender-based violence. We aim to extract sentiment, which involves determining whether the message contained in the tweet is negative, positive, or neutral, and to analyze the underlying emotion.
Gender-based violence is a significant social issue that should not be underestimated. Through this type of analysis, policymakers, organizations, and activists can monitor public opinion regarding this topic to evaluate the level of awareness, support, and outrage. These insights can, in turn, inform the development of effective policies and interventions. Furthermore, social media platforms like Twitter can serve as breeding grounds for disinformation and the proliferation of negative attitudes. Sentiment analysis can be instrumental in identifying and countering false narratives and prejudices, ultimately promoting a

more empathetic and informed society.

1.2 Research Question

In this thesis we seek to explore the effectiveness of various sentiment analysis and emotion detection approaches, including lexicon-based methods, traditional machine learning algorithms and state-of-the-art Transformer-based models.

In this literature similar analyses have been conducted, demonstrating how Transformer-based models, such as BERT, yield the best results. However, it is important to note that these results mainly apply to the English language, which provides many more resources for training large language models. Studies related to Italian literature, such as Catelli et al. [1], Magnini et al. [2], and Bacco et al. [3], have shown that transformer-based models do not always offer better results compared to more traditional techniques.
In particular, Magnini et al. [2] demonstrate that for some NLP tasks, such as sentiment analysis, the BERT model does not achieve the same results as traditional machine learning models. Bacco et al. [3] analyze the results obtained using support vector machines and BERT in the context of Italian reviews in the healthcare domain. In their study, they have shown that SVM performs better in terms of F1-Score.
Finally, Catelli et al. [1] compare the use of Italian lexicons and BERT models for sentiment analysis of tweets. Based on the results obtained, they suggest that lexicon-based models are preferable when dealing with small datasets and limited computational resources.

In light of these considerations, the objective of this thesis is to evaluate the performance of different methods on a sample of italian tweets related to the topic of gender-based violence, in order to compare their performance and understand their strengths and weaknesses. Furthermore, the existing literature primarily focuses on sentiment analysis, which typically involves classifying text as positive, neutral, or negative. Some studies, such as Catelli et al. [2] and Bacco et al. [3], only consider the positive and negative classes, limiting the analysis to binary classification. In this thesis, in addition to analyzing more than two classes for sentiment, we also consider emotion detection for seven different emotions, expanding the problem to multi-class classification.

1.3 Background

Sentiment analysis is a field of study whose main objective is to identify and classify person's opinion. An opinion refers to an individual's subjective viewpoint, assessment, or attitude toward a particular subject, often expressed in the form of text. Sentiment analysis, also known as opinion mining, involves the use of natural language processing and computational techniques to determine the sentiment or emotional

tone conveyed in a piece of text.
Although throughout the thesis we will refer to sentiment classification when discussing it, in reality sentiment analysis is a broad field that encompasses various tasks. For instance:

- Emotion Detection: involves identifying specific emotions expressed in a given text.
- Subjectivity Classification: involves determining the existence of subjectivity in text, often considered the first step in sentiment analysis.
- Opinion Spam Detection: involves identifying false or fake reviews that promote or discredit a product.
- Implicit Language Detection: involves detecting humor, sarcasm and irony in text.
- Aspect Extraction: involves retrieving the target entity and aspects of the target entity in a document.

Moreover, sentiment analysis has been investigated on several levels: document, sentence, phrase and aspect.

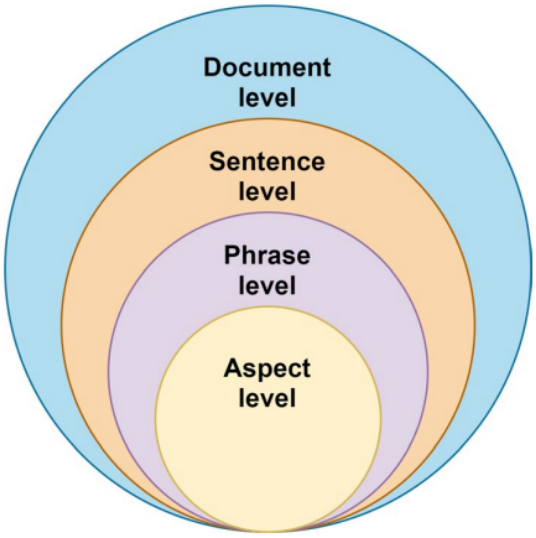


Figure 1.3.1: Sentiment Analysis Levels. (Source: [4])

Document-level sentiment analysis involves evaluating the sentiment of an entire document, assigning a single polarity to the entire content. This approach is less commonly used but can be applied to classify entire chapters or pages of a book as positive, negative, or neutral. In sentence-level sentiment analysis, each sentence within a document is individually analyzed to determine its corresponding polarity. This approach proves particularly valuable when a document exhibits a diverse range of sentiments. The resulting individual sentence polarities can be aggregated to derive the overall sentiment of the document

or used independently. However, sentence level sentiment analysis suffers the same problem as the document level, i.e they may contain multiple entities with different aspects. Aspect-based sentiment analysis (ABSA) aims at addressing these limitations by focusing on specific aspects or features within a piece of text. This approach provides a more granular understanding of sentiment by identifying and analyzing sentiments associated with individual aspects or entities. Considering the following example, "The food is good but the service is bad", we can see clearly that the sentence is positive with respect to the aspect "food" but is negative with respect to the aspect "service".
Finally, there are many techniques that can be used to perform sentiment analysis. An overview is given in Figure 1.3.2

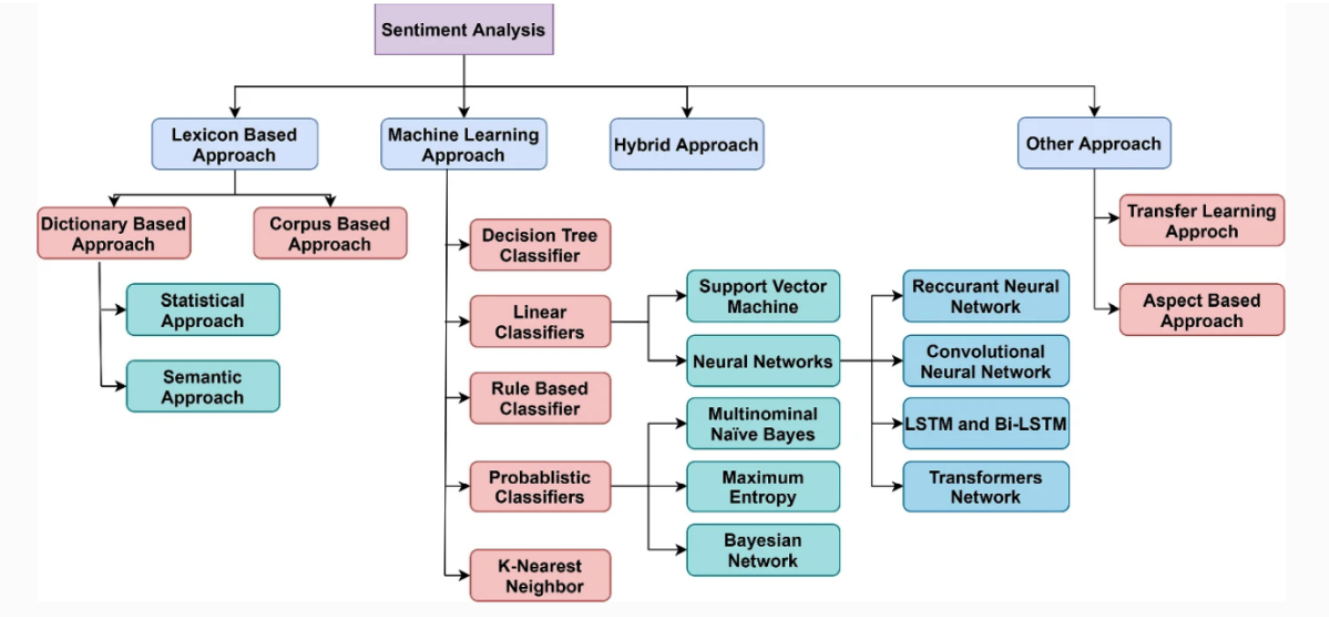


Figure 1.3.2: Overview of Sentiment Analysis Techniques. (Source: [4])

As previously explained, we will try and compare different methods for our specific tasks. For instance we consider:

- The Lexicon-based approach: use precompiled sentiment lexicons containing different words and their polarity to classify a given word into positive or negative sentiment class labels. It will be discussed in section 3.1.
- The Machine learning approach: use the machine learning algorithms to classify the words into their corresponding sentiment labels. The main benefit is their ability to learn sophisticated non linear patterns by learning from the training data. It will be discussed in section 3.2
- The Transformers approach: even though they are incorporated into the machine learning approach in Figure 1.3.2, they revolutionized sentiment analysis through contextual understanding, bidirectional processing, and attention mechanisms, enabling to capture intricate language dependencies. It will be discussed in chapter 4.

,

Chapter 2

Data and Methodology

In this chapter we start with an exploratory data analysis on the tweets underlying the study. Moreover, data preprocessing and feature extraction techniques are discussed in order to understand how the data is managed. Finally, a complete overview of the evaluation metrics is given.

2.1 Structure of the Datasets and Visualization

The data is composed of a collection of italian tweets provided by ISTAT (Istituto Nazionale di Statistica), each of them related to gender-based violence theme. The tweets are split into two different datasets: one for sentiment analysis, the other for emotion detection. Each dataset is again divided into three splits: train, validation and test. Each tweet has been reviewed and manually labelled by a team of experts, either by sentiment or by emotion.

Sentiment Category
Negative
Neutral
Positive

Table 2.1.1: Sentiment Categories

Emotion Category
Tristezza
Gioia
Amore
Rabbia
Paura
Sorpresa
Neutra

Table 2.1.2: Emotion Categories

Tables 2.1.1 and 2.1.2 list the categories for the sentiment and emotion analysis that will be conducted using the datasets. They show the range of emotions and sentiments that the dataset covers, which is important for understanding the diversity of responses and opinions expressed in the tweets.

Moreover, the sample size is quite different between the two datasets, with the sentiment one being relatively small compared to the emotion dataset.

Split	Features	Number of Rows
Train	[‘sentiment’, ‘text’]	715
Test	[‘sentiment’, ‘text’]	216
Validation	[‘sentiment’, ‘text’]	91

Table 2.1.3: Sentiment Dataset Information

Split	Features	Number of Rows
Train	[‘emotion’, ‘text’]	1781
Test	[‘emotion’, ‘text’]	543
Validation	[‘emotion’, ‘text’]	318

Table 2.1.4: Emotion Dataset Information

However, it must be considered that since the emotion detection task requires to classify the tweets in seven different categories (instead of three as for sentiment analysis), it is crucial to have more training data to achieve good performance.
To get a clear idea of the data structure, here are shown the first tweets of the training set for both datasets:

sentiment	text
NEU	La lotta contro il bodyshaming è una cosa, promuovere l'obesità è un'altra
POS	La marginalità è un luogo radicale di possibilità, uno spazio di #resistenza. Ne parla @Racheleborghi in questo podcast pubblicato da #TRANSfemmIN...
NEU	@ilgiomba @GiovannaSerra3 @voilaloves @ArthurMeurs @LuniVale @antrichelieu @diegodemme4 @fabyo255 Io spero solo, che in cuor suo, Giovanna abbia ca...
POS	Seppellire l'odio sotto una montagna di amore #ProudBoys
NEU	#iorestoacasama non dimentico. E SE IO LOTTO DA PARTIGIANA Raccolta delle biografie delle partigiane a cura di @NonUnaDiMenoMI https://t.co/KUlwqE...

Table 2.1.5: Tweet Sentiment Train Dataset

emotion	text
NEUTRA	Il ministro della Salute
GIOIA	"Ho parlato di bodyshaming e fatshaming in Parlamento, perché credo che solo nominarli faccia bene e perché molta pubblicistica ha relegato qu...
NEUTRA	Se gli editori dell'Aie sono in polemica per l'intervento sugli sconti
RABBIA	1 Maggio 2020 Milano, duplice violenza sessuale: preso maniaco seriale nigeriano. Clandestinamente in Italia dal 2018 aveva segnalazioni per atti ...
AMORE	amore mio per sempre

Table 2.1.6: Tweet Emotion Train Dataset

These tweets capture a wide range of language styles and tones, reflecting the diversity in the datasets. Some tweets exhibit a casual or colloquial style, characterized by the use of slang and informal expressions such as hashtags and emojis. Others are characterized by a more formal and professional tone, reminiscent of content found in journals or articles.

Another important feature to analyze is the distribution of the classes among the three splits, i.e the proportion of different sentiment or emotion categories within the dataset. Understanding and examining these class distributions across different data splits, such as training, validation, and testing, is a critical preliminary step that can greatly influence the effectiveness and reliability of subsequent analytical processes.

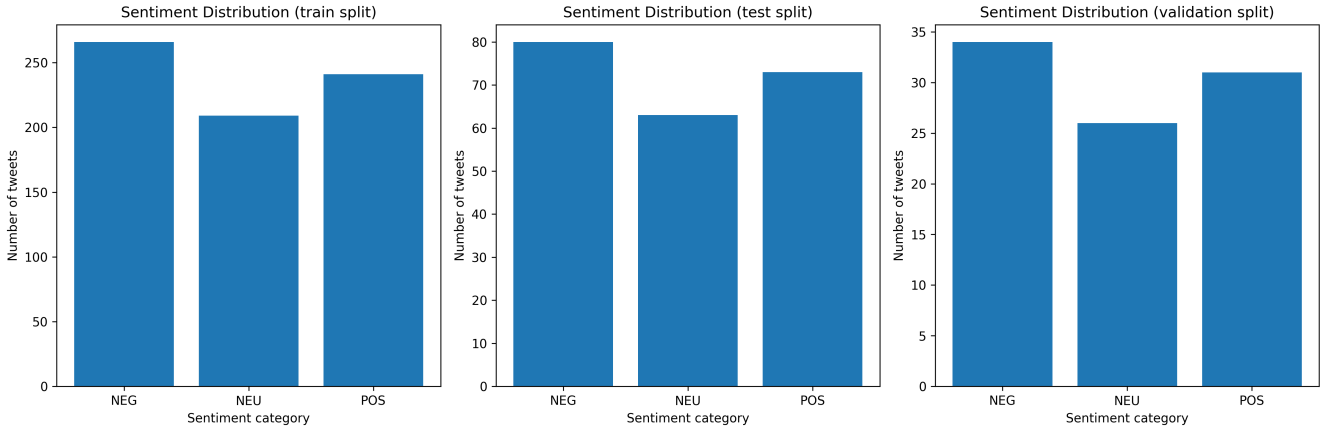


Figure 2.1.1: Sentiment Distribution

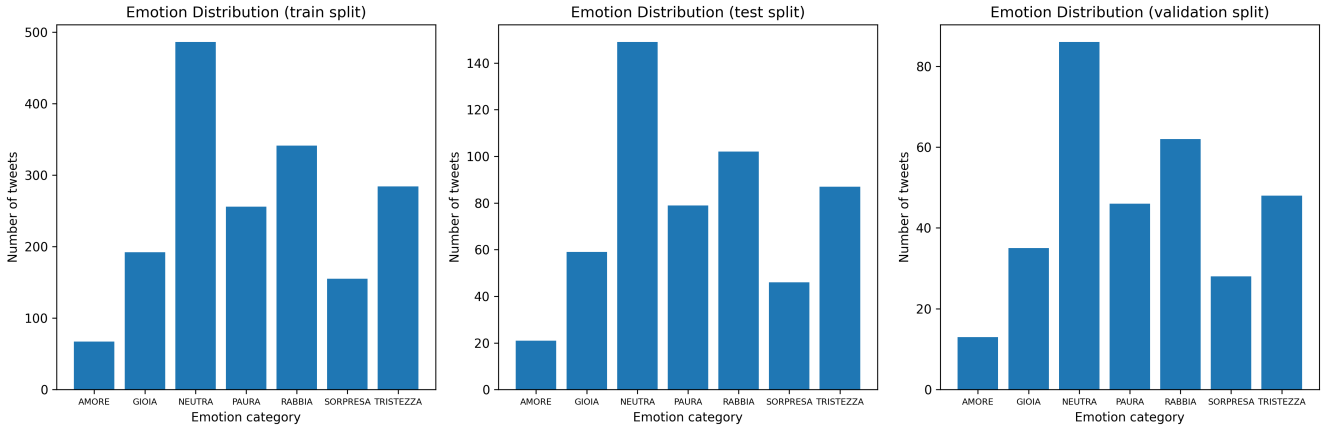


Figure 2.1.2: Emotion Distribution

As we can see from figure 2.1.1 and 2.1.2, each dataset split is created by using a stratified sampling method: it consists on dividing the tweets into distinct subgroups based on the different categories, then they are sampled proportionally to ensure that each category is adequately represented in each dataset split. As for the sentiment dataset, table 2.1.7, the distribution is almost uniform among classes, with the exception of few more negative tweets and less neutral ones.

However, looking at table 2.1.8, there is more variability in terms of class representation in the emotion dataset. For example, giving the high numerosity of 'NEUTRA' class (27.5%), it is expected the models to perfom better about this one with respect, for example, to 'AMORE' class (3.9%).

Given the presence of unbalanced classes, accuracy is not the only metrics considered, as it will be exhaustively explained further on.

Sentiment Category	Percentage (%)
NEG	37.1
POS	33.8
NEU	29.1

Table 2.1.7: Normalized and Rounded Sentiment Distribution (In Decreasing Order of Percentage)

Emotion Category	Percentage (%)
NEUTRA	27.5
RABBIA	19.0
TRISTEZZA	15.9
PAURA	14.4
GIOIA	10.8
SORPRESA	8.5
AMORE	3.9

Table 2.1.8: Normalized and Rounded Emotion Distribution (In Decreasing Order of Percentage)

Finally, to get a glimpse of the tweets content, two word clouds are showed, respectively for sentiment and emotion. Word clouds, visual representations of text data, are created to offer a quick and intuitive overview of the most frequently occurring words within a dataset. They serve as valuable tools to reveal the main themes and topics present in the datasets, making them particularly useful to gain insights into the predominant emotions and sentiments expressed in the domain of gender-based violence. Only nouns have been included in the word clouds in order to get a better visual representation.

(a) Sentiment Word Cloud

(b) Emotion Word Cloud

Figure 2.1.3: Sentiment and Emotion Keyword Visuals

2.2 Data Preprocessing

Preprocessing the data is the process of cleaning and preparing the text for classification [5]. It refers to the set of techniques and steps applied to raw text data to transform it into a suitable format for sentiment classification tasks. Text data is often unstructured, making it challenging to directly apply machine learning algorithms. Preprocessing helps extract relevant features and eliminate noise, improving the accuracy and effectiveness of sentiment analysis models.
By looking at tables 2.1.5 and 2.1.6, it is evident the presence of irrelevant elements that do not help in determining the sentiment or emotion of the sentence, such as user mentions, URLs and so on. Moreover, they can interfere and deteriorate the models performance, thus they should be handled in proper ways.
Data preprocessing has been applied using popular NLP libraries such as NLTK, re and simplemma. The main steps are the following:

1. Punctuation removal: punctuation refers to symbols like periods, commas, exclamation marks, question marks, and more. Removing punctuation is important because they often do not carry sentiment information and can add noise to the text data. By eliminating punctuation, the text becomes cleaner and more suitable for analysis.

2. URL removal: URLs or web links are usually not relevant to sentiment analysis and can introduce noise into the text. Removing URLs helps in focusing on the actual textual content and avoids any bias or confusion that might arise from the presence of URLs.

3. User Mention removal: user mentions are references to other users on social media platforms, typically starting with the '@' symbol. Sentiment analysis aims to understand the sentiment of the content itself, not the individuals mentioned.

4. Tokenization: is the process of breaking down a piece of text (the tweet) into individual words or 'tokens'. Tokenization is a necessary step to perform in order to further apply feature extraction techniques.

5. Lemmatization: is the process of reducing words to their base or root form. Lemmatization is important because it ensures that variations of words are treated as the same word.

After these steps, we can compare the processed tweets with the original ones to observe how they differ.

sentiment	text
NEU	lotta bodyshaming essere cosa promuovere obesità essere altro
POS	marginalità essere luogo radicale possibilità spaziare resistenza parlare podcast pubblicare TRANSfemmINonda voce essere corpo tempo coronavirus
NEU	sperare solo cuore Giovanna avere capire parola bullismo uso sintetizzare
POS	seppellire odio sotto montagna amore ProudBoys
NEU	non dimenticare lottare partigiano raccolto biografia partigiano cura

Table 2.2.1: Preprocessed Tweet Sentiment Train Dataset

emotion	text
NEUTRA	ministro Salute
GIOIA	avere parlare bodyshaming fatshaming parlamento credere solo nominarli bene perchè molto pubblicistica avere relegare tema qualcosa riguardare donna potere avere significare parlare voce maschile
NEUTRA	editore aia essere polemica intervento sconto
RABBIA	1 maggio 2020 Milano duplice violenza sessuale prendere maniaco seriale nigeriano Clandestinamente Italia 2018 avere segnalazione atto persecutore violenza sessuale resistenza pubblico ufficiale finalmente finire dietro sbarra sinistro volere ginocchio
AMORE	amore sempre

Table 2.2.2: Preprocessed Tweet Emotion Train Dataset

2.3 Feature Engineering

Feature extraction is the next step to finally convert raw text data into a format that can be easily processed by machine learning algorithms. There are various techniques available for feature extraction in NLP, each with its own strengths and weaknesses. As evidenced by [6], the choice of one method rather than another could lead to significant differences in model performance. In this study the focus is on two techniques: Bag of Words and TF-IDF.

Before delving into the specifics of these two methods, it's essential to begin with a brief introduction to common terms that will lay the groundwork for our discussion. These terms encompass fundamental concepts essential for a comprehensive understanding of the topics at hand.

- Corpus (c): The total number of words present in the dataset.
- Vocabulary (V) — Total number of unique words available in the corpus.

- Document (D) — There are multiple records in a dataset, so a single record or tweet is referred to as a document.
- Word (w) — Words that are used in a document are known as Word.

2.3.1 Bag of Words

This is perhaps the most simple vector space representational model for unstructured text [7]. A vector space model is simply a mathematical model to represent unstructured text as numeric vectors, such that each dimension of the vector is a specific feature attribute. The bag of words model represents each text document as a numeric vector where each dimension is a specific word from the corpus and the value represents its frequency in the document. The model's name is such because each document is represented literally as a 'bag' of its own words, disregarding word orders, sequences and grammar [8].

	bullismo	**violenza**	**amore**	**donna**	**stupro**
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	0	0	1	0	0
4	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮
710	0	0	2	0	0
711	0	0	1	0	0
712	0	0	0	0	0
713	0	1	0	1	0
714	0	0	0	0	0

Table 2.3.1: Bag of Words Representation

Table 2.3.1 represents a sketch of the entire matrix: each row represents a tweet, meanwhile each column is a word in the vocabulary. Due to the high sparsity of the matrix and for the sake of visualization, only some of the most characterizing words are shown with their frequencies. For example, the word 'amore' appears once in the 4th tweet, twice in the 710th tweet and so on. However, there are few key points and drawbacks that must be discussed when talking about Bag of Words:

- Importance: all word are treated equally, regardless of their meaning, significance or importance in the document. This means that, ideally, a stop word could have the same impact, if not more, of other words. This is one of the reasons why preprocessing is fundamental in sentiment analysis.
- Semantic Information: words are treated in isolation and the order is ignored. This leads to a loss of the context which, in many cases, could determine the sentiment and/or emotion of the sentence.

- Frequency-based importance: if all words are said to be treated equally, the decision rule is based on the frequency of their appearance. Words that appear more frequently in a document will have higher values in the corresponding dimensions of the BoW vector. This can somewhat reflect the idea that more frequent words may be more important in representing the content of a document.

2.3.2 TF-IDF

Essentially, TF-IDF works by determining the relative frequency of words in a specific document compared to the inverse proportion of that word over the entire document corpus [9]. Intuitively, this calculation determines how relevant a given word is in a particular document. Words that are common in a single or a small group of documents tend to have higher TF-IDF numbers than common words such as articles and prepositions. This measure can be decomposed by the product of two statistics, *term frequency* and *inverse document frequency*.

Term Frequency

Term frequency, $tf(w, D)$, is the relative frequency of term *w* within document *D*,

$$tf(w, D) = \frac{f_{w,D}}{\sum_{w' \in D} f_{w',D}} \tag{2.1}$$

where $f_{w,D}$ is the raw count of a word in a document, i.e the number of times that word w occurs in document D. The denominator is simply the number of words in document D.

Inverse Document Frequency

The inverse document frequency is a measure of how much information the word provides, i.e., if it is common or rare across all documents,

$$idf(w) = \log\left(\frac{1+N}{1+|\{D : w \in D\}|}\right) + 1 \tag{2.2}$$

where $N = |D|$ is the total number of documents, and the denominator is the the number of documents containing the word w.

Finally, we have that TF-IDF is simply given by:

$$tf\text{-}idf(w, D) = tf(w, D) \cdot idf(w) \tag{2.3}$$

A higher TF-IDF score for a term within a document suggests that the term is both frequent in that document and relatively rare in the others, making it more significant for that document's content. Conversely, a lower TF-IDF score indicates that the term is either common across many documents or infrequent in

the document, making it less relevant.

	bullismo	**violenza**	**amore**	**donna**	**stupro**
0	0.00	0.00	0.00	0.00	0.00
1	0.00	0.00	0.00	0.00	0.00
2	0.19	0.00	0.00	0.00	0.00
3	0.00	0.00	0.20	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00
⋮	⋮	⋮	⋮	⋮	⋮
710	0.00	0.00	0.23	0.00	0.00
711	0.00	0.00	0.18	0.00	0.00
712	0.00	0.00	0.00	0.00	0.00
713	0.00	0.16	0.00	0.17	0.00
714	0.00	0.00	0.00	0.00	0.00

Table 2.3.2: TF-IDF Representation

By making a comparison between table 2.3.1 and 2.3.2, some considerations should be highlighted. First, words with the same BoW value may not have the same in TF-IDF. This is due to the inverse document frequency weighting. Second, BoW just creates a set of vectors containing the count of word occurrences in the document (tweets), while the TF-IDF model contains information on the most important words and the less important ones as well.
The BoW and TF-IDF representations have been prepared and are now ready to serve as inputs for machine learning models. In the upcoming chapter, Support Vector Machine (SVM) and Naive Bayes algorithms will be introduced to work with these feature representations.

2.4 Evaluation Metrics

Both sentiment analysis and emotion detection are multi class classification tasks, i.e they involve more than two classes. Performance indicators are very useful when the aim is to evaluate and compare different classification models or machine learning techniques [10]. Furthermore, they are essential for discerning the classes where models excel and those where they underperform.

Confusion Matrix

The confusion matrix is a table used to evaluate the performance of a classification algorithm [10], where the rows represent the actual class and the columns represent the predicted class. The classes are listed in the same order in the rows as in the columns, therefore the correctly classified elements are located

on the main diagonal from top left to bottom right and they correspond to the number of times that an instance is predicted to its true/actual class.

		PREDICTED classification				
	Classes	a	b	c	d	Total
ACTUAL classification	a	6	0	1	2	9
	b	3	9	1	1	14
	c	1	0	10	2	13
	d	1	2	1	12	16
	Total	11	11	13	17	**52**

Figure 2.4.1: Example of a Multi-Class Classification Matrix. (Source: Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for Multi-Class Classification: an Overview. 2020)

Before going into the details of the metrics, it is important to highlight some building block concepts. They are adapted from the classical binary classification to the multi-class case [11]:

- **True Positive (TP)**: It refers to the number of instances that are correctly classified as belonging to a specific class.
- **False Positive (FP)**: It refers to the number of instances that are incorrectly classified as belonging to a specific class when they don't actually belong to that class.
- **False Negative (FN)**: It refers to the number of instances that are incorrectly classified as not belonging to a specific class when they actually belong to that class.

Each of these values can be calculated separately for each class and they are the starting point to evaluate the model performances.
Notice that, differently than the binary classification case, it is not considered the True Negative (TN) concept. This is due to the fact that it loses significance because its meaning it's incorporated by the True Positive in the multi-class scenario.

Accuracy

Accuracy measures the proportion of correctly classified tweets from the total number in the dataset. Thus, it is an overall measure of how much the model is correctly predicting on the entire set of data.

$$\text{Accuracy} = \frac{\text{Total number of corrected classified tweets}}{\text{Total number of tweets}} \tag{2.4}$$

Precision

In multi-class classification, precision is the fraction of tweets correctly classified as belonging to a specific class out of all tweets the model predicted to belong to that class.

$$\text{Precision}_{\text{class A}} = \frac{\text{TP}_{\text{class A}}}{\text{TP}_{\text{class A}} + \text{FP}_{\text{class A}}} \tag{2.5}$$

Precision can be interpreted as the conditional probability of the tweet truly belonging to a certain class, given the model predicted to be that class.

Recall

In multi-class classification, recall is the fraction of instances in a class that the model correctly classified out of all instances in that class.

$$\text{Recall}_{\text{class A}} = \frac{\text{TP}_{\text{class A}}}{\text{TP}_{\text{class A}} + \text{FN}_{\text{class A}}} \tag{2.6}$$

Recall can be interpreted as the model's ability to recover all the tweets belonging to a certain class.

