PM

# Sentiment Analysis of Restaurant Reviews in Portuguese
## A transfer learning and ensemble approach with edge computing

MASTER'S DEGREE PROJECT

**Alexandre João Jardim Branco**
MASTER IN ELECTRICAL ENGINEERING - TELECOMMUNICATIONS

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

February | 2024

# Sentiment Analysis of Restaurant Reviews in Portuguese

## A transfer learning and ensemble approach with edge computing

MASTER'S DEGREE PROJECT

**Alexandre João Jardim Branco**
MASTER IN ELECTRICAL ENGINEERING - TELECOMMUNICATIONS

SUPERVISOR
Fernando Manuel Rosmaninho Morgado Ferrão Dias

CO-SUPERVISOR
Fábio Ruben Silva Mendonça

UNIVERSIDADE da MADEIRA

Faculty of Exact Sciences and Engineering

**Master thesis in Electrical Engineering – Telecommunications**

# Sentiment Analysis of Restaurant Reviews in Portuguese: A Transfer Learning and Ensemble Approach with Edge Computing

**Alexandre João Jardim Branco**

Supervisor:
Fernando Manuel Rosmaninho Morgado Ferrão Dias
Co-Supervisor:
Fábio Ruben Silva Mendonça

Funchal – Portugal

February 5th, 2024

# Abstract

This research focuses on a case of applying transfer learning and transformer-based pre-trained models to sentiment analysis in Portuguese in restaurant reviews. Specifically, we employ BERT and RoBERTa, two strong Language Models that fit into limited computational resources, like edge computing, to build a sentiment review classifier. The classifier's performance is evaluated using accuracy and AUC ROC as the primary metrics. Our results demonstrate that the classifier developed using ensemble techniques outperforms the baseline model in accurately classifying restaurant reviews. This research contributes to sentiment analysis by exploring the effectiveness of transfer learning and transformer-based models in the context of Portuguese restaurant reviews.

This work highlights the importance of considering the Portuguese language in sentiment analysis tasks. Furthermore, this study investigates the deployment of the model on edge computing platforms, making sentiment analysis more accessible in resource-constrained environments. Combining deep learning techniques, transfer learning, and edge computing offers promising real-time sentiment analysis application opportunities. This research provides valuable insights for researchers and practitioners interested in sentiment analysis, natural language processing, and text analysis in the context of restaurant reviews.



**Keywords**: Sentiment analysis, natural language processing, Portuguese language, edge-computing, transfer-learning, transformers.

# Resumo

Este trabalho de investigação tem como foco melhorar a análise de sentimentos em avaliações de restaurantes, utilizando *transfer learning* e modelos pré-treinados baseados em *transformers*. Especificamente, foram aplicados o BERT e o RoBERTa, dois modelos de última geração, para construir um classificador de avaliações de sentimentos. O desempenho do classificador é avaliado utilizando *accuracy* e *AUC ROC* como principais métricas. Os resultados demonstram que o classificador desenvolvido utilizando técnicas de *ensemble* supera o modelo de referência na classificação precisa das avaliações de restaurantes. Este trabalho contribui para a análise de sentimentos, explorando a eficácia do *transfer learning* e de modelos baseados em *transformers* no contexto das avaliações de restaurantes em Português.

Este trabalho, destaca a importância de considerar a língua portuguesa em tarefas de análise de sentimentos. Além disso, este estudo investiga a implementação do modelo em plataformas de *edge computing*, tornando a análise de sentimentos mais acessível em ambientes com recursos limitados. A combinação de técnicas de *deep learning*, *transfer learning* e *edge computing* oferece oportunidades promissoras para aplicações de análise de sentimentos em tempo real. Este trabalho fornece indicações relevantes para investigadores e profissionais interessados em análise de sentimentos, processamento de linguagem natural e análise de texto no contexto de avaliações de restaurantes.

**Palavras-chave**: Análise de sentimento, processamento linguagem natural, língua Portuguesa, computação de borda, transferência de conhecimento, *transformers*.

# Acknowledgment

Thanks to my thesis supervisors, Professor Dr. Morgado Dias for his patience and assistance, and Professor Dr. Fábio Mendonça for his essential advice, and insightful suggestions that greatly improved this work.

I am deeply grateful to all the professors, including Eng. Filipe, who guided me during the Master of Electrical Engineering - Telecommunications program, generously sharing their knowledge and motivating me and challenging me to grow as a student and individual.

Special thanks to my wife, Sofia, whose understanding, tolerance, and unwavering support enabled me to successfully complete my studies. Throughout life's challenges, she has been my unwavering support, guiding me and standing consistently beside me during both joyous and difficult moments - truly, "you raise me up".

I would like to express my gratitude to my entire family, my parents, Jacinta and António, mainly my sister Marta and my brothers António and Miguel, for their firm support during difficult times.

To my dear classmates, especially Daniel, who shared this journey, offering constant support, encouragement, and assistance. I cherish the friendship and the valuable lessons learned from each of them, and they will always be lifelong friends.

This research received support from the project "RRSO - Restaurant Review Sentiment Output" under the scope of the project M1420-01-0247-FEDER-000055, under the Incentive System PROCiência 2020.

# Index

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| ACC | Accuracy |
| AdaBoost | Adaptive Boosting |
| AI | Artificial Intelligence |
| ANN | Artificial Neural Network |
| ATE | Aspect Term Extraction |
| BERT | Bidirectional Encoder Representations from Transformers |
| Bi-LSTM | Bidirectional Long-Short Term Memory |
| CLM | Casual Language Model |
| CM | Confusion Matrix |
| CNN | Convolutional Neural Networks |
| DL | Deep Learning |
| DLI | Deep Learning Institute |
| DNN | Deep Neural Networks |
| EL | Ensemble learning |
| ELMo | Embeddings from Language Models |
| FFNN | Feedforward Neural Network |
| FN | False Negative |
| FP | False Positive |
| FPGA | Field Programmable Gate Array |
| FPR | False Positive Rate |
| GD | Gradient Descent |
| GPT | Generative Pre-trained Transformers |
| GPU | Graphics Processing Unit |
| LSTM | Long-Short Term Memory |
| ML | Machine Learning |
| MLM | Masked Language Modeling |
| MSE | Mean Square Error |
| NLP | Natural Language Processing |
| NN | Neural Network |
| NSP | Next Sentence Prediction |
| OS | Operating System |
| OvA | One-vs-All |

| | |
|---|---|
| OvO | One-vs-One |
| OvR | One-vs-Rest |
| PE | Positional Encoding |
| PLC | Programmable Logic Controller |
| PPV | Positive Predictive Value |
| QA | Question Answering |
| ReLU | Rectified Linear Unit |
| RNN | Recurrent Neural Networks |
| RoBERTa | Robustly Optimized BERT Pretraining Approach |
| SA | Sentiment Analysis |
| SAMME | Stagewise Additive Modeling using Multiclass Exponential loss function |
| SDK | Software Development Kit |
| SOE | Sentiment Orientation Extraction |
| TL | Transfer Learning |
| TLM | Transformer Language Models |
| TN | True Negative |
| TNR | True Negative Rate |
| TP | True Positive |
| TPR | True Positive Rate |
| ULMFiT | Universal Language Model Finetuning |

# 1. Introduction

The concept of Machine Learning (ML) appeared in the 1950s as a branch of Artificial Intelligence (AI) focused on developing algorithms that enable computers to learn and improve their performance on specific tasks without being explicitly programmed. In the 1960s and 1970s, significant progress in developing ML algorithms was made, such as decision trees and Neural Networks (NN). In the 1980s, Natural Language Processing (NLP) began to gain shape as a subfield of AI and linguistics. Researchers focused on developing algorithms and models to analyze and understand natural language text, allowing computers to process and generate human language.

Machine Learning is a subfield of AI that enables computers to learn from data, identify patterns, and make decisions without being explicitly programmed. It involves algorithms and statistical models that analyze and learn from data to make predictions, decisions, or perform tasks. Over the next few decades, ML and NLP continued to evolve and advance, developing new algorithms and techniques and increased computational power and more significant amounts of data. ML and NLP are used in a wide variety of applications, including language translation and Sentiment Analysis (SA), as demonstrated in various scientific articles [1], [2], [3]. A significant benefit of ML algorithms is their capacity to manage vast data and make forecasts based on that information.

Sentiment analysis emerged as a subfield of NLP in the early 2000s, with the increasing availability of large amounts of online text data and the development of new ML techniques. SA is an area in active development in NLP and used in various applications, including customer service, marketing, and social media analysis based on extracting objectivity/subjectivity, polarity (positive, negative, and neutral), or more complex emotions (sadness, anger, happiness, and others) from natural language data.

Deep learning (DL) is a subset of ML that uses NN with multiple layers to learn from large amounts of data. In recent years, DL techniques have achieved remarkable success in various fields, including image classification, NLP, and speech recognition [4], [5].

Classification of sentiments in restaurant reviews has been an active area of research in NLP and SA [6], [7], [8]. The task involves determining the overall sentiment expressed in a written review, such as positive, negative, or neutral. However, these techniques require large amounts of annotated data and computationally expensive algorithms, which can limit their practical applicability in resource-constrained environments. With the increasing availability of customer feedback data online, automated SA techniques are applied to gain valuable insights into customer satisfaction and preferences. It can also provide valuable information for businesses to improve their products and services.

Additionally, it can help monitor a restaurant's reputation, identify the most common issues mentioned by customers, and determine overall customer satisfaction. Moreover, this task can also contribute to the advancement of NLP research by exploring the complexities of SA in real-world situations. There is a growing interest in developing efficient DL methods that can work effectively with less data and computational resources to address this challenge.

The specific technique or combination of techniques used in ML can vary depending on the type of problem being solved and the available data. The most common ones are supervised learning, where the algorithm is trained on a labeled dataset and makes predictions based on this training, unsupervised learning, where the algorithm learns from the data without any labels or pre-defined categories, semi-supervised learning, reinforcement learning, Transfer Learning (TL), and Ensemble Learning (EL).

Transfer learning, a technique that involves transferring knowledge from one domain to another, has proven effective in improving the performance of NLP models. By leveraging the knowledge learned from large-scale pre-trained models, such as Bidirectional Encoder Representations from Transformers (BERT) [1], Robustly Optimized BERT Pretraining Approach (RoBERTa) [2], and Generative Pretraining Transformer (GPT-4) [9], researchers have been able to achieve state-of-the-art results in various NLP tasks, including SA.

In the context of SA, TL could involve using a pre-trained DL model, such as a language model, to extract features from text and then using those features to train a separate model for sentiment classification. This technique can improve the sentiment classification model's performance by leveraging the pre-trained model's knowledge and expertise. These concepts can be further improved using ensemble learning, a ML technique that combines multiple individual models to produce a more accurate and robust prediction. The basic idea behind EL is that by combining the predictions of multiple models, it is possible to create a new model that is more robust and accurate than any of the individual models.

## 1.1. Problem Statement

The Restaurant Review Sentiment Output – RRSO project served as the foundation for this academic work focused on SA in Portuguese reviews, leveraging the data obtained from Zomato company. Zomato is an online food delivery and restaurant discovery platform. The project was undertaken to classify sentiments in Portuguese reviews from Zomato's restaurant dataset. The project lasted one year and aimed to develop a SA model trained specifically for Portuguese reviews. The team collected Portuguese reviews, cleaned, and structured the data to be used in algorithms to train the model. The model learned from the review patterns to accurately classify sentiments as positive, negative, or neutral. Throughout the year, the SA model was refined and optimized, experimenting with different techniques. The model's accuracy (ACC) in classifying sentiments measured the project's success. Once integrated into Zomato's platform, the model could enhance the user experience and help restaurants address feedback effectively.

The classification of sentiment in restaurant reviews in Portuguese involves determining the overall sentiment expressed in a written review. However, classifying sentiments based on text is challenging due to ambiguity, irony and sarcasm, cultural context, subjectivity, and language evolution. Furthermore, to overcome these challenges and enhance ACC, using pre-trained models becomes essential, allowing for the leverage of their benefits while ensuring they are lightweight enough for deployment on low-cost hardware.

Furthermore, the emergence of pre-trained models like BERT, which falls under the category of Transformer Language Models (TLM) and was introduced by Devlin et al. [1], offers the possibility of fine-tuning for various tasks, including SA or Question Answering (QA), through the application of Transfer Learning techniques [10]. It is worth noting that a limitation of BERT lies in its primary pre-training on English datasets. In response to this limitation, some researchers have endeavoured to adapt this model to languages other than English, with a particular emphasis on Portuguese variations, as is the focus of this thesis. Noteworthy efforts in this direction include BERTimbau [11] and Albertina [12]. The dearth of research pertaining to languages other than English has also been underscored in the work of Khurana et al. [10] and is identified as a potential avenue for future research.

Scholars have observed a conspicuous gap in research concerning languages beyond English, signifying a pertinent research frontier that has been addressed to some extent in prior

studies. The principal objective of this thesis is to construct models grounded in Transformer architectures and ensemble techniques for the task of SA. Certain investigations have already proposed model architectures capable of discerning substantial information from textual reviews. It is worth noting that the majority of studies featured in the State-of-the-Art section center on advancements in the Portuguese language, which is a focal point of this thesis. These studies exhibit considerable diversity in their chosen solutions, reflecting the relative underdevelopment of the Portuguese language within the field of NLP.

The lack of support for the Portuguese language in NLP is an issue that needs careful consideration. This limitation significantly restricts the available options in the field of NLP. Research focused on NLP models for SA primarily conducted in English has left a noticeable gap in our understanding of SA in Portuguese. This underscores the importance of extending NLP capabilities to include Portuguese, opening new avenues for research and application.

The challenge of incorporating edge computing into our research should not be overlooked. This issue presents a unique set of limitations and opportunities. In the realm of SA, the exploration of low-cost hardware inference by DL models requires us to leverage both generic and specialized single-board computers, as well as specialized processors. These hardware components play a crucial role in the efficient implementation of our SA model at the edge. By addressing the challenges and limitations in edge computing, we aim to enhance the practicality and accessibility of our approach. These platforms offer powerful computing capabilities at a fraction of the cost of traditional high-end specialized processors such as those used in cloud computing. Generic and specialized single-board computers are complete computers on a single board, widely used for their popularity and affordability. On the other hand, specialized processors are specifically designed to accelerate the performance of AI tasks.

## 1.2. Objectives

This research focuses on enhancing sentiment classification in restaurant reviews using DL techniques. The approach leverages TL with supervised learning on pre-trained BERT and RoBERTa models, using EL techniques and evaluating the overall performances using a cross-validation technique. The primary objective is to improve the ACC of sentiment classification in Portuguese. The research aims to develop a model to accurately classify Portuguese reviews as positive, negative, or neutral. With the ability to deploy our model on a range of low-cost hardware platforms, we believe our approach has the potential to facilitate access to real-time solutions, opening new opportunities in a variety of fields such as social media monitoring, market research, and customer feedback analysis, making it a valuable tool for real-time SA applications for researchers and practitioners in the field.

Building SA models from the ground up is a time-consuming endeavor, especially when considering the complexity and nuances of language, such as Portuguese. The process entails substantial data collection, annotation, model architecture design, and training, often spanning several months or even years. In contrast, leveraging transfer learning enables us to significantly compress the model development timeline. This expedited approach allows us to focus our efforts on the fine-tuning and optimization of pre-trained models for the specific nuances of Portuguese SA, ultimately delivering faster results without compromising accuracy or quality.

Transfer learning stands as a pivotal choice in our research methodology. Developing models from scratch demands an extensive amount of time, computational resources, and labeled data. In the context of SA, transfer learning allows us to leverage pre-existing models, like BERT, that have been trained on massive datasets in various languages, although

predominantly in English. This approach significantly accelerates the development process, as we can fine-tune these pre-trained models on Portuguese SA tasks, benefiting from the knowledge these models have already acquired. Thus, transfer learning serves as an efficient pathway to achieve State-of-the-Art results in SA while conserving valuable resources.

Ensemble techniques play a vital role in enhancing the robustness and accuracy of our SA models. Rather than relying on a single model's predictions, ensemble methods combine the outputs of multiple models, each with its unique strengths and weaknesses. By integrating these diverse perspectives, we aim to attain more reliable and comprehensive sentiment predictions. In essence, ensemble techniques serve as a risk mitigation strategy, reducing the potential for model bias or errors that could arise from a single model. This approach bolsters the overall performance of our system and ensures more dependable sentiment analysis outcomes.