F1-Score

Due to their nature, precision and recall are in a trade-off relationship. The F1-score can be interpreted as a harmonic mean of the precision and recall, where an F1-score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score is equal.

$$\text{F1-Score} = 2 \cdot \left(\frac{\text{precision} \cdot \text{recall}}{\text{precision + recall}} \right) \tag{2.7}$$

To give some intuition about the F1-Score behaviour, we review the effect of the harmonic mean on the final score. As an example, we consider Model A with Precision equal to Recall (80%), and Model B whose precision is 60% and recall is 100%. Arithmetically, the mean of the precision and recall is the same for both models, but using the harmonic mean, i.e. computing the F1-Score, Model A obtains a score of 80%, while Model B has only a score 75% [10].

Chapter 3

Lexicon-Based and Machine Learning Approaches

This chapter primarily focuses on two main approaches for sentiment classification: the machine learning approach and the lexicon-based approach.
The *Lexicon-based Approach* relies on a sentiment lexicon, a collection of known and precompiled sentiment terms.
The *Machine Learning Approach* applies the traditional ML algorithms and relies on linguistic features.

In this chapter we start with VADER, a lexicon and rule-based sentiment analysis tool. Furthermore, two machine learning algorithms are explained in details, since they represent the core of the machine learning approach: Naive Bayes and Support Vector Machine.

3.1 Lexicon-Based Approach

Lexicons are collections of words where each word is assigned with a predefined score which indicates the neutral, positive and negative nature of the text [4]. The score is assigned based on polarity, i.e whether the word is negative or positive. Moreover, intensity of polarity may be accounted for, thus the the score could be any value contained in the range [-1,+1], where +1 represent highly positive and -1 highly negative. In the Lexicon-Based Approach, when analyzing a given tweet or text, the scores of individual words are aggregated separately for positive, negative, and neutral sentiments. Ultimately, the overall polarity of the text is determined based on the highest value among these individual scores. To achieve this, the document is initially tokenized, then the polarity of each token is calculated and then aggregated at the end. Notably, it operates without the need for training data, making it similar to an unsupervised technique. Conversely, its primary drawback lies in its susceptibility to domain-specific variations. This stems from the fact that words can possess multiple meanings and connotations, causing a word with a positive sentiment in one domain to exhibit a negative sentiment in another. For example,

consider the word 'small' in the sentences 'The TV screen is too small' and 'This camera is extremely small'. In the first sentence, 'small' carries a negative sentiment since people generally prefer larger screens. Conversely, in the second sentence, it conveys a positive sentiment because a small camera is convenient to carry. To address this challenge, one can mitigate domain dependence by creating a sentiment lexicon tailored to a specific domain or by adapting an existing lexicon to suit the context.

Given the structure of this approach, emotion detection task can not be performed with the standard lexicons. This is due to the fact the output is a 'compound score' that summarizes the polarity of the sentence. For example, if we get a negative compound score, we can not say if that sentence expresses anger, sadness or fear. There exists specific lexicons for emotion detection, such as NRC Emotion Lexicon[1], however its categories are not the same as the ones considered in this study[2]. For this reason, I decided not to include the lexicon-based approach as a method to perform emotion detection, focusing only on the sentiment classification.

Summarizing, sentiment lexicons can be divided in three categories [12]:

1. **Semantic Orientation (Polarity-based) Lexicons**: words are categorized into binary classes (i.e., either positive or negative). Examples are: LIWC[3], GI[4] and Hu and Liu[5].

2. **Sentiment Intensity (Valebce-based) Lexicons**: other than the binary polarity, words are given also a value of intensity, which is the strength of the sentiment expressed. Examples are: ANEW[6], SentiWordNet[7] and SenticNet[8].

3. **Lexicons and Context-Awareness**: it is possible to improve sentiment analysis performance by understanding deeper lexical properties (e.g., parts-of-speech) for more context awareness. A lexicon may be further tuned according to a process of word-sense disambiguation, i.e the process of identifying which sense of a word is used in a sentence when the word has multiple meanings.

3.1.1 VADER

VADER (Valence Aware Dictionary and sEntiment Reasoner) is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media [12]. It takes into account both the polarity and the intensity, making it a good starting point for sentiment analysis tasks. The VADER lexicon has been constructed following these steps:

[1] NRC Emotion Lexicon (EmoLex)
[2] EmoLex categories are: anger, anticipation, disgust, fear, joy, sadness, surprise, trust.
[3] LIWC
[4] GI
[5] Hu and Liu
[6] ANEW
[7] SentiWordNet
[8] SenticNet

1. Examine lexical features of existing and well-established sentiment lexicons.
2. Addition of lexical features typical of social media text such as emojis, slangs and so on.
3. Wisdom-of-the-crowd approach[9] to get point estimations of the score for the lexical features.
4. Keeping only the lexical features with non-zero mean rating and with a standard deviation less than 2.5.

After these processes, the result is the final gold-standard VADER's sentiment lexicon. A small glimpse is given at table 3.1.1.

Word	**Compound Score**	**Standard Deviation**	**Raw Scores**
lousiest	-2.6	0.8	[-4, -2, -3, -3, -1, -2, -3, -3, -2, -3]
lousily	-1.2	0.9798	[-1, -1, -2, -2, -2, -2, 0, -1, 1, -2]
lousiness	-1.7	0.64031	[-2, -2, -1, -1, -2, -2, -3, -2, -1, -1]
lousing	-1.1	0.9434	[-3, 0, 0, 0, -1, -2, -1, -2, -1, -1]
lousy	-2.5	0.67082	[-2, -4, -2, -3, -2, -3, -2, -2, -3, -2]
lovable	3.0	0.63246	[3, 3, 3, 4, 3, 2, 3, 3, 2, 4]
love	3.2	0.4	[3, 3, 3, 3, 3, 3, 3, 4, 4, 3]
loved	2.9	0.7	[3, 3, 4, 2, 2, 4, 3, 2, 3, 3]
lovelies	2.2	0.74833	[3, 3, 3, 1, 2, 2, 3, 2, 1, 2]
lovely	2.8	0.6	[2, 3, 3, 3, 2, 3, 4, 3, 2, 3]
lover	2.8	0.87178	[3, 1, 2, 3, 4, 3, 2, 3, 4, 3]

Table 3.1.1: Glimpse of VADER lexicon

Notice that the compound score ranges from -4 (extremely negative) to +4 (extremely positive). Moreover, qualitative analysis techniques has been used to define some important baselines to better capture the intensity of the sentiment, taking into account not only the words themselves but also the relationships among them:

1. Punctuation can influence the intensity of the sentiment, for example using '!' would increase the strength.
2. Capitalization, since writing word in CAPS may emphasize the magnitude of the emotion without changing the semantic orientation.
3. Degree adverbs, for example using 'very slow' instead of 'slow'.

[9]The wisdom-of-the-crowd approach leverages the collective opinions or judgments of a diverse group of individuals to make more accurate decisions or predictions than those made by any single member of the group.

4. Using contrastive conjunction such as ‘but’ may indicate the presence of different polarities in the same sentence.

5. Negation: negations reverse the polarity orientation of the lexical particles they are referred to. The investigation of Trigram preceding sentiment-laden terms enables the identification of negations for that specific term.

In this analysis, a slightly modified version of VADER has been employed because the original one was designed primarily for the English language. Instead, a multilanguage version[10] that integrates the Google Translate API through the translatte Python library has been used.
To visualize how it works, in table 3.1.2 are shown some examples of sentiment classification.

Text	**Negative**	**Neutral**	**Positive**	**Compound**
lotta bodyshaming essere cosa promuovere obesità essere altro	0.232	0.536	0.232	0.0
marginalità essere luogo radicale possibilità spaziare resistenza parlare podcast pubblicare TRANSfemmINonda voce essere corpo tempo coronavirus	0.0	1.0	0.0	0.0
sperare solo cuore Giovanna avere capire parola bullismo uso sintetizzare	0.264	0.541	0.196	-0.25
seppellire odio sotto montagna amore ProudBoys	0.311	0.336	0.353	0.128
non dimenticare lottare partigiano raccolto biografia partigiano cura	0.306	0.476	0.218	-0.0772

Table 3.1.2: VADER Sentiment Analysis Results

The columns Negative, Neutral and Positive represent the proportion of text that falls in each category, according to the lexicon. They sum up to one and they give a first summary of the sentiment expressed in the tweet. However, it is important to highlight that they do not take into account the VADER rule-based enhancements, thus it is not possible to classify the sentences just by looking at them. The most important feature is the value reported in the Compound column: it is the normalized score of the sum of valences computed following the lexicon and the five heuristics, i.e taking into account both the polarity and the intensity. Moreover, it is said to be normalized since the values fall in [-1,+1]. To classify the sentences according to VADER, there are thresholds established by the literature[13]:

[10]VADER Sentiment Analysis Multilanguage

1. positive sentiment: compound score ≥ 0.05
2. neutral sentiment: -0.05 $<$ compound score $<$ 0.05
3. negative sentiment: compound score $\leq$ -0.05

3.2 Machine Learning Approach

Differently from lexicon-based approach, supervised machine learning (ML) approach requires training data to perform the task[11]. Then, it relies on syntactic and linguistic factors to extend sentiment analysis to a standard classification task. Given a set of training records, each associated to a specific label class, the ML algorithm learn these examples and predict the label class for new unseen records. When the ML model assigns just one label to an instance, it is said to be a hard classification problem. In the other hand, when a probability distribution among the entire set of labels is given, it is called soft classification problem.

These supervised machine learning algorithms offer diverse approaches to solve classification problems. Some popular choices include Decision Trees, Random Forest, Naive Bayes, Support Vector Machines and Neural Networks, each with its own strengths and weaknesses. Decision Trees are known for their interpretability and ease of use, while Random Forest combines multiple trees to improve predictive accuracy. Naive Bayes is based on the bayes' theorem and is particularly useful for text classification tasks. Support Vector Machines excel in handling high-dimensional data and complex decision boundaries. Neural Networks, on the other hand, are versatile and capable of learning intricate patterns from data but often require substantial computational resources and extensive training data.

The choice of the most suitable algorithm depends on the specific characteristics of the sentiment analysis task, such as the complexity of the data, the availability of training data, and the computational resources. Researchers often experiment with different algorithms to determine which one performs best for their particular problem, as discussed in [4] and [14].

3.2.1 Naïve Bayes Classifier

Naïve Bayes classifier, NB from now on, is a supervised learning algorithm that belongs to the class of probabilistic classifiers. It aim at maximizing the posterior probability of belonging to a certain class given a set of features. It relies on the famous bayes theorem, which is represented by the Bayes' rule:

$$P(x \mid y) = \frac{P(y \mid x) \cdot P(x)}{P(y)} \tag{3.1}$$

[11] The discussion is based entirely on supervised techniques.

Let C represents the class variable[12] and D a document (i.e tweet) to be classified, where each document can be represented by a feature vector $f = (f_1, f_2, \cdots, f_p)$ [15]. Then, NB would solve [16]:

$$\hat{c} = \arg\max_{c \in C} P(c \mid D) = \arg\max_{c \in C} \frac{P(D \mid c) \cdot P(c)}{P(D)} \tag{3.2}$$

Notice that it is possible to discard the denominator in the maximization problem since $P(D)$ does not depend on the class c. This leads to:

$$\hat{c} = \arg\max_{c \in C} P(c \mid D) = \arg\max_{c \in C} P(D \mid c) \cdot P(c) \tag{3.3}$$

By looking at the above equation, it is straightforward to understand that the chosen class is the one that have the highest product between the likelihood of the document, $P(D \mid c)$, and the prior of the class, $P(c)$. The likelihood represents the probability of observing the given document (D) under a particular class (c). A higher likelihood indicates that the document is more likely to belong to that class based on the model's fit to the data. The prior is the probability of a document belonging to a specific class without considering any specific evidence (i.e., before observing the document). It reflects the prior knowledge or beliefs about the class distribution. For example, if a class has a high prior probability, it will be more likely to be chosen as the final classification, even if the likelihood for that class is not exceptionally high. As said before, the document can be represented by a set of feature:

$$\hat{c} = \arg\max_{c \in C} P(f_1, f_2, \ldots, f_p \mid c) \cdot P(c) \tag{3.4}$$

Unfortunately, equation 3.4 is computationally challenging to evaluate directly. Without making simplifying assumptions, estimating the probability of every conceivable combination of features would necessitate a huge number of parameters and exceedingly large training datasets. Here is where the term 'naïve' comes to play: in this algorithm we rely on the naïve assumption of conditional independence between every pair of features given the value of the class variable. This means that the presence or absence of one feature does not affect the presence or absence of any other feature within the same class. While this assumption is often unrealistic in practice, it significantly reduces the computational complexity and makes the algorithm more manageable. Thus, the likelihood can be expressed as follows:

$$P(f_1, f_2, \ldots, f_p \mid c) = P(f_1 \mid c) \cdot P(f_2 \mid c) \cdot \ldots \cdot P(f_p \mid c) \tag{3.5}$$

Finally, putting everything together, Naïve Bayes classifier selects the class according to:

$$c_{\text{NB}} = \arg\max_{c \in C} \left(P(c) \prod_{j=1}^{p} P(f_j \mid c) \right) \tag{3.6}$$

[12]Tables 2.1.1 and 2.1.2 represent the elements of the class variable for each task.

The different Naïve Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(f_i \mid c)$[13]. Because of the project's structure, the preferred option is the Multinomial Naive Bayes classifier. This classifier is well-suited for data with a multinomial distribution, making it a popular choice for text classification tasks. The distribution is characterized by $\theta_c = (\theta_{c1}, \ldots, \theta_{cp})$ for each class c, where p is the number of features. θ_{cj} represents the probability $P(f_j \mid c)$ of feature j appearing in a sample belonging to class c.
The parameters θ_c are estimated by a smoothed version of maximum likelihood, i.e relative frequency counting:

$$\hat{\theta}_{cj} = \frac{N_{cj} + \alpha}{N_c + \alpha n} \tag{3.7}$$

In the numerator, N_{cj} represents the count of feature j occurring in samples that belong to class c. In the denominator, N_c represents the total count of all features in class c. α is a smoothing parameter used to avoid zero probabilities and handles cases where certain features may not appear in some classes. The purpose of this equation is to estimate the probability $\hat{\theta}_{cj}$, which represents the likelihood of observing feature j in a sample belonging to class c. The numerator smooths the count of feature j in class c to prevent zero probabilities, and the denominator normalizes the estimate by taking into account all features and their counts in the class, as well as the smoothing factor.

3.2.2 Support Vector Machines

The support vector machines, SVM from now on, is a generalization of a simple and intuitive classifier called the maximal margin classifier [17]. It aims at finding the hyperplane that best separates the data into distinct classes, in the sense that allows to maximizing the generalization ability of a model [18].
To better understand how the algorithm works, following [17], we'll begin with the simplest configuration and gradually progress towards the actual algorithm implemented in this thesis.
Suppose we have an $n \times p$ data matrix that consists on n training observations in a p-dimensional space. Thus, each document (i.e tweet) D_i can be represented as a set of feature vector f_i = ($f_{i1}, f_{i2}, \cdots, f_{ip}$), with $i = 1 \ldots n$. For the sake of visualization, we assume that $p = 2$, so that we can discuss support vector machines in a two-dimensional space. Moreover, we assume there are only two classes, such that $c_1, c_2, \cdots, c_n \in \{-1, 1\}$.

[13] Naive Bayes - scikit learn

Maximal Margin Classifier

A hyperplane is defined as a flat affine subspace of dimension p-1. A generic hyperplane is defined by:

$$\beta_0 + \beta_1 f_1 + \beta_2 f_2 + \ldots + \beta_p f_p = 0 \tag{3.8}$$

We can think of the hyperplane as a boundary that partitions the p-dimensional space into two distinct regions. Suppose it is possible to create a hyperplane that perfectly separates training observations into two classes, labeled as class 1 and class -1. Then, it holds this condition:

$$c_i(\beta_0 + \beta_1 f_{i1} + \beta_2 f_{i2} + \ldots + \beta_p f_{ip}) > 0 \quad \forall i = 1, \ldots, n \tag{3.9}$$

If a separating hyperplane exists, we can leverage it to create an intuitive classifier: the class assigned to a test observation depends on which side of the hyperplane it falls. We classify the test observation D^*, represented by features f^*, based on the sign of $g(f^*) = \beta_0 + \beta_1 f_1^* + \beta_2 f_2^* + \ldots + \beta_p f_p^*$. If $g(f^*)$ is positive, then we assign the test observation to class 1, and if $g(f^*)$ is negative, then we assign it to class -1.

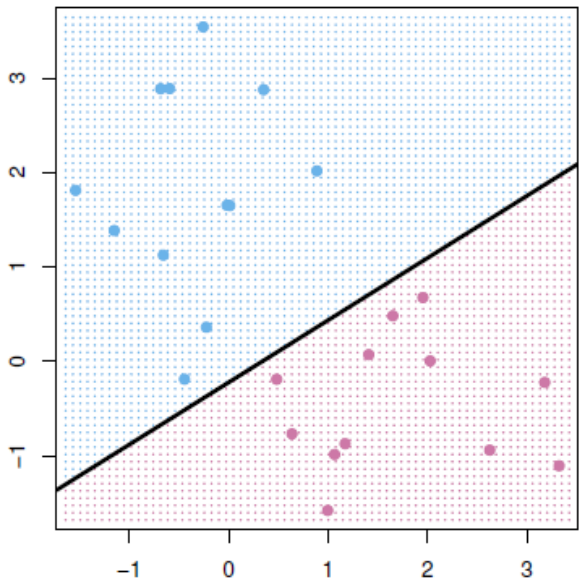


Figure 3.2.1: Example of a separating hyperplane. (Source: Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning. Springer Texts in Statistics. Springer, 2021)

.

However, if our data can be perfectly separated using a hyperplane, then there will in fact exist an infinite number of such hyperplanes. This is because a given separating hyperplane can usually be shifted a tiny bit up or down, or rotated, without coming into contact with any of the observations. To choose the optimal one, we rely on the maximal margin hyperplane: is the hyperplane that maximizes the distance

(margin) between itself and the nearest data points from each class, i.e it has the farthest minimum distance to the training observations (figure 3.2.2).

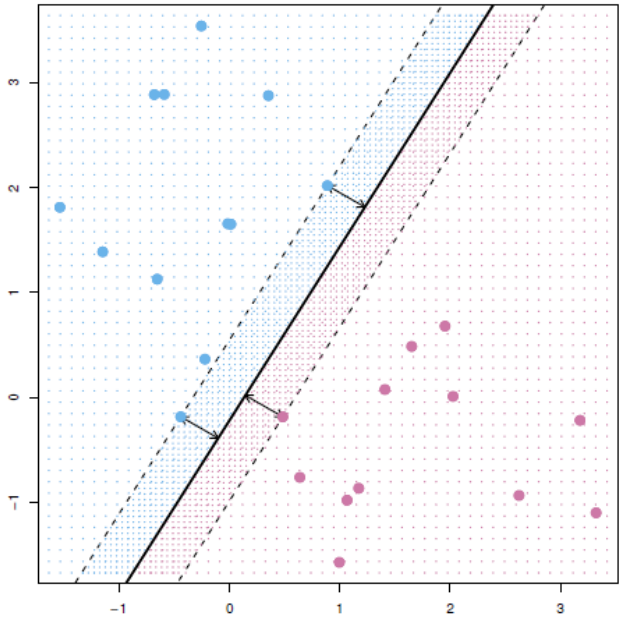


Figure 3.2.2: Maximal margin hyperplane. (Source: Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning. Springer Texts in Statistics. Springer, 2021)

.

Mathematically. it solves the following optimization problem:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,M}{\text{maximize}}\ M \tag{3.10}$$

$$\text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1 \tag{3.11}$$

$$c_i(\beta_0 + \beta_1 f_{i1} + \beta_2 f_{i2} + \ldots + \beta_p f_{ip}) \geq M \quad \forall i = 1, \ldots, n. \tag{3.12}$$

We want to maximize the margin, M, making sure that all the training data are on the correct side of the hyperplane (actually, we are imposing them to be also on the correct side of the margin).

Support Vector Classifier

In many cases no separating hyperplane exists, and so there is no maximal margin classifier. Also, there are cases where a separating hyperplane exists, however it may not be the best solution. In fact, the maximal margin classifier is really sensitive to individual observation and it may overfit the data.
In this scenario, one might contemplate utilizing a classifier based on a hyperplane that doesn't achieve perfect separation between the two classes, for two main reasons:

1. By allowing some margin for error, the classifier becomes more robust to outliers or extreme data points.

2. Prioritizing the classification of the majority of training data points can lead to a better overall performance, even if a few observations are misclassified.

These are the base concepts of the support vector classifier, which allows some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane[14].
The optimization problem is:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,\varepsilon_1,\ldots,\varepsilon_n,M}{\text{maximize}} M \tag{3.13}$$

$$\text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1 \tag{3.14}$$

$$c_i(\beta_0 + \beta_1 f_{i1} + \beta_2 f_{i2} + \ldots + \beta_p f_{ip}) \geq M(1 - \varepsilon_i) \quad \forall i = 1, \ldots, n. \tag{3.15}$$

$$\varepsilon_i \geq 0, \quad \sum_{i=1}^{n} \varepsilon_i \leq C \tag{3.16}$$

As before, we want to maximize the margin, M. However, individual observations are allowed to be misclassified by the presence of the slack variables $\varepsilon_1, \ldots, \varepsilon_n$. They provide valuable information about the positioning of the ith observation concerning the hyperplane and the margin: if $\varepsilon_i = 0$, it means the ith observation is correctly positioned on the appropriate side of the margin, if $\varepsilon_i > 0$ the ith observation is on the wrong side of the margin, and if $\varepsilon_i > 1$ the ith observation is on the wrong side of the hyperplane. The hyperparameter C serves as an upper bound on the sum of ε_i values, determining the number and severity of margin violations (and violations to the hyperplane) that the model can tolerate. The higher the value of C, the higher the possibility to misclassify the observations. The optimization problem demonstrates that only certain observations have a significant impact on the determination of the hyperplane: only those that either lie on the margin or violate the margin. These observations are referred to as 'support vectors'.

[14]This is why it is also called *soft margin classifier*.

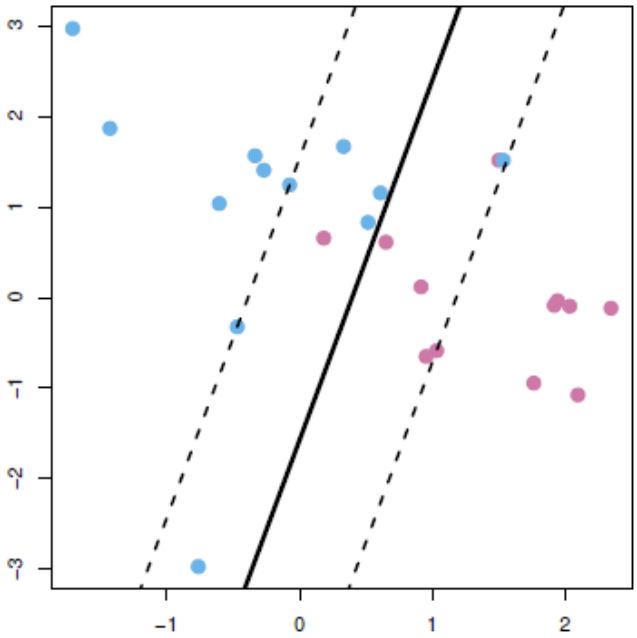


Figure 3.2.3: Example of Support Vector Classifier. (Source: Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning. Springer Texts in Statistics. Springer, 2021)

.

Support Vector Machines

In practice, most real world applications have non linear boundaries, and this is particularly true in the field of NLP. A solution to this problem is to enlarge the feature space, for example including some functions of the predictors. In the enlarged feature space, if chosen correctly, the decision boundary is indeed linear. However, it is not hard to see that there are many possible ways to enlarge the feature space, and that unless we are careful, we could end up with a huge number of features.
The support vector machine (SVM) is an extension of the support vector classifier that results from enlarging the feature space in a specific way, using kernels.

To better understand how kernels work, we turn back to the optimization problem of the support vector classifier. It can be shown that its solution involves only the inner product of the observations. Thus, the linear support vector classifier can be represented as:

$$g(f^*) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle f^*, f_i \rangle \tag{3.17}$$

$$\text{where} \quad \langle f_i, f_{i'} \rangle = \sum_{j=1}^{p} f_{ij} f_{i'j} \tag{3.18}$$

To evaluate the function $g(f^*)$, it is necessary to calculate the inner product between the new point f^* and each of the training points f_i. However, an important observation is that the values of α_i are non-zero only for the support vectors within the solution. In other words, if a training observation is not classified as a support vector, then its corresponding α_i is equal to zero.
The inner product can be generalized by some function $K(f_i, f_{i'})$, which we refer to as kernel. It is a function that quantifies the similarity of two observations. For example, if we take a linear kernel

$$K(f_i, f_{i'}) = \sum_{j=1}^{p} f_{ij}, f_{i'j} \tag{3.19}$$

we turn back to the classical support vector classification. Thus, in the general case, the decision function takes the form:

$$g(f^*) = \beta_0 + \sum_{i \in S} \alpha_i K(f^*, f_i) \tag{3.20}$$

where S represent the training observations that are support vectors.
A popular choice of non-linear kernel, the one used in this study, is the radial kernel:

$$K(f_i, f_{i'}) = \exp\left(-\gamma \sum_{j=1}^{p} (f_{ij} - f_{i'j})^2\right) \tag{3.21}$$

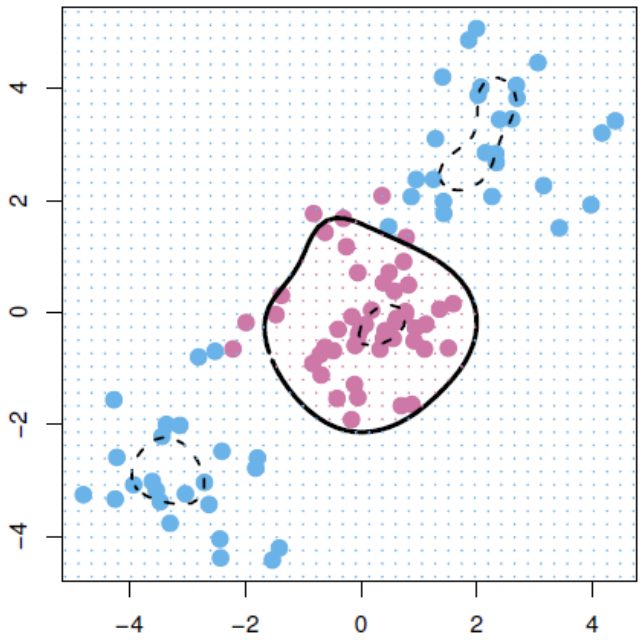


Figure 3.2.4: Example of Support Vector Machine with radial kernel. (Source: Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning. Springer Texts in Statistics. Springer, 2021)

.

In equation 3.21, γ represents a positive constant. Given a test observation f^*, if it is far from a training observation f_i in terms of Euclidean distance, then $\sum_{j=1}^{p}(f_j^* - f_{ij})^2$ will be large, and so $K(f^*, f_i)$ will be small (i.e, f_i does not contribute much in determining $g(f^*)$). This shows the local behaviour of the radial kernel, in the sense that the class label of a test observation is primarily influenced by training observations that are close.

Finally, to extend the support vector machine in our application, which is not a two-dimensional case, the *one-versus-all* approach is considered: it is a strategy used for multi-class classification. It involves training multiple binary classifiers, each focused on distinguishing one class from the rest. These individual classifiers are then combined to make predictions for multiple classes, effectively breaking down a multi-class problem into a set of binary classification tasks.

Chapter 4

Transformers Approach

The introduction of the Transformer architecture by Vaswani et al. [19] has reshaped the landscape of natural language processing (NLP), redefining the state-of-the-art in this field.

This chapter begins by providing an overview of Transformers, followed by an in-depth exploration of the BERT model's architecture [20], which will be the baseline model from now on. Moving forward, our attention will be directed towards the key stages of sentiment analysis and emotion detection tasks, including tokenization, embeddings processing, output representations and fine-tuning.

4.1 Introduction to Transformers

In recent years, the field of NLP has undergone a profound revolution, primarily driven by the introduction of transformer models. These innovative architectures have marked the beginning of a new era of performance improvements across a variety of NLP applications, including machine translation, text summarization and sentiment analysis. In contrast to traditional methods like Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Gated Recurrent Neural Networks, Transformers have demonstrated remarkable advantages.

Those traditional models that rely on sequential processing suffer from inherent limitations, particularly when dealing with training in parallel and handling longer text sequences. One of the key challenges they face is the vanishing gradient problem.

The vanishing gradient problem occurs when gradients, during training, become extremely small as they are propagated backward through the network during the learning process. This problem is particularly pronounced in RNNs, where the influence of earlier time steps on later ones diminishes rapidly. Consequently, when dealing with long sequences, RNNs struggle to capture and retain relevant information from distant past or future time steps, leading to a loss of context and an inability to effectively learn dependencies over extended distances.

Modern approaches like Transformers, which employ attention mechanisms and can process information

in parallel, have alleviated many of these issues. Transformers have become the foundation for state-of-the-art natural language processing models, as they can capture effectively long-range dependencies in text .

The transformer is composed by two blocks: an encoder and a decoder. The encoder is responsible for processing and encoding the input sequence. The decoder is responsible for generating the output sequence based on the encoded input.

Given that the chosen model to perform sentiment analysis and emotion detection is BERT, which relies solely on the encoder part of the Transformer, in the next sections a more in-depth explanation will be provided for the encoder, as opposed to the decoder.
For clearer understanding, figure 4.1.1 shows the Transformers architecture:

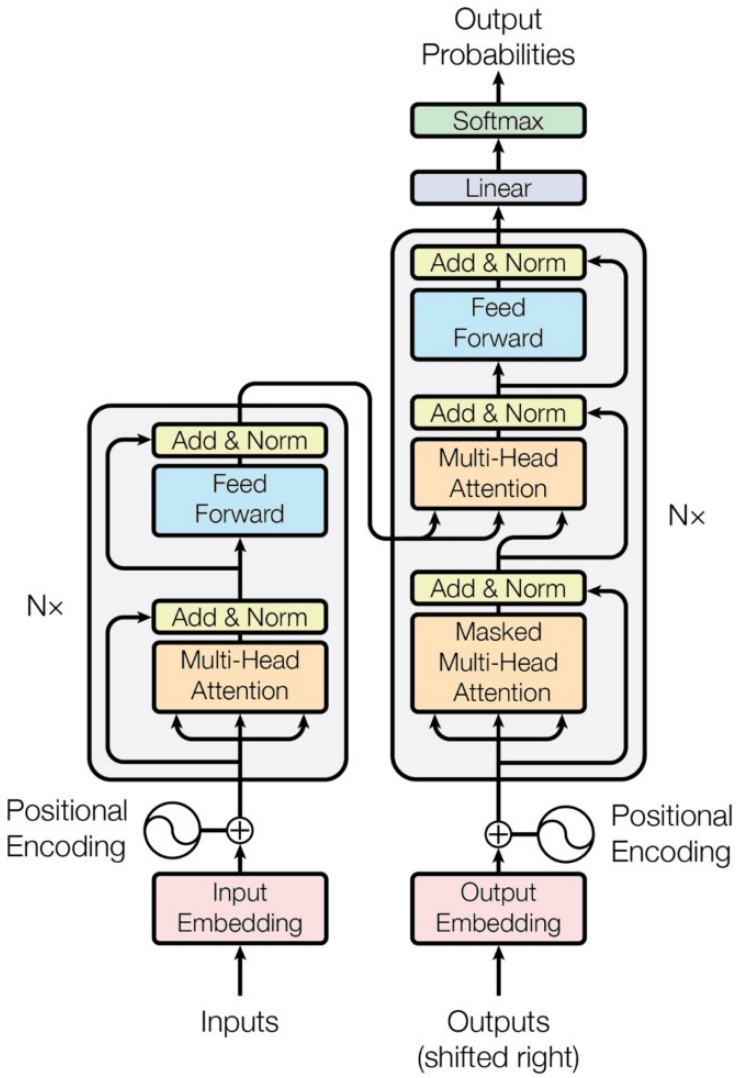


Figure 4.1.1: Transformer architecture. Encoder (left) and Decoder (right). (Source: Vaswani et al. [19]).

The initial step in processing input data for a transformer model involves utilizing a word embedding layer. This layer links the input words to a vector of continuous values, enabling effective representation. To help the model understand the order of words in a sequence without using recurrence or convolution, the authors introduced 'positional encodings' based on sine and cosine functions. These encodings provide information about the position of each word in the sequence.

4.1.1 Encoder Stack

The primary objective of the encoder is to transform input sequences into an abstract continuous representation that captures the learned information for the entire sequence. This process involves two essential sub-modules: multi-head attention and a fully connected neural network. Surrounding each of these sub-modules, there are also residual connections, which facilitate information flow, followed by layer normalization to stabilize the output.

In the encoder, there is a specialized attention mechanism known as self-attention, which enables the model to establish connections between individual words in the input by allowing each word to be associated with other words. This self-attention mechanism involves three separate fully connected layers to generate, starting from the input embeddings, the query, key, and value vectors. Attention is computed through a dot product of query and key, followed by scaling the scores and applying the softmax function to produce attention weights. These attention weights determine the importance of each word in the input sequence. Finally, the value vector is multiplied by the attention weights to produce the output vector.
In the multi-headed attention approach, query, key, and value vectors are split into h separate vectors, and each self-attention 'head' independently produces an output vector. These output vectors are then concatenated and passed through a linear layer to enhance the model's representation capacity.
The output of multi-headed attention is combined with the original positional input embedding using a residual connection, enabling smoother gradient flow through the network. This output undergoes layer normalization for stability. Following this, a pointwise feed-forward network is applied, consisting of two linear layers with ReLU activation, further enriching the representation of the attention outputs.
Overall, the encoder layer facilitates the transformation of input sequences into an abstract representation by utilizing self-attention, multi-headed attention, residual connections, layer normalization, and a feed-forward network, enhancing the model's ability to capture meaningful information from the input sequence.

4.1.2 Decoder Stack

The primary objective of the decoder is to generate a meaningful and coherent output sequence based on the information it has received from the encoder and its own previous predictions. The decoder has a similar structure as the encoder: it has two multi-head attention layers, a pointwise feed-forward layer,

residual connections and layer normalization after each sub-layer. These sub-layers behave similarly to the ones in the encoder, however the first multi-head attention layer has a different job.

In the decoder's first multi-head attention layer, a unique approach is employed due to the autoregressive nature of the decoder, which generates the sequence one word at a time. To ensure that the decoder does not condition on future tokens during this generation process, a mechanism called masking is used: the goal is to prevent the decoder from considering tokens that have not been generated yet. To achieve this, a look-ahead mask is applied. This mask is a matrix of the same size as the attention scores, filled with 0's and negative infinities. When added to the scaled attention scores, it effectively sets the upper-right triangle of the score matrix to negative infinity. The reason for this masking is that, when softmax is applied to the masked scores, the negative infinities become zero, resulting in zero attention scores for future tokens. Despite the masking, the first multi-head attention layer maintains its structure with multiple heads. The mask is applied to each head before the scores are concatenated and passed through a linear layer for further processing. The output of this layer is a masked output vector that guides how the model should attend to the decoder's input, ensuring that future tokens are excluded from consideration.

For the second multi-head attention layer, the keys and the values are the encoder's outputs, while the queries are the output of the previous masked-multi-head attention layer. From now on, the mechanism is the same as for the encoder part.
The final pointwise feedforward layer in the decoder is followed by a linear classifier that has as many output units as there are classes (for example, they could be the words in a vocabulary). The output of this classifier is passed through a softmax layer, generating probability scores for each class. The class with the highest probability is the chosen one. The decoder continues decoding by taking this prediction as input and repeats the process until it predicts an end token.

4.2 Introduction to BERT

In the field of NLP, pre-trained language models (PTMs) have revolutionized the way we approach NLP tasks by leveraging large-scale unsupervised learning [21]. Pre-trained models are deep neural networks that have been trained on massive amounts of unlabeled text data. Their fundamental idea is to learn the rich structure and patterns of language from vast corpora of text, allowing them to capture a wide range of linguistic knowledge. Unlike traditional NLP models that require substantial labeled data for each specific task, PTMs have the ability to transfer this pre-acquired language understanding to various downstream tasks. Over the past few years, the advent of the transformer architecture and the substantial advancements in computational capabilities, made pre-trained language models the dominant choice for the majority of NLP tasks.

BERT (Bidirectional Encoder Representations from Transformers) is a language model representation introduced by researchers at Google AI Language [20]. The main characteristic of BERT is the bidirectionality: it processes the entire input sequence in both directions (left-to-right and right-to-left) simultaneously. This means that BERT can consider context from both before and after a word in the input text, making it capable of capturing more complex relationships between words and their meanings.
It is worth noting that there are other transformer-based language models, such as the Generative Pre-trained Transformer (GPT), that follow a unidirectional approach. Unlike BERT, these models operate solely in a left-to-right manner and can only take into account preceding contex. The main difference is that BERT is based on the encoder part of the transformer, meanwhile autoregressive models like GPT on the decoder part[1].

To comprehend the bidirectional nature of BERT, we examine the tasks involved during its pre-training phase, which include Masked Language Modeling (MLM) and Next Sentence Prediction (NSP).

4.2.1 Masked Language Modelling

The first task is the one strictly related to the bidirectionality of BERT and consists of replacing 15% of the words in a given text sequence with a [MASK] token. The model's objective during pre-training is to predict the original words that have been masked, based on the context provided by the surrounding words in the sequence. This task encourages the model to learn a deep understanding of language by considering how words relate to one another and how they can be inferred from the surrounding context.

[1] BERT and GPT have distinct architectures because they are designed for different tasks: BERT focuses on understanding bidirectional context for tasks like text classification, while GPT excels at autoregressive text generation and completion.

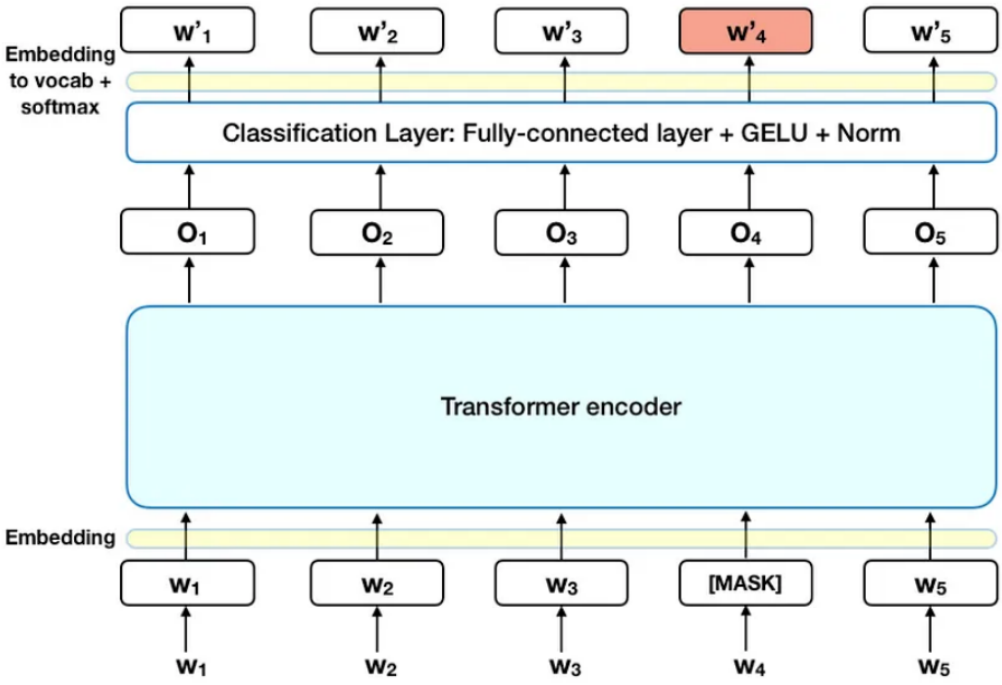


Figure 4.2.1: Masked Language Modelling. (Source: Horev [22])

As we can see from figure 4.2.1, on top of the encoder output is built a classification layer. The output vectors are multiplied by the embedding matrix in order to transform them into vocabulary dimension and then the probability of each word in the vocabulary is computed using the softmax function. In computing the loss function during training, only the prediction of the masked words are considered. This choice is made to increase the context awareness of the model, even though it leads to a lower convergence rate.

Although masked language modelling allows BERT to be bidirectional, a downside is that it creates a mismatch between pre-training and fine-tuning[2], since the [MASK] token does not appear during fine-tuning.
To address this problem, the masking procedure works as follows:

- 80% of the time the selected words are replaced with a [MASK] token.

$$\text{You are a wizard, Harry} \Longrightarrow \text{You are a [MASK], Harry}$$

- 10% of the time the selected words are replaced with random words.

$$\text{You are a wizard, Harry} \Longrightarrow \text{You are a table, Harry}$$

[2]Fine-tuning is a machine learning technique where a pre-trained model is further trained on a more specific dataset to adapt and optimize its performance for a particular task or problem.

- 10% of the time the selected words are left unchanged.

$$\text{You are a wizard, Harry} \Longrightarrow \text{You are a wizard, Harry}$$

The reason why the authors chose this approach is motivated by the following considerations [22]:

- Using [MASK] exclusively in all instances would not guarantee effective token representations for unmasked words, as the model's optimization primarily centers around predicting masked words while still utilizing non-masked tokens for contextual understanding.
- Applying [MASK] for 90% of the instances and introducing random words for the remaining 10% would suggest the model that the observed word is consistently incorrect.
- Using [MASK] 90% of the time while maintaining the same word 10% of the time, the model might easily adopt a non-contextual embedding approach to replicate the input word.

4.2.2 Next Sentence Prediction

The second pre-training task is next sentence prediction, in which the model is provided with pairs of sentences as input and is trained to predict whether the second sentence in each pair is indeed the subsequent sentence in the original document. When choosing the sentences A and B for each pre-training examples, 50% of the time B is the actual subsequent sentence in the original document (IsNext), while in the other 50% it is a random sentence from the corpus (NotNext). The fundamental idea here is that the randomly chosen sentence is expected to have no substantial link to the initial sentence.
Before feeding the model with the pair of sentences, some adjustments are performed to help the model understand which words belong to which sentence:

1. A [CLS] token is added at the beginning of the first sentence
2. After each sentence, a [SEP] token is added to indicate the end of the sentence
3. A sentence embedding indicating Sentence A or Sentence B is added to each token
4. A positional embedding is added to each token to indicate its position in the sequence

The result is the one showed in figure 4.2.2:

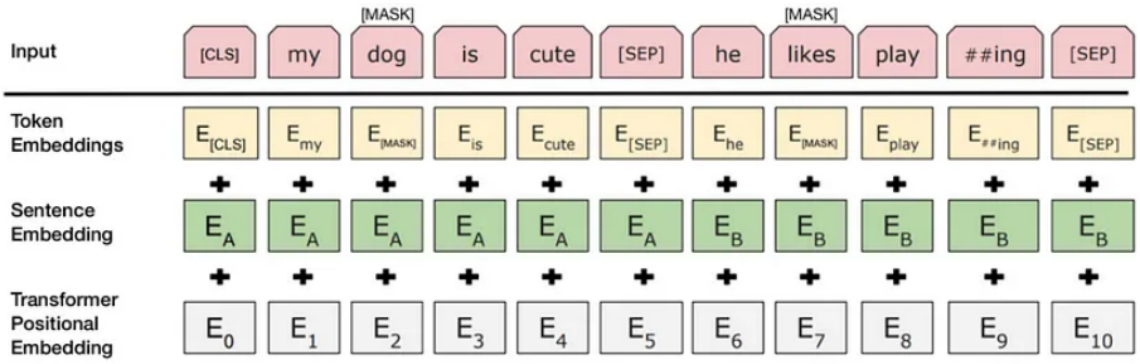


Figure 4.2.2: Next Sentence Prediction. (Source: Horev [22])

The entire input sequence, composed by the pair of sentences, is fed to the model. The output of the [CLS] token is the representation of the entire sequence after it has been processed by the Transformer layers. To transform the [CLS] token representation into a 2x1 vector, BERT uses a simple classification layer. This classification layer is essentially a fully connected neural network layer with learned weight matrices and bias terms. Finally, the probability of IsNext is computed by a softmax function.

It is important to highlight that, as shown in figure 4.2.2, Masked Language Modelling and Next Sentence Prediction are trained together, and the goal is to minimize the combined loss function of the two approaches.
Summarizing, Masked Language Modeling helps BERT understand words and their context, while Next Sentence Prediction helps it grasp the relationships between sentences. Together, these tasks make BERT versatile and powerful in understanding both word-level details and sentence-level structure, which is crucial for various NLP tasks.

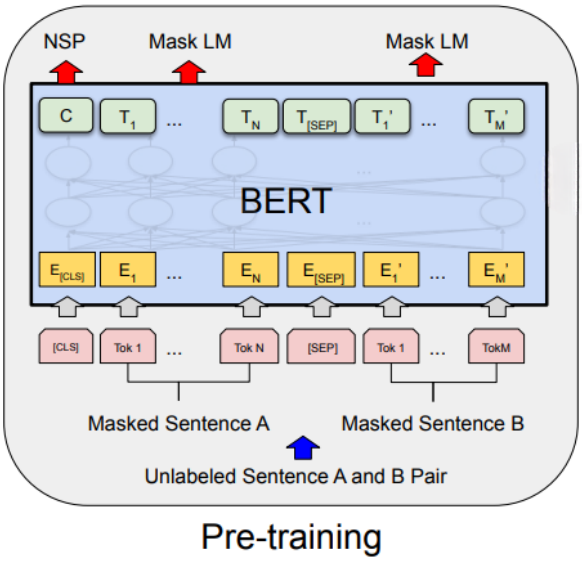


Figure 4.2.3: BERT pre-training. (Source: Devlin et al. [20])

4.3 BERT Encoding Pipeline

Having provided a comprehensive overview of the Transformers architecture, with specific attention to BERT, we are now prepared to explore the practical details of how tweets are processed within these advanced language models. Our exploration will follow a systematic path, beginning with the important step of breaking down tweets into smaller parts (tokenization), moving on to how they are represented using embeddings, and finally, we will examine how BERT transforms them through its contextualizing process. Together, these steps constitute the essence of the BERT Encoding Pipeline, which we will exhaustively explain in the next sections.

4.3.1 BERT Tokenization

Transformer models, like other machine learning algorithms, are unable to directly process raw text. Thus, the initial stage of our task involves the transformation of text inputs into numerical representations that the model can comprehend. This is achieved through a tokenizer [23], which performs the following tasks:

1. Segmentation of the input into tokens, which can be words, subwords, or symbols (as punctuation).
2. Assigning a unique integer to each token.
3. Incorporating any supplementary inputs that might enhance the model's understanding of the data.

All this preprocessing needs to be done in exactly the same way as when the model was pretrained, thus it is important to use the correct tokenizer. In this thesis we will consider two different BERT models: 'bert-base-multilingual-cased' and 'dbmdz/bert-base-italian-xxl-cased'[3]. From now on we will show the codes for the model 'bert-base-multilingual-cased', however there are no relevant differences with respect to the other chosen model.

To understand how the three steps are computed, we first look at the tokenization algorithm behind BERT tokenizer.

WordPiece

There are many subword tokenization algorithms such as Byte-Pair Encoding (BPE) used by GPT-2, WordPiece used by BERT and Unigram used by T5.
WordPiece starts from a small vocabulary including the special tokens[4] used by the model and the initial alphabet. It identifies subwords by adding a prefix (##), so that each word is initially split by adding that

[3] We will explain in detail the differences between the two models in Section 4.4.
[4] [PAD], [UNK], [CLS], [SEP], [MASK]

prefix to all the characters inside the word. For example, 'word' gets split like w ##o ##r ##d. Therefore, the initial alphabet encompasses all characters found at the start of a word and those within a word, provided they are preceded by the WordPiece prefix. Then, WordPiece algorithm learns merge rules to expand the vocabulary: it computes a score for each pair of subwords based on a weighted frequency formula.

$$\text{score} = \frac{\text{freq\_of\_pair}}{\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element}}$$

The algorithm gives preference to merging pairs in which the individual components have lower frequencies within the vocabulary by dividing the frequency of the pair by the product of the frequencies of its constituent parts.
The WordPiece algorithm proceeds by iteratively selecting and merging subword pairs with the highest scores. This process expands the vocabulary while preserving the special tokens and the initial alphabet. It stops when a predefined vocabulary size is reached or when no more suitable subword pairs can be merged.
The benefit of this approach is that it adapts to the characteristics of the training data. Rare and out-of-vocabulary words can be effectively represented as combinations of more common subwords. This not only helps with handling previously unseen words but also contributes to reducing the overall vocabulary size, which can be particularly advantageous for memory and efficiency in natural language processing tasks.

Example of Tokenization

The first step is to import the exact same tokenizer used for the pre-training of our model.

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('bert-base-multilingual-cased')
```

To get a clear understanding of the tokenization process, we start by showing the steps for a single tweet, then we will discuss the implications of tokenizing the entire dataset.

As example, we consider the first tweet of the sentiment dataset train split:

```
example = dataset['train']['text'][0]
print(example)
```

La lotta contro il bodyshaming è una cosa, promuovere l'obesità è un'altra

When we apply the tokenizer to the tweet we get multiple outputs:

```
encoding = tokenizer(example)
print(encoding.keys())
```

dict_keys([’input_ids’, ’token_type_ids’, ’attention_mask’])

We will see in detail each of them, since they represent the inputs for our model.
However, it is important to first look at how the tokenizer has divided the tweet into tokens following the WordPiece algorithm:

```
print(encoding.tokens())
```

[’[CLS]’, ’La’, ’lotta’, ’contro’, ’il’, ’body’, ’##sha’, ’##ming’, ’è’, ’una’, ’cosa’, ’,’, ’promu’, ’##overe’, ’l’, ’[UNK]’, ’ob’, ’##esi’, ’##tà’, ’è’, ’un’, ’[UNK]’, ’altra’, ’[SEP]’]

There are some observations that deserve to be highlighted:

- Tokens such as ‘lotta’ and ‘contro’ are part of the final vocabulary of the tokenizer, thus they are not split into subwords.
- Tokens such as ‘##sha’ and ‘##ming’, which contain the WordPiece prefix, are subword units that are combined to reconstruct complete words. For example, those two combine to form ‘shaming’.
- The presence of the [UNK] tokens means there are characters (in this case the apostrophes) which are not in the vocabulary.
- In this example we can see the presence of the [CLS] token, which indicates the beginning of the tweet, and the [SEP] token, which indicates the end.

Moving to the three main outputs of the tokenizer, we start by analyzing the input_ids which are the numerical representation of the tokens:

```
print(encoding['input_ids'])
```

[101, 10159, 55569, 13722, 10154, 14333, 23315, 16405, 262, 10153, 18887, 117, 77980, 81636, 180, 100, 17339, 17582, 13339, 262, 10119, 100, 15603, 102]

- The values 101, 102 and 100 are special tokens that represents, respectively, the [CLS] token, the [SEP] token and the [UNK] token.
- The subsequent numerical values, such as 10159, 55569, 13722, and so on, correspond to specific tokens in the input text. These tokens could represent words or subwords, and they are encoded as unique numerical identifiers. Each of these values maps to a specific word or subword in the model’s vocabulary.
- input_ids is one of the two inputs that will be fed to the BERT model for sentiment analysis and emotion detection.

The second output are the token_type_ids:

```
print(encoding['token_type_ids'])
```

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

These token type IDs are used to distinguish different sentences within a single input sequence. For example, in question-answering task, we might have two segments: one for the question and another for the answer. However, in text classification tasks like sentiment analysis and emotion detection this output is not useful and thus it will not be considered in the actual tokenization of the dataset.
The last output is attention_mask:

```
print(encoding['attention_mask'])
```

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

This is the second input that will be fed to our model. They indicate which tokens in the input sequence should be attended to and which ones should be ignored during processing. The possible values are 1 (token should be attended) and 0 (token should not be attended). In this example, the model should consider all the tokens in the input sequence, however, as we will see, when considering the entire dataset this is not always the case.

Dataset Tokenization

While the concepts discussed in the preceding sections still hold, tokenization of multiple tweets introduces additional complexities and limitations that must be carefully managed to ensure the model's effectiveness.
A primary concern arises from the varying lengths of tweets, making it impractical to directly input them into the model since it requires fixed-size inputs. To address this issue, we employ two strategies:

- **Padding**: is a text processing technique used to ensure that shorter tweets match the desired length. It involves adding extra tokens, denoted as [PAD], to the original tokenized tweet until it reaches the correct length.
- **Truncation**: is a text processing technique used to ensure that longer tweets match the desired length. Any text exceeding the specified length is removed, retaining only a portion of the original tweet.

```
max_length = 256

def tokenize_text(dataset):
    return tokenizer(
        text=dataset['text'],
        add_special_tokens=True,
        return_token_type_ids=False,
        max_length=max_length,
        padding='max_length',
        truncation=True,
        return_tensors='tf',
        verbose=True
        )
```

In the code the maximum length of the tweet has been set as 256. When choosing that parameter there are some considerations to keep in mind: setting an higher number may help preserving the context, reducing the truncation and the number of special tokens. However, it would increase the memory usage and make slower the processing. The majority of BERT models, as the ones we consider in this thesis, can get as input sequences of a maximum of 512 tokens. However, since we are working with tweets that generally are short texts, the choice has fallen on 256.

Considering these strategies, the previous example tweet would be tokenized as follows:

['[CLS]', 'La', 'lotta', 'contro', 'il', 'body', '##sha', '##ming', 'è', 'una', 'cosa', ',', 'promu', '##overe', 'l', '[UNK]', 'ob', '##esi', '##tà', 'è', 'un', '[UNK]', 'altra', '[SEP]', '[PAD]', ...]

```
input_ids tf.Tensor(
[[  101 10159 55569 13722 10154 14333 23315 16405   262 10153 18887   117
  77980 81636   180   100 17339 17582 13339   262 10119   100 15603   102
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0]], shape=(1, 256), dtype=int32)
attention_mask tf.Tensor(
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0]], shape=(1, 256), dtype=int32)
```

Figure 4.3.1: Example tweet with padding and truncation.

To get a final overview we can make these considerations:

- Each tweet in our dataset is transformed into a numerical vector, represented as input_ids. This vector has a consistent length, where each token within the tweet is associated with a unique numerical value. Notice that the [PAD] token, represented by a 0, is added as many times as necessary after the [SEP] token.
- Differently from the previous section, the attention_mask vector does not contain only 1s, but also some 0s. This is due to the fact that our model should not attend to the [PAD] tokens, which do not carry any relevant information for the text classification task and may introduce noise.

4.3.2 BERT Embeddings

Before a string of text is passed to the BERT model, the BERT Tokenizer is used to convert the input from a string into a list of integer Token IDs, where each ID directly maps to a word or part of a word in the original string. For each unique Token ID, the BERT model contains an embedding that is trained to represent that specific token. The Embedding Layer within the model is responsible for mapping tokens to their corresponding embeddings.

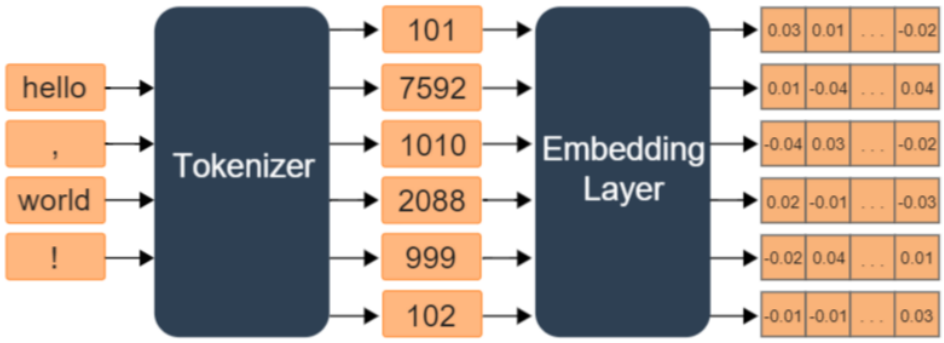


Figure 4.3.2: BERT Embedding Layer. (Source: [24])

An Embedding is a trained numerical representation of a categorical feature[5]. In practice, this means a list of floating point values that are learned during model training.

```
embedding_layer = bert.bert.embeddings
print(embedding_layer.weights)
```

```
<tf.Variable 'tf_bert_model/bert/embeddings/word_embeddings/weight:0' shape=(119547, 768) dtype=float32, numpy=
array([[ 0.02595074, -0.00617341, -0.00409975, ...,  0.02965234,
         0.02417551,  0.01970279],
       [ 0.01038065, -0.0136286 ,  0.00672081, ...,  0.01237162,
         0.0267217 ,  0.03370738],
       [ 0.0220679 , -0.00360613,  0.01932366, ...,  0.0069061 ,
         0.026809  ,  0.00498276],
       ...,
       [ 0.00684139,  0.01885802,  0.02666426, ...,  0.02292391,
         0.06465269,  0.04373793],
       [ 0.0183579 ,  0.01480132,  0.02434449, ...,  0.03205629,
         0.00708906,  0.02039703],
       [ 0.02139908,  0.01879423, -0.01343376, ..., -0.00597953,
         0.00583893, -0.00586251]], dtype=float32)>
```

The embeddings are returned as a 119,547 x 768 matrix, or 2-dimensional tensor:

- The first dimension of this tensor is the size of the BERT tokenizer's vocabulary, which is 119,547 for bert-base-multilingual-cased.
- The second dimension is the embedding size, which is also called the Hidden Size. This is the number of trainable weights for each token in the vocabulary. Both the models we consider have a Hidden Size of 768, however there are other variations that have been trained with smaller and larger values.

The embeddings for each token ID are not random numbers since the values were learned when the BERT model was trained, which means that each embedding encodes the model's understand of that particular token.
In addition to the Token Embeddings BERT also relies on Position Embeddings, as we can see at the

[5]We can think of the input_ids as a list of categorical features for each tweet.

beginning of Figure 4.1.1.While Token Embeddings are used to represent each possible word or subword that can be provided to the model, Position Embeddings represent the position of each token in the input sequence.