It is worth noting that the landscape of SA in Portuguese remains relatively unexplored within the broader context of NLP. Previous work in this area has primarily concentrated on English SA, leaving a noticeable void in our understanding of sentiment expression in Portuguese. This deficiency in prior research underscores the urgency and significance of our work, as it seeks to fill a critical gap in NLP research by developing robust SA models tailored to the intricacies of the Portuguese language.

Therefore, this work aims to investigate and develop effective techniques for sentiment classification in Portuguese restaurant reviews and explore the potential of using TL and ensemble techniques to optimize the performance of DL models in resource-constrained environments.

Based on these objectives, the formulated research questions are:

1. How suitable are pre-trained DL models for analyzing NLP tasks in Portuguese?
2. Considering their computational limitations and resource constraints, are edge devices viable for performing NLP tasks?

## 1.3. Thesis Organization

Chapter 2 delves into essential concepts related to ML, transformers, and two techniques: TL and boosting.

In Chapter 3, the focus is on the field of SA within NLP. This chapter covers a wide range of topics, including data preprocessing, the models' architecture, hyperparameter optimization, evaluation of the models, and exploring Adaboost as an ensemble technique.

Chapter 4 showcases the practical application of the techniques discussed in the previous chapters.

Chapter 5 addresses the crucial aspects of inference and implementation of the models across various platforms.

Chapter 6 presents the comprehensive results.

The final chapter, Chapter 7, reflects on the accomplishments made throughout the study and provides insights into potential future developments and possibilities.

# 2. Artificial Neural Networks

Based on the knowledge of the human brain's capabilities and recognizing that there are tasks not easily performed by computers, several initiatives have emerged to reproduce brain functioning. The aim is to emulate its performance in crucial tasks, such as image analysis or identification of emotional states, which prove challenging for computers.

## 2.1. Basics of Neural Networks

Neural networks can be defined as a set of neurons linked together, similar to the human brain [13], with a structure as represented in Figure 2.1.

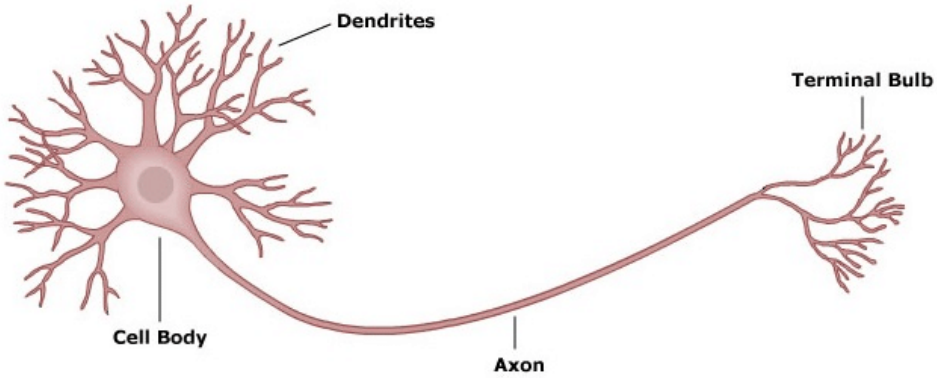



Figure 2.1 – Constitution of a neuron [14].

Neural networks exhibit two fundamental similarities to the human brain: knowledge acquisition occurs through learning from the environment, and synaptic weights, representing the connection strengths between neurons, store the acquired knowledge [13].

Neurons are considered the fundamental structural unit of the nervous system. Each neuron is made up of a cell body containing the nucleus as well as specialized prolongations called dendrites and axons. The neuron's body calculates the sum of the inputs affected by applying a nonlinear function to the signals it receives. Dendrites are responsible for receiving signals and information from other neurons, while axons (single nerve fibers) transmit the signals to other neurons. The point of contact between the axon of one neuron and the dendrite of another neuron is called the synapse, and the neuronal membrane delimits the cell body, separating the intracellular and extracellular medium. This complex structure of neurons allows communication and the transmission of information within the nervous system.

### 2.1.1. Artificial Neurons

Although the structural levels of organization are a unique feature of the brain, not being found anywhere in a digital computer, the Artificial Neural Network (ANN) were produced to follow the direction of a hierarchy similar to the one described, reproducing some brain-related concepts.

An artificial neuron is an information processing unit crucial for an ANN's operation. The block diagram in Figure 2.2 illustrates the model of an artificial neuron, which serves as the foundation for the design of ANN [13].

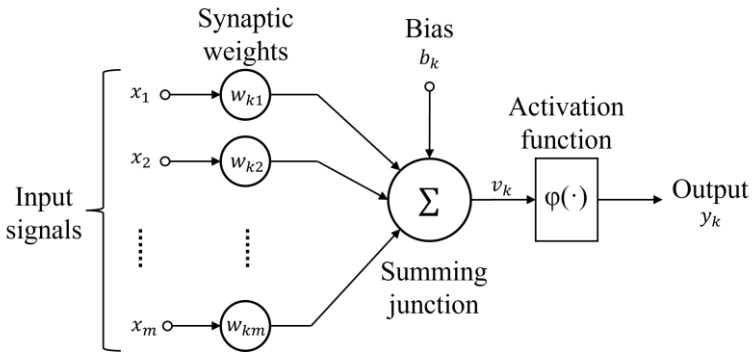



Figure 2.2 – Nonlinear model of an artificial neuron.

This diagram identifies three fundamental elements of the neuronal model: a set of synaptic connections, an adder, and an activation function.

Each of these connections has its weight. Upon receiving an $x$ signal, the $j$ synapse connected to artificial neuron $k$ performs a calculation by multiplying that signal by the corresponding synaptic weight $w$. The way the synaptic weight indices $w$ are written is essential: the first index refers to the artificial neuron in question, while the second refers to the input terminal of the specific synapse to which the weight is associated. One difference between the synapses of the brain and the synapses of artificial neurons is that the synaptic weight of the latter can vary in a range that includes both negative and positive values.

The purpose of the adder is to add the input signals, considering the weights of the synapses corresponding to the artificial neuron. These operations form what is known as a linear combiner.

The activation function is used to control the amplitude of the output of an artificial neuron. This function also limits the range allowed for the amplitude of the output signal to a finite value. Typically, the normalized range of a neuron's output amplitude is represented by the closed interval [0, 1], the unit interval, or the closed interval [-1, 1].

Take a Feedforward Neural Network (FFNN) as an example, as they are one of the most widely used ANNs that are constructed with sequences of layers with multiple artificial neurons in each layer [15]. The artificial neuron $k$ can be described in mathematical terms by:

$$v_k = \left( \sum_{j=1}^{m} (w_{kj} x_j) + b_k \right), \tag{2.1}$$

and

$$y_k = \varphi(v_k), \tag{2.2}$$

where the input signals are $x_1, x_2, \ldots, x_m$; the synaptic weights of the artificial neuron $k$ are $w_{k1}, w_{k2}, \ldots, w_{km}$; the output of the linear combiner resulting from the input signals is $v_k$; bias is $b_k$; the activation function is $\varphi(\cdot)$; and the output signal from the neuron is $y_k$.

Looking at equation 2.2, the output of an artificial neuron is defined by the activation function $\varphi(\cdot)$, where different activation functions can be applied to neurons, with the most commonly used ones being the sigmoid, hyperbolic tangent, Rectified Linear Unit (ReLU), identity, and softmax functions.

Equations 2.3, 2.4, 2.5, and 2.6 present the mathematical formula for the sigmoid, hyperbolic tangent, ReLU, and softmax functions, respectively [16],

$$\varphi(x) = \frac{1}{1 + e^{-x}}, \tag{2.3}$$

$$\varphi(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{2.4}$$

$$\varphi(x) = \begin{cases} 0, & if\ x < 0 \\ x, & if\ x \geq 0 \end{cases}, \tag{2.5}$$

$$\varphi(x) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}. \tag{2.6}$$

which are partially represented in Figure 2.3.

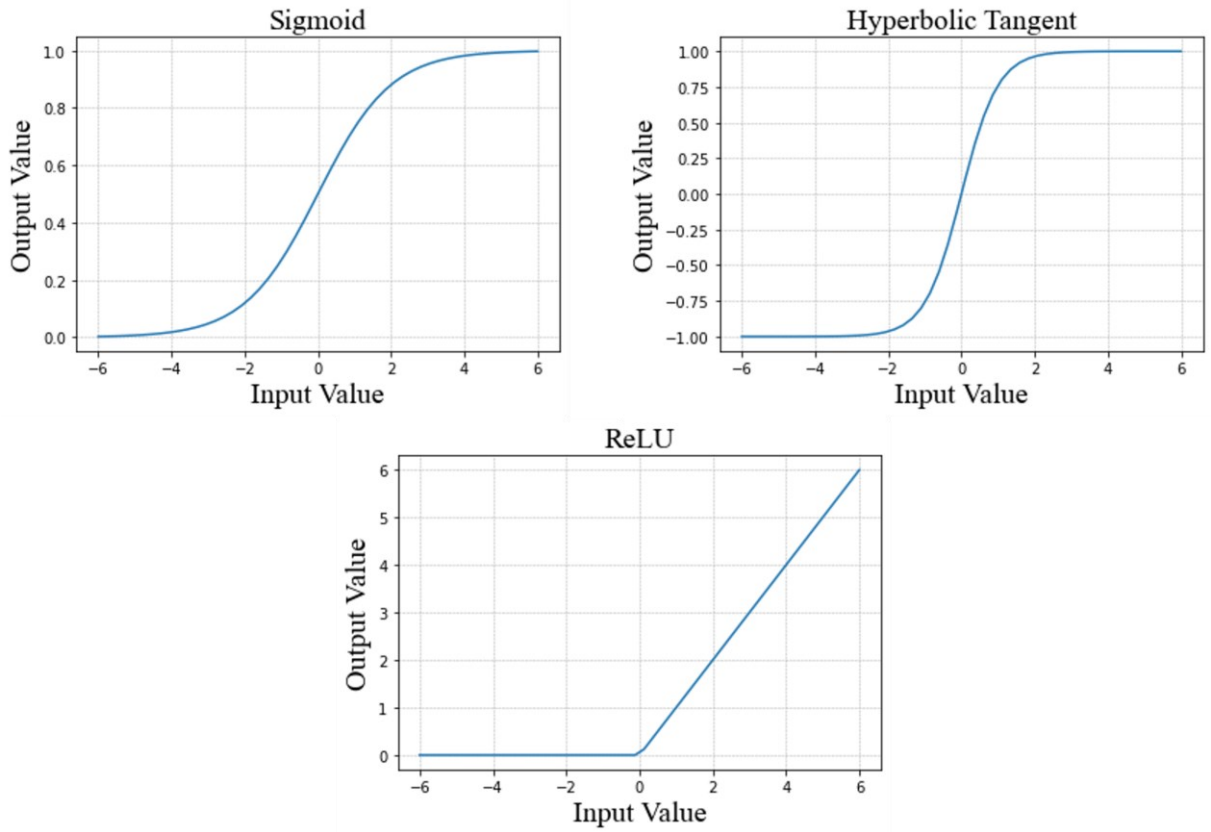



Figure 2.3 - Sigmoid, Hyperbolic Tangent, and ReLU Activation functions.

In networks with a small number of hidden layers, the sigmoid and hyperbolic tangent functions are commonly chosen as the activation functions for the artificial neurons. However, the preference shifts towards ReLU and its variants for networks with more hidden layers. This preference is motivated by the vanishing gradient problem. Regarding regression tasks, the identity function is frequently employed in the output layer neurons. On the other hand, in classification problems, the softmax function is predominantly used. This is because the softmax function produces probability values indicating the likelihood of a sample belonging to a particular class [17].

### 2.1.2. Layers

In ANNs, particularly in a FFNN, the neurons implement a function defined by equation 2.2, providing the possibility of choosing different activation functions, thus shaping the response of the neurons. The relevance of NN was previously shown as with only one hidden layer, and it can behave as a universal approximator, meaning it can mimic the behavior of any function, even if it is possible to use multilayer networks [18].

Considering that FFNNs can have multiple layers, a composite function is created. In the case of two layers, the result is calculated by

$$y = \Phi_1\left(\sum_{j=1}^{mh} w'_{j1}\left(\sum_{l=1}^{mI} w_{lj}x_l\right)\right). \tag{2.7}$$

An example of this scenario is illustrated in Figure 2.4.

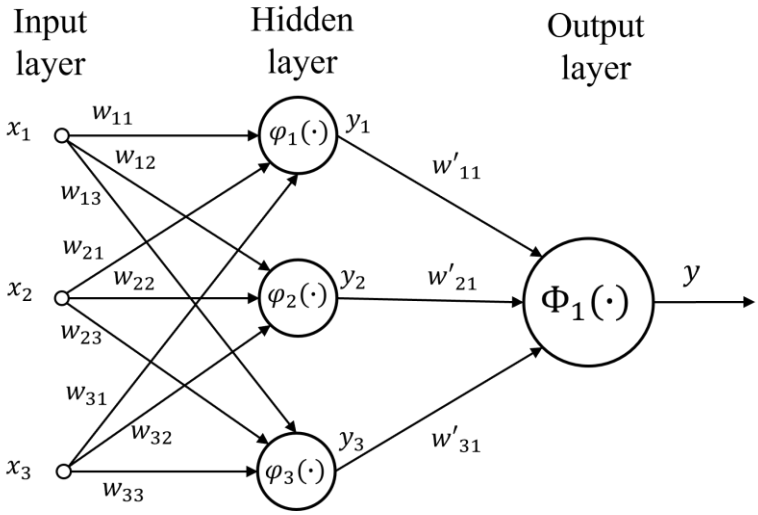



Figure 2.4 - Representation of a FFNN with one hidden layer.

Neural networks learn by presenting examples that result in modifications in the network weights during the training phase. The training of NN is usually carried out using the backpropagation algorithm, whose fundamental principle is to adjust the weights along the network based on the output error so that the answer is correct. ANN weights are updated iteratively, starting with an initial estimate and updating them using,

$$W_{k+1} = W_k + \alpha K.\rho K, \tag{2.8}$$

where $\rho K$ represents the search direction, and $\alpha K$ is the learning rate used. The various algorithms have different methods for choosing these values, considering the context of the problem. When dealing with minimization functions, which are often quadratic, it is common to employ a derivation process to find the desired minimum, according to

$$W = W_k - \alpha G(x, K), \tag{2.9}$$

where $G$ represents the gradient of the function to be minimized in iteration $k$.

## 2.2. Training Process

During the training process, the ANN learns by iteratively performing the task and assessing the level of error estimation.

### 2.2.1. Optimization

The computations conducted by individual neurons, equations 2.1 and 2.2 that describe the artificial neuron computation, we can summarize the calculation of neurons to,

$$y = \varphi\left(\sum_{j=1}^{m}\left(w_{kj}x_j\right) + b_k\right), \tag{2.10}$$

with $\varphi$ denoting the activation function, $w_{kj}$ representing the weight associated with input $j$, $b_k$ as the bias term, $x_j$ signifying input $j$, and $m$ indicating the count of inputs.

As previously seen, during the training process, the weights and biases used in the weighted sums are adjusted to obtain more accurate predictions in the output layer using different algorithms.

### 2.2.2. Backpropagation Algorithm

All training algorithms based on derivatives are classified as backpropagation. Although frequently described as an optimization algorithm in academic literature, Backpropagation encompasses a broader scope. It entails the computation of error linked to individual neurons within a NN, rooted in the output error. In other words, it entails the backward propagation of errors, progressing from the output to the input.

This fundamental process is a cornerstone in all algorithms utilized with NN, as it is imperative to gauge the error magnitude at the output of each neuron for the purpose of fine-tuning the associated weights.

The widely recognized backpropagation algorithm is essentially an implementation of the steepest descent technique and the chain rule of differentiation. This algorithm uses a loss or performance function and changes the weights and biases to minimize this function. The value of the loss function reflects the error presented by the model. A commonly used loss function is the Mean Square Error (MSE). Its value is calculated as

$$E = \frac{\sum_{i=1}^{n}(t_i - p_i)^2}{n}, \tag{2.11}$$

where $t_i$ represents the target value (label) for sample $i$, $pi$ is the prediction (output) for sample $i$, and $n$ is the total number of samples [16].

During each iteration, the weights and biases are adjusted based on gradient values. Consequently, when the output value $o_i$ is dependent upon the weight $w_{ij}$, the gradient value is determined using

$$\frac{\partial_E}{\partial w_{ij}} = \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial w_{ij}}. \tag{2.12}$$

Equation 2.13 gives the formula for weight adjustment, with $\frac{\partial E}{\partial w_{ij}}$ signifying the gradient value, $\alpha$ representing the learning rate, and $k$ indicating the layer [19].

$$\Delta w_{ij}^{k} = -\alpha \frac{\partial E}{\partial w_{ij}} \tag{2.13}$$

The most common cost function for classification is cross-entropy. It is calculated for a single sample using

$$E = -\sum_{c=1}^{n} t_c . log(p_c), \tag{2.14}$$

where $t_c$ represents the target value for class $c$, $p_c$ is the prediction for that sample's class $c$, and $n$ is the total number of classes [17].

### 2.2.3. Adam and AdaGrad Optimizer

Various training algorithms employ different formulas to update the weights, with the simplest one being the Gradient Descent (GD) algorithm.

There are several optimization algorithms, including Adam, one of the best known and used in DL, based on the junction of two other algorithms, AdaGrad and RMSProp.

The AdaGrad algorithm [20] accelerates the GD process, considering the exponentially weighted means of the gradients. By using these averages, the algorithm converges to the minimums quicker. The means are calculated by

$$W_{t+1} = W_t - \alpha m_t, \tag{2.15}$$

$$m_t = \beta m_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right], \tag{2.16}$$

where $w$ is the weights, $m$ the gradients, $\beta$ is the moving average parameter $\alpha$ is the learning rate, $L$ the loss function, and $t$ the time.

RMSProp (Root Mean Square Prop) is an adaptive learning algorithm that departs from the conventional approach of accumulating the squared gradient [21]. Instead, it utilizes exponential moving averages, and its computation is determined by the following equations.

$$W_{t+1} = W_t - \frac{\alpha_t}{(\upsilon_t + \xi)^{1/2}}\left[\frac{\delta L}{\delta w_t}\right] \tag{2.17}$$

$$\upsilon_t = \beta \upsilon_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right]^2 \tag{2.18}$$

where $\upsilon_t$ represents the sum of past gradient squares, $w$ denotes the weights, $m$ signifies the gradients, $\beta$ is the moving average parameter, and $\alpha$ is the learning rate. The Adam optimizer inherits the favorable characteristics of both methods and constructs a more optimized gradient descent, as given by

$$m_t = \beta m_{t-1} + (1-\beta_1)\left[\frac{\delta L}{\delta w_t}\right] \upsilon_t = \beta_2 m_{t-1} + (1-\beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2 \tag{2.19}$$

### 2.2.4. Training

Overfitting and underfitting are potential issues if precautions are not taken during training. Overfitting occurs when the model is too adapted to the training set, making it unable to generalize and classify new data. This happens because the model fits the noise, errors, and specific details (that are not a general pattern) present in the training data [22]. In contrast, underfitting happens when the model fails to fit the data adequately [23]. Balancing between overfitting and underfitting is often referred to as the bias-variance trade-off. Several methodologies can be used to prevent overfitting.

Early stopping is a technique used to stop model training when consecutive validation checks fail to show improvement. The validation set is evaluated at each epoch using the loss function to monitor error changes. Training is stopped if the error fails to decrease over several sequential epochs to prevent overfitting. This technique reduces training time and is controlled by the maximum number of failed validation checks [24].

In NN, dropout is employed as a regularization technique to mitigate overfitting. It involves randomly selecting a subset of neurons and setting their outputs to zero with a certain probability during training. This process compels the network to learn more resilient and independent features, as it cannot depend on specific neurons exclusively.

Incorporating dropout into NN enhances their performance in supervised learning tasks across diverse domains, including vision, speech recognition, and document classification. However, tests on text datasets revealed that the observed enhancement was smaller than the improvements in the vision and speech datasets [25].

Before starting the training process, the provided data must be partitioned into training, test, and validation sets. The purpose of the training set is to train the model, while the test set is utilized to evaluate the model's performance on unseen data. The validation set plays a role in finetuning the learning parameters and facilitating early stopping. Typically, the data is divided in a standard manner where 70% of the samples are allocated for training, 15% for testing, and the remaining 15% for validation.

Randomly selecting review data from the dataset to create sets that match defined percentages could limit our ability to assess the model's capacity to generalize to novel, previously unseen reviews. These unfamiliar reviews may exhibit different characteristics not represented in the dataset.

An alternative option is to implement k-fold cross-validation. Here, the dataset is divided into k sets known as folds. Each fold is used in turn as the test set, while the remaining folds are used for training and validation purposes. This process is repeated k times, allowing for a comprehensive evaluation of the model's performance across different subsets of the data. By employing k-fold cross-validation, we can obtain reliable performance metrics and assess the model's ability to generalize on unseen data.

### 2.2.5. Cross-Validation

The K-fold cross-validation technique divides the data into K equally sized folds or subsets. In each iteration of K-fold cross-validation, the model is trained on $K-1$ folds and evaluated on the remaining fold. This process is repeated K times, with each fold serving as the validation set precisely once. The model's performance is averaged over the K iterations for a more robust estimate. The susceptibility of this method to bias prompted the adoption of the stratified k-fold technique.

On the other hand, stratified k-fold cross-validation is a method for evaluating the performance of an ML model on a dataset with data unbalance. Specifically, stratified sampling is a technique where the proportions of different classes in the dataset are preserved in each fold. Each fold is stratified, containing a representative sample of each class or category in the data. For example, each fold would contain a representative sample of positive, neutral, and negative labels in a SA dataset. It is especially recommended when the data is imbalanced, as often in SA datasets [26].

For example, in a dataset of 100 randomly generated input reviews, represented in Figure 2.5, three classes split unevenly across reviews. Four splits of the data will be performed to visualize the stratified k-fold [26].

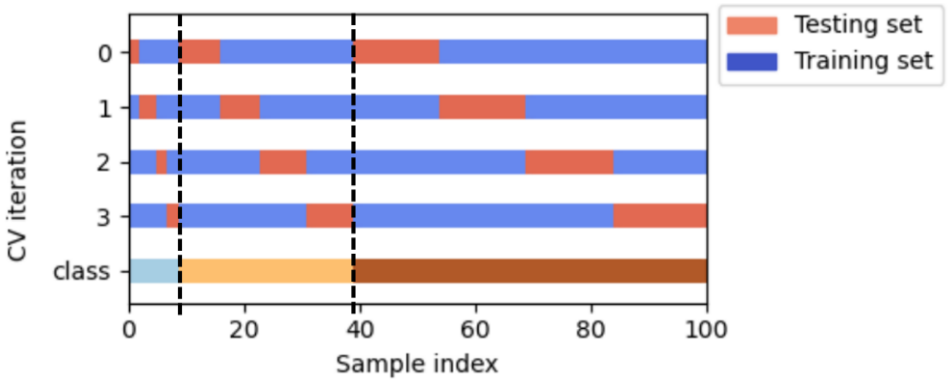



Figure 2.5 – Graphical representation of StratifiedKFold [26].

### 2.2.6. Benefits and Limitations

Stratified k-fold cross-validation has several advantages for evaluating the performance of an ML model. Dividing the data into multiple folds and using each fold as a test set once provides a more precise estimation of the model's performance on unseen data. Additionally, it exhibits greater resilience to variations in the data as the model undergoes training and evaluation multiple times on different data splits.

One of this technique's limitations is that small datasets can lead to higher performance estimate variability due to limited samples in each fold. It requires additional computational time and accurate class labels for effective stratification. Evaluation results may not reflect model performance on unseen data with the same class distribution.

## 2.3. Performance Evaluation Metrics

Performance evaluation metrics play a vital role in assessing the efficacy and quality of DL models. These metrics serve as quantitative measures to evaluate various aspects of the model's performance, enabling researchers and practitioners to assess its effectiveness in solving specific tasks. By examining these metrics, we can comprehensively understand the model's performance and make informed decisions about its application in real-world scenarios.

In the context of ML, it's important to distinguish between micro and macro metrics. Micro metrics focus on assessing the model's performance at the individual instance level, providing insights into precision, recall, accuracy, and other per-instance-based measures. These metrics are valuable for pinpointing specific areas of model errors or improvements. Conversely, macro metrics aggregate performance across different classes or groups within the dataset, offering a broader perspective on how well the model generalizes. Leveraging both micro and macro metrics offers a holistic view of a ML model's capabilities.

The Confusion Matrix (CM) is a tabular representation that records the number of occurrences between two raters: the true/actual classification and the predicted classification. The CM for three class problems and taking into consideration the class Negative for the calculation of the different metrics is depicted in Figure 2.6.

| | | Predicted Values | | |
|---|---|---|---|---|
| | | **Negative** | **Neutral** | **Positive** |
| **Actual Values** | **Negative** | TP (True Positive) | FN (False Negative) | FN (False Negative) |
| | **Neutral** | FP (False Positive) | TN (True Negative) | TN (True Negative) |
| | **Positive** | FP (False Positive) | TN (True Negative) | TN (True Negative) |

Figure 2.6 – Visualization of CM example for three classes.

This representation considers the definitions:

- True Positive (TP): This occurs when the actual and predicted values are identical.
- False Negative (FN): This indicates that the actual value is positive, but the model has made an incorrect prediction, presenting a negative value instead. In other words, the negative output obtained is inaccurate. The False-negative value for a particular class is determined by summing the values of corresponding rows, excluding the TP value.
- False Positive (FP): This refers to the actual negative value while the model's prediction is positive. Consequently, the positive output obtained is erroneous. The False-positive value for a specific class is obtained by summing the values of the corresponding column, excluding the TP value.
- True Negative (TN): Similar to TP, this occurs when both the actual and predicted values are the same. The True-negative value for a class is calculated by summing the values of all columns and rows except for the values of that specific class for which we are computing the values.

In the case of a multi-class classification problem, obtaining TP, TN, FP, and FN values is more challenging. Therefore, for validation, we must calculate these values for each class individually [27].

Heuristic methods can split a multi-class classification problem into multiple binary classification datasets and train a binary classification model for each class. There are two of these heuristic methods for using binary classification algorithms for multi-class classification, that is: One-vs-Rest (OvR), also referred to as One-vs-All (OvA), and One-vs-One (OvO). In this work, we have used the method OvR since it is the most common.

The OvR process entails dividing the multi-class dataset into several binary classification tasks. For each binary classifier trained in the OvR approach, we obtain a numerical score or probability representing the classifier's prediction confidence. This score indicates how confident the classifier is about its prediction for a given instance.

During the prediction phase, we apply all the binary classifiers to a new, unseen instance, and each classifier produces a confidence score or probability. The model with the highest confidence score among all the classifiers is considered the most confident model [28].

For example, suppose we have three classes (A, B, and C) and have trained three binary classifiers (A vs. B and C, B vs. A and C, and C vs. A and B) when we feed a new instance to these classifiers. In that case, they will each produce a confidence score or probability for their corresponding class. If classifier A vs. B and C gives a confidence score of 0.8, classifier B vs. A and C gives a confidence score of 0.6, and classifier C vs. A and B gives a confidence score of 0.9. We would select the C vs. A and B classifier as the most confident model for this particular instance.

Ultimately, the class associated with the most confident model is considered the predicted class for the given instance in the multiclass classification problem.

By utilizing the CM, we can derive two crucial metrics: Recall and Precision. These, alongside ACC and the AUC-ROC curve, are commonly employed to assess the performance of ML models. The CM, in this work, employs the PyCm library.

The following formulas were adapted for calculating the metrics for the three classes, where the calculation is done by class and then made an average.

**ACC** – also known by various names, such as overall agreement, overall classification rate, overall predictive ability, total accuracy, or overall success rate, represents the proportion of correct classifications in relation to the total number of samples without considering the individual class classification performance. The ACC measures the overall correctness of the classification and can be expressed as

$$ACC_o = \frac{1}{N}\sum_{c=1}^{C} TP_c, \tag{2.20}$$

Its value ranges from 0 (indicating no correctly classified samples) to 1 (representing a perfect classification) [29], [30].

**Precision** – also named Positive Predictive Value (PPV), is a measure of the model's ability to avoid false positives, in other words, to avoid wrong predictions [29], [30]. It is defined as the proportion of correct positive predictions made by the model on the dataset and is given by

$$PPV = \frac{1}{C}\sum_{c=1}^{C} \frac{TP_c}{TP_c + FP_c}. \tag{2.21}$$

**Specificity** – also known as the True Negative Rate (TNR), indicates the proportion of all negative samples that are accurately predicted as negative by the classifier [29], [30]. The calculation is performed using

$$TNR = \frac{1}{C}\sum_{c=1}^{C} \frac{TN_c}{TN_c + FP_c}. \tag{2.22}$$

**Sensitivity** – also known as True Positive Rate (TPR), is a measure of the model's ability to avoid false negatives [29], [30]. It is defined as the proportion of actual positive cases correctly predicted by the model on the dataset and can be expressed as

$$TPR = \frac{1}{C}\sum_{c=1}^{C}\frac{TP_c}{TP_c + FN_c}. \quad (2.23)$$

**F1 Score** – is the harmonic mean of precision and recall. It is a weighted average that considers both the model's precision and recall [29], [30]. The F1 score is calculated as shown in the following formula:

$$F1 = 2 \times \frac{(PPV \times TPR)}{(PPV + TPR)}. \quad (2.24)$$

F1 score is more valuable than ACC, especially when it has an uneven class distribution, showing how balanced the classifications are between classes. This measure can be computed in terms of the metrics mentioned earlier or in terms of the CM as

$$F1 = \frac{1}{C}\sum_{C=1}^{C}\frac{2TP_c}{2TP_c + FP_c + FN_c}. \quad (2.25)$$

**AUC- ROC** – quantifies the overall two-dimensional area beneath the entire ROC curve, spanning from the point (0,0) to (1,1) as in Figure 2.7. AUC is a comprehensive performance metric summarizing the classifier's ability across all possible classification thresholds. It can be computed as

$$AUC \approx \frac{1}{C}\sum_{c=1}^{C}\frac{TNR_c + TPR_c}{2}. \quad (2.26)$$

The ROC curve is a graphical representation of the performance of a binary classification model. It plots the TPR against FPR at different classification thresholds [31]. The ROC curve allows the comparison of different models on the same dataset, and it is often used in SA to evaluate the performance of different models [29], [30]. Figure 2.7 shows examples of ROC curves for each class using a dummy dataset with three classes.

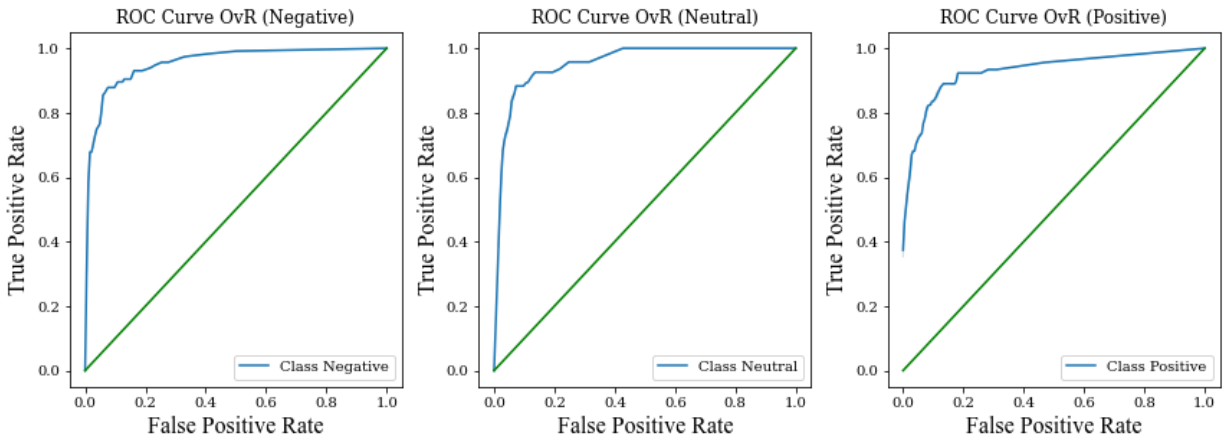



Figure 2.7 – Visualization of ROC curve for each class.

In this example, the calculated values are presented in TABLE I, where is the calculated AUC for each specific class.

TABLE I – Example of AUC ROC values with OvR and average final value for three class tasks.

| Classes | Value |
|---|---|
| Negative | 0.9573 |
| Neutral | 0.9712 |
| Positive | 0.9642 |
| Average | 0.9642 |

Looking into TABLE I, the average ROC AUC with OvR, in this case, is 0.9642, an excellent score reflecting how well the classifier predicted each class.