```
print(embedding_layer.position_embeddings)
```

```
<tf.Variable 'tf_bert_model/bert/embeddings/position_embeddings/embeddings:0' shape=(512, 768) dtype=float32, numpy=
array([[-0.02327054,  0.00622581, -0.01144287, ...,  0.01739656,
        -0.00897003,  0.00576193],
       [-0.00969618, -0.01121458,  0.03502079, ...,  0.01846941,
        -0.00101469, -0.0076937 ],
       [-0.00575436, -0.0042865 ,  0.0217139 , ...,  0.00676087,
         0.00478133, -0.00317525],
       ...,
       [ 0.01440435,  0.05431867, -0.00102155, ..., -0.01291058,
        -0.00273864,  0.04081771],
       [ 0.01337033,  0.03064424, -0.00682547, ..., -0.01116468,
        -0.01331095,  0.03087371],
       [ 0.01368413,  0.07561649,  0.00604935, ..., -0.01477607,
        -0.0141621 ,  0.03253375]], dtype=float32)>
```

In this case we get a 2-dimensional tensor of shape 512 x 768, since the maximum length of the input sequence that BERT model can accept is 512. However, as we said in the tokenization part, we will use only the first 256 positional encodings embeddings as we set a limit to the maximum length of the tokenized tweets.
Finally, BERT model has a last type of embedding representing the token type, i.e whether the token belongs to sentence A or sentence B. As we've seen, we do not need this embedding to perform sentiment analysis and emotion detection, thus we will not entrer in the details.

Summarizing, Bert Embedding Layer is responsible for computing a final embedding for each input token by combining the different embeddings discussed so far. The final embedding for each token is computed by summing the embeddings for its specific token ID, position, (and token type when necessary), and then applying normalization on the sums.

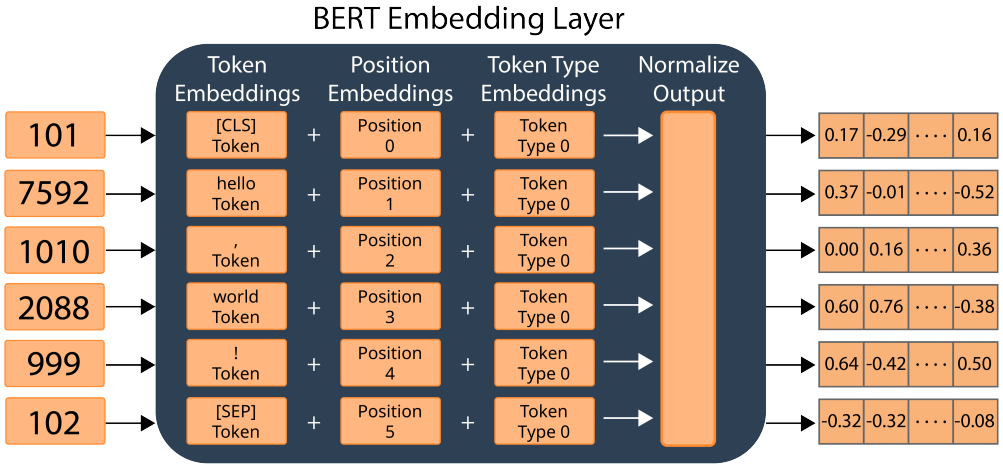


Figure 4.3.3: BERT Embedding Layer Summary. (Source: [24])

To get a final visualization of the BERT Embedding Layer, we examine the output of the the same tweet used as example in the tokenization part:

```
example = dataset['train']['text'][0]
encoding = tokenizer(
    text=example,
    add_special_tokens=True,
    return_token_type_ids=False,
    max_length=max_length,
    padding='max_length',
    truncation=True,
    return_tensors='tf',
    verbose=True
)
input_ids = tf.convert_to_tensor(encoding['input_ids'])
final_embeddings = embedding_layer(input_ids)
print(final_embeddings)
```