## 2.4. Transformers Architecture

Transformers were first introduced by Vaswani et al. [32], who were affiliated with Google in 2017. At the time of their introduction, language models primarily used Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN) to handle NLP tasks.

The transformer architecture, represented in Figure 2.8, symbolizes a breakthrough in the field of NLP due to its ability to address a key limitation of earlier sequence-to-sequence (seq-to-seq) models like RNNs. These models struggled to capture long-term dependencies in text, which the transformer architecture overcame with impressive results. This breakthrough paved the way for developing revolutionary NLP architectures, creating pre-trained models like BERT, which was trained on massive amounts of language data before its release.

At the core of transformer models lies an attention mechanism commonly named self-attention. The introduction of this architecture can be attributed to the seminal 2017 paper "Attention is All You Need" [32]. The transformer architecture is built upon an Encoder-Decoder Architecture, which comprises two critical components allowing for more efficient and effective sequential data processing, including natural language.

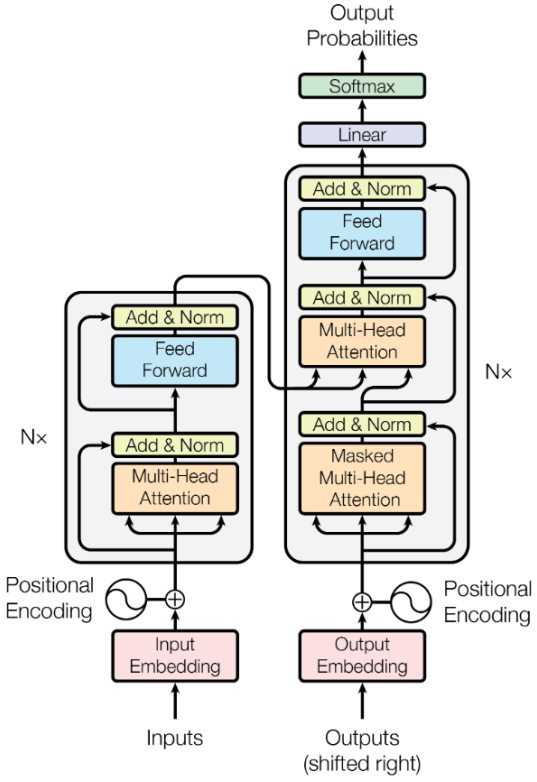



Figure 2.8 – The Transformer – model architecture [32].

The first component is the encoder, which is crucial in processing the input sentence. Specifically, it is responsible for accepting the input sentence and converting it into a hidden representation that captures the most critical information while filtering out any irrelevant or redundant information. Once the encoder generates this condensed representation, it is passed to the decoder, responsible for generating the target sentence. As this work focuses on the pre-trained BERT model, only the encoder component will be discussed in detail, as it is the only part of the architecture utilized by BERT.

### 2.4.1. Encoder

The Transformer's encoder consists of a stack of $N = 6$ layers that are the same. Each layer has two sub-layers: a multi-head self-attention mechanism and a simple, fully connected FFNN. We use a residual connection around each sub-layer, followed by a normalization layer. This means that the output of each part is added to the original input and then normalized. All parts of the model, including the embedding layers, produce outputs of dimension $d_{model} = 512$.

### 2.4.2. Self-attention Mechanism

Attention mechanisms were initially developed for neural machine translation to align source and target sentences. In seq-to-seq models, attention is used to determine which tokens in the source sentence are most relevant for generating the target sentence token by token. This is achieved by calculating attention scores from hidden state representations in the encoder and decoder and generating context vectors as input for the decoder.

In the concept of self-attention or scaled dot-product, the basic idea is to calculate attention scores while mapping a sentence onto itself. For instance, consider the sentence: "During the concert, John was on stage playing the guitar. He played an amazing solo."

Humans naturally understand that "he" in the sentence refers to John. However, self-attention is employed for a language model to understand this connection. At a high level, each word in the sentence is compared against every other word in the sentence to identify the relationships and better comprehend the context. This process is particularly useful for capturing long-range dependencies in a sentence, which traditional seq-to-seq models often struggle with.

A visual representation of attention was performed using an open-source tool that visualizes attention at multiple scales, providing a unique perspective on the attention mechanism [33]. Figure 2.9 shows the connection between the two sentences. This was done using the 12 layers and the 12 heads from the BERT transformer.

In this tool, it is possible to select a word from the left-hand sentence and view, within the same sentence on the right, where the greatest attention is directed in terms of word connections. We can see that when selecting the word "He," the attention mechanism places greater emphasis on the word "John," and punctuation establishing the reference for the word "He" by giving a more pronounced tonality to the words it pays closer attention to, assigning them greater importance.

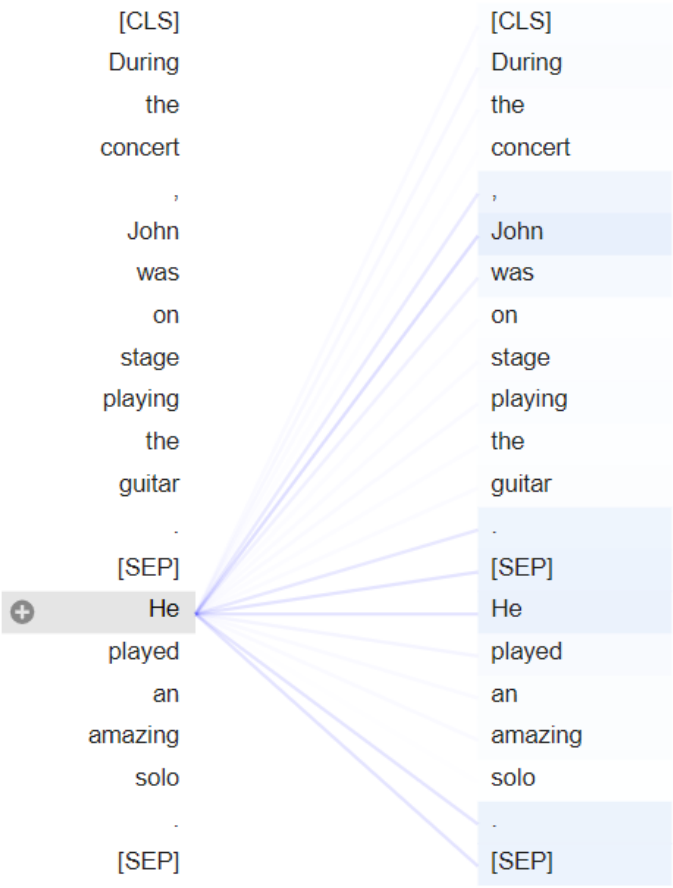



Figure 2.9 - Attention pattern in BERT related to coreference.

The neuron visualization, depicted in Figure 2.10, offers insights into the distinct neurons present in the query and key vectors, illustrating their interplay in generating attention. Upon selecting a token chosen by the user (left), this visualization tracks the attention computation from that specific token to other tokens in the sequence (right).

In this representation, blue coloration is assigned to positive values, while negative values are denoted by orange. The intensity of the colors varies based on the magnitude of the value. The connecting lines are weighted to reflect the attention relationships between the words.


Figure 2.10 - Illustration of the interplay in generating attention, showing the attention computation.

The words in the sentence are converted into an input matrix to generate embeddings for the input sentence by finding embeddings for each word. The embeddings can be generated using simple tokenization and one-hot encoding or with more sophisticated embedding algorithms such as BERT [34]. The resulting input matrix $X$ has the dimensions of the sentence length by the embedding dimension. This input matrix serves as the foundation for the subsequent steps in the model.

To enable self-attention, the input matrix $X$ is transformed into three new matrices: Query ($Q$), Key ($K$), and Value ($V$). This is achieved by multiplying the input matrix $X$ with three randomly initialized weight matrices - $Wq$, $Wk$, and $Wv$. The output matrices $Q$, $K$, and $V$ are used in the subsequent steps of the model. During the training process, the optimal values for the weight matrices $Wq$, $Wk$, and $Wv$ are learned to improve the ACC of the values for $Q$, $K$, and $V$.

Considering that $q_i$, $k_i$, and $v_i$, represent the values of $Q$, $K$, and $V$ for the i-th word in the sentence. The dot-product between the $Q$ and $K$ in the Transformer's self-attention mechanism (2.28) determines the degree of relatedness between words in a sentence as

$$Q.K^T = \begin{matrix} w_1 \\ w_2 \\ w_3 \end{matrix} \begin{bmatrix} q_1k_1 & q_1k_2 & q_1k_3 \\ q_2k_1 & q_2k_2 & q_2k_3 \\ q_3k_1 & q_3k_2 & q_3k_3 \end{bmatrix}. \quad (2.27)$$

This is reflected in the first row of the output matrix, where the higher the dot-product value, the stronger the association. The relationship between $Q$ and $K$ can be compared to information retrieval, where $Q$ represents the term being searched and $K$ represents a set of keywords for comparison and matching. Figure 2.11 is the representation of self-attention.

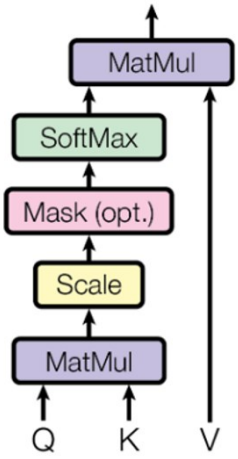



Figure 2.11 – Self-Attention or scaled dot-product [32].

It is necessary to stabilize the gradients to prevent the value from exploding during the multiplication operation of calculating the dot-product in the previous step. One way to do this is by dividing the dot product of $Q$ and $K$-transpose by the square root of the embedding dimension ($d_k$). The compute of the matrix of output becomes

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \quad (2.28)$$

The softmax function is applied to the scaled dot-product to normalize the values between 0 and 1. This ensures that the cells with high dot-products are further emphasized while low values are suppressed, resulting in a clearer distinction between the matching word pairs. The resulting output matrix can be viewed as a score matrix, which we can refer to as $S$.

The next step involves calculating the attention matrix $Z$ by multiplying the values matrix $V$ with the score matrix $S$ obtained from the previous step. The reason for this multiplication is to weigh the importance of each value vector based on its corresponding score. For example, if the score vector for the i-th word in a sentence is $Si = [0.9, 0.07, 0.03]$, then the attention matrix for that i-th word, $Zi$, will be calculated by multiplying this score vector with the corresponding value matrix $V$ as,

$$Zi = [0.9 \times V1 + 0.07 \times V2 + 0.03 \times V3]. \quad (2.29)$$

This weighting process helps to emphasize the more important words in the sentence and suppress the less important ones. It can be said that the contribution of each word in the attention matrix $Z$ is indicative of its importance in understanding the context of the i-th word. For example, if 90% of the value of the attention score comes from $V1$ (the vector representation of word 1), it could be concluded that more attention should be paid to that word to understand the context of the i-th word. Therefore, the higher the contribution of a word in the $Zi$ representation, the more critical and related it is to the other words.

### 2.4.3. Multi-head Attention Mechanism

The multi-head attention, presented in Figure 2.12, is one of the most important components of the encoder mechanism. When the score matrix is biased towards a specific word representation, it can mislead the model and produce inaccurate results. To understand this better, let us consider the following examples applying (2.29):

Example 1:

Input: “All is well.”

Score matrix $S1 = [0.6, 0.0, 0.4]$

Attention matrix for “well,”

$$Z(well) = 0.6 \times V(all) + 0.0 \times V(is) + 0.4 \times V(well)$$

In this example, more importance is given to $V(all)$ while calculating $Z(well)$, even more than $V(well)$ itself.

Example 2:

Input: “The dog ate the food because it was hungry.”

Score matrix $S2 = [0.0, 1.0, 0.0, \ldots, 0.0]$

Attention matrix for “it”

$$Z(it) = 0.0 \times V(the) + 1.0 \times V(dog) + 0.0 \times V(ate) + \cdots + 0.0 \times V(hungry)$$

In this example, all the importance is given to $V(dog)$ while calculating $Z(it)$, and the scores for the rest of the words are 0.0, including $V(it)$ itself. This seems reasonable as the “it” word is ambiguous, and it makes more sense to relate it to another word than the word itself. This is the whole purpose of calculating self-attention, of handling the context of ambiguous words in input sentences.

However, giving more importance to another word while calculating self-attention is not always advisable, as it can be misleading for the model. To avoid this issue, we can calculate multiple attention matrices (z1, z2, z3, ….., zm) and concatenate them to derive the final attention matrix. This is what multi-head attention is all about, and it allows more confidence in the attention matrix.

Figure 2.12 visually depicts multi-head attention’s application on the encoder’s input.

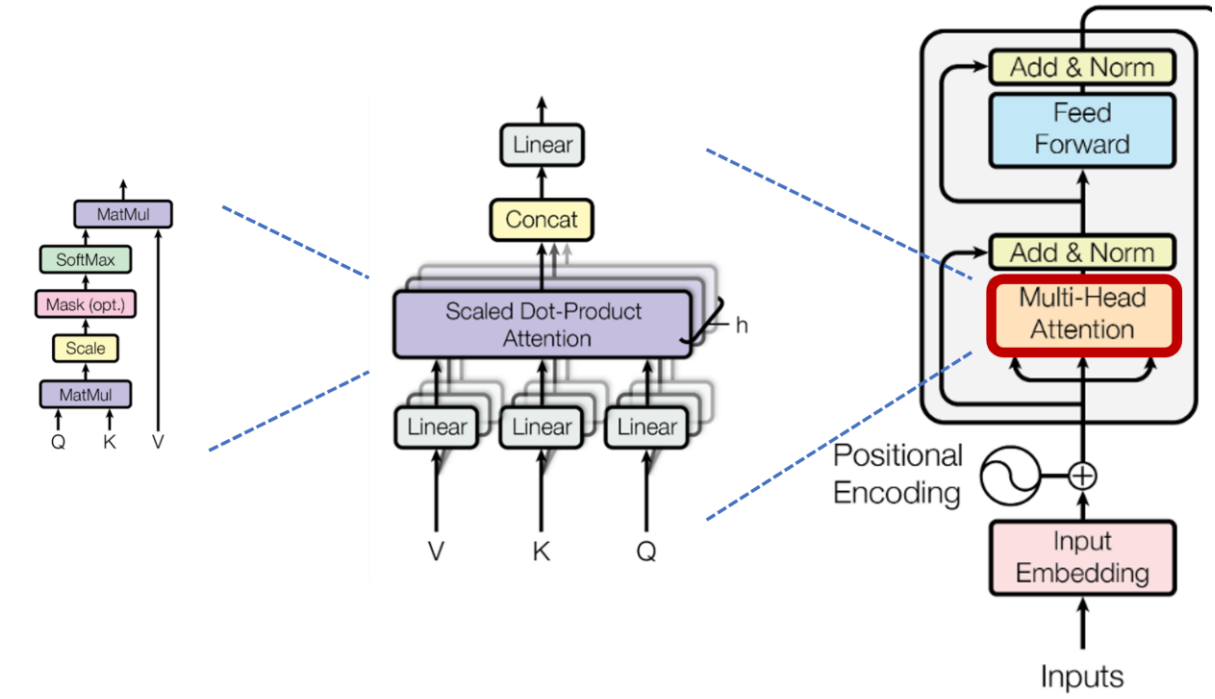



Figure 2.12 – Multi-head attention on encoder representation [32].

It shows how the input is transformed into Q, K, and V matrices using learnable weight matrices ($Wq, Wk, Wv$).

### 2.4.4. Positional Encoding

In seq-to-seq models, the input sentence is fed to the model one word at a time, allowing it to understand the positions of each word relative to others. However, the transformer model takes a different approach. Instead of processing the input sequentially, it is fed in parallel, which reduces training time and helps to learn long-term dependencies. However, with parallel input, the model loses the word order information essential for proper sentence meaning. A new matrix called "positional encoding" ($P$) is introduced to address this. This matrix is combined with the input matrix $X$ to include the word order information. The dimensionality of the positional encoding matrix is the same as that of the input matrix $X$ for compatibility.

The positional encoding, $PE$, is calculated as,

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \text{ if } i \text{ odd}, \tag{2.30}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \text{ if } i \text{ even}, \tag{2.31}$$

where $pos$ is the position of the word in the sentence, $d_{model}$ is the dimension of the word/token embedding, and $i$ represents each dimension in the embedding. By adding this positional encoding matrix to the input matrix, the transformer model can preserve the sequential order of the words while processing the input in parallel.

During the calculation of positional encoding, the dimension of the word/token embedding ($d$) remains fixed, but the position of the word in the sentence ($pos$), and the dimension ($i$) varies. Specifically, if the value of $d$ is set to 512, then the possible range of $i \in [0, 255]$, as $2i$ is used in the calculation.

Figure 2.13, displayed below, illustrates an instance of a positional encoding vector representation along with different variable values based on (2.30).

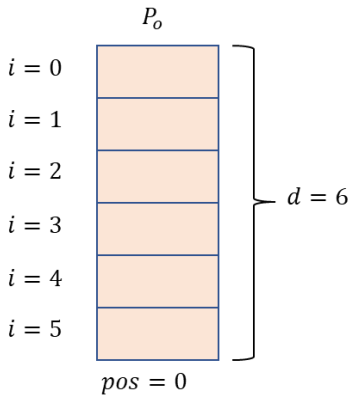



Figure 2.13 – Positional encoding vector representation.

Examining (2.30) and knowing that the values of $pos$ vary while $i$ is kept constant together with $d$, it will create identical positional encoding as shown in Figure 2.14.

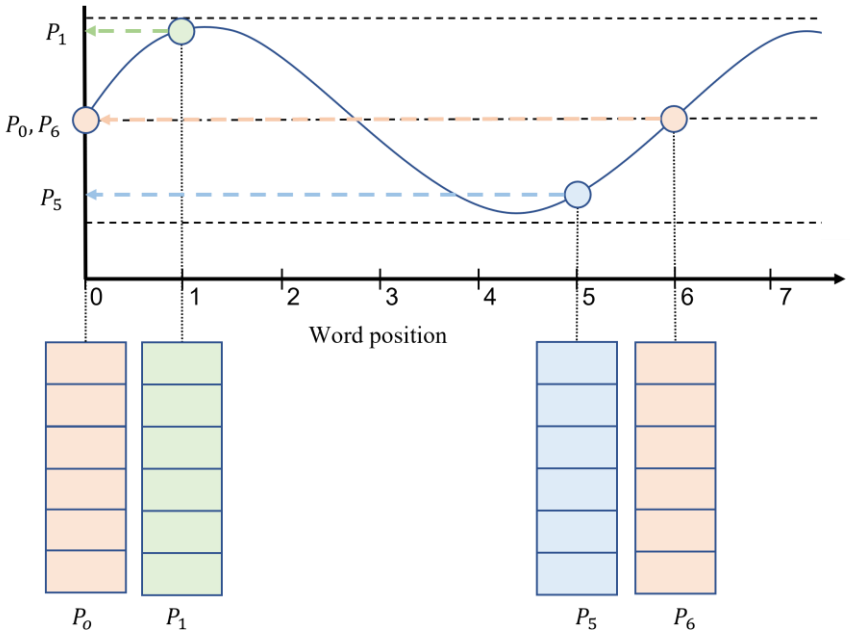



Figure 2.14 – Position encoder vector representation with $i$ and $d$ constant. (curves are not drawn to scale).

Since sinusoidal waves repeat themselves periodically, the encoding vectors for $pos = 0$ and $pos = 6$ are the same. However, it is not ideal to have identical positional encoding vectors for different values of $pos$. It needs different positional encoding vectors for different values of $pos$.

To ensure that different positional encoding vectors are produced for different values of $pos$, one can vary the frequency of the sinusoidal wave. In Figure 2.15, we can see that as the value of $i$ varies, the frequency of the sinusoidal waves also varies, resulting in different waves and, thus, different values for each positional encoding vector.

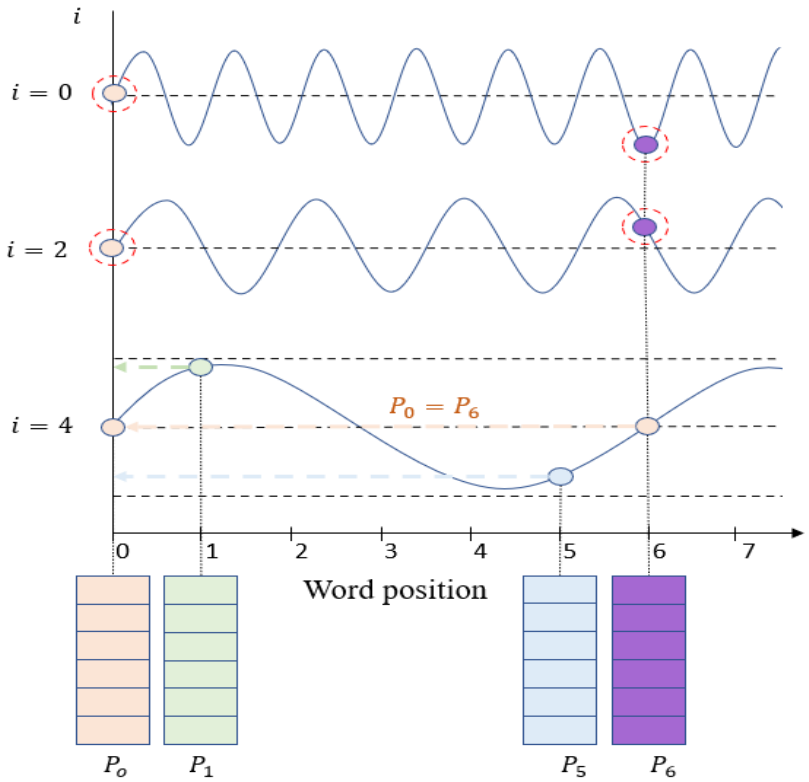



Figure 2.15 - Position encoder vector representation with varying $pos$ and $i$. (curves are not drawn to scale).

This achieves the desired result of having distinct positional encoding vectors for different positions in a sentence.

After calculating the positional encoding matrix ($P$) using the formula and variable values, it is added to the input matrix ($X$) to incorporate the information about word order. As displayed in Figure 2.16, the resulting matrix is then fed into the encoder of the transformer model to be processed.

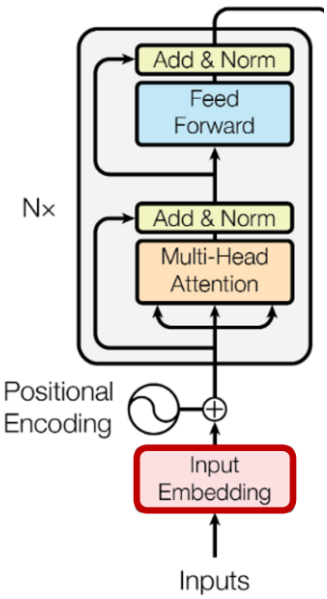



Figure 2.16 – Adding positional encoding to the input embeddings [32].

### 2.4.5. Feedforward Network

The sublayer within the encoder block is a standard NN with two dense layers and ReLU activations. It receives the input from the multi-head attention layer, applies a nonlinear transformations to the input, and produces contextualized vectors, as represented in Figure 2.17. The fully-connected layer is responsible for processing each attention head and extracting relevant information from them. Because the attention vectors are not dependent on each other, they can be processed in a parallelized manner, enabling faster computation within the transformer network.

The final component of the Encoder block is the Add & Norm layer, which consists of a residual layer followed by layer normalization. The residual layer ensures that critical information related to the input of sub-layers is preserved during processing, and the normalization layer promotes faster model training and prevents drastic changes in values.

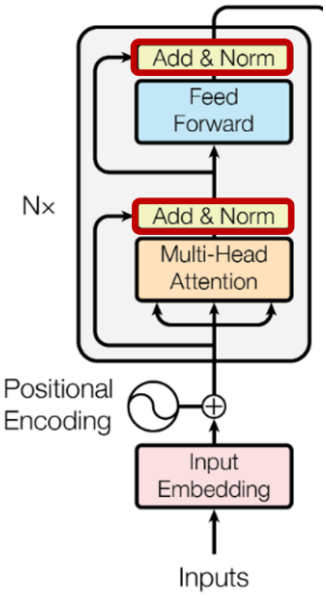



Figure 2.17 – Encoder components with add & norm layers [32].

The encoder has two add & norm layers to connect the input and output of the multi-head attention sub-layer and feedforward network sub-layer. These layers add the input and the output from each sub-layer, then normalize the result to ensure the training process remains stable. This marks the end of the description of the internal workings of the encoder.

## 2.5. Transfer Learning

Transfer learning refers to leveraging knowledge from one task to improve performance on another related task. This approach has gained significant popularity due to its ability to enhance model performance, reduce training time, and alleviate the need for large amounts of labeled data [35] [36].

### 2.5.1. Methods and Concepts

Transfer learning methods can be categorized based on different criteria, such as the homogeneity of source and target data, the label-setting of source and target data, and the applied approaches. The most common applied approaches are model-based, which use pre-trained models and adjust them for target data by freezing, finetuning, or adding layers. Finetuning involves taking a pre-trained model and adapting it to a new task by further training on a smaller dataset.

The adding layers approach consists in adding new layers on top of the pre-trained model. These additional layers can be randomly initialized and then trained during training. Adding new layers allows the model to learn task-specific features and capture complex patterns specific to the target task [36].

### 2.5.2. Pretraining Techniques

Pretraining techniques are an integral part of TL. Pretraining involves training a model on a large dataset and then using the learned representations as a starting point for a target task.

There are two main challenges: catastrophic forgetting and overly biased pre-trained models. Catastrophic forgetting occurs when a pre-trained model loses its previous knowledge when finetuned on a new task. Overly biased pre-trained models occur when a pre-trained

model cannot learn new features from target data due to frozen layers. Possible solutions include progressive learning, which adds new layers to a frozen pre-trained model, and vertical expansion, which adds new nodes to frozen pre-trained layers.

## 2.6. Ensemble and Boosting

Ensemble learning and boosting are two distinct approaches employed in ML to enhance the performance and precision of predictive models. Nevertheless, they vary in terms of their fundamental principles and methodologies.

Ensemble learning combines multiple models to produce a final prediction, leveraging their diversity to enhance performance. The final prediction is obtained through voting or averaging the individual models' predictions. In EL, models are independently created using different algorithms, architectures, or data subsets. Each model contributes equally to the final prediction, and they can be trained independently and in parallel without explicit feedback or adjustment between them.

Boosting is an iterative process that enhances the performance of a weak learner by training new models focused on correcting misclassifications. The final prediction is obtained through an ensemble methodology, such as weighted voting, combining the predictions of all the weak learners. In boosting, models are constructed sequentially to rectify previous model errors. Notable boosting algorithms include Adaptive Boosting (AdaBoost), Gradient Boosting, and XGBoost. These algorithms dynamically adjust the weights of training samples to focus on misclassified instances. The choice of base learners and the criterion for updating sample weights vary across different boosting variants. AdaBoost is one of the most well-known boosting algorithms [37] and has contributed significantly to the popularity of this approach since the influential works of Freund and Schapire [38].

### 2.6.1. Adaboost Algorithm

Following their contributions to boosting algorithms, Freund and Schapire introduced the AdaBoost algorithm [38], [39], [40]. Unlike previous boosting methods, AdaBoost utilizes weighted versions of the same training data instead of random subsamples. This eliminates the need for a massive training set. AdaBoost has gained substantial recognition as a well-known and extensively studied technique for constructing high-performing classifier ensembles.

The algorithm employs a weak learner to sequentially generate a set of classifiers that collectively form the final classifier. The weak classifiers are trained using re-weighted versions of the training data, where the weights depend on the ACC of the previous classifiers. The training set remains consistent throughout the iterations, with each training instance being assigned a weight based on its correct or incorrect classification by the previous weak classifiers. This enables the weak learner in each iteration to focus on patterns not effectively classified by the previous weak classifiers.

Selecting suitable weak learners to generate the base classifiers is essential. Ensuring these weak learners can learn effectively without excessively reducing the weight assigned to previously correctly classified instances is vital. If a base learner is overly powerful, it might achieve high ACC but at the cost of assigning significant weight to outliers and noisy instances in the subsequent iterations of the algorithm [37]. The structure of AdaBoost is depicted in Figure 2.18.

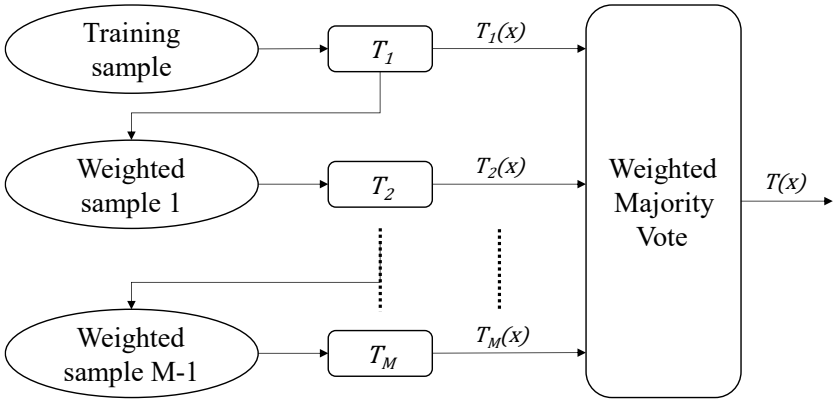



Figure 2.18 – Representation of the Adaboost algorithm. Adapted from [37].

AdaBoost has demonstrated exceptional success in generating precise classifiers for two-class classification tasks. However, when applied to multiclass problems, although AdaBoost has been adapted for such scenarios (Freund & Schapire 1997) [37], its performance may not be as outstanding. Due to this, a variation of the AdaBoost algorithm was used for multiclass boosting [41], referred to by the author as Stagewise Additive Modeling using a Multiclass Exponential loss function (SAMME), described in **Algorithm 1**.

**Algorithm 1** SAMME

1: Initialize the observation weights $w_i = 1/n, i = 1, 2, \ldots, n$.

2: For m =1 to M:

(a) Fit a classifier $T^{(m)}(x)$ to the training data using weights $w_i$.

(b) Compute

$$err^{(m)} = \sum_{i=1}^{n} w_i \mathbb{I}\left(c_i \neq T^{(m)}(x_i)\right) / \sum_{i=1}^{n} w_i.$$

(c) Compute

$$\alpha^{(m)} = \log \frac{1-err^{(m)}}{err^{(m)}} + \log (K-1). \qquad (2.32)$$

(d) Set

$$w_i \leftarrow w_i * \exp\left(\alpha^{(m)} * \mathbb{I}\left(c_i \neq T^{(m)}(x_i)\right)\right), i = 1, 2, \ldots, n.$$

(e) Re-normalize $w_i$.

3: Output

$$C(x) = argmax \sum_{m=1}^{M} \alpha^{(m)} * \mathbb{I}\left(T^{(m)}(x) = k\right)$$

**Algorithm 1** (SAMME) resembles AdaBoost, yet with a notable distinction in 2.32. In SAMME, for $\alpha^{(m)}$ to be positive, it suffices that $(1 - err^{(m)}) > 1/K$, implying that the ACC of each weak classifier must surpass random guessing rather than merely $1/2$. Consequently, SAMME assigns greater weight to misclassified data points in point (2d) compared to

AdaBoost. Moreover, SAMME combines weak classifiers in a slightly different manner than AdaBoost, specifically varying by $log(K-1)\sum_{m=1}^{M}\mathbb{I}\left(T^{(m)}(x)=k\right)$. Notably, SAMME reduces to AdaBoost when $K=2$. The additional term $log(K-1)$ in 2.32 renders the algorithm equivalent to fitting a forward stagewise additive model using a multiclass exponential loss function.

The difference between AdaBoost and SAMME (when $K=3$) lies in the fact that AdaBoost ceases to be effective once the $err^{(m)}$ surpasses $1/2$. In contrast, SAMME continues to perform well even if $err^{(m)}$ can be bigger than $1/2$ (or equal to $1/2$), the $\alpha^{(m)}$ is still positive, resulting in an increased emphasis on misclassified training samples and a continuous decrease in test error.

### 2.6.2. Soft Voting

The voting used to produce the ensemble is usually either a hard or a soft approach. Hard voting predicts the final class label based on the most frequently predicted class label among the classification models [42].

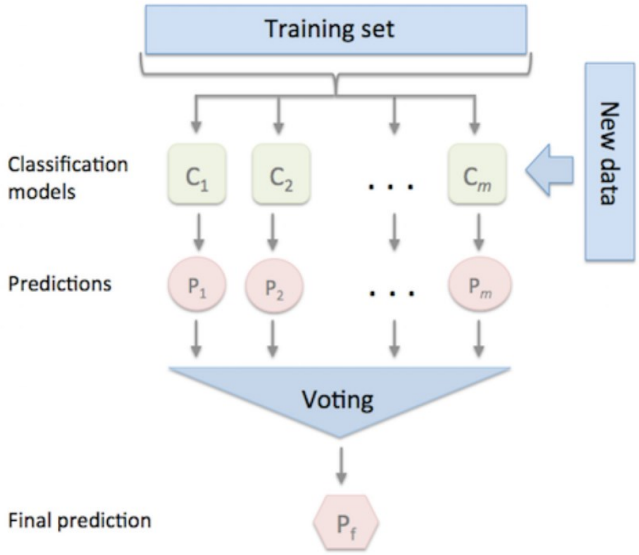



Figure 2.19 – Generalization of ensemble voting [42].