```
tf.Tensor(
[[[-0.38289857  0.12982383 -0.15237723 ...  0.28270796 -0.4924395
   -0.02894128]
  [ 0.49540383 -0.18398705  0.08900508 ...  0.5321044  -0.5706177
    0.44490868]
  [ 0.42196965 -0.47777924  1.7480626  ...  0.01645073  0.7944187
    0.49709898]
  ...
  [ 0.44564253 -0.61237437 -0.1198297  ...  0.565477    0.5501125
    0.44818127]
  [ 0.3008211  -0.35981387  0.13772365 ...  0.4765409   0.723824
    0.6512687 ]
  [ 0.5864303  -0.9104549   0.3044351  ...  0.4166163   0.45579207
    0.40734544]]], shape=(1, 256, 768), dtype=float32)
```

It is important to take a look at the dimensions of the final embedding of the tweet: (1, 256, 768). The first number (1) represents the batch size, which in this case is one since we got the embedding of only one tweet. The second number (256) is the number of input tokens, i.e the sequence length of each tweet. Finally, the third number (768) represents the hidden size.

Following Figure 4.1.1, we now have everything we need to enter the encoder part of the Transformer.

4.3.3 BERT Encoder Layer

The Encoder is the main building block of the BERT model architecture. As input, the Encoder takes text embeddings produced by the Embedding Layer, and as output, it returns modified embeddings of the same shape [25]. Because the input and output of the Encoder have the same shape, it is possible to chain

multiple Encoders together such that the output of one becomes the input to next. In fact, the original BERT model architecture includes 12 Encoders chained together in this fashion.

As mentioned, the BERT Encoder takes embeddings as input and produces embeddings as output. What is the difference between them?
To answer this question, we first need to consider the embeddings produced by the embedding layer. For each input token, the Embedding Layer produces an embedding that represents that specific token, as well as its position in the input sequence of text. The individual word and its position, however, are not enough to understand a word's meaning in a text. The problem is that the same word could have different meanings depending on the context. That is precisely the main purpose of the BERT Encoder Layer: it gets as input embeddings, and it returns as output *contextualized* embeddings. To achieve this objective it uses a specialized attention mechanism known as self-attention, which enables the model to establish connections between individual words in the input by allowing each word to be associated with other words. To understand this mechanism we break down the BERT encoder and we follow step by step the passages to further process the embeddings.

The encoder involves two essential sub-modules: multi-head attention and a fully connected neural network. Surrounding each of these sub-modules, there are also residual connections, which facilitate information flow, followed by layer normalization to stabilize the output (Figure 4.1.1).
At first the embeddings produced by the emebdding layer are passed separately through three fully-connected layers called **Query**, **Key** and **Value**, producing a modified embedding for each word. Attention is computed as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V \tag{4.1}$$

1. **Dot product of Query and Key**: The result is a NxN score matrix, where N represents the number of words in the input sequence (in our case 256). The score matrix determines the degree of attention that each word should give to other words. Each word is assigned a score that corresponds to its relevance to other words, where an higher score indicates a greater degree of focus. To prevent the model to calculate the score with respect to the [PAD] tokens, the attention masks, which were the second output of the tokenization process, indicate which are the tokens that should be left out of the calculation marking them with a 0 score.

2. **Softmax of Scaled Scores**: Afterward, the scores are scaled down by dividing them by the square root of the dimensions of the query and key vectors. The dimensionality of the query and key vectors depends on the hidden size of the model and the number of heads employed in the multi-head attention mechanism[6]. In our case we have $\sqrt{d_k} = \sqrt{\frac{768}{12}} = 8$. This scaling operation is

[6]The dimensionality of the query and key vector is given by $\frac{\text{Hidden Size}}{\text{Number of attention heads}}$.

performed to ensure stable gradients since direct multiplication can lead to exploding values. Following the scaling, the softmax function is applied to the scaled scores. This softmax operation produces attention weights, converting the scores into probability values ranging from 0 to 1.

3. **Output and Multi-Head Attention**: Following the computation of softmax scores, we proceed to multiply these attention weights by the value vector. This results in an output vector where words with higher softmax scores retain their values, emphasizing the words the model deems important, while the words with lower scores are diminished, reducing the impact of less relevant words. This output vector is then passed through a linear layer for further processing.
   To implement multi-headed attention, we first split the query, key, and value vectors into *h* separate vectors before applying the self-attention mechanism. Each of these separate self-attention processes is referred to as a 'head'. Each head independently produces an output vector. These output vectors are concatenated into a single vector before being passed through the final linear layer.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \cdot W^O$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, \quad W^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}}$$

In essence, each head has the potential to learn different patterns or relationships within the data, enhancing the model's representation capacity.

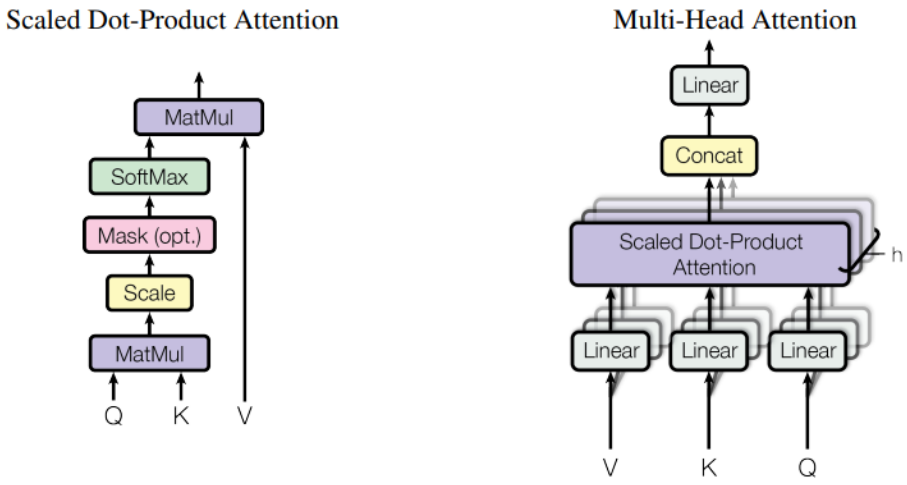


Figure 4.3.4: Scaled dot produt and multi-head attention. (Source: Vaswani et al. [19]).

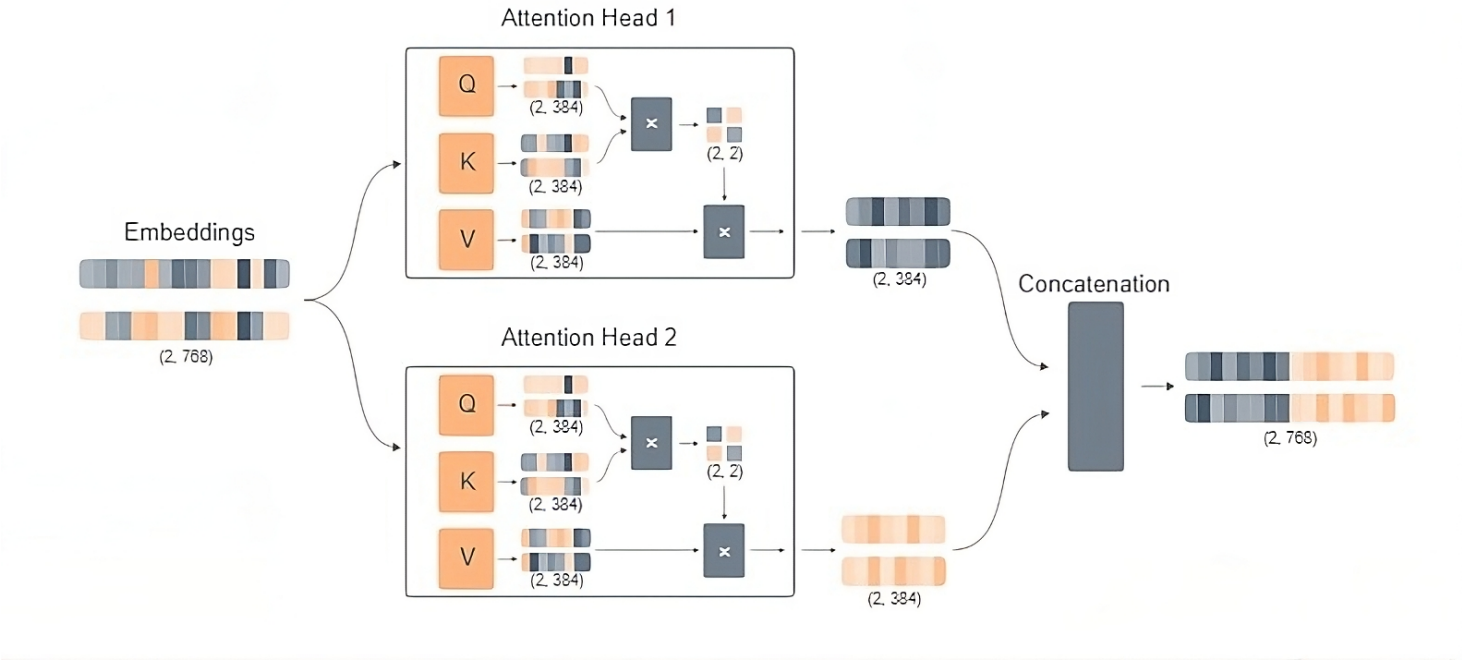


Figure 4.3.5: Example of multi-head attention. (Source: [25]).
Note: The shapes of the inputs and intermediate outputs are included to illustrate how multi-head attention produces outputs in the same shape as the input. The values above assume an input length of 2 tokens and an embedding dimension (i.e. hidden size) of 768.

The input embeddings consist of vectors with 768 elements, whereas the query, key, and value vectors calculated within the attention heads have only 384 values each. In this example, the number 384 is referred to as the 'Attention Head Size,' and it is determined by dividing the hidden size by the number of attention heads. This design choice ensures that when we concatenate the outputs from all attention heads, the final output maintains the same shape as the input embeddings.

4. **Residual Connections, Layer Normalization, and Feed Forward Network**: The multi-head attention output vector is combined with the original input embedding using a residual connection. This technique helps gradients flow more smoothly through the network. The output of this connection then undergoes layer normalization for stability.
   Subsequently, the normalized output from the residual connection is passed through a pointwise feed-forward network for further processing. This feed-forward network consists of a couple of linear layers with a ReLU activation in between. The output of this network is again added to the input of the pointwise feed-forward layer and further normalized.
   These residual connections facilitate training by enabling gradient flow directly through the networks. The layer normalization steps are employed to stabilize the network, significantly reducing the required training time. Additionally, the pointwise feedforward layer projects the attention outputs, potentially enriching their representation.

4.4 Model Architecture and Fine-Tuning

Now that we have a solid grasp of how BERT operates, we enter into the final stages required for sentiment analysis and emotion detection. To recap, the initial steps involve tokenizing the raw tweets and converting them into embeddings. Subsequently, these embeddings undergo additional processing to yield what we refer to as 'Hidden States'.
In order to fine-tune the model and perform text classification using its output, we need to add an additional component known as a classification head (Figure 4.4.1).

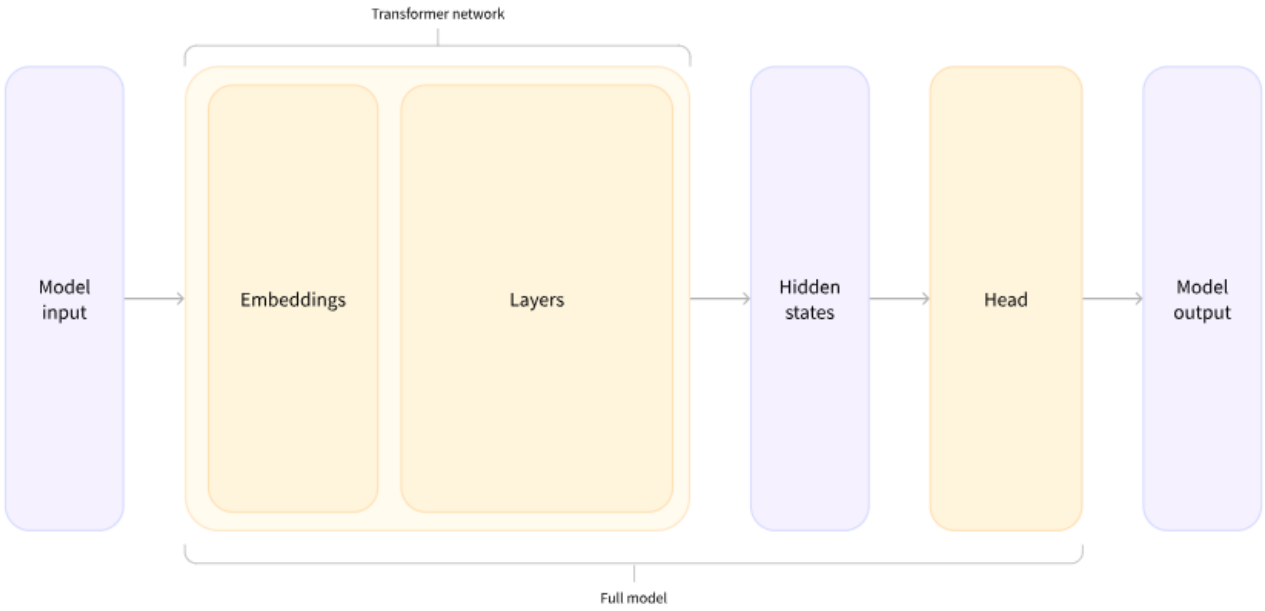


Figure 4.4.1: Model Architecture (Source: Hugging Face [23])

This classification head typically consists of two key elements: a dense layer and a softmax activation function.

1. **Dense Layer**: The dense layer is used to transform the high-dimensional hidden states into a format that is suitable for the specific classification task. This transformation typically involves reducing the dimensionality of the data and capturing relevant features that are important for classification.

2. **Softmax Activation**: After the dense layer, we apply the softmax activation function. The softmax function takes the transformed features and normalizes them to produce probability scores. These scores represent the likelihood of the input belonging to each class.

Due to the multi-layer structure of Transformers, different layers capture different levels of representations [26]. For this reason, we can explore different representations of BERT output to use in the classification head.

Finally, to improve the model's performance for our specific downstream task, we fine-tune it. This involves taking the pre-trained model and training it on our specific datasets (sentiment and emotion separately). The pre-trained model's parameters are then adjusted using gradients derived from our datasets. The primary objective is to make little modifications to the model's parameters while preserving the knowledge acquired during the initial pre-training phase.

4.4.1 Model Variants

The two specific checkpoints chosen in this analysis are 'bert-base-multilingual-cased' and 'dbmdz/bert-base-italian-xxl-cased'[7][27]. Even though they both rely on the BERT architecture, they differ in the data used for the pre-training phase. The dataset used to train the multilingual model is wikipedia from Hugging Face, which contains 104 languages. For the italian language, the subset is composed by 1,743,035 training texts from a variety of subjects. The italian BERT, other than wikipedia, has been trained on various texts from the OPUS corpora and data from the italian part of the OSCAR corpus, achieving a final training corpus of 81GB size and 13,138,379,147 tokens.
In their names both models share the words 'base' and 'cased': the former represents the model size and the latter the fact that they make a distinction between uppercase and lowercase letters. To understand the model size, it is important to note that in the original paper, Devlin et al. [20] propose two different model size: $\textbf{BERT}_{\textbf{BASE}}$ and $\textbf{BERT}_{\textbf{LARGE}}$.

Model	Layers (L)	Hidden Size (H)	Attention Heads (A)	Total Parameters
$\textbf{BERT}_{\textbf{BASE}}$	12	768	12	110M
$\textbf{BERT}_{\textbf{LARGE}}$	24	1024	16	340M

Table 4.4.1: Specifications of $\textbf{BERT}_{\textbf{BASE}}$ and $\textbf{BERT}_{\textbf{LARGE}}$ Models

- **Layers (L)**: it represents the number of Transformers layers in the BERT model. Increasing the number of layers allows the model to learn more complex patterns and dependencies but also makes the model computationally more intensive.
- **Hidden Size (H)**: it denotes the dimensionality of the model's hidden states. The hidden size determines the capacity of the model to represent and capture information. A higher hidden size allows the model to store more detailed information but can also increase memory and computational requirements.
- **Attention Heads (A)**: represents the number of attention heads used in the multi-head self-attention mechanism within each transformer layer. Each attention head can focus on different parts of the input sequence, allowing the model to capture various types of relationships and dependencies.

[7] In the results we will refer to as BERT and itaBERT, respectively.

4.4.2 Output Representations

In HuggingFace Transformers there are 3 possible outputs [28]:

- **Pooler output** (batch size, hidden size): Last layer hidden state of the first token of the sequence [CLS].
- **Last Hidden State** (batch size, sequence length, hidden size): Last layer hidden states of the entire sequence.
- **Hidden States** (number of layers, batch size, sequence length, hidden size): Hidden states for all layers and for all ids.

The batch size represents the number of tweets processed at a time and it remains unspecified until the fine-tuning phase. The sequence length is specified in the tokenizer and in this study has been chosen as 256. Finally, the hidden size represents the dimensionality of the hidden states (in our models is 768).