On the other hand, in soft voting, the class labels are predicted by considering the predicted probabilities $p$ from each classifier,

$$\hat{y}=argmax\sum_{j=1}^{m}w_j p_{ij}, \tag{2.33}$$

where $w_j$ is the weight that can be assigned to the $j$th classifier.

In a multiclass classification task with class labels represented as $i\in\{0,1,2\}$, let us consider an example where our ensemble of classifiers makes the following prediction for a given input $x$,

$$C_1(x)\rightarrow[p0_1,p1_1,p2_1],$$

$$C_2(x)\rightarrow[p0_2,p1_2,p2_2],$$

$$C_3(x)\rightarrow[p0_3,p1_3,p2_3].$$

Assigning the weights $\{w_1, w_2, w_3\}$ to the classifiers, and computing the average probabilities would yield the predicted class label ŷ,

$$p(i_0 \mid x) = w_1 \times p0_1 + w_2 \times p0_2 + w_3 \times p0_3 \quad (2.34)$$

$$p(i_1 \mid x) = w_1 \times p1_1 + w_2 \times p1_2 + w_3 \times p1_3 \quad (2.35)$$

$$p(i_2 \mid x) = w_1 \times p2_1 + w_2 \times p2_2 + w_3 \times p2_3 \quad (2.36)$$

$$\hat{y} = argmax[p(i_0 \mid x), p(i_1 \mid x), p(i_2 \mid x)] \quad (2.37)$$

this approach is only recommended if the classifiers are well-calibrated, as it yields more reliable and accurate results.

## 2.7. Key Remarks

Neural networks serve as the fundamental building blocks of ML, drawing upon a combination of various techniques to create increasingly robust and accurate models. One such technique, stratified k-fold cross-validation, proves instrumental in effectively handling datasets, especially those that are imbalanced, thus enhancing overall performance.

For a thorough assessment of the efficiency of DL models, performance evaluation metrics play a pivotal role. By offering quantitative measures to evaluate a model's proficiency in different tasks, these metrics enable informed decisions concerning its practical application in real-world scenarios.

Transformers, such as BERT and RoBERTa, are designed to pre-train deep bidirectional representations from unlabeled text, employing joint conditioning on both left and right context throughout all layers. This unique feature empowers these models to capture essential contextual information within sentences, a crucial aspect for numerous NLP tasks, including SA.

In TL, the pre-trained model undergoes further fine-tuning or adaptation to a target task using a smaller labeled dataset—an inherent characteristic of TL. This fine-tuning step allows the model to better adjust its learned representations, thereby enhancing its performance and generalization capabilities for the specific task at hand.

When it comes to ensemble techniques like AdaBoost and SAMME, an interesting distinction arises. While AdaBoost becomes ineffective once the error surpasses 1/2, SAMME continues to perform well even when the error exceeds 1/2. The result is that SAMME places a stronger emphasis on misclassified training samples, leading to a continuous reduction in test error. Additionally, in ensemble modeling, the adoption of soft voting proves valuable as it considers the predicted probabilities from each classifier. By incorporating the confidence levels of each class label prediction, this approach offers a more nuanced and probabilistic prediction mechanism.

# 3. Sentiment Analysis in NLP

SA, also known as opinion mining, is a research area that examines the sentiments, opinions, and emotions expressed in the written text towards various subjects and their characteristics. The field of NLP has been actively studying SA, primarily focusing on written text. Through NLP, SA usually categorizes words as positive, negative, or neutral. However, SA has also attracted significant attention in data mining, web mining, and information retrieval due to the prevalence of textual data [43].

Figure 3.1 presents an illustration depicting the interrelation between different fields. An ellipse in the image's background symbolizes AI, serving as the overarching domain encompassing various subfields. One of these subfields is ML, which is closely linked to AI. ML focuses on creating algorithms that enable computer systems to learn and make predictions or decisions based on data without explicit programming.

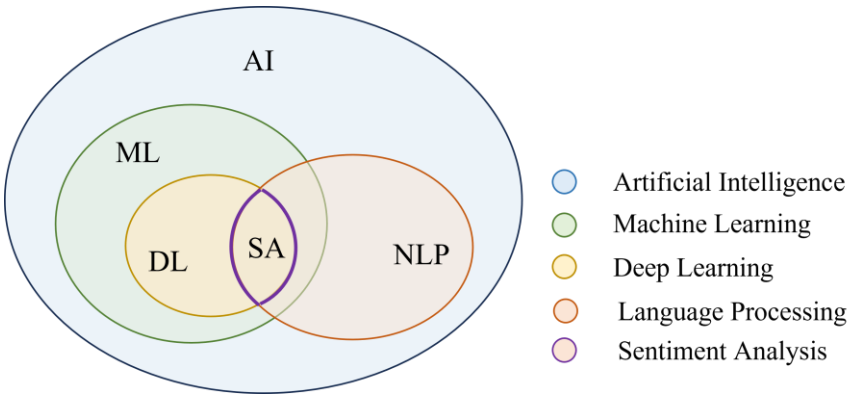



Figure 3.1 – Representative relation between fields.

Bringing all these fields together and incorporating NLP techniques as a crucial component, we find SA, which aims to analyze and extract opinions, sentiments, and emotions from textual data. However, SA can be performed using various ML, DL, or other NLP techniques. It is not limited to a specific combination of these techniques, as Figure 3.1 suggests.

## 3.1. Dataset Preprocessing

In the context of dataset preprocessing, it's important to consider the use of TL with BERT or RoBERTa models for review analysis. These models, built on transformer architectures, inherently capture all aspects of reviews, including case sensitivity and punctuation. As a result, any additional preprocessing steps or techniques related to these aspects become unnecessary.

However, due to an imbalanced dataset, there is a need to assign class weights to improve performance and reduces bias towards more represented classes [44] [45]. The method for assigning class weights used was inverse class frequency. The inverse class frequency method assigns weights to classes inversely proportional to the number of examples in each class, giving higher weights to classes with fewer examples. The inverse class frequency is typically computed using

$$W_c = \frac{N}{N_c \times K}, \tag{3.1}$$

where, $W_c$ is the weight assigned to the class, $N$ is the total number of samples in the dataset, $N_c$ is the number of samples belonging to the class, $K$ is the number of classes in the dataset.

## 3.2. Architecture of Models

Following the release of the original Transformer model, several language models utilizing the same architecture emerged, incorporating various architectural variations, diverse pretraining methods, and distinct training datasets.

### 3.2.1. BERT

Bi-directional Encoder representations BERT is the most popular transformer model due to its effectiveness in pre-training through a self-supervised procedure using a large corpus, followed by finetuning for specific tasks, resulting in impressive outcomes [1] [46]. Unlike the Generative Pre-trained Transformers (GPT), which employs a stack of $N$ decoder blocks for Casual Language Modeling (CLM) using masked multi-head attention, BERT takes a different approach. Its key innovation lies in its bi-directional nature, enabling the model to consider the context from both sides of a sequence. The objective is to achieve a model with a deeper and more comprehensive understanding of language compared to previous unidirectional models [46]. This allows the model to capture the context of words in a sentence, which is essential for many NLP tasks, including SA.

### 3.2.2. Pretraining

BERT introduces a unique approach called Masked Language Modeling (MLM) for language modeling. MLM involves a "fill-in-the-blank" task, where the model attempts to predict the masked words within a given sequence. For example, if the original sequence is "I like Portuguese food," the model might receive the input "I [MASK] Portuguese food" and aims to predict the masked term.

In BERT, a specific procedure is followed. Given an input sequence $t = [t_1, t_2, \dots, t_N]$ with $N$ tokens, $k$ tokens are randomly chosen from all the tokens, up to 15% of the total. From this random selection, each token can be replaced in one of the following ways:

- 80% of the time, the selected token is replaced with a [MASK] token.
- 10% of the time, a random token is used for replacement.
- The remaining probability leaves the selected token unchanged.

It is important to note that not every token in the sequence undergoes this masking process.

In addition to the MLM pretraining task, Next Sentence Prediction (NSP) was employed during the pretraining of BERT. In this task, BERT is given a pair of sentences (A and B) and attempts to predict whether the second sentence entails the first. The dataset consists of 50% sequential sentence pairs and 50% randomly selected pairs from the Corpus with no entailment relationship. A simple binary classification model is used to solve this problem, to classify the relationship between the two sentences. The input sequence is prepended with a Classification token ([CLS]), and the model makes the final prediction using the representation of the CLS token through a softmax function applied on top of the last encoder layer.

Since BERT takes a pair of sentences, it utilizes both Positional Encoding (PE) embeddings from the original Transformer and segmentation embeddings to indicate whether a token belongs to sentence A or B. A special token ([SEP]) denotes the end of a sentence.

These two pretraining tasks, MLM and NSP, effectively employ multi-task learning. BERT simultaneously learns to represent word information through MLM and semantic sentence information through NSP. Both tasks are trained jointly but in separate layers, with one layer dedicated to each task. The losses of both tasks are combined and back-propagated during training.

### 3.2.3. Input Representation

Like FastText, BERT does not perceive tokens as complete words but as subwords. The input text undergoes a segmentation process using WordPiece [47]. WordPiece, trained separately from BERT on the Corpus, constructs a vocabulary of words. In this vocabulary, the most frequent tokens remain intact (as full words), while fewer common words are broken down into their constituent parts. For example, the word "playing" might be segmented into "play" and "##ing." To obtain accurate embeddings, the model must encounter the word "play" a sufficient number of times. By observing different word forms (e.g., plays, played, play, playing), the model can associate these variants with similar embeddings and better understand the base word "play." However, it is important to note that subword tokens like "##ing" or "##ed" may be associated with multiple verbs, potentially leading to misrepresentations and generating similarities between unrelated words. The input representation of a token is created by adding together the respective token, segment, and position embeddings. A visualization of this construction can be observed in Figure 3.2.

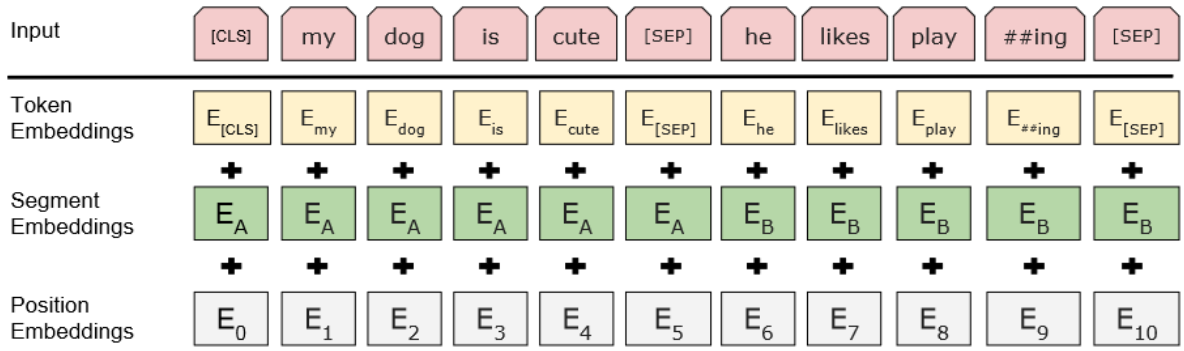



Figure 3.2 – BERT input representation. Segment embedding is added to position and token embeddings to create the input embedding [47].

The original model is a Transformer designed for sequence-to-sequence tasks, such as translating from French to English. It encodes the input sequence into a "machine language" representation and then decodes it to another language. BERT, however, utilizes only the encoder, transitioning from sequence-to-sequence to more of a sequence-to-context approach. The context holds all the condensed information from the "machine language" representation. Consequently, it uses this context to make predictions using a linear layer.

### 3.2.4. Finetuning

After completing the pretraining task, two main approaches are typically followed: finetuning the pre-trained model for a specific task or utilizing the learned output representations

as feature vectors for input in another model. As the primary focus is achieving high performance and no specific architecture requirements exist, finetuning is the preferred option.

This process involves adding new layers on top of the pre-trained model and updating all the layers, including the pre-trained ones, using labeled data from the new task. In this approach, no layers are frozen, and all are updated to adapt the model to the new task.

### 3.2.5. RoBERTa

The architecture of the Robustly Optimized BERT Pretraining Approach, RoBERTa, closely resembles that of BERT. According to the authors of RoBERTa, BERT training was found to be insufficiently extensive [2]. RoBERTa, on the other hand, was trained with a significantly larger amount of data than BERT. While BERT training data consisted of only 16 GB from the Wikipedia English Corpus and BookCorpus, RoBERTa utilized the same Corpus as BERT and an additional 144 GB of new training data, totaling 160 GB. RoBERTa is also trained for longer, from 100K to 500K steps. Moreover, RoBERTa omits the NSP pre-training task employed in BERT's original implementation, as it did not yield significant improvements and sometimes resulted in inferior downstream outcomes [2].

By controlling pretrained data, RoBERTa authors observe that RoBERTa provides a large improvement over the original $\text{BERT}_{\text{Large}}$ that served as an objective in their research. The advancement in pretraining techniques leads to state-of-the-art results in three distinct tasks, sometimes even without the need for multi-task finetuning or additional data [2].

## 3.3. Hyperparameters Optimization

Hyperparameters are external configuration variables used to manage ML models. They are manually set before training a model and differ from parameters internally derived during the learning process. Hyperparameters determine essential aspects such as model architecture, learning rate, and complexity.

Selecting the proper set of hyperparameters is crucial for model performance and accuracy [48] [49]. There are no fixed rules or optimal values for hyperparameters, so experimentation is required based on prior knowledge or heuristics. This process, known as hyperparameter tuning or optimization, allows the adjustment of model performance for optimal results.

Some of the hyperparameters are:

- Maximum length,
- Batch size,
- Epochs,
- Patience,
- Delta,
- Learning rate,
- Number of hidden layers,
- Number of neurons in each hidden layer,
- Activation function,
- Regularization.

Some of these will remain unused due to the utilization of TL, with the final four already pre-defined within the model. However, they may undergo modifications during the finetuning process of the overall model. Patience and Delta serve as two essential parameters regulating the early stopping function. Patience determines the duration to wait after the last observed improvement in validation loss. At the same time, Delta sets the minimum change required in the monitored quantity to be considered an improvement.

Determining the maximum length relies on the particular dataset and task requirements, considering factors such as the tokenizer and available computational resources. It defines the maximum number of words that the model accepts as inputs. Its value is based on the distribution of the dataset's number of words per review. Input with fewer words than is padded to a fixed length, while input with more is truncated to a fixed length. Its value may be changed based on the tokenizer used.

The batch size denotes the number of training examples processed together in each iteration of the training algorithm. This aspect is crucial due to memory limitations associated with larger batch sizes. On the other hand, an epoch signifies a complete iteration through the entire training dataset.

The learning rate involves identifying an optimal value that enables the model to converge effectively and achieve satisfactory performance. Typically, these hyperparameters are mainly determined through heuristic methods.

## 3.4. Evaluation of the Models

In SA, evaluation metrics evaluate an ML model's performance on a text data dataset. These metrics provide a quantitative measure of the model's ability to correctly classify the text data's sentiment.

Many different metrics can be used for SA, depending on the dataset's specific characteristics and the research goals [31]. Some standard metrics [30] used in SA include ACC, precision, sensitivity, specificity, F1 score, and the AUC ROC curve [29] [50]. All metrics used in this work are macro, and all models were validated using a cross-validation process due to the capability to obtain reliable performance metrics and assess the model's ability to generalize to unseen data.

## 3.5. Boosting

In the context of SA, using SAMME in conjunction with ensemble methods and weighted voting is aimed at enhancing performance. In this work, SAMME is developed to incorporate strong learners, and combining ensemble techniques and weighted voting seeks to achieve superior results in SA. The use of strong classifiers is applied in our approach due to the appearance of pre-trained models and improvements in model processing speed, such as Graphics Processing Unit (GPU), opening the possibility of using better algorithms to make more robust ensembles as there is a growing trend in utilizing different ensemble techniques in Portuguese to achieve new state-of-the-art results.

## 3.6. State-of-the-art

The preprocessing of the dataset, due to highly unstructured and noisy data, is something that has been argued recently for the various types of Deep Neural Networks (DNN)

architectures such as Long-Short Term Memory (LSTM), Bidirectional Long-Short Term Memory (Bi-LSTM), and CNN when using BERT embeddings [51].

Due to the limitations posed by static representations, where words are consistently mapped to the same vector regardless of their context, and the subsequent failure to capture the full semantic and syntactic meanings of words in different sentences, the need for deep contextual representations became evident. Researchers sought to address this issue by introducing new models like Embeddings from Language Models (ELMo) [52], Universal Language Model Finetuning (ULMFiT) [53], and InferSent, which adopted a BiLSTM architecture to incorporate context-dependency. However, the transformative moment came with the emergence of transformers [32], quickly becoming the standard in nearly all NLP tasks due to their remarkable efficacy. Large pre-trained models like BERT, RoBERTa, and XLNet are prominent examples of these context-dependent architectures, demonstrating their utility across various NLP tasks [54], namely in SA [55]. According to [55], utilizing a monolingual pre-trained BERT model, specifically in Portuguese, yields superior outcomes to multilingual BERT.

In addition, articles demonstrate that transformer-based models achieve superior results when compared to previous models, such as the case of LSTM and CNN [56]. To achieve better results, it is possible to adjust the hyperparameters in the training of the BERT and RoBERTa models. Some of these experiments have been carried out recently [48]. It is necessary to be careful with these adjustments to avoid problems such as the fading of the gradient or the instability when fine-tuning is done [49].

In ML, ensembles of classifiers are usually built by combining multiple learners (week or strong), following the strategy of boosting rather than relying on a single strong classifier. This idea has gained interest in recent years [57], as it is often easier to train and combine several simple classifiers than to learn a complex one. There is a growing trend in utilizing different ensemble techniques and single-language pre-trained models, such as RoBERTa and BERT, rather than multi-language models to achieve new state-of-the-art results [58] [55] [54].

Gomes et al.[58] in 2022, have employed cutting-edge Transformer models to tackle two specific subtasks within the realm of Aspect Term Extraction (ATE) and Sentiment Orientation Extraction (SOE). Their assertion is that they have attained the highest level of performance in both subtasks, surpassing previously established benchmarks and setting new standards for the Portuguese language.

In the case of ATE, the methodology involved the utilization of an ensemble comprising models from RoBERTa and mDeBERTa, which were trained on Portuguese and multilingual datasets, respectively, achieving 67.1% of ACC. For the SOE subtask, a voting ensemble consisting of PTT5 large models was employed without reliance on external data sources.

Regarding SOE, the optimal outcomes were realized through the utilization of PTT5 Large in conjunction with the conditional text generation training strategy, reaching 82.4% of ACC. This approach entailed presenting the complete review alongside the aspect term as input to the model, all without reliance on any external data sources.

Lopes et al.[55] present an approach designed for extracting aspects from Portuguese-language reviews by employing pre-trained BERT models. They conducted a performance comparison between Google's multilingual BERT and BERTimbau. Remarkably, BERTimbau attains a balanced ACC rate of up to 93% when applied to a corpus of hotel reviews. To assess the efficacy of these models in aspect extraction, the authors employ ACC and F1-score as evaluation metrics, with their findings indicating that BERTimbau outperforms the multilingual BERT model across both metrics.

Furthermore, the authors incorporate an additional step termed post-training. This post-training phase involves enhancing BERTimbau to cater to a specific domain, aligning it with its intended field of application. The initial results, when post-training is not employed, yield an F1-score of up to 70% using polarity auxiliary sentences. The introduction of post-training with polarity sentences yields enhanced performance, with the most noteworthy results reaching a 77% F1-score after 5k and 10k post-training iterations, albeit with a decrease in stability observed after the 10k steps. The model's ACC also reaches up to 80% with post-training.

Moura et al.[54] in their article, evaluates different methods for creating sentence embeddings that can be used to cluster user intents in dialog data. Compares six transformer-based models (BERT, RoBERTa, GPT-2, XLNet, ALBERT, and ELECTRA) for text representation. Also evaluates two pre-trained Siamese transformers (SBERT and SRoBERTa) and studies the impact of retraining them on domain data. Moreover, the article explores the use of ensemble methods to combine different embeddings and clustering algorithms. The article systematically assesses various approaches to generate sentence embeddings aimed at clustering user intents within dialog data. Moreover, the article delves into the exploration of ensemble methods, seeking to amalgamate diverse embeddings and clustering algorithms for enhanced results.

## 3.7. Key Remarks

The method of inverse class frequency provides weight allocations to classes based on their proportion to the number of examples, granting higher weights to classes with fewer instances. This approach effectively balances the dataset, leading to enhanced performance and reduced bias towards classes that are heavily represented.

BERT, building upon the original Transformer, incorporates PE embeddings and segmentation embeddings to handle sentence pairs effectively. Multi-task learning is employed, with the BERT model integrating two specific tasks during its training process, specifically, MLM and NSP. Masked language modeling focuses on predicting masked tokens to derive meaningful word representations, while NSP captures semantic information within sentences. These tasks are concurrently trained in distinct layers, utilizing combined losses for backpropagation and model updates.

The significance of hyperparameters cannot be understated, as they influence various aspects of a model, such as its architecture, learning rate, and complexity. Selecting the appropriate set of hyperparameters is crucial for achieving optimal model performance and ACC. Due to the absence of fixed rules or universally optimal values for hyperparameters, experimentation becomes imperative, often relying on prior knowledge or heuristics. This iterative process, known as hyperparameter tuning or optimization, enables fine-tuning the model's performance to achieve the best possible outcomes.

In SA, standard metrics like ACC, precision, sensitivity, F1 score, and ROC curve are often relied upon to evaluate performance. The metrics employed in this work are of the Macro type, and cross-validation techniques are utilized for model validation. This approach is preferred as it ensures the acquisition of reliable performance metrics and facilitates an assessment of the model's ability to generalize to unfamiliar data.

To enhance SA performance, Adaboost, ensemble methods, and weighted voting are incorporated. Utilizing strong learners due to pre-trained models and faster GPU processing speed further contributes to the model's effectiveness. A notable trend in Portuguese SA involves the utilization of different ensemble techniques, which have been shown to yield superior results.

Researchers have identified a significant research gap marked by the limited number of studies conducted in languages other than English. This presents a valuable research opportunity, which some studies have already begun to explore. This thesis focuses on developing models for SA using Transformers and ensemble techniques. Previous studies have introduced effective model architectures for extracting insights from text reviews. Most notably, most of the research in the State-of-the-Art section centers on advancements in the Portuguese language, a key area of concern in this study. These studies employ diverse approaches, reflecting the evolving landscape of solutions in a field that has seen relatively fewer explorations in the realm of NLP.

# 4. Practical Case

The main dataset used in this work was a collaborative effort between Zomato's technical team and the RRSO project team. The team was able to extract the dataset from Zomato's database and structure using MongoDB, a flexible and scalable NoSQL database. Unlike traditional relational databases that store data in tables, MongoDB stores data as files, making it well-suited for handling large and complex datasets.

## 4.1. Exploratory Analysis

The structured dataset consisted of 537,083 reviews, considering the removal of null lines and two columns, including "text review" and "rating." The text reviews were customer feedback on restaurants, while the ratings were scores assigned to the reviews by the customers. The data was gathered from April 1st, 2014, to September 2nd, 2022, and provided a comprehensive snapshot of the sentiment and opinion of customers towards restaurants over an extended period. This dataset was an essential resource for the project team and provided valuable insights for SA and restaurant review classification. The reviews were labeled with a rating of 1 to 5 in increments of 0.5, assigned by the reviewer.

Looking more in detail, it turns out that about 513K of these reviews are in Portuguese, around 23K in different languages, and around 13K of those in English. No other languages were considered, and test users were removed. The goal here is to regulate the language used. This work aims to evaluate sentiments in Portuguese since little work is done in this language with three classes: negative, neutral, and positive. A translation script was used, based on Google Translate, to translate all reviews to Portuguese, and coding was employed to separate the ratings into the three classes.

It is important to note that no additional preprocessing steps were applied to the datasets apart from those required by the Transformer models. Furthermore, no measures were taken to modify or improve the quality of the translated sentences after they were retrieved from Google Translate.

Figure 4.1 shows, by looking at the frequency of word counts in a dataset, that most reviews used between 30 and 60 words.

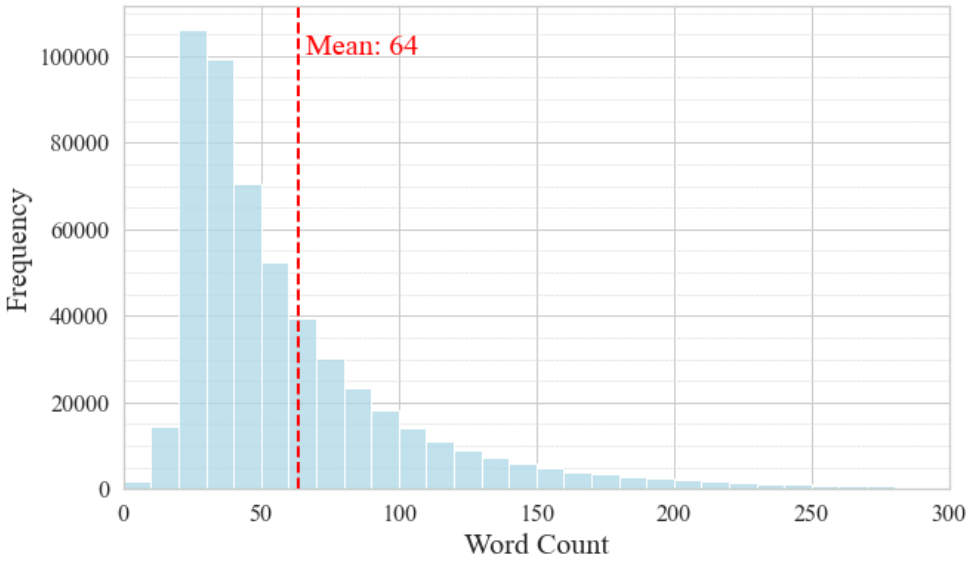



Figure 4.1 – Word count distribution.

Considering the average of 64 words, and the chosen models use all the punctuation, which makes the punctuation count as words, it was decided, heuristically, to make an increment of about a third, thus using a maximum length of 100 words.

### 4.1.1. Data Visualization and Insights

The label distribution is shown in Figure 4.2, and it is heavily imbalanced due to its nature, making it difficult for supervised training. The separation was done based on a heuristic decision [59].

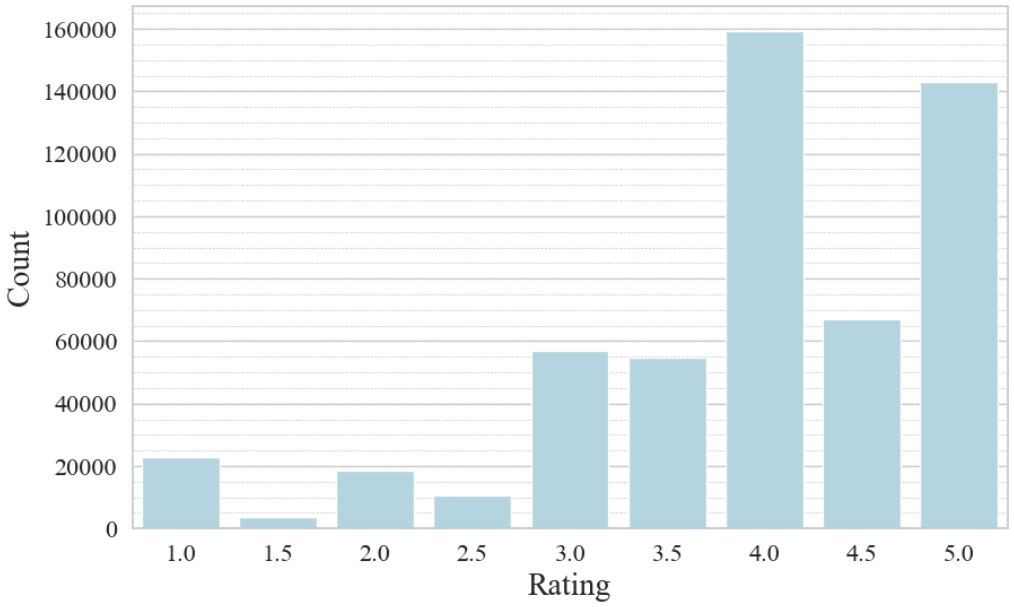



Figure 4.2 – Raw dataset rating distribution.

Most of the work was carried out using 3-class classification models, with the original nine label values being re-sampled to negative (0), neutral (1), and positive (2) classes, as shown in Figure 4.3.

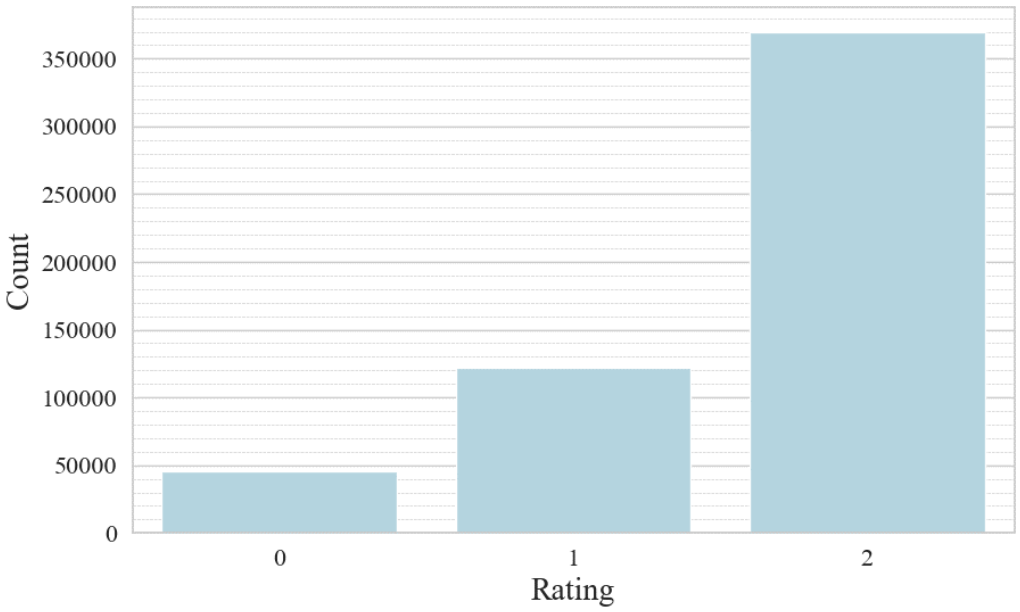



Figure 4.3 – Re-sampled dataset rating distribution.

### 4.1.2. Preprocessing

There was a need to handle an imbalanced dataset because the data were not equally distributed. This can cause a bias towards the more represented classes in the model. The inverse class frequency method is applied to balance the data, assigning weights to each class, with higher weights given to underrepresented classes and lower weights given to more represented classes. This increases the significance of underrepresented classes during the training process, leading to a more accurate capture of patterns in the data.

Figure 4.4 shows the representation of the training dataset and the weights assigned to each class.

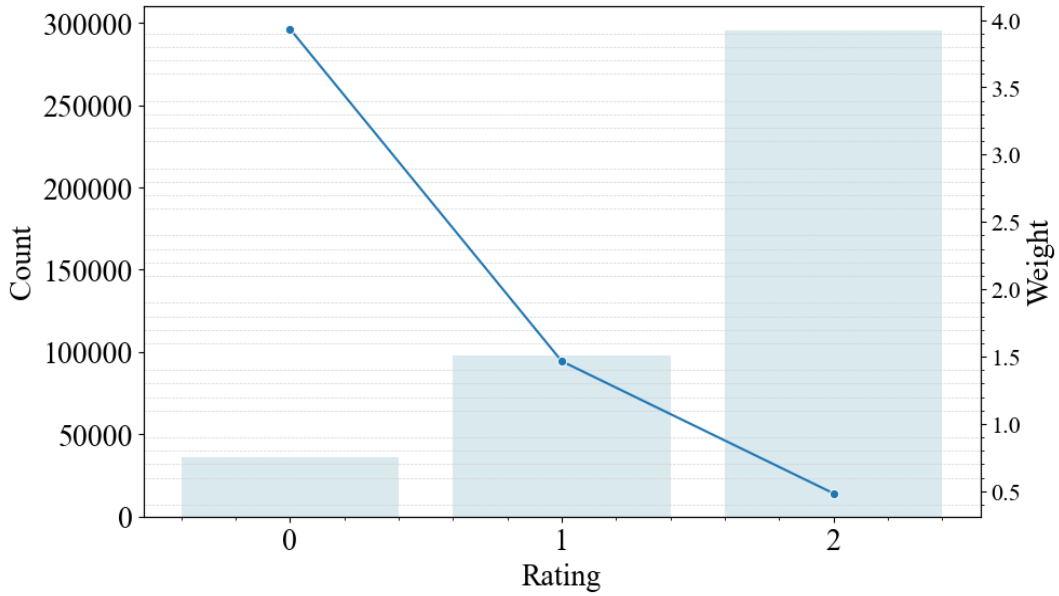



Figure 4.4 – Train dataset weight rating distribution and weight assigned.