In figure 4.4.2 we can see the representations of two of the afromentioned outputs [8]. In the left there is the last hidden state, where each row represents a tweet, each column a token in the sequence and the depth is the hidden size. In the right there is the representation of the pooler output, since only the [CLS] token of each tweet is taken into consideration. Finally, to visualize the last possible output (Hidden States), we can imagine to have the left representation once for each layer in the model: in this case we would have 12 hidden states, each capturing different pieces of information.

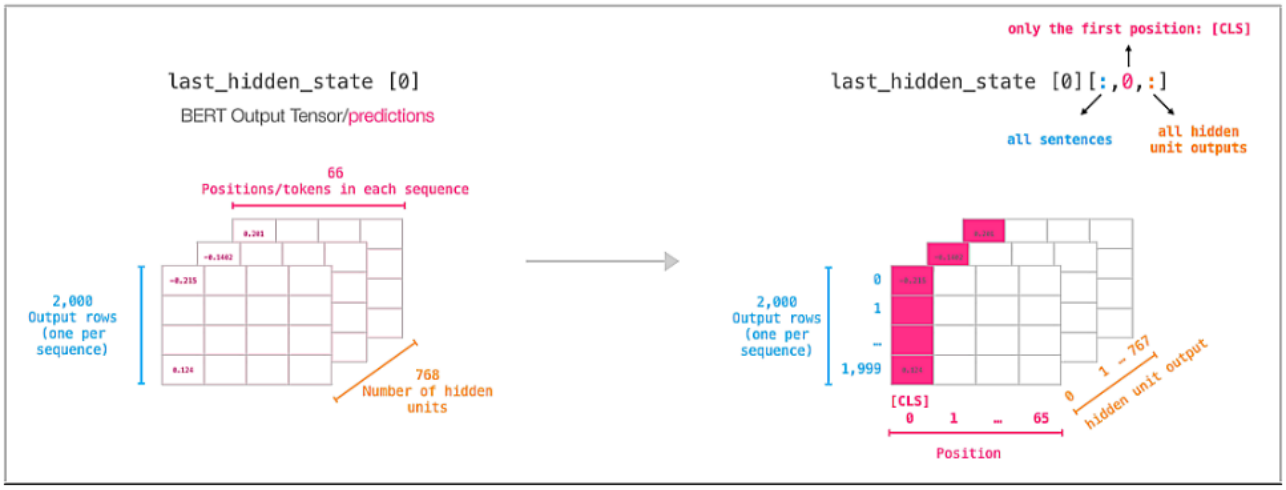


Figure 4.4.2: Output Representations (Source [28])

[8] This figure is intended for visualization purposes, so the specific dimensions should not be taken into account.

Pooler Output - CLS Token

The most straightforward approach for conducting sentiment analysis involves using the final layer hidden state of the [CLS] token as input for the classification head [26]. This choice is made because the [CLS] token should effectively capture the entire context of the tweet. To understand why this is the case, we recall how BERT model was trained. As we mentioned in section 4.2.1, masked language modelling helps the model to be bidirectional. Moreover, the model uses a [CLS] token for predicting whether the second sentence of a pair is indeed the subsequent (4.2.2). The model is trained to understand the relationships between sentences and to generate a representation for the [CLS] token that encodes the semantic information of the entire input sequence. Even though in sentiment analysis and emotion detection the input is composed just by one tweet, the [CLS] can still retain the meaning of the entire sequence.

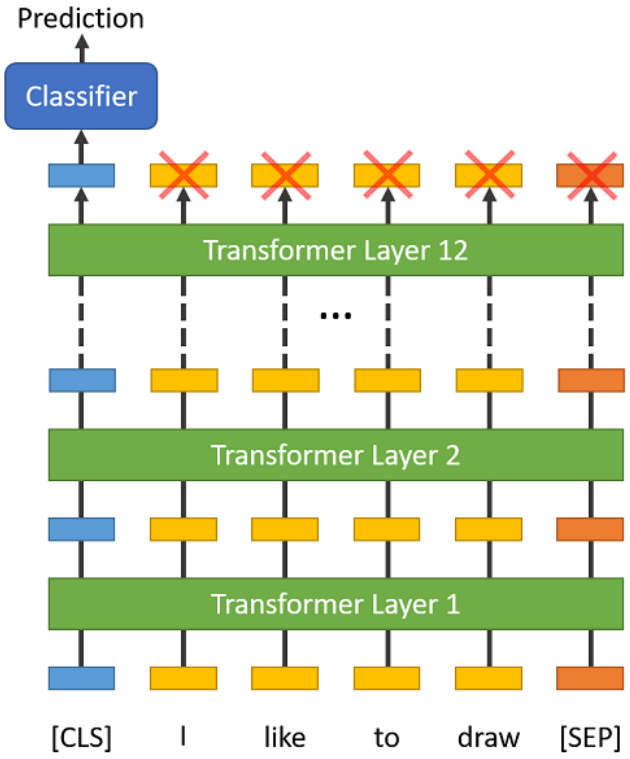


Figure 4.4.3: Pooler Output - CLS Token (Source [28])

Last Hidden State - Max Pooling

The second approach consists in using the entire last hidden state. Instead of feeding to the classification head only the [CLS] token, we first apply a max pooling over the entire tokens of the tweet. Max pooling is a mathematical operation that selects the maximum value from a set of values. In the context of NLP and Transformers, it is used to identify the most important feature within a sequence. By applying max pooling over the last hidden state, we effectively capture the most critical information within the tweet, regardless of its position in the sequence.

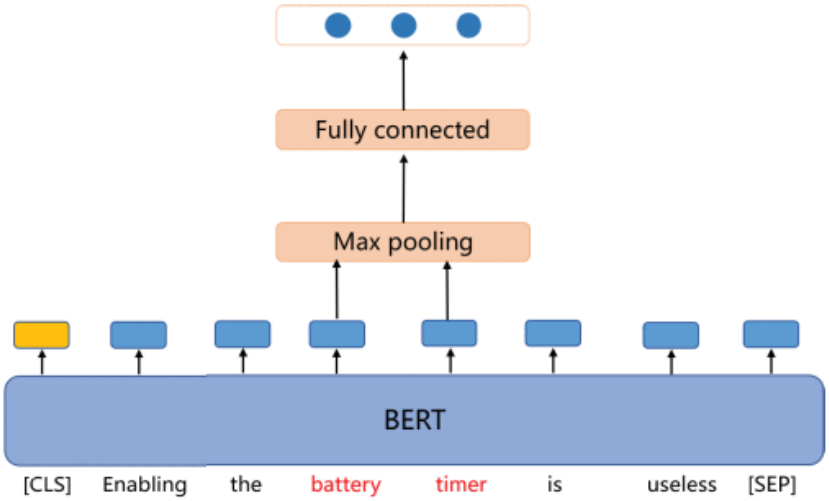


Figure 4.4.4: Last Hidden State - Max Pooling (Source [28])

Hidden States - LSTM Pooling

The last technique we explore consists in concatenating the hidden states of the [CLS] token from all L layers, $\mathbf{h}_{CLS} = \{h^1_{CLS}, h^2_{CLS}, \ldots, h^L_{CLS}\}$. Because LSTM networks are well-suited for handling sequential data, we employ it to link all the intermediate representations of the [CLS] token. The final representation is determined by the output of the last LSTM cell.

$$o = h^L_{LSTM} = LSTM(h^i_{CLS}), \quad i \in [1, L] \tag{4.2}$$

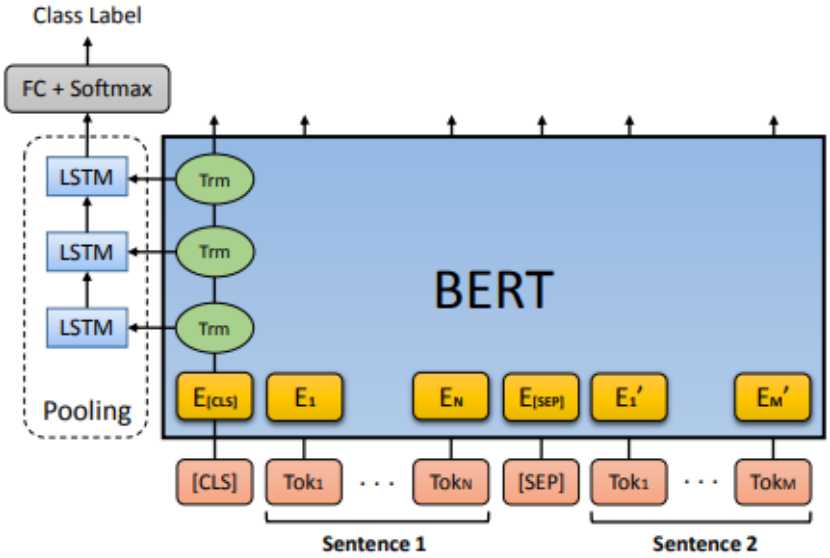


Figure 4.4.5: Hidden States - LSTM Pooling (Source [26])

4.4.3 Fine-Tuning

The last step before evaluating the results is fine-tuning. To do it we need to define a loss function, to choose an optimizer and to train our models on labeled data.
As we have already seen in tables 2.1.3 and 2.1.4, our datasets are divided into train, validation and test. We fit the models on the train split to update their weights, while the validation data is used to monitor the models performance during training and to prevent overfitting. Finally, we evaluate the results using the test split, which is composed by new unseen data. To better understand this procedure, we explain one by one each component.

Loss Function

The loss function chosen to fine-tune our models is the sparse categorical cross entropy[9]. It relies on the cross entropy loss and in a multi-class classification problem is computed as follows:

$$L_{CE} = -\sum_{c=1}^{M} y_c \, log(p_c) \tag{4.3}$$

In equation 4.3 c represents the class, M the total number of classes (3 for sentiment and 7 for emotion), y_c is a binary indicator (0,1) for the true class, p_c is the probability given by the softmax activation function of a tweet belonging to that class. It is important to notice that the indicator y_c is always 0 except for the correct class, thus for each tweet we would only compute the loss for the actual class. For example, if the model predicts a tweet to be positive with 100% of confidence and it is indeed positive, then we would have a zero loss for that observation.
The choice of the '*sparse* categorical cross entropy', instead of 'categorical cross entropy' is given by the way in which the observations are labeled: if they were represented as a one-hot encoded vector (such as [0,1,0]) then the second would have been the corrected loss function. In our datasets labels are represented as integers (such as [0], [1]...) which are the correct representation for the *sparse* one.

Optimizer

The optimizer is responsible for adjusting the model's parameters during the training process to minimize the loss function.
One of the most popular optimizers to train deep learning models is Stochastic Gradient Descent (SGD), which basically differs from the classical Gradient Descent by the fact that the gradient of the loss function is not computed on the entire dataset but by choosing randomly one observation at a time.
However, by making numerous trials with different optimizers, the best one has been AdamW [29]. It stands for 'Adaptive Moment Estimation' with decoupled weight decay and it is characterized by using

[9] Sparse Categorical Cross Entropy - Keras

'momentum', by adapting learning rates and by applying weight decay. To understand how it works and why it is better than SGD and Adam, we take a look at its algorithm:

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $t \leftarrow t + 1$
5: $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
6: $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
7: $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
8: $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9: $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
10: $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
13: **until** *stopping criterion is met*
14: **return** optimized parameters $\boldsymbol{\theta}_t$

Figure 4.4.6: AdamW Algorithm. (Source: [29])

- **(1-3)**. α represents the learning rate, while β_1 and β_2 serve as smoothing factors for momentum and squared gradient, respectively. Additionally, ε is a constant used to prevent division by zero, and λ is the weight decay parameter. Then, the parameters and other vectors are initialized, marking the beginning of the optimization loop.

- **(4-6)**. We increment the time step by 1, and at each time step, we compute the gradient using the parameters obtained at the previous time step (t-1). It's worth noting that in the standard Adam formulation (indicated in violet), the weight decay penalty has a direct impact on the gradient. The issue with basic Adam is that it inadvertently applies weight decay (L2 regularization) to both the model's weights and the moving averages, which wasn't the intended behavior. Weight decay should only apply to the model's weights to prevent overfitting by penalizing large parameter values.

- **(7-8)**. We update the first moment vector, denoted as m_t, which helps smoothing the optimization process and speed up convergence. Similarly, we update the second moment vector, v_t, which is responsible for dynamically adjusting the learning rates for the parameters at each step.

- **(9-10)**. Both momentum (m_t) and squared gradient (v_t) moving averages can be biased towards zero at the beginning of training (especially when β_1 and β_2 are close to 1). AdamW includes bias correction steps to account for this bias.

- **(11-14)**: Finally, the updated parameters are computed using the calculated gradients, first and second moment estimates. This is the key step of the AdamW algorithm:

$$\theta_t = \theta_{t-1} - \eta_t(\alpha\hat{m}_t/(\sqrt{\hat{v}_t} + \varepsilon) + \lambda\theta_{t-1}) \tag{4.4}$$

  The crucial part of AdamW is the fact that weight decay only affects the models weights and does not influence the momentum (m_t) or the squared gradient (v_t).

Learning Rate, Batch Size and Number of Epochs

In fine-tuning there are some hyperparameters that play an important role because they influence how models learn from the training data and ultimately influence their performance. Most of them are left unchanged with respect to their values used in pre-training except for:

- **Learning Rate**: It controls the step size at which a model's parameters are updated during the training process. An high learning rate may lead to a faster convergence, however it could skip optimal solutions. In the other hand, a low learning rate helps preventing that problem but could get stuck in local minima.

- **Batch Size**: During fine-tuning, training data is typically divided into smaller batches for processing. Batch size is the number of training examples used in a single forward and backward pass through the model. Smaller batch sizes may result in noisier updates, but they can help the model generalize better. Larger ones may provide more stable updates but require more memory and computation.

- **Number of Epochs**: An epoch is one complete pass through the entire training dataset during the training of a model, thus its number specifies how many times the model will see the entire training dataset. Training for more epochs can help a model learn better, up to a point, but too many epochs can lead to overfitting, where the model learns the training data too well but fails to generalize on new data.

To determine the best set of hyperparameters a grid search has been performed. We first specify a set of possible values for each hyperparameter and then we crate a grid, i.e all possible combinations of hyperparameters values. For each combination we train a model using the training data and evaluate its performance using the validation split. The best yielding results combination will be one used to predict the test data.
Following the original paper of BERT [20] and by making some trials, the values chosen for the grid search are:

Hyperparameter	Possible Values
Batch Size	16, 32
Learning Rate (AdamW)	4e-5, 3e-5, 2e-5, 1e-5
Number of Epochs	2, 3, 4

Table 4.4.2: Hyperparameter Values for Grid Search

However, in the actual grid search performed the values of 'Number of Epochs' were not considered since an early stop callback has been applied[10]: it keeps track of the validation loss during training and, if it stops improving for a certain number of epochs, it stops the training. Furthermore, an argument called 'restore_best_weights = True' has been applied, meaning that after training the model's weights yielding the best results are restored, helping the model not to overfit.
After the hyperparameter tuning, the chosen values are:

Hyperparameter	Chosen Value
Batch Size	16
Learning Rate (AdamW)	2e-5 & 1e-5

Table 4.4.3: Chosen Hyperparameter Values.
Note: 2e-5 is used for the Italian BERT, meanwhile 1e-5 for the multilingual one.

[10]Early Stopping - Keras

Chapter 5

Results and Discussion

In this chapter, we will analyze the outcomes of the different models used for sentiment analysis and emotion detection. We begin by examining the confusion matrix to gain an initial visual understanding of the results. Then, we will look into the metrics to evaluate the models' performance. Additionally, for the Transformers models, we will explore the training process, including the evolution of training and validation loss across epochs.

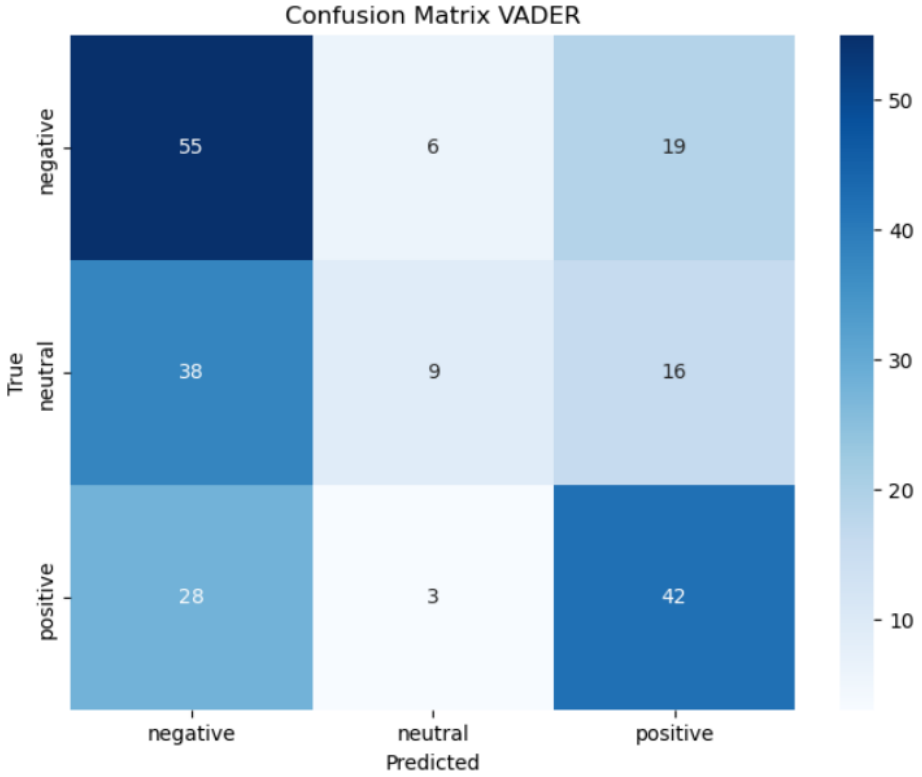
5.1 Sentiment Analysis

Before discussing the results, it is important to highlight that, as shown in Figure 2.1.1 and Table 2.1.7, the sentiment dataset is almost balanced. Each class is adequately represented, meaning that any particular behaviour in some of the sentiment classes may be attributed to the model itself. In a balanced dataset, there is less risk of model bias towards any specific sentiment class. If one class is heavily overrepresented, the model might perform well on that class but poorly on others.

VADER

When it comes to sentiment analysis, the VADER model exhibits the poorest performance with an accuracy of 49%. As depicted in Figure 5.1.1, it performs reasonably well in classifying negative and positive tweets, but it struggles with the neutral class. Notably, the bad performance in the neutral class is evident not only when the true class is indeed neutral, but also in the misclassifications of the other two classes, which tend to be categorized as either positive or negative rather than neutral. A closer look at the F1-Score in Table 5.1.1 reveals a value of 0.22 for the neutral class, which is the lowest among all the models. This poor performance may be attributed to the choice of the threshold values used to classify sentiment classes. As explained previously, the range of compound scores designated for the neutral class in VADER is represented by the interval of [-0.05, 0.05], which is relatively narrow compared to the broader ranges used for positive and negative sentiment.
One potential approach to improve performance could involve conducting cross-validation on the thresh-

old parameters. This method would allow for the selection of optimal threshold values specifically for this task. However, it's important to note that such an approach was not explored in the present analysis.



Class	Precision	Recall	F1-Score	Support
Negative	0.45	0.69	0.55	80
Neutral	0.50	0.14	0.22	63
Positive	0.55	0.58	0.56	73
Accuracy			0.49	216
Macro avg	0.50	0.47	0.44	216
Weighted avg	0.50	0.49	0.46	216

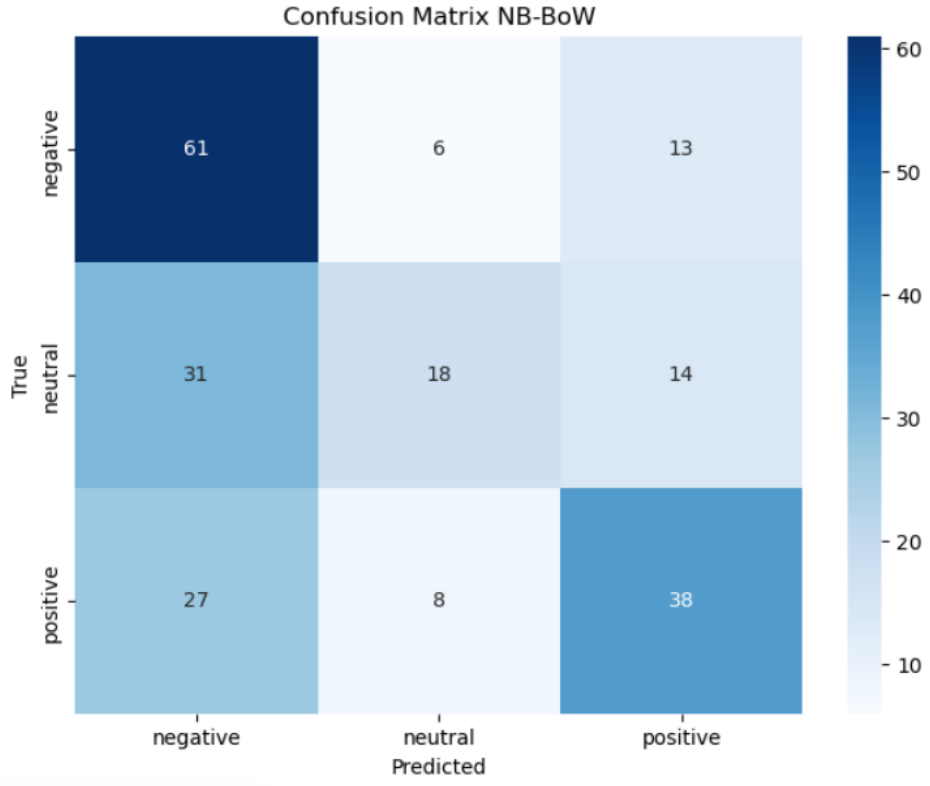
Figure 5.1.1: Confusion Matrix and Classification Report VADER

Naive Bayes and Support Vector Machines

Machine Learning algorithms demonstrated small improvements in performance when compared to VADER sentiment analysis, achieving an overall accuracy that falls within a range spanning from 53% to 55%, depending on the specific model and features employed. This improvement suggests a higher capacity of effectively capturing and analyzing complex patterns in text data, resulting in a more accurate and precise understanding of sentiment compared to traditional rule-based approaches like VADER. However, it must be highlighted that even though they perform better, these results are not satisfying

since that, on average, less than 2 out of 3 tweets are correctly classified.

We begin by examining the confusion matrices for the Naive Bayes classifier in the context of two different feature extraction techniques: Bag of Words (BoW) and TF-IDF.
In comparison to the VADER sentiment analysis results, the performance of Naive Bayes with BoW features (Figure and Table 5.1.3) shows a similar pattern, but there is a noticeable improvement in the neutral class, even though it is not on the same level as the Transformers models.

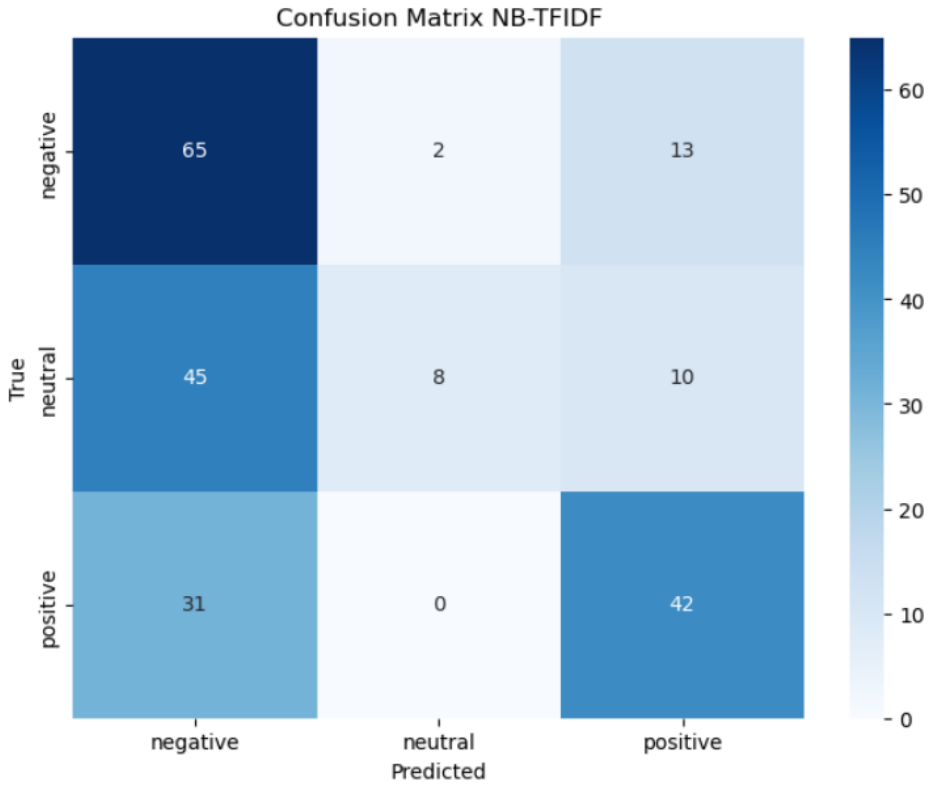


Class	Precision	Recall	F1-Score	Support
Negative	0.51	0.76	0.61	80
Neutral	0.56	0.29	0.38	63
Positive	0.58	0.52	0.55	73
Accuracy			0.54	216
Macro avg	0.55	0.52	0.51	216
Weighted avg	0.55	0.54	0.52	216

Figure 5.1.2: Confusion Matrix and Classification Repor NB-BoW

In the other hand, when we shift our focus to the results obtained using TF-IDF (Figure and Table 5.1.3), we observe a strong negative bias. The recall for the negative class is 0.81, the highest among all models in the sentiment analysis task. Furthermore, over 70% of the neutral tweets are misclassified as negative,

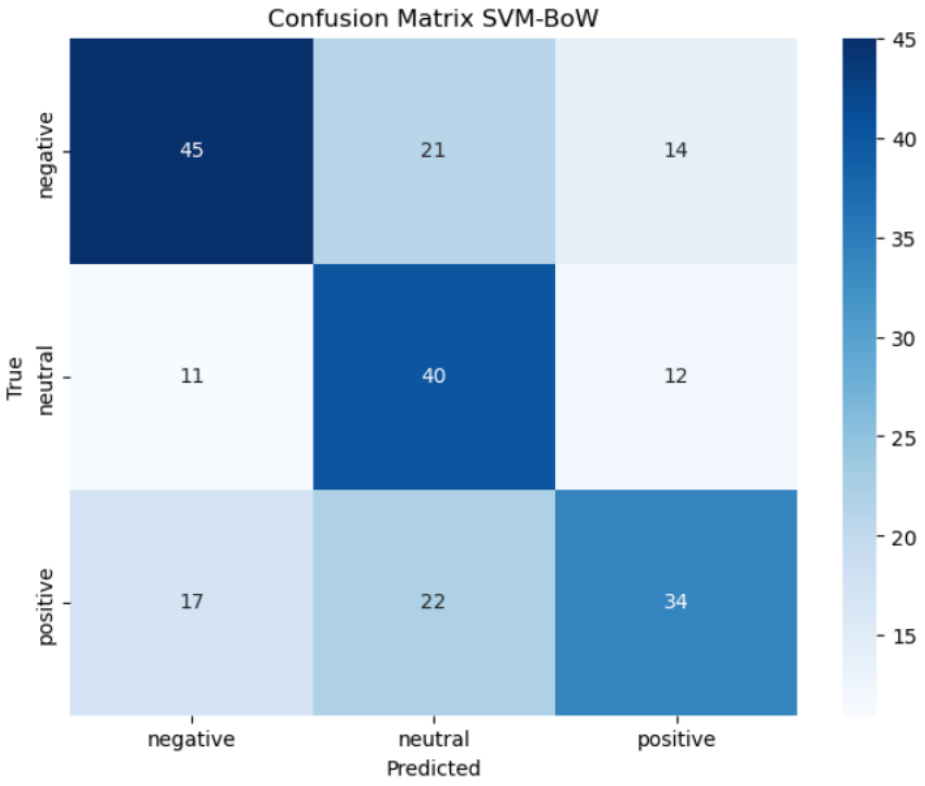
providing further evidence of the bias. Again, the primary issue with this approach remains the neutral class, which maintains the same F1-Score as in the case of VADER, at 0.22. Even though the precision for neutral tweets is 0.80, indicating that when a tweet is predicted as neutral it is usually correct, the recall is only 0.13, as only 10 out of 216 tweets were predicted belonging to that class.



Class	Precision	Recall	F1-Score	Support
Negative	0.46	0.81	0.59	80
Neutral	0.80	0.13	0.22	63
Positive	0.65	0.58	0.61	73
Accuracy			0.53	216
Macro avg	0.64	0.50	0.47	216
Weighted avg	0.62	0.53	0.49	216

Figure 5.1.3: Confusion Matrix and Classification Report NB-TFIDF

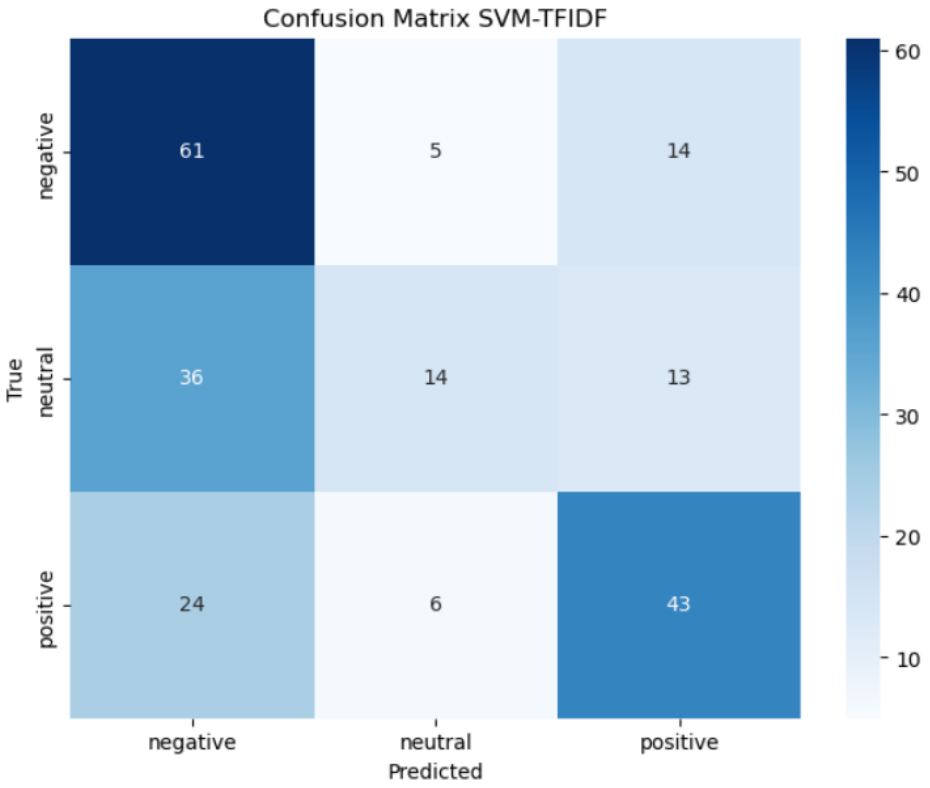
In the context of SVM with BoW features (Figure and Table 5.1.4), we observe a relatively balanced performance. Each sentiment class achieves an F1-Score above 0.50, which suggests that this model does not suffer from a low recall for the neutral class. In fact, the recall for the neutral class is 0.63, which is the highest among all models, including the Transformer-based ones.



Class	Precision	Recall	F1-Score	Support
Negative	0.62	0.56	0.59	80
Neutral	0.48	0.63	0.55	63
Positive	0.57	0.47	0.51	73
Accuracy			0.55	216
Macro avg	0.56	0.55	0.55	216
Weighted avg	0.56	0.55	0.55	216

Figure 5.1.4: Confusion Matrix and Classification Report SVM-BoW

However, when we examine the results obtained using SVM with TF-IDF features (Figure and Table 5.1.5), we find strong results for the negative and positive sentiment classes since they have both an F1-Score higher than 0.60. Nevertheless, similar to the previous models, SVM with TF-IDF struggles with the recall for the neutral class.

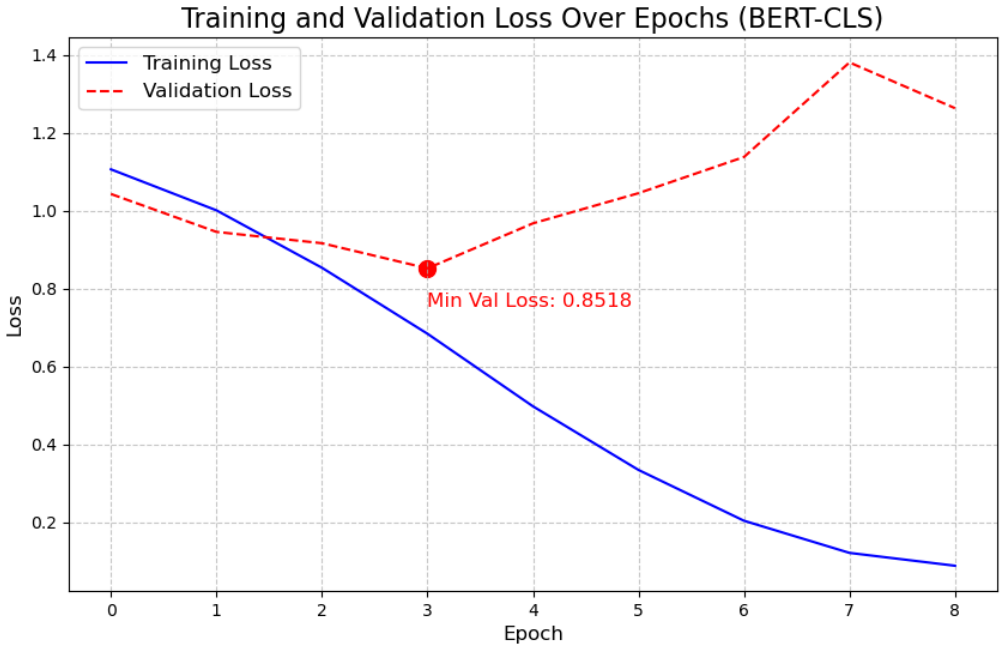


Class	Precision	Recall	F1-Score	Support
Negative	0.50	0.76	0.61	80
Neutral	0.56	0.22	0.32	63
Positive	0.61	0.59	0.60	73
Accuracy			0.55	216
Macro avg	0.56	0.52	0.51	216
Weighted avg	0.56	0.55	0.52	216

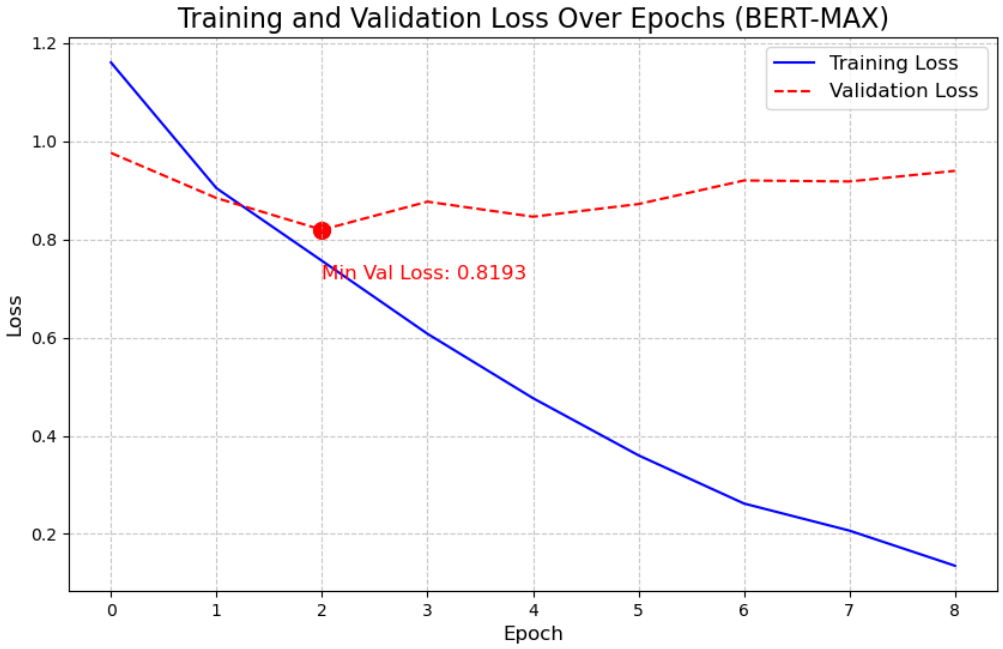
Figure 5.1.5: Confusion Matrix and Classification Report SVM-TFIDF

BERT Models

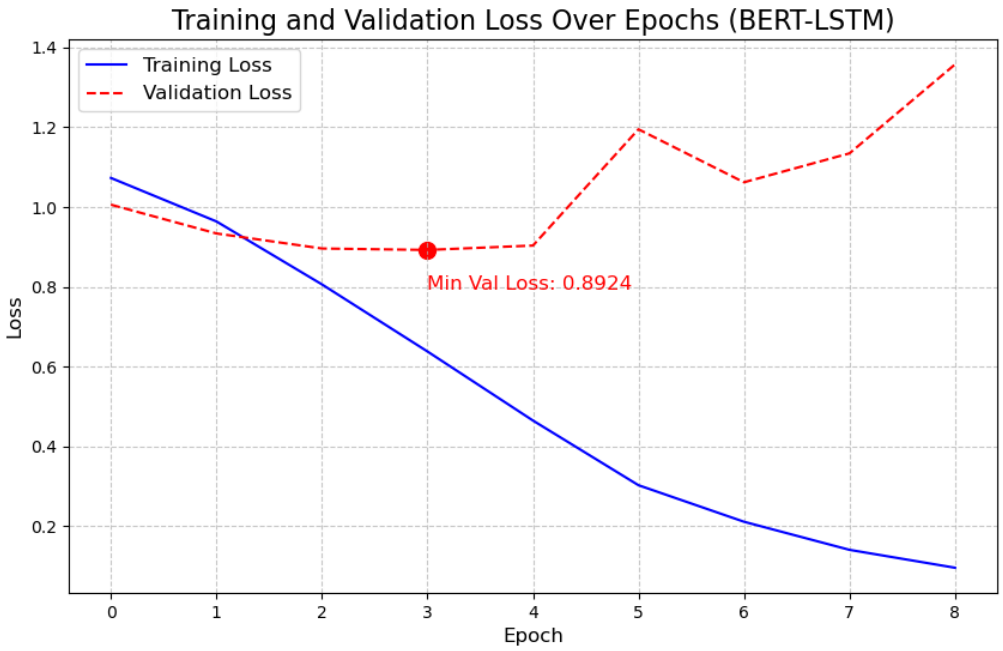
Lastly, we explore the results of the BERT-based models in the sentiment analysis task. Before looking at the confusion matrices and the evaluation metrics, it is worth making some comments on their training and validation loss. As we can see from Figures 5.1.6 and 5.1.7, the train loss decays almost linearly over the epochs, while the validation one starts decreasing in the first few epochs and then it increases afterward.



(a) multiBERT CLS LOSS

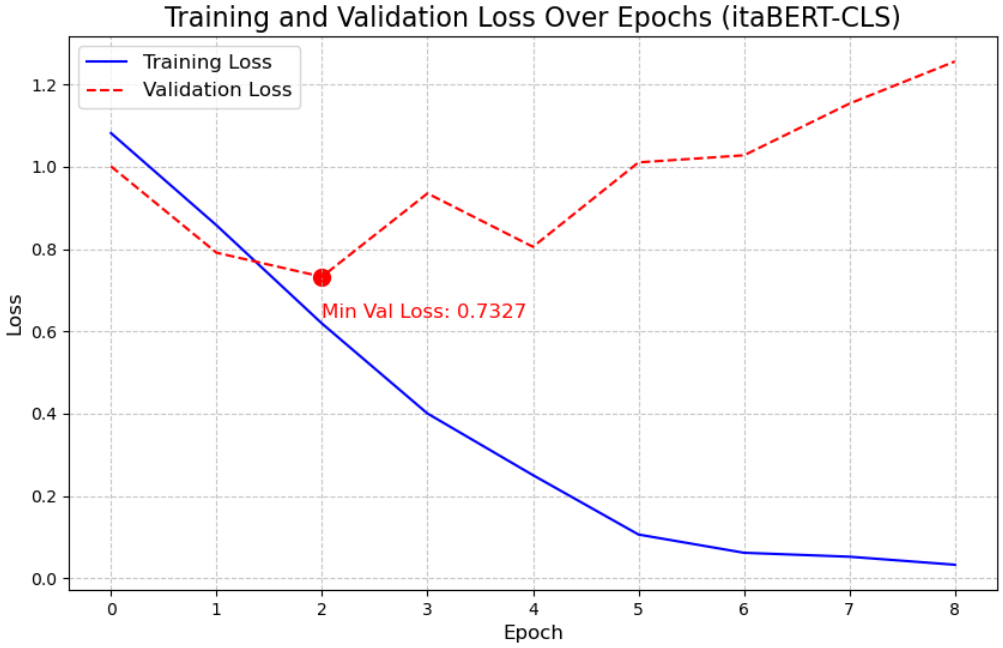


(b) multiBERT MAX LOSS

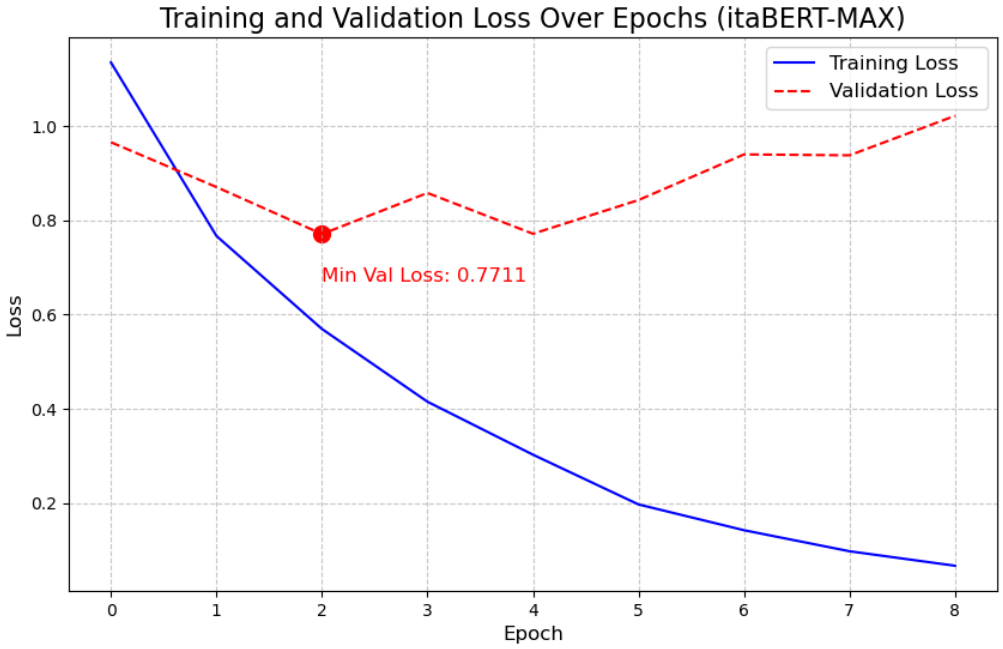


(c) multiBERT LSTM LOSS

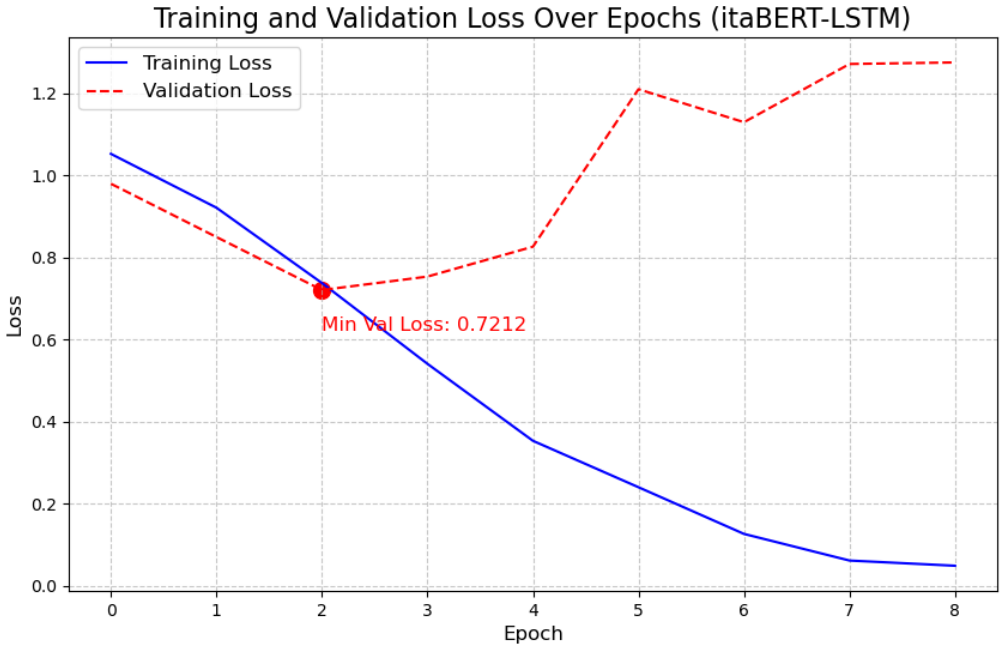
Figure 5.1.6: Training and Validation Loss multiBERT



(a) itaBERT CLS LOSS



(b) itaBERT MAX LOSS



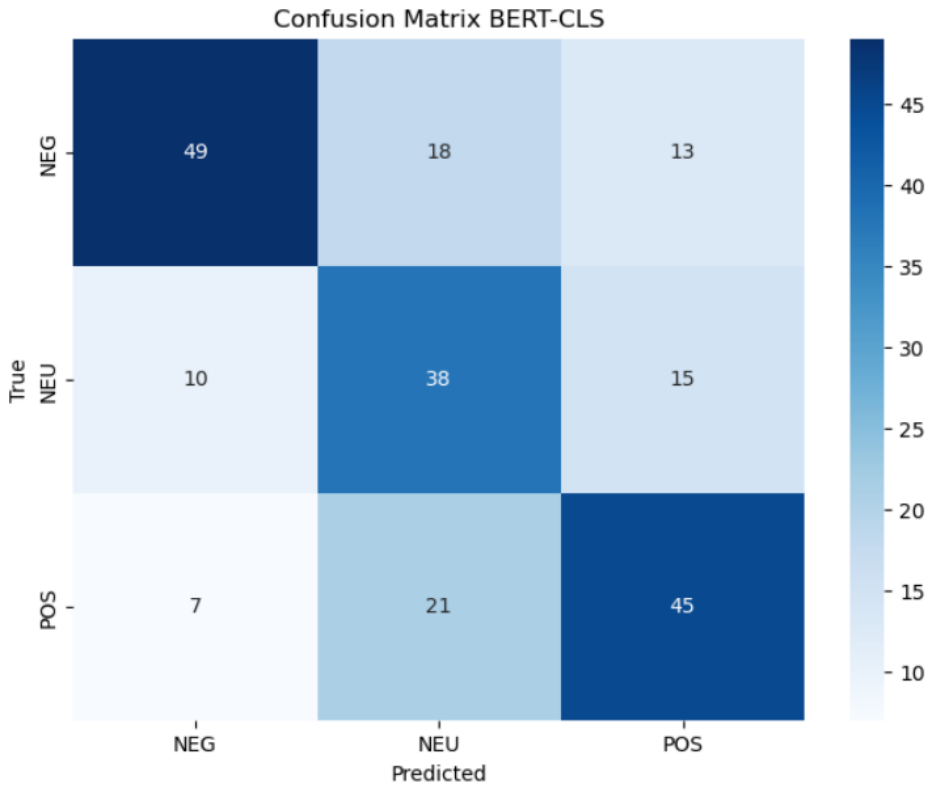
(c) itaBERT LSTM LOSS

Figure 5.1.7: Training and Validation Loss itaBERT

The best weights, those who minimize the validation loss, are obtained within the fourth epoch for the multilingual BERT, yielding a validation loss ranging in the interval [0.81, 0.90]. On the other hand,

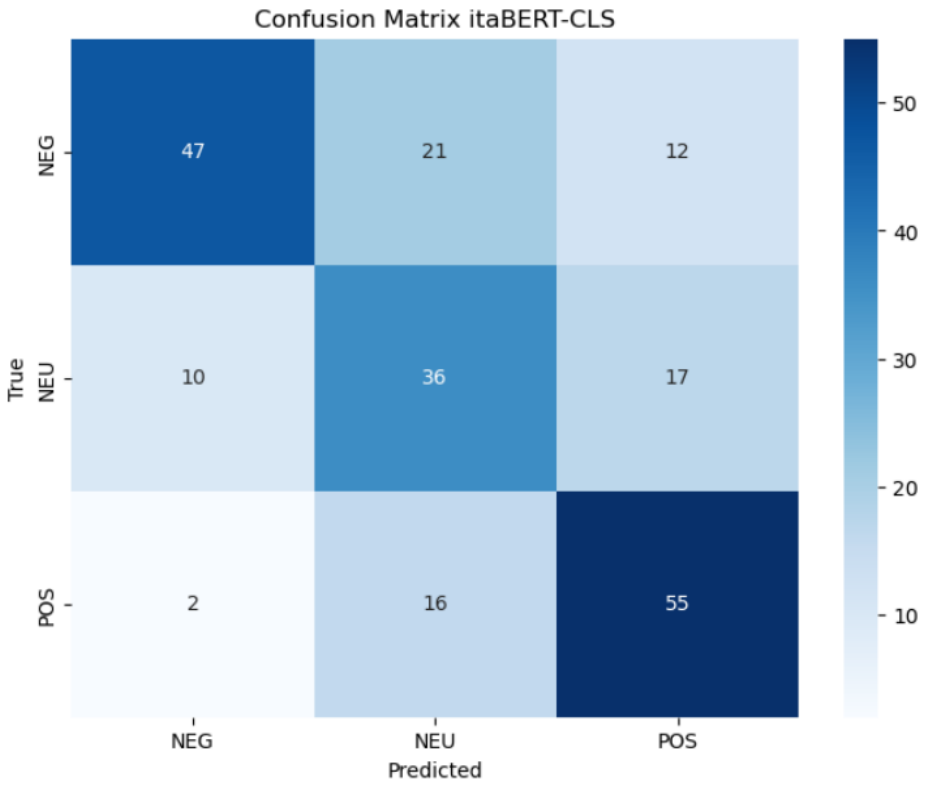
given that the italian BERT works better with an higher learning rate, the best weights are obtained in the third epoch, with a lower validation loss in the interval [0.72, 0.77].
As expected, the BERT models are the ones performing better in the sentiment analysis task. Each representation of the multilingual BERT achieve an accuracy of $\sim 60\%$, while the italian BERT reaches the best results with an overall performance of $\sim 65\%$.

The confusion matrices for the BERT-CLS models (Figures and Tables 5.1.8 and 5.1.9) exhibits similarities with the one produced by the SVM model using Bag of Words features, i.e they are well balanced. With respect to the SVM, the multiBERT-CLS demonstrates a remarkable improvement of approximately 10% in terms of F1-Score for both the positive and negative classes. The itaBERT-CLS is really close to the multilingual, with a notable improvement in the positive class.



Class	Precision	Recall	F1-Score	Support
NEG	0.74	0.61	0.67	80
NEU	0.49	0.60	0.54	63
POS	0.62	0.62	0.62	73
Accuracy			0.61	216
Macro avg	0.62	0.61	0.61	216
Weighted avg	0.63	0.61	0.62	216

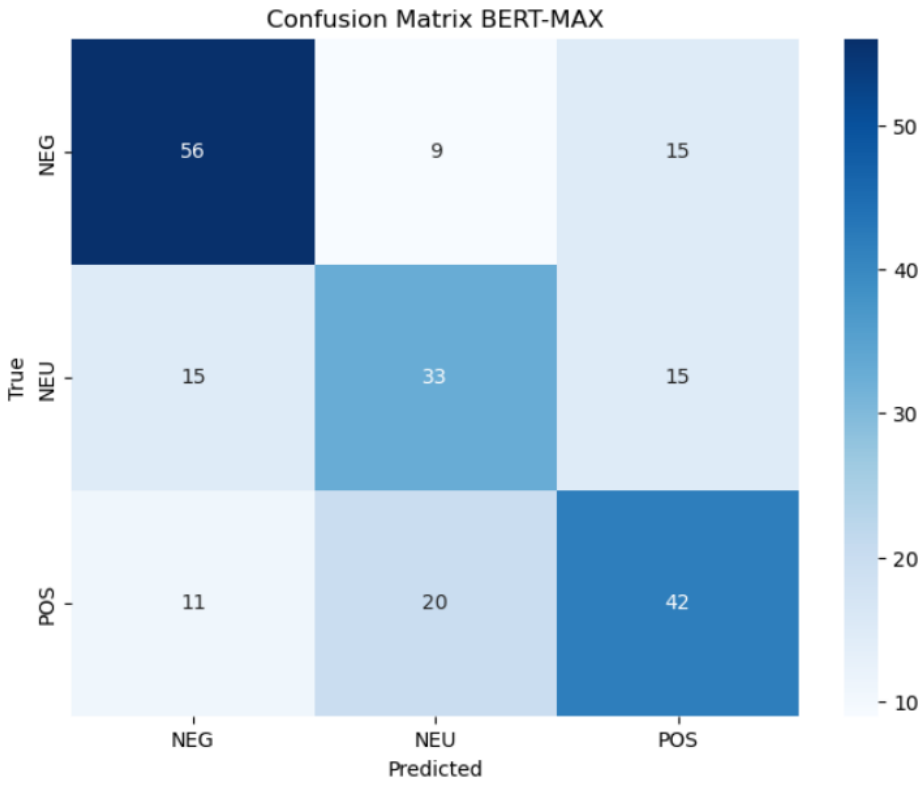
Figure 5.1.8: Confusion Matrix and Classification Report multiBERT-CLS



Class	Precision	Recall	F1-Score	Support
NEG	0.80	0.59	0.68	80
NEU	0.49	0.57	0.53	63
POS	0.65	0.75	0.70	73
Accuracy			0.64	216
Macro avg	0.65	0.64	0.64	216
Weighted avg	0.66	0.64	0.64	216

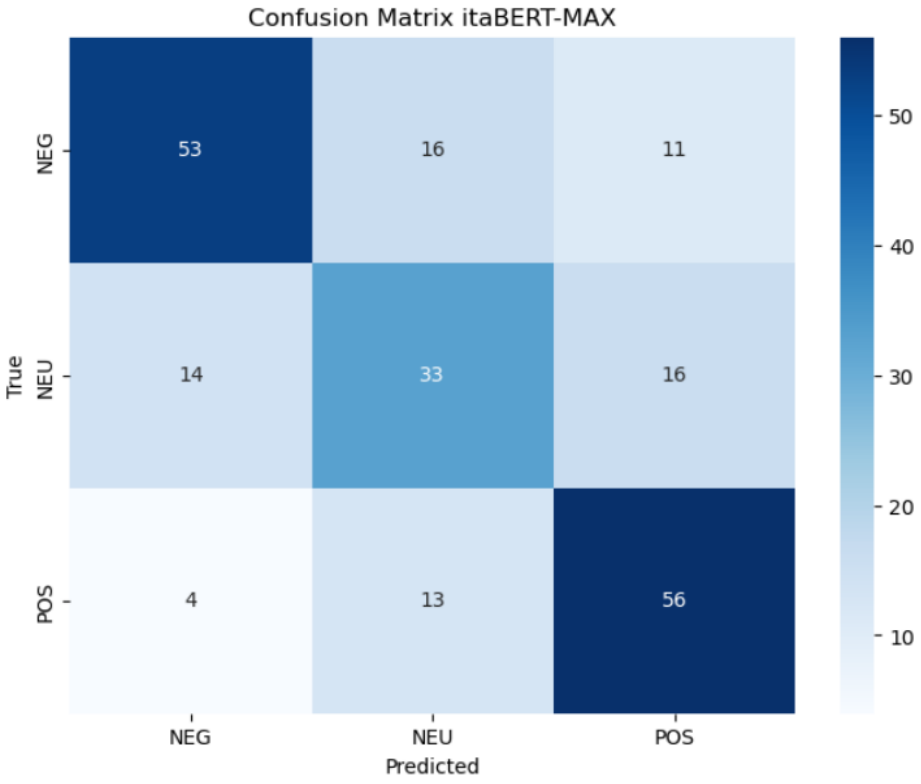
Figure 5.1.9: Confusion Matrix and Classification Report itaBERT-CLS

Shifting our focus to the BERT-MAX models (Figures and Tables 5.1.10 and 5.1.11), we observe that the differences in output representation do not affect much the results. For the multilingual variant, the negative class displays a marginal increase in accuracy at the cost of a reduced performance in the neutral and positive classes. For what concerns the italian one, it improves slightly both the negative and the positive sentiments.



Class	Precision	Recall	F1-Score	Support
NEG	0.68	0.70	0.69	80
NEU	0.53	0.52	0.53	63
POS	0.58	0.58	0.58	73
Accuracy			0.61	216
Macro avg	0.60	0.60	0.60	216
Weighted avg	0.61	0.61	0.61	216

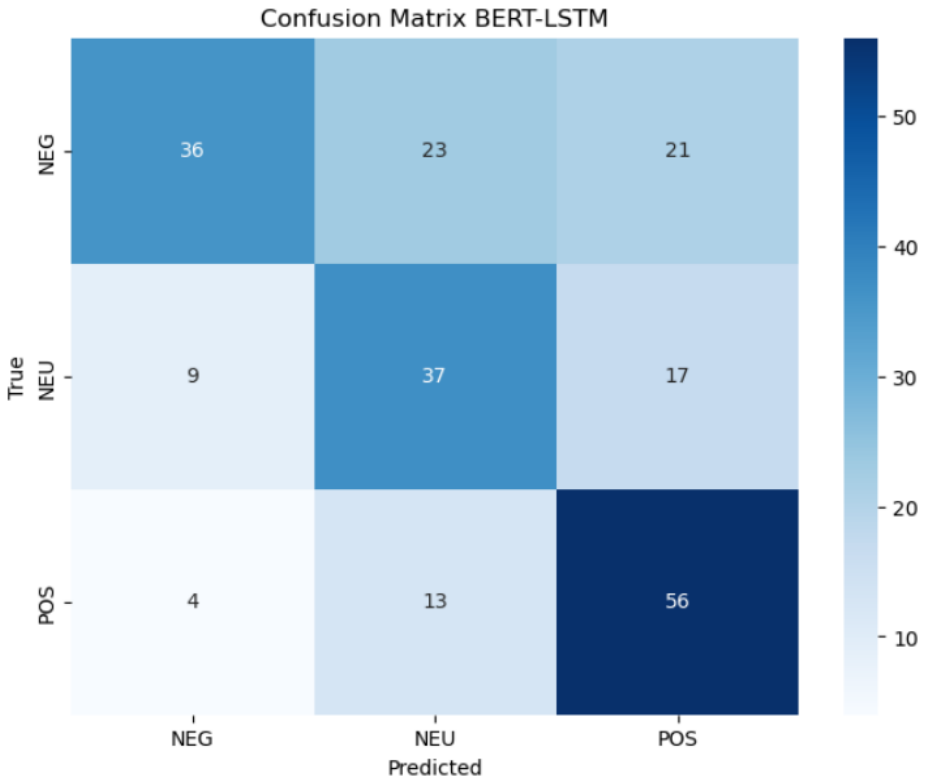
Figure 5.1.10: Confusion Matrix and Classification Report multiBERT-MAX



Class	Precision	Recall	F1-Score	Support
NEG	0.75	0.66	0.70	80
NEU	0.53	0.52	0.53	63
POS	0.67	0.77	0.72	73
Accuracy			0.66	216
Macro avg	0.65	0.65	0.65	216
Weighted avg	0.66	0.66	0.66	216

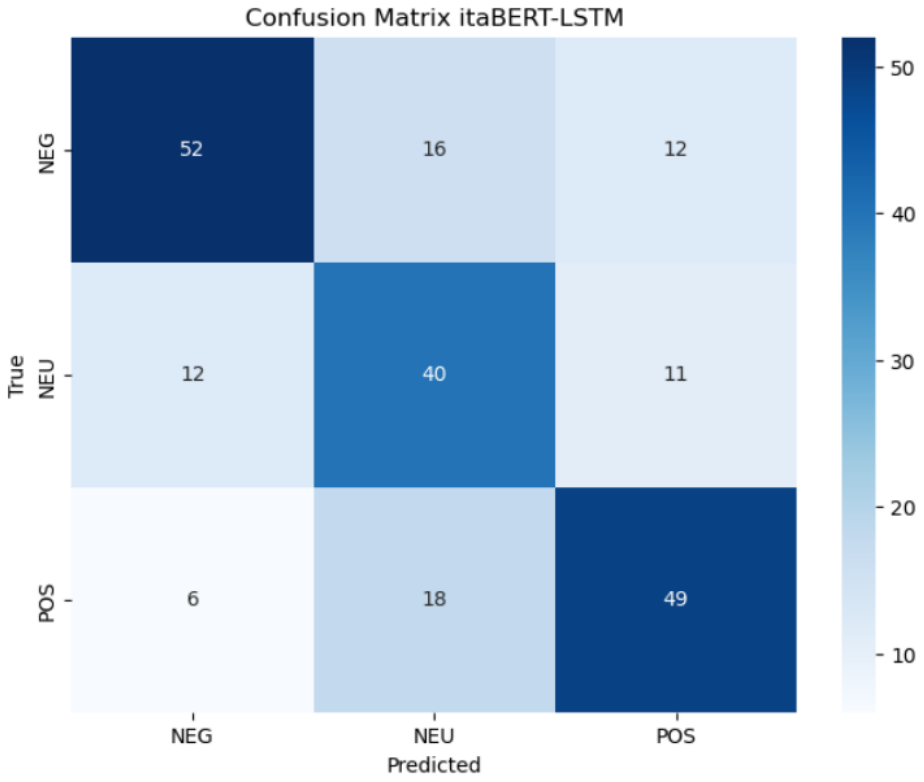
Figure 5.1.11: Confusion Matrix and Classification Report itaBERT-MAX

Moving on to the BERT-LSTM models (Figures and Tables 5.1.12 and 5.1.13), they still give consistent results, but they present some peculiarities. The multilingual variant is the first model in our analysis that appears to yield results with a positive bias. Surprisingly, it encounters greater challenges in classifying the negative class, as evidenced by an F1-Score decrease of approximately 12-15% with respect to the previous BERT models. Nonetheless, it still manages to maintain a respectable score in this regard. What stands out is its elevate F1-Score of 0.67 in the positive class compared to the other two multilingual representations. Finally, the italian BERT with LSTM output representation is the best performing in the neutral class, achieving an F1-Score of 0.58.



Class	Precision	Recall	F1-Score	Support
NEG	0.73	0.45	0.56	80
NEU	0.51	0.59	0.54	63
POS	0.60	0.77	0.67	73
Accuracy			0.60	216
Macro avg	0.61	0.60	0.59	216
Weighted avg	0.62	0.60	0.59	216

Figure 5.1.12: Confusion Matrix and Classification Report multiBERT-LSTM



Class	Precision	Recall	F1-Score	Support
NEG	0.74	0.65	0.69	80
NEU	0.54	0.63	0.58	63
POS	0.68	0.67	0.68	73
Accuracy			0.65	216
Macro avg	0.65	0.65	0.65	216
Weighted avg	0.66	0.65	0.66	216

Figure 5.1.13: Confusion Matrix and Classification Report itaBERT-LSTM

To summarize:

- The Lexicon-based model is the one yielding the worst results, in particular for the neutral class. However, it may be improved by finding better thresholds.
- Machine Learning models perform slightly better. The Bag of Words representation seems to give balanced results meanwhile TF-IDF is good at the extremes (negative and positive classes) but struggles in identifying the neutral tweets.
- The BERT models give the best results and the different output representations seem not to alter much the performance. There is a decent difference of $\sim 5\%$ in the performance between the multilingual and italian BERT.

Model Name	Accuracy	Macro Avg F1-Score	Weighted Avg F1-Score
VADER	0.49	0.44	0.46
Naive Bayes BoW	0.54	0.51	0.52
Naive Bayes TFIDF	0.53	0.47	0.49
SVM BoW	0.55	0.55	0.55
SVM TFIDF	0.55	0.51	0.52
multiBERT-CLS	0.61	0.61	0.62
multiBERT-MAX	0.61	0.60	0.61
multiBERT-LSTM	0.60	0.59	0.59
itaBERT-CLS	0.64	0.64	0.64
itaBERT-MAX	**0.66**	**0.65**	**0.66**
itaBERT-LSTM	0.65	**0.65**	**0.66**

Table 5.1.1: Sentiment Analysis Summary

5.2 Emotion Detection

Differently from the sentiment dataset, here we are confronted with an unbalanced dataset (as shown in Figure 2.1.2 and Table 2.1.8). Models trained on unbalanced datasets tend to develop a bias towards the majority class. In this case, the model may become highly accurate at predicting NEUTRA emotions but may perform poorly on the other emotions, especially AMORE, as it lacks sufficient examples to learn from. For this models we will not only look at the overall accuracy, but sometimes we will refer to the macro average F1-Score[1] and/or the weighted average F1-Score[2]. Nevertheless, the presence of many more training observations in the emotion dataset leads to an overall better performance for both the machine learning methods and the Transformers ones.

Naive Bayes and Support Vector Machines

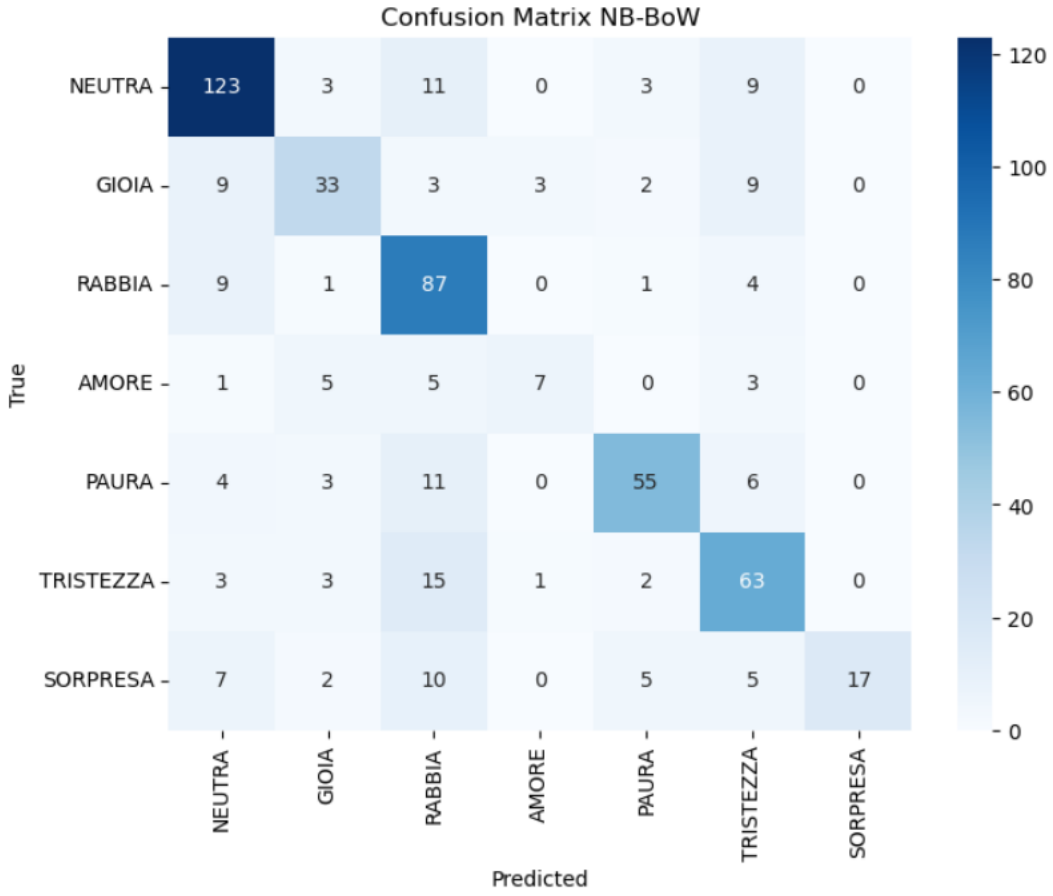
Given that for the emotion detection part we don't use any lexicon-based model, we start directly with the machine learnings algorithms.

The Naive Bayes with BoW fetures (Figure and Table 5.2.1) is almost balanced in all classes and performs especially well in the NEUTRA class, which is the most represented. It is interesting to observe that every tweet predicted as SORPRESA belongs indeed to that class, but this is not unexpected if we also look at the recall which is very low (0.37). Moreover, even though lots of tweets are predicted correctly to be in the class RABBIA, there is also a decent amount of false positives in that class, a behaviour not seen in other models. For this model we see that the overall accuracy (0.71) does not differ much from both the macro average F1-Score (0.65) and the weighted average F1-Score (0.70), confirming the

[1]The macro-averaged F1 score is computed by taking the arithmetic mean of all the per-class F1 scores.

[2]The weighted-averaged F1 score is calculated by taking the mean of all per-class F1 scores while considering each class's support.

decent generalization ability of the model.

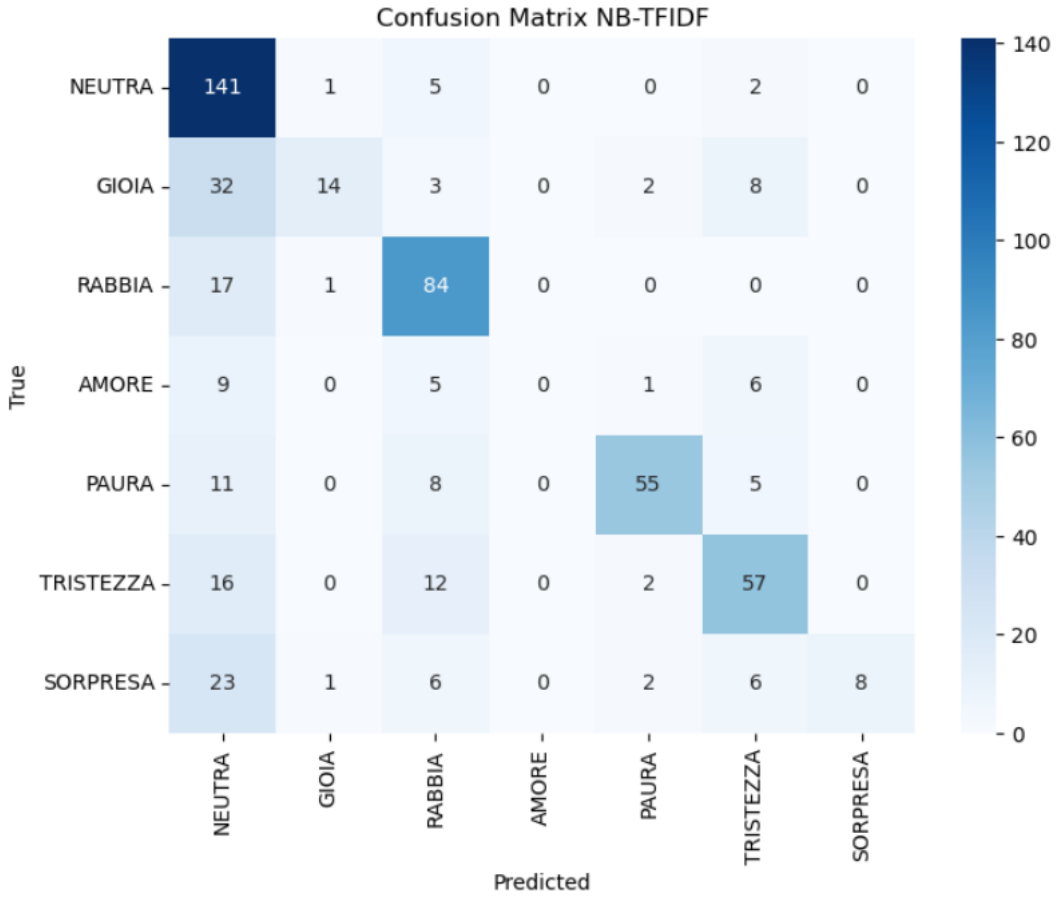


Class	Precision	Recall	F1-Score	Support
AMORE	0.64	0.33	0.44	21
GIOIA	0.66	0.56	0.61	59
NEUTRA	0.79	0.83	0.81	149
PAURA	0.81	0.70	0.75	79
RABBIA	0.61	0.85	0.71	102
SORPRESA	1.00	0.37	0.54	46
TRISTEZZA	0.64	0.72	0.68	87
Accuracy			0.71	543
Macro avg	0.73	0.62	0.65	543
Weighted avg	0.73	0.71	0.70	543

Figure 5.2.1: Confusion Matrix and Classification Report NB-BoW

Turning to the Naive Bayes with TF-IDF features, we get worse results because the majority of the tweets are predicted to be NEUTRA. If we compare the model's recall in that class with respect to the previous using BoW features, we see a huge difference (0.95 vs 0.83). The recall for the least represented class

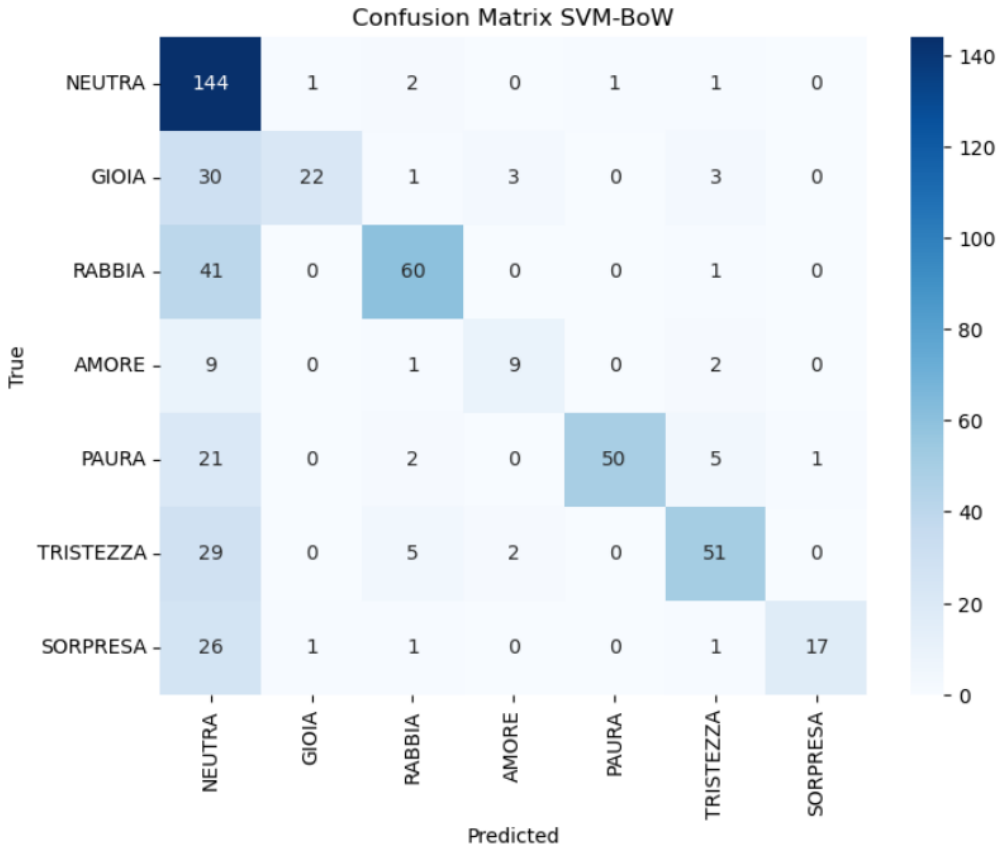
such as AMORE, GIOIA and SORPRESA is very low, indicating the model to be inefficient for these classes. An important analytics to highlight is that none of the 543 tweets has been classified as belonging to the class AMORE. Looking closely at the classification report (Table 5.2.2), the macro average F1-Score is $\sim$ 15% lower than the accuracy (0.66), confirming the poorness of this model in some classes.



Class	Precision	Recall	F1-Score	Support
AMORE	0.00	0.00	0.00	21
GIOIA	0.82	0.24	0.37	59
NEUTRA	0.57	0.95	0.71	149
PAURA	0.89	0.70	0.78	79
RABBIA	0.68	0.82	0.75	102
SORPRESA	1.00	0.17	0.30	46
TRISTEZZA	0.68	0.66	0.67	87
Accuracy			0.66	543
Macro avg	0.66	0.50	0.51	543
Weighted avg	0.70	0.66	0.62	543

Figure 5.2.2: Confusion Matrix and Classification Report NB-TFIDF

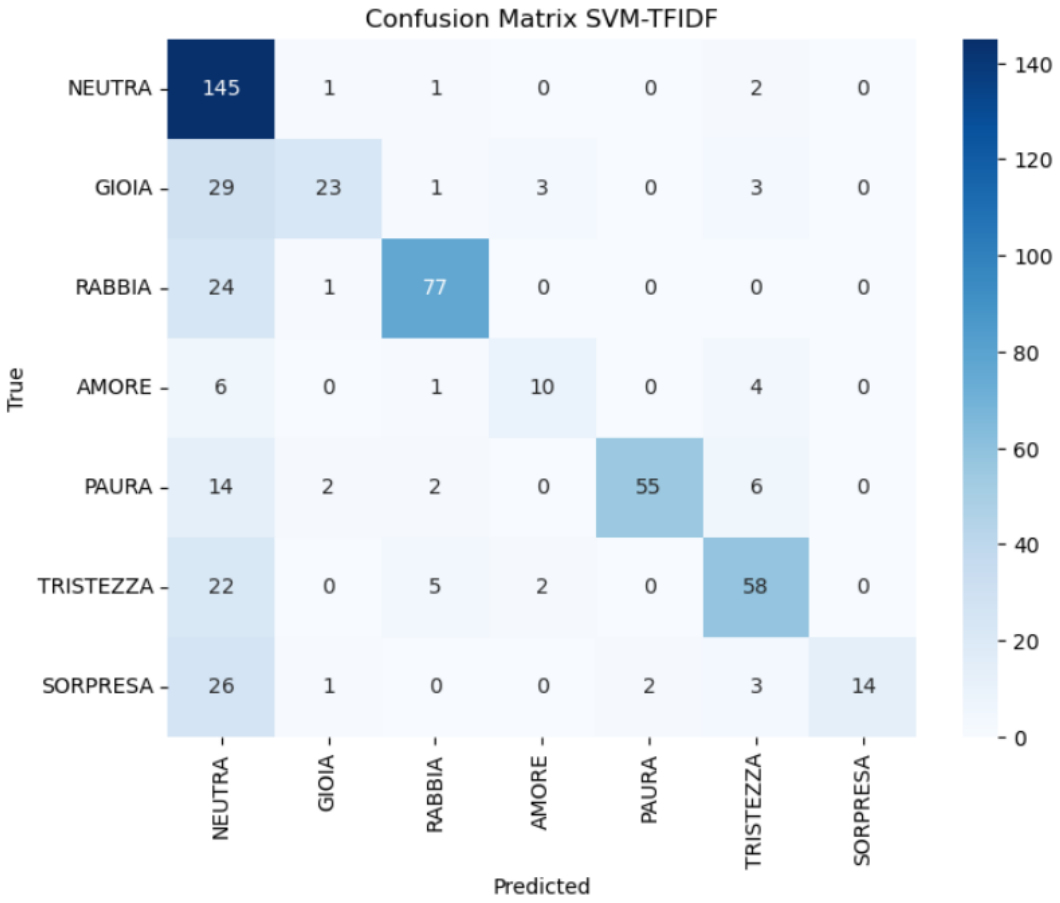
If we consider Support Vector Machines with BoW representation (Figure and Table 5.2.3) we get a similar confusion matrix as the one of Naive Bayes with TF-IDF feautres, but the problem with the NEUTRA class is even more evident: 300 out of 543 tweets were predicted in that class, when the actual number is 149. This fact leads to two consequences: the first is that the precision of the other classes is very high, since that if a tweet is not predicted as NEUTRA then it is very likely to belong in the predicted class. Second, for some classes there are more misclassified tweets than correct ones.



Class	Precision	Recall	F1-Score	Support
AMORE	0.64	0.43	0.51	21
GIOIA	0.92	0.37	0.53	59
NEUTRA	0.48	0.97	0.64	149
PAURA	0.98	0.63	0.77	79
RABBIA	0.83	0.59	0.69	102
SORPRESA	0.94	0.37	0.53	46
TRISTEZZA	0.80	0.59	0.68	87
Accuracy			0.65	543
Macro avg	0.80	0.56	0.62	543
Weighted avg	0.76	0.65	0.65	543

Figure 5.2.3: Confusion Matrix and Classification Report SVM-BoW

Things go better when we analyze Support Vector Machines with TF-IDF features (Figure and Table 5.2.4): the structure of the confusion matrix does not change from the previous model, but we have fewer misclassified tweets as belonging to NEUTRA and higher numbers on the main diagonal. It is worth noting that this model has the best results for the class RABBIA, yielding an F1-Score of 0.81.

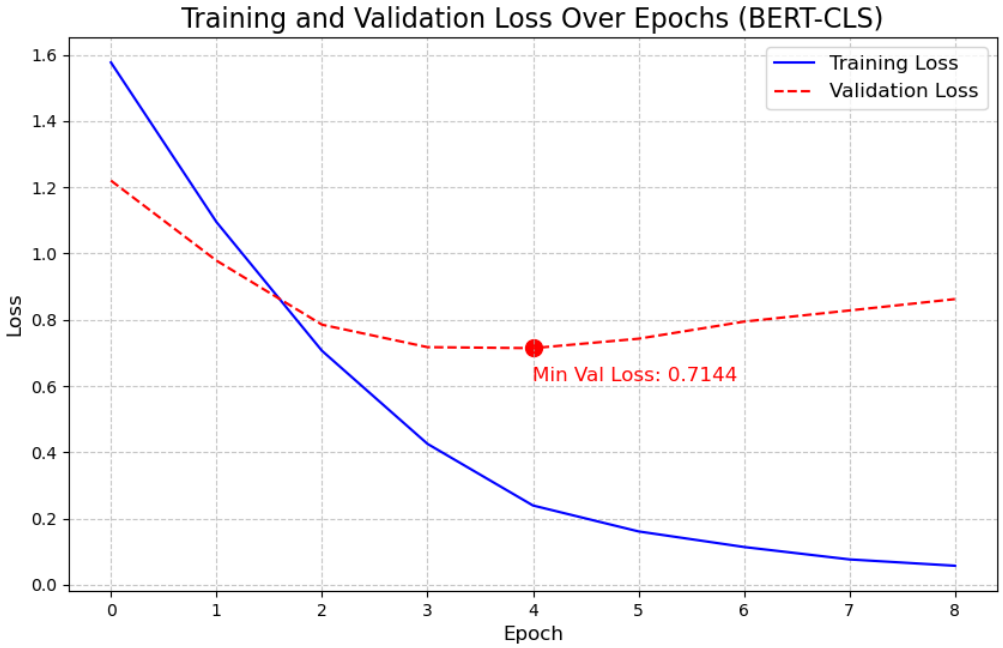


Class	Precision	Recall	F1-Score	Support
AMORE	0.67	0.48	0.56	21
GIOIA	0.82	0.39	0.53	59
NEUTRA	0.55	0.97	0.70	149
PAURA	0.96	0.70	0.81	79
RABBIA	0.89	0.75	0.81	102
SORPRESA	1.00	0.30	0.47	46
TRISTEZZA	0.76	0.67	0.71	87
Accuracy			0.70	543
Macro avg	0.81	0.61	0.66	543
Weighted avg	0.78	0.70	0.69	543

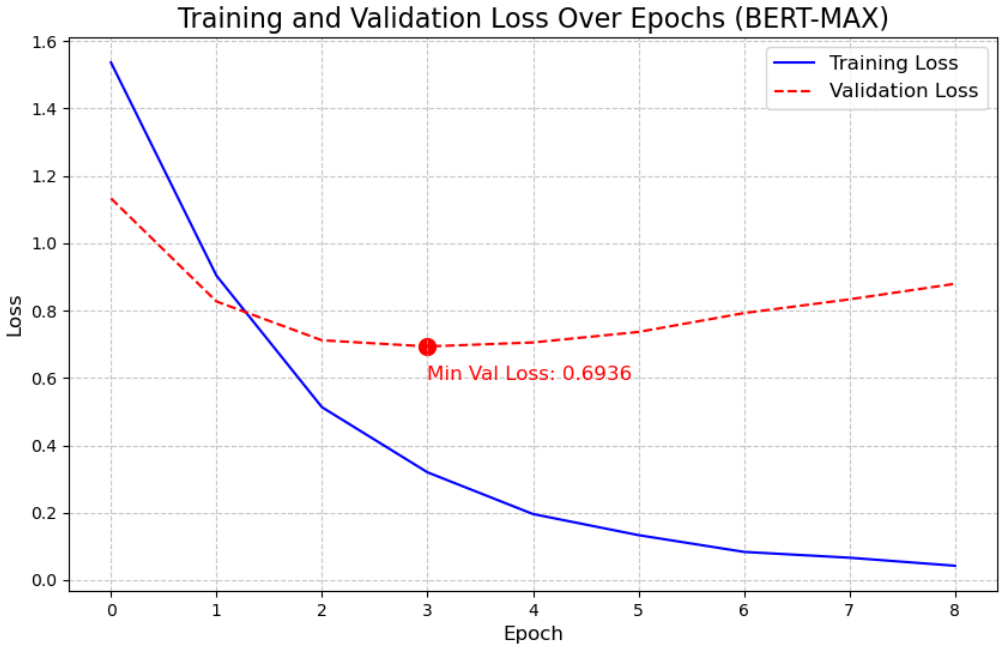
Figure 5.2.4: Confusion Matrix and Classification Report SVM-TFIDF

BERT Models

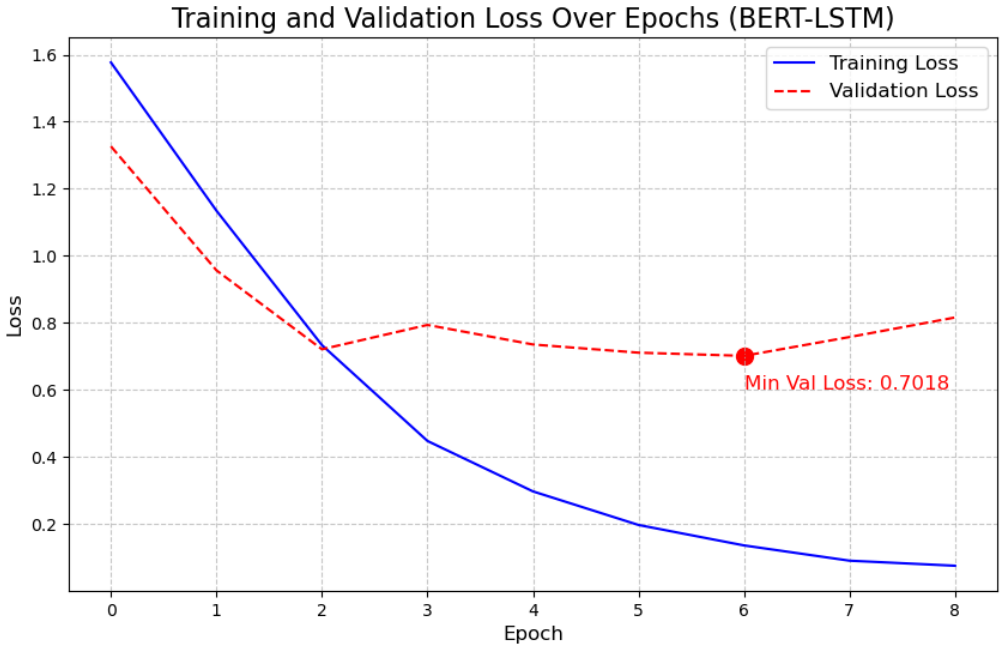
As we did for the sentiment analysis task, we start by exploring the training and validation loss of the BERT models during training. As we can see from Figures 5.2.5 and 5.2.6, the evolution of the training loss is a concave function, i.e it decreases faster in the first few epochs and then it becomes flatter. For the validation loss, the behaviour is similar to the one seen in the sentiment analysis, but the minimum value for the multilingual BERT-LSTM is only obtained after 7 epochs (even though that value is very close to the one obtained after the third).



(a) multiBERT CLS LOSS

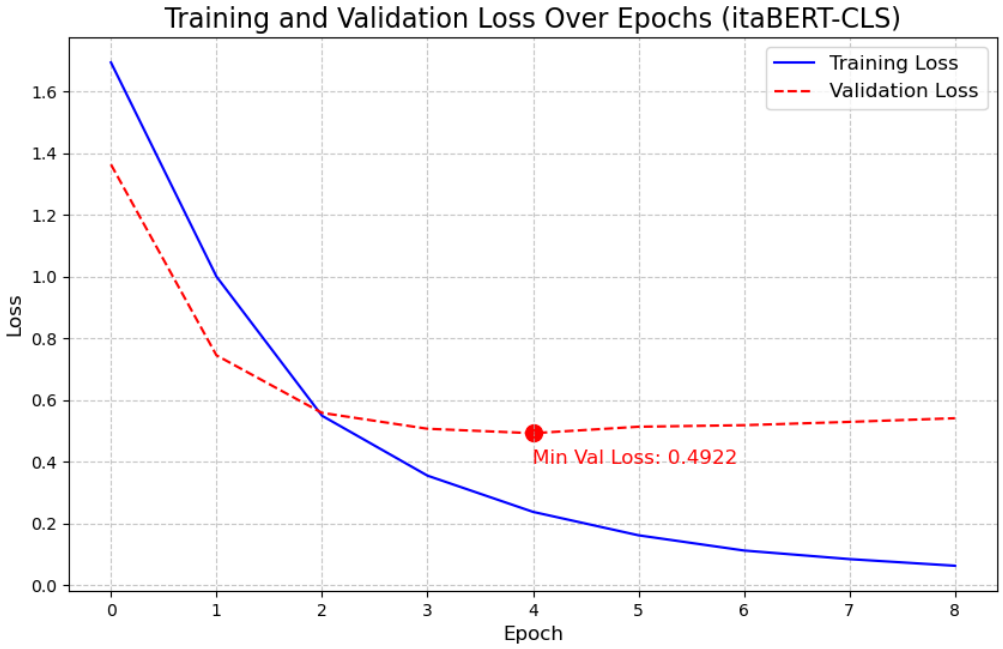


(b) multiBERT MAX LOSS

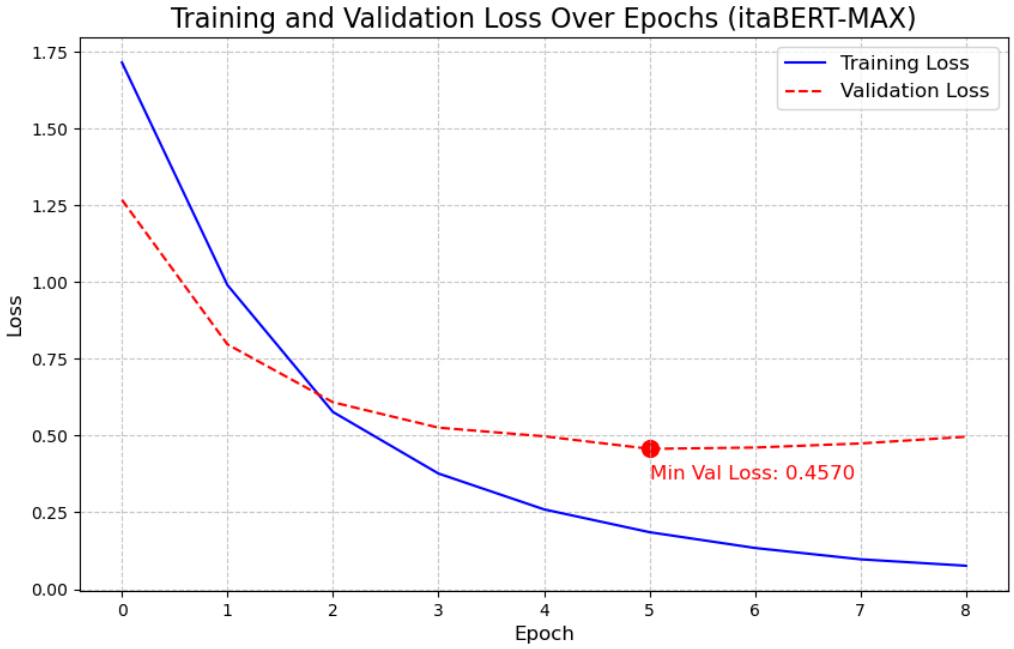


(c) multiBERT LSTM LOSS

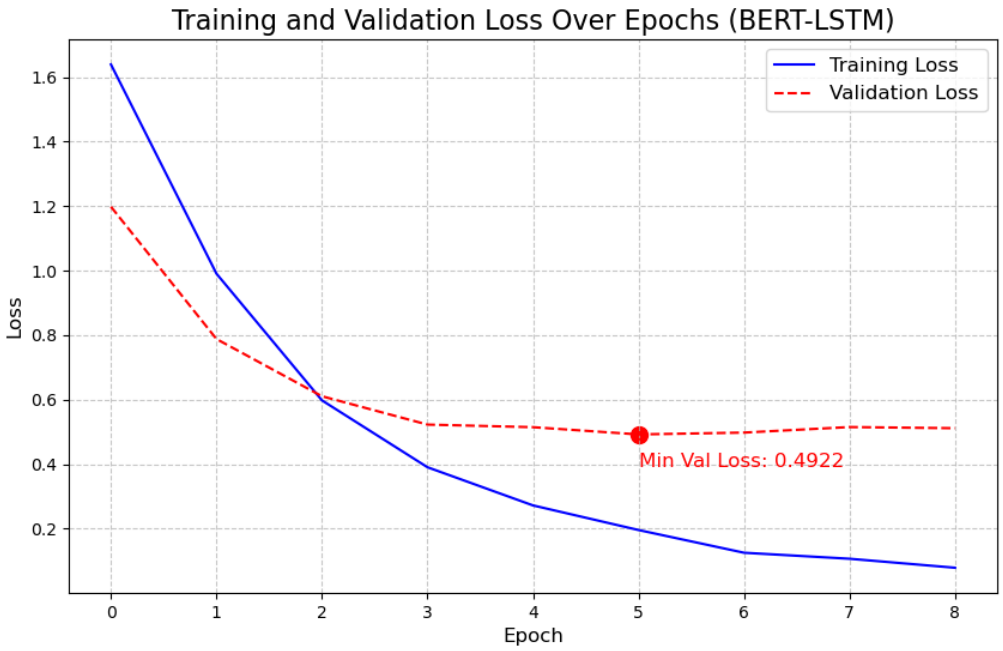
Figure 5.2.5: Training and Validation Loss multiBERT



(a) itaBERT CLS LOSS



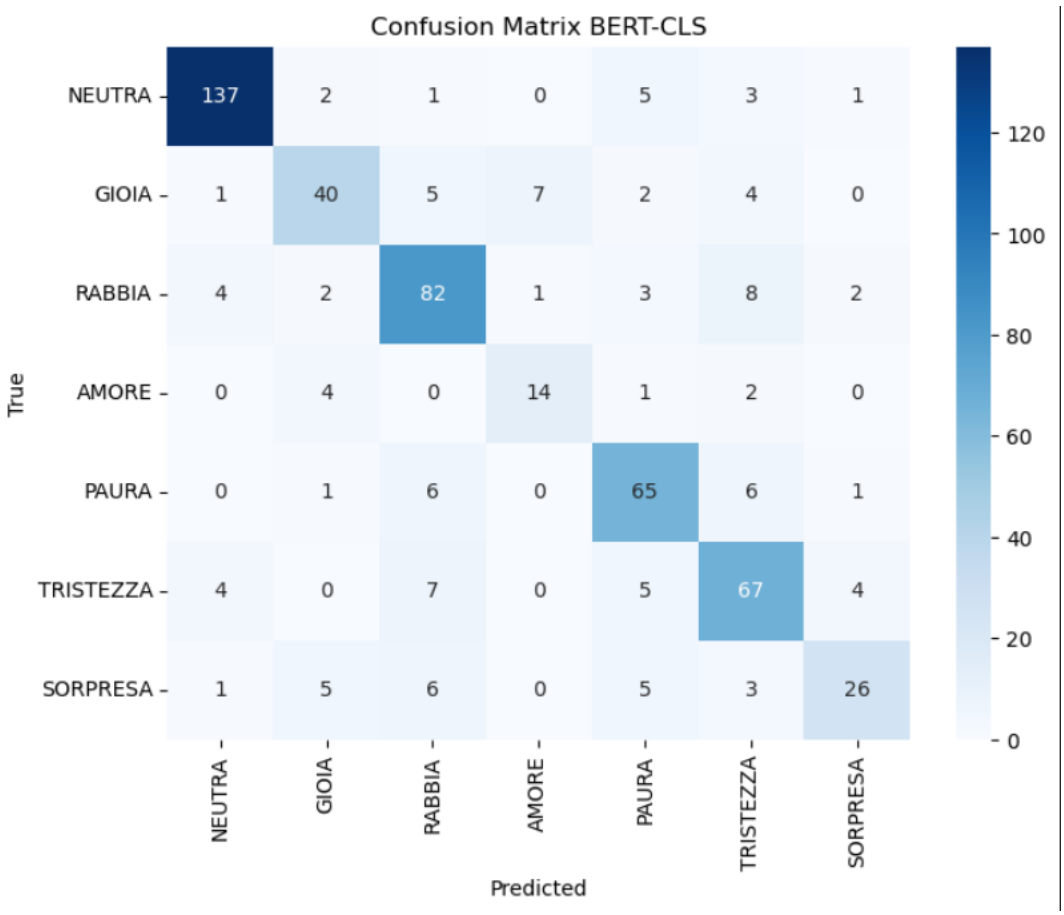
(b) itaBERT MAX LOSS



(c) itaBERT LSTM LOSS

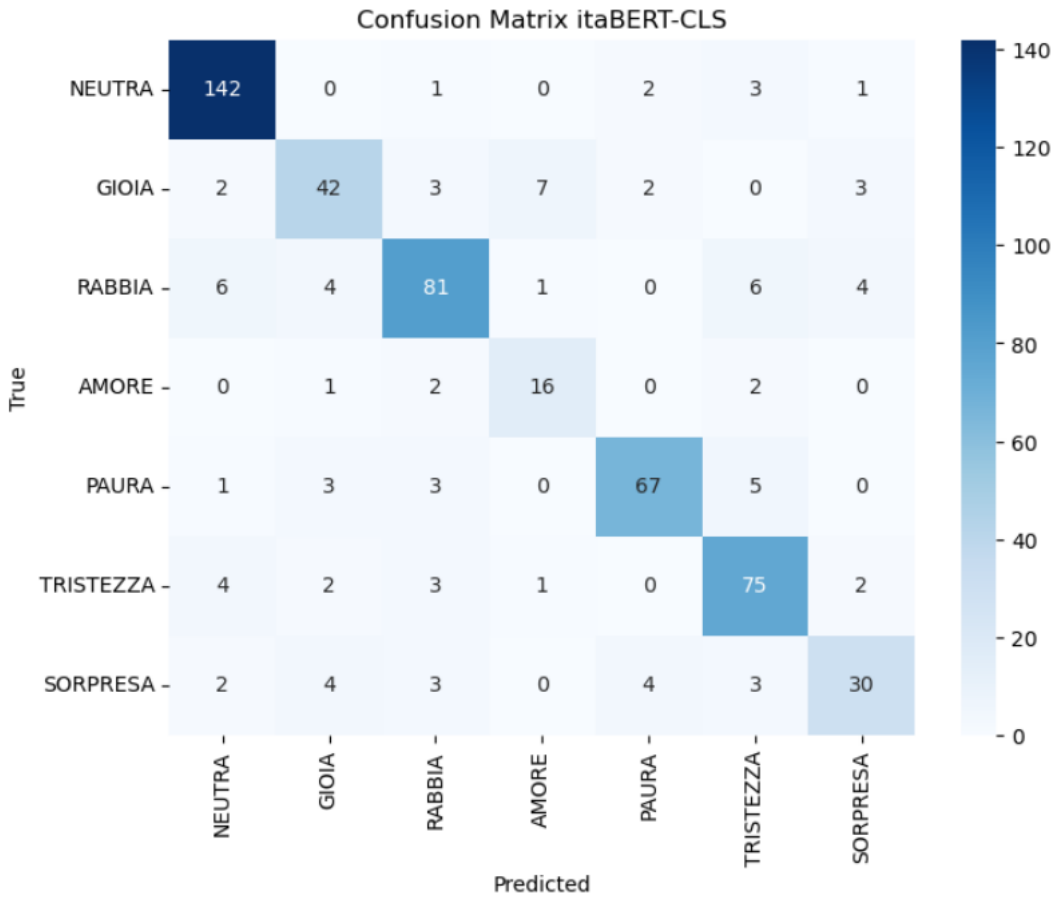
Figure 5.2.6: Training and Validation Loss itaBERT

In this framework the minimum value for the multilingual models falls in a smaller interval [0.69, 0.71], indicating not much difference on their performance on the validation set. If we focus on the italian ones, the validation loss lies in the interval [0.45, 0.49], suggesting again a better performance. By looking at the confusion matrices and the classification reports of the BERT models (from 5.2.7 to 5.2.12) we can see they give the excellent results, regardless the different output representations.



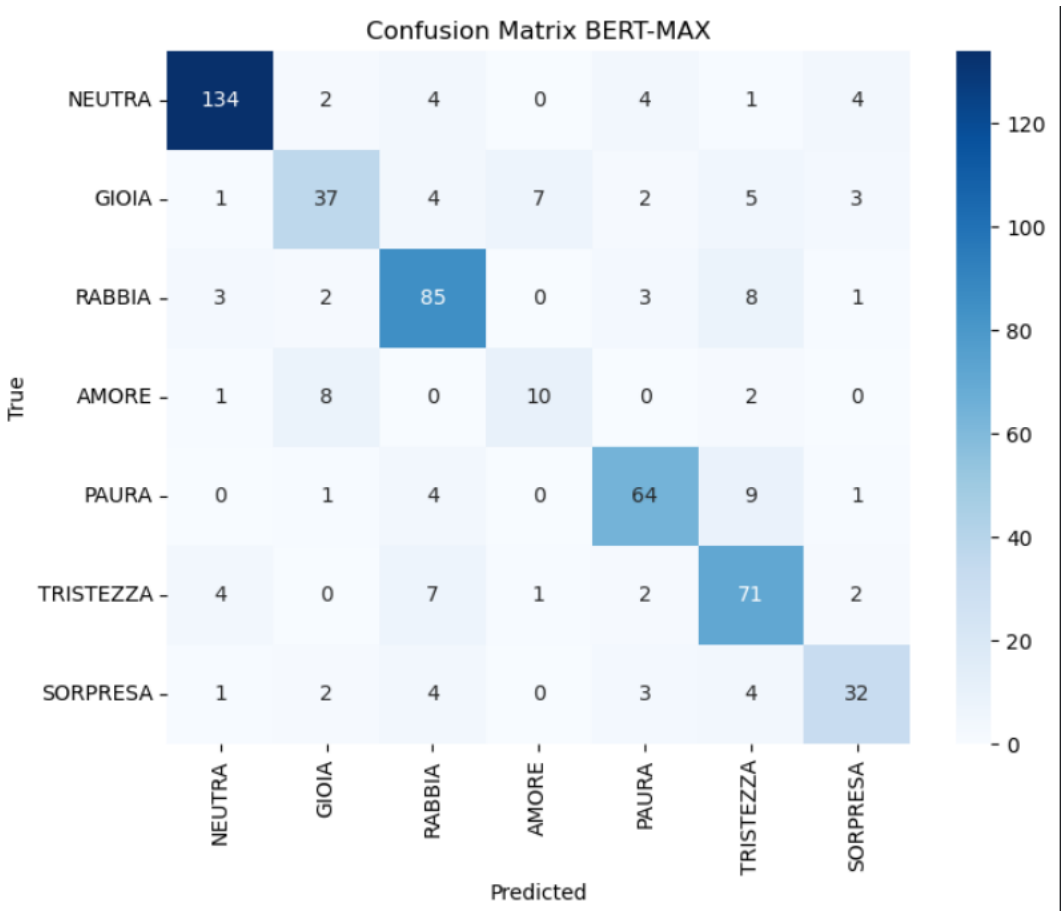
Class	Precision	Recall	F1-Score	Support
AMORE	0.64	0.67	0.65	21
GIOIA	0.74	0.68	0.71	59
NEUTRA	0.93	0.92	0.93	149
PAURA	0.76	0.82	0.79	79
RABBIA	0.77	0.80	0.78	102
SORPRESA	0.76	0.57	0.65	46
TRISTEZZA	0.72	0.77	0.74	87
Accuracy			0.79	543
Macro avg	0.76	0.75	0.75	543
Weighted avg	0.79	0.79	0.79	543

Figure 5.2.7: Confusion Matrix and Classification Report multiBERT-CLS



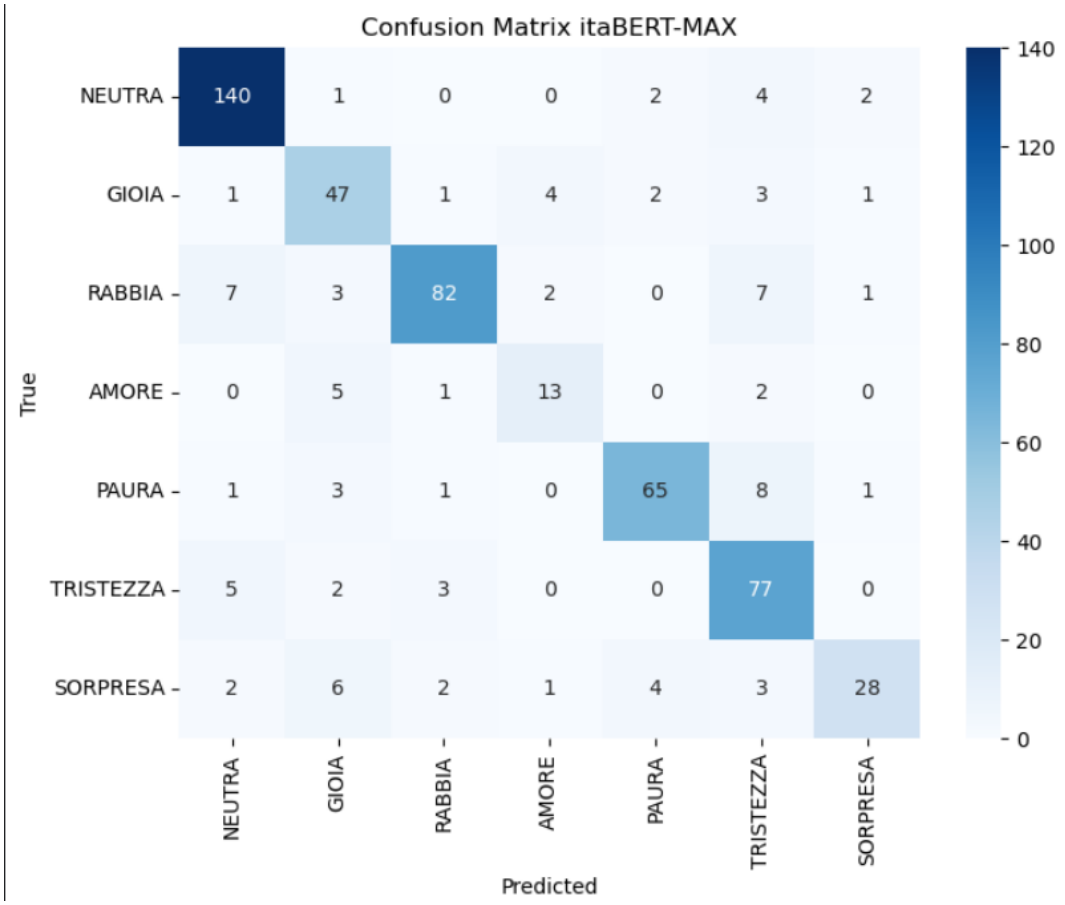
Class	Precision	Recall	F1-Score	Support
AMORE	0.64	0.76	0.70	21
GIOIA	0.75	0.71	0.73	59
NEUTRA	0.90	0.95	0.93	149
PAURA	0.89	0.85	0.87	79
RABBIA	0.84	0.79	0.82	102
SORPRESA	0.75	0.65	0.70	46
TRISTEZZA	0.80	0.86	0.83	87
Accuracy			0.83	543
Macro avg	0.80	0.80	0.80	543
Weighted avg	0.83	0.83	0.83	543

Figure 5.2.8: Confusion Matrix and Classification Report itaBERT-CLS



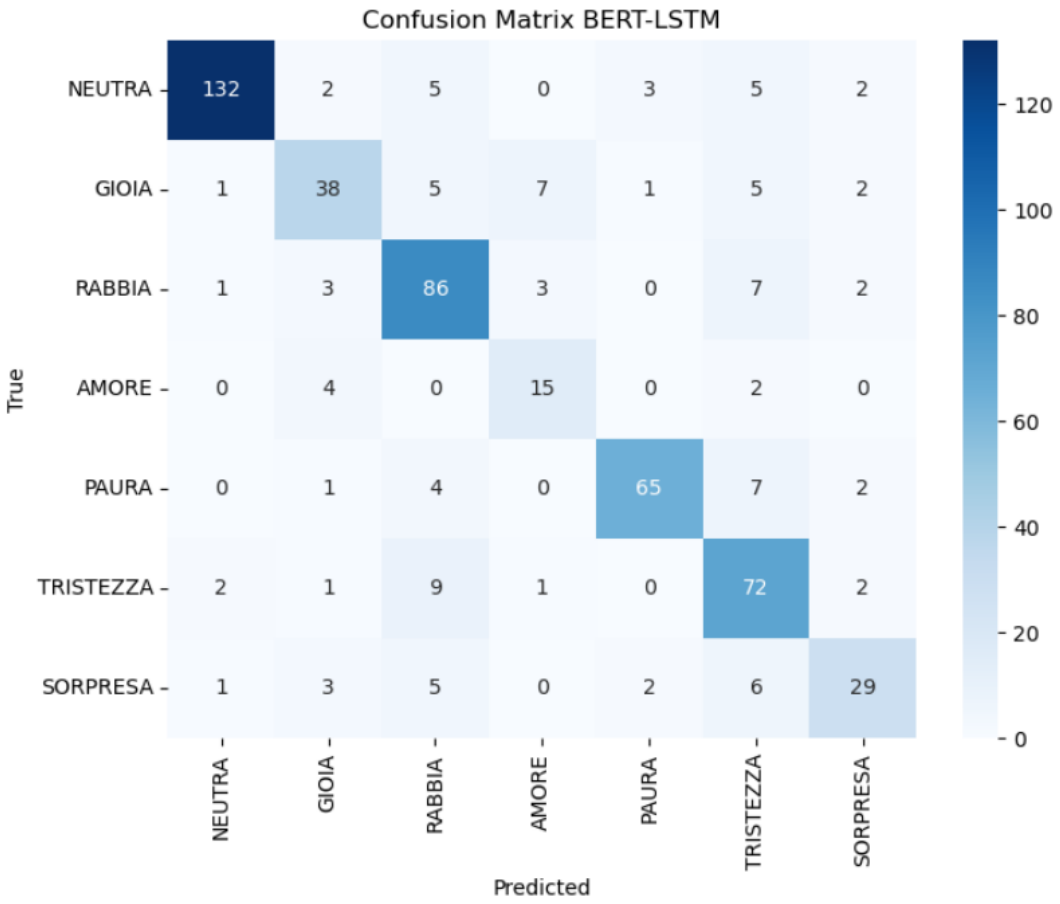
Class	Precision	Recall	F1-Score	Support
AMORE	0.58	0.71	0.64	21
GIOIA	0.73	0.64	0.68	59
NEUTRA	0.96	0.89	0.92	149
PAURA	0.92	0.82	0.87	79
RABBIA	0.75	0.84	0.80	102
SORPRESA	0.74	0.63	0.68	46
TRISTEZZA	0.69	0.83	0.75	87
Accuracy			0.80	543
Macro avg	0.77	0.77	0.76	543
Weighted avg	0.81	0.80	0.81	543

Figure 5.2.9: Confusion Matrix and Classification Report multiBERT-MAX



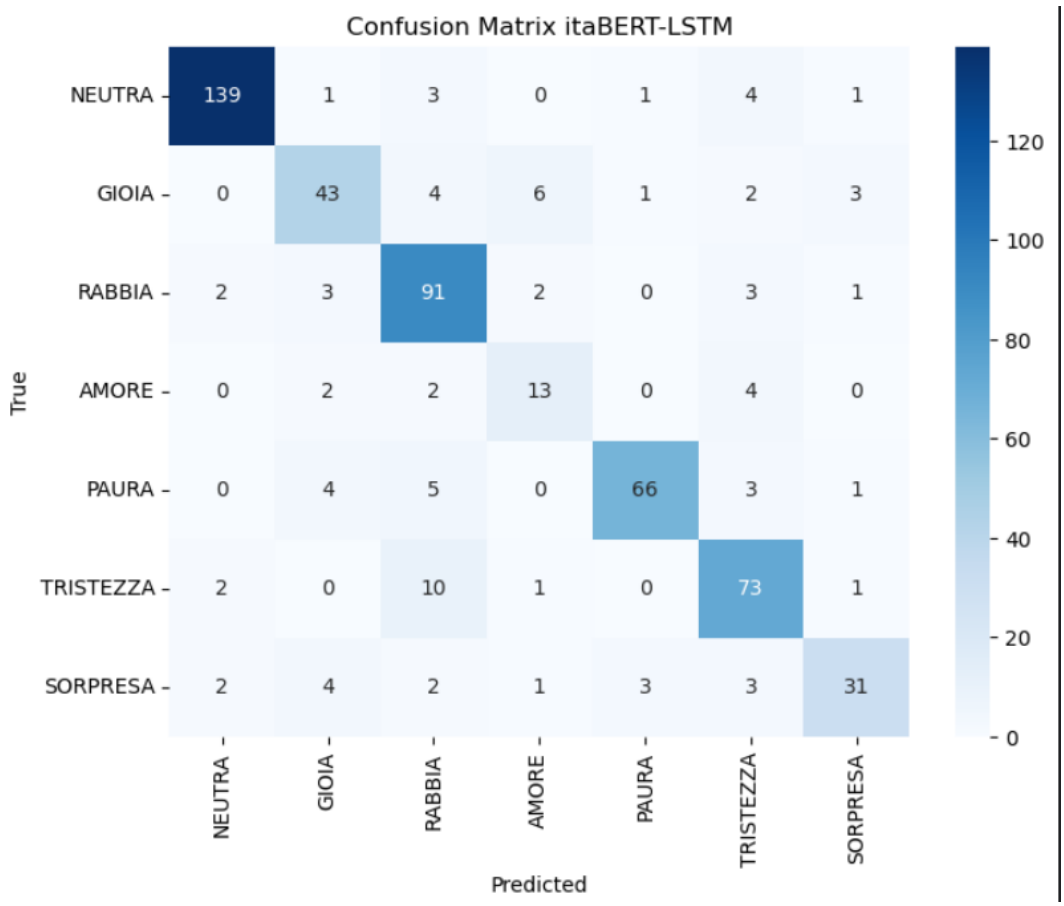
Class	Precision	Recall	F1-Score	Support
AMORE	0.65	0.62	0.63	21
GIOIA	0.70	0.80	0.75	59
NEUTRA	0.90	0.94	0.92	149
PAURA	0.89	0.82	0.86	79
RABBIA	0.91	0.80	0.85	102
SORPRESA	0.85	0.61	0.71	46
TRISTEZZA	0.74	0.89	0.81	87
Accuracy			0.83	543
Macro avg	0.81	0.78	0.79	543
Weighted avg	0.84	0.83	0.83	543

Figure 5.2.10: Confusion Matrix and Classification Report itaBERT-MAX



Class	Precision	Recall	F1-Score	Support
AMORE	0.58	0.71	0.64	21
GIOIA	0.73	0.64	0.68	59
NEUTRA	0.96	0.89	0.92	149
PAURA	0.92	0.82	0.87	79
RABBIA	0.75	0.84	0.80	102
SORPRESA	0.74	0.63	0.68	46
TRISTEZZA	0.69	0.83	0.75	87
Accuracy			0.80	543
Macro avg	0.77	0.77	0.76	543
Weighted avg	0.81	0.80	0.81	543

Figure 5.2.11: Confusion Matrix and Classification Report multiBERT-LSTM



Class	Precision	Recall	F1-Score	Support
AMORE	0.57	0.62	0.59	21
GIOIA	0.75	0.73	0.74	59
NEUTRA	0.96	0.93	0.95	149
PAURA	0.93	0.84	0.88	79
RABBIA	0.78	0.89	0.83	102
SORPRESA	0.82	0.67	0.74	46
TRISTEZZA	0.79	0.84	0.82	87
Accuracy			0.84	543
Macro avg	0.80	0.79	0.79	543
Weighted avg	0.84	0.84	0.84	543

Figure 5.2.12: Confusion Matrix and Classification Report itaBERT-LSTM

Each multilingual model has an overall accuracy of $\sim$ 80%, which is definitely higher than the one obtained using the machine learning models, which ranged from 65% to 70% depending on the algorithm and features. Moreover, the macro average F1-Score is only $\sim$ 4% lower than the accuracy and the weighted average is always in line.

The italian BERT models achieve the state-of-the-art results in this thesis, by reaching an overall accuracy to $\sim$ 84%. It is worth noting that the itaBERT-CLS is the most balanced, since each class has an F1-Score of at least 0.70, making it the models with the highest macro average F1-Score. Finally, the highest F1-Score reached is represented by the 0.95 of the NEUTRA class in the italian BERT-LSTM.

However, it is important to stress that it is not by chance that the best performing classes are the most represented ones: NEUTRA, RABBIA, PAURA and TRISTEZZA. Still, BERT models are able to improve the F1-Score of the least represented classes by $\sim$ 10-15% compared to the machine learning models.

To summarize:

- Machine Learning models are biased toward the most represented class (NEUTRA). Naive Bayes works generally better using BoW features and, in the other hand, Support Vector Machines with TF-IDF.
- As for sentiment analysis, BERT models give the state-of-the-art results and the different output representations seem not to alter much the performance. Among them, the italian variants confirm to be the best performing ones.

Model Name	**Accuracy**	**Macro Avg F1-Score**	**Weighted Avg F1-Score**
Naive Bayes BoW	0.71	0.65	0.70
Naive Bayes TFIDF	0.66	0.51	0.62
SVM BoW	0.65	0.62	0.65
SVM TFIDF	0.70	0.66	0.69
multiBERT-CLS	0.79	0.75	0.79
multiBERT-MAX	0.80	0.76	0.81
multiBERT-LSTM	0.80	0.76	0.81
itaBERT-CLS	0.83	**0.80**	0.83
itaBERT-MAX	0.83	0.79	0.83
itaBERT-LSTM	**0.84**	0.79	**0.84**

Table 5.2.1: Emotion Detection Summary

Chapter 6

Conclusions

The aim of this work was to carry out a comparative study on Italian tweets of different sentiment analysis and emotion detection approaches. We covered a variety of models and experimented different features and representations. While each model exhibited its unique strengths and weaknesses, the clear stand-outs were the Transformer-based models, emphasizing their robust performance across both sentiment analysis and emotion detection. Furthermore, the findings reinforced the critical role of data quantity and quality in pre-training Transformer models. The success of these models is intricately tied to the richness and diversity of the training data, showcasing the importance of well-curated datasets for optimal performance.

Future research in this field could explore the effectiveness of different Transformer-based models beyond BERT. For istance, RoBERTa relies on the same architecture of BERT but it modifies training objectives and hyperparameters, resulting competitive on various NLP tasks, including sentiment analysis.

Bibliography

[1] Rosario Catelli, Serena Pelosi, and Massimo Esposito. *Lexicon-Based vs. Bert-Based Sentiment Analysis: A Comparative Study in Italian*. Electronics, 2022.

[2] Bernardo Magnini, Alberto Lavelli, and Simone Magnolini. *Comparing Machine Learning and Deep Learning Approaches on NLP Tasks for the Italian Language*. European Language Resources Association, 2020.

[3] Luca Bacco, Andrea Cimino, Luca Paulon, Mario Merone, and Felice Dell'Orletta. *A Machine Learning approach for Sentiment Analysis for Italian Reviews in Healthcare*. 2020.

[4] Mayur Wankhade, Annavarapu Chandra Sekhara Rao, and Chaitanya Kulkarni. *A survey on sentiment analysis methods, applications, and challenges*. Artifcial Intelligence Review, 2022.

[5] Emma Haddia, Xiaohui Liua, and Yong Shib. *The Role of Text Pre-processing in Sentiment Analysis*. Procedia Computer Science, 2013.

[6] Ravinder Ahuja, Aakarsha Chug, Shruti Kohli, Shaurya Gupta, and Pratyush Ahuja. *The Impact of Features Extraction on the Sentiment Analysis*. Procedia Computer Science, 2019.

[7] Rishi Kumar. *Natural Language Processing. Feature Extraction Techniques.* Towards Data Science, 2021.

[8] Wisam A. Qader, Musa M.Ameen, and Bilal I. Ahmed. *An Overview of Bag of Words;Importance, Implementation, Applications, and Challenges*. Fifth International Engineering Conference on Developments in Civil Computer Engineering Applications, 2019.

[9] Juan Ramos. *Using TF-IDF to Determine Word Relevance in Document Queries*. 2003.

[10] Margherita Grandini, Enrico Bagli, and Giorgio Visani. *Metrics for Multi-Class Classification: an Overview*. 2020.

[11] Bex T. *Comprehensive Guide to Multiclass Classification Metrics*. Towards Data Science, 2021.

[12] C.J. Hutto and Eric Gilbert. *VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text*. Eighth International Conference on Weblogs and Social Media, 2014.

[13] Aditya Beri. *SENTIMENTAL ANALYSIS USING VADER*. Towards Data Science, 2020.

[14] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. *Sentiment analysis algorithms and applications: A survey*. Ain Shams Engineering Journal, 2014.

[15] Harry Zhang. *The Optimality of Naive Bayes*. The Florida AI Research Society, 2004.

[16] Daniel Jurafsky and James H. Martin. *Naive Bayes and Sentiment Classification*. Stanford University, 2023.

[17] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer Texts in Statistics. Springer, 2021.

[18] Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. *A comprehensive survey on support vector machine classification: Applications, challenges and trends*. Neurocomputing, 2019.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018.

[21] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. *Pre-trained models: Past, present and future*. AI Open, 2021.

[22] Rani Horev. *BERT Explained: State of the art language model for NLP*. Towards Data Science, 2018.

[23] Hugging Face. *The Hugging Face Course - NLP Course*. Hugging Face, 2022.

[24] *BERT Embeddings*. Tinkerd, 2023.

[25] *BERT Encoder Layer*. Tinkerd, 2023.

[26] Youwei Song, Jiahai Wang, Zhiwei Liang, Zhiyue Liu, and Tao Jiang. *Utilizing BERT Intermediate Layers for Aspect Based Sentiment Analysis and Natural Language Inference*. CoRR, 2020.

[27] Marco Pota, Mirko Ventura, Rosario Catelli, and Massimo Esposito. *An Effective BERT-Based Pipeline for Twitter Sentiment Analysis: A Case Study in Italian*. Sensors, 2020.

[28] rhtsingh. *Utilizing Transformer Representations Efficiently*. Kaggle, 2021.

[29] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. ICLR, 2019.