The inverse class frequency method, also called class weight, is only applied in the training dataset. In this case, it was used the "sklearn" module "*compute_class_weight*."

## 4.2. Training and Validation

For the development and training of the model to achieve optimal performance, it is essential to understand the parameters of the machine on which it will be trained. This knowledge allows us to ensure the best possible training performance and outcomes. All the training were performed on the GPU RTX 3090, known for its computational power, which is particularly advantageous for complex NLP tasks. The NVIDIA GeForce RTX 3090 is a top-of-the-line graphics card renowned for its outstanding performance in both gaming and professional applications. Powered by the Ampere architecture, it boasts 10,496 CUDA cores and comes with 24 GB of GDDR6X video memory, providing a bandwidth of 936 GB/s. The RTX 3090 particularly shines in real-time ray tracing and AI acceleration, making it compatible with popular frameworks like TensorFlow and PyTorch.

The code was developed using the "Spyder 5.4.1" interface within the "Anaconda Navigator" environment, providing a robust and user-friendly development environment.

### 4.2.1. Training Parametrization

The fact that we have a GPU with great capacity and performance allows us to use larger batches, making training faster. As the models used were of substantial size and based on PyTorch rather than TensorFlow, specifically with BERTimbau, the code evolved to incorporate the latest trends in PyTorch. The primary libraries utilized were "torch, version 1.13," "numpy, version 1.23.5," and "pandas, version 1.5.3," alongside other common ones such as "transformers," "sklearn," "torchmetrics," and "pycm".

Considering the previous chapters, as well as some knowledge shared in the scientific community, the hyperparameters were determined as follows:

- max_len = 100,
- batch_size = 128,
- epochs = 20,
- patience = 10,
- delta = 0.0003,
- dropout = 0.2,
- learning_rate = 2e-5.

Another crucial aspect is data separation, carried out heuristically to prioritize training. This separation was achieved using the "train_test_split" function, which is a part of the sklearn (scikit-learn) library. The training dataset is assigned 80% of the samples, while the remaining 20% is divided into 14% for validation and the remaining 6% for the test dataset. This division emphasizes greater attention to the training process, particularly since the model is pre-trained and requires fine-tuning the output layers.

An essential part when using transformers is setting up a scheduler function for controlling the learning rate during training. The scheduler is used to adjust an optimizer's learning rate over training to improve the model's performance.

In this specific work, the scheduler used is "get_linear_schedule_with_warmup," a function from the "transformers" library commonly used in NLP tasks.

The learning rate starts from 0 and gradually increases during the warmup phase using a linear scheduler with a warmup function. After the warmup, the learning rate follows a linear schedule, typically decreasing as training progresses to help the model converge to an optimal solution.

The warmup was set to 10% of the total steps required to cover all the training samples for the specified number of epochs. This avoids getting stuck in suboptimal solutions.

One of the significant challenges in NLP models is domain-specific biases. As general datasets are difficult to find and train due to their generalized information, NLP models are often trained on single-domain corpora.

The validation is performed using 2-fold cross-validation, which allows for comprehensive assessment and validation of the SA model generalization capabilities.

## 4.3. Sentiment Analysis using BERTimbau

BERTimbau, a language model developed by NeuralMind [60], is built upon BERT architecture but tailored explicitly for the Portuguese language.

Distinctively, NeuralMind adapted the BERT architecture to create BERTimbau with a primary focus on optimizing performance for the intricacies of Portuguese. Notably, the model is case-sensitive, distinguishing between lowercase and uppercase characters, which is particularly significant for Portuguese due to the grammatical information carried by case distinctions.

The pre-training process of BERTimbau involves exposing the model to an extensive and diverse dataset of Brazilian Portuguese texts, BrWaC (Brazilian Web as Corpus), a large Portuguese corpus, for one million steps, using a whole-word mask [11]. By learning from various linguistic contexts, BERTimbau gains a robust understanding of Portuguese grammar, syntax, and semantics. Figure 4.5 shows the graphical representation of the BERTimbau.

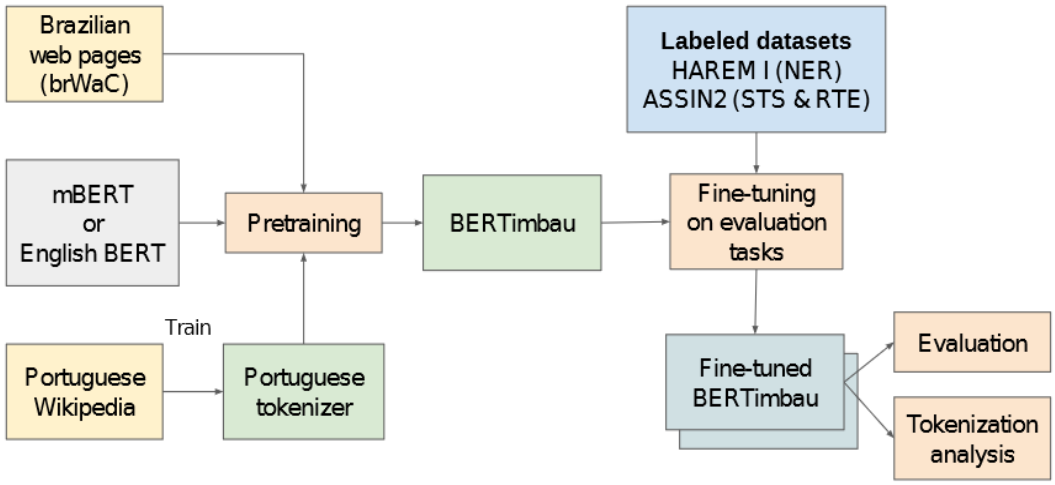



Figure 4.5 – Graphical representation of BERTimbau [11].

BERTimbau was prepared for using TensorFlow and PyTorch, and it is available in two sizes, Base with 12 layers and Large with 24 layers, as represented in Figure 4.6, offering flexibility to cater to different computational needs.

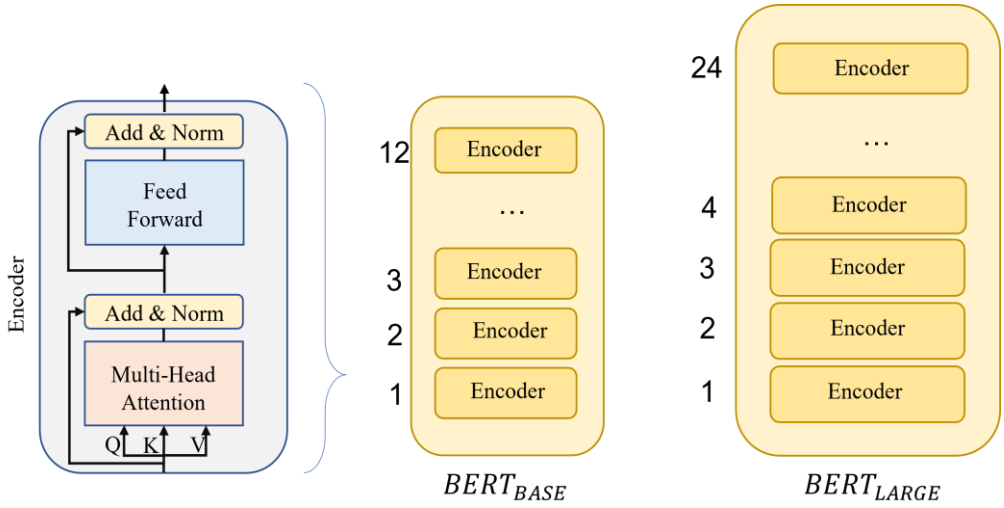



Figure 4.6 - $BERT_{BASE}$ and $BERT_{LARGE}$ encoder stack representation.

TABLE II details insights about these two models, showing that the large model is almost double the size of the base model.

TABLE II - Difference in parameters between $BERT_{BASE}$ and $BERT_{LARGE}$

| | $BERT_{BASE}$ | $BERT_{LARGE}$ |
|---|---|---|
| Layers | 12 | 24 |
| Hidden Size | 768 | 1024 |
| Heads | 12 | 16 |
| Parameters | 110M | 335M |

BERTimbau has achieved state-of-the-art performance across three essential NLP tasks:

- Named Entity Recognition,
- Sentence Textual Similarity,
- Recognizing Textual Entailment [11].

Among the noteworthy features that distinguish BERTimbau, its exceptional adaptability stands out prominently. This adaptability is achieved through the fine-tuning process, even when working with smaller datasets for specific downstream tasks. This remarkable capability empowers the model to not only adeptly adjust to the intricacies of domain-specific requirements but also markedly elevate task performance, resulting in a more finely tailored and effective solution for a variety of applications and domains, all while maximizing its potential with limited data resources.

Thanks to its inclusion in the Hugging Face model repository, BERTimbau is easily accessible. The user-friendly interface and API provided by Hugging Face enable seamless integration and utilization of BERTimbau in a wide range of NLP projects and applications.

The capability of adjusting to domain-specific requirements is explored to produce sentiment classification in this work with three classes, taking advantage of the Portuguese learned by the model in the pre trained process.

In the process of fine-tuning the model, it is crucial to underscore that two additional layers were meticulously engineered, designed, and implemented, reflecting a unique contribution to this research. The first of these layers is a regularization layer, which serves to enhance the model's robustness and generalization capabilities. Following this, a fully-connected dense layer was thoughtfully crafted, contributing to the model's depth and capacity for learning complex patterns. It is essential to emphasize that these layers were not adopted from pre-existing models but were specifically developed as part of this work. For a visual representation of this architecture, please refer to Figure 4.7. These customized layers were then further augmented with a Softmax function to derive the computed predictions.

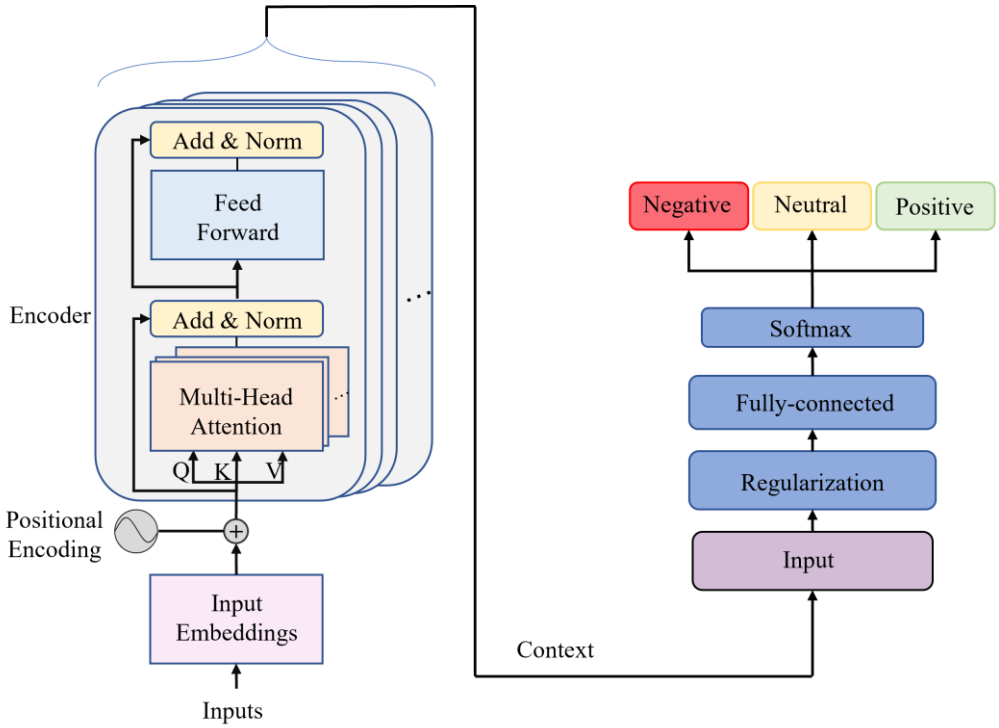



Figure 4.7 – Representation of a fully developed model.

The pre-trained model was directly loaded into the "SentimentClassifier" object, as shown in Figure 4.7. The tokenizer was loaded right after, and the dataset split was performed. A new object called "dataloader" was created, considering the data, the specified length, and the batch size. This ensures that during training, the samples are appropriately sized and allocated. The pre-trained model and the tokenizer used in this process are part of the "transformers" library, providing essential functionalities for NLP tasks. After all this process, the model can be trained for the SA task.

The sequence of layers with the fully detailed BERTimbau architecture is provided below, where the connectivity between layers can be observed, particularly in the output layers.

```
SentimentClassifier(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(29794, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_af-
ine=True)
```

```
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_af-
fine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (drop): Dropout(p=0.2, inplace=False)
  (linear): Linear(in_features=768, out_features=3, bias=True)
)
```

## 4.4. Sentiment Analysis using RoBERTa

RoBERTa is a variant of BERT that involves some adjustments to the critical hyperparameters and minor changes to the embeddings. The architecture developed with the RoBERTa model remains consistent with the original BERT model (Figure 4.6). Consequently, we added the same layers as in the BERTimbau model, including the regularization layer and fully-connected layer, to create the sentiment classifier block, as shown in Figure 4.8, and then applied a Softmax function in the output, as depicted in Figure 4.7.

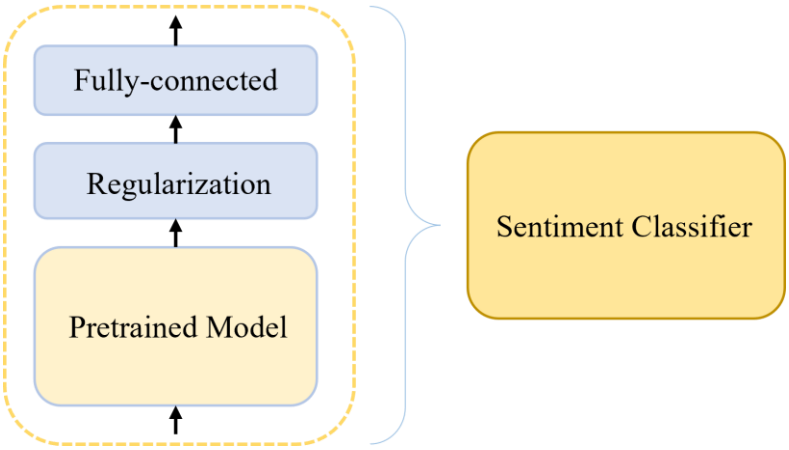



Figure 4.8 – Architecture developed for SA.

This RoBERTa model is available in the Hugging Face model repository by team thegoodfellas [61] and is a fine-tuned version of the xlm-roberta-base on the BrWac dataset. TABLE III presents the base parameters that RoBERTa uses, as in the original paper [2].

TABLE III – Base parameters that RoBERTa uses that are similar to $BERT_{LARGE}$.

| | $RoBERTa_{BASE}$ |
|---|---|
| Layers | 24 |
| Hidden Size | 1024 |
| Heads | 16 |
| Parameters | 355M |

The complete architectural sequence of the RoBERTa model developed, along with detailed layer configurations, is listed underneath.

```
CustomRobertaModel(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(250002, 1024, padding_idx=1)
      (position_embeddings): Embedding(514, 1024, padding_idx=1)
      (token_type_embeddings): Embedding(1, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-23): 24 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=1024, out_features=1024,
bias=True)
              (key): Linear(in_features=1024, out_features=1024, bias=True)
              (value): Linear(in_features=1024, out_features=1024,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=1024, out_features=1024,
bias=True)
              (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_af-
fine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=1024, out_features=4096, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=4096, out_features=1024, bias=True)
            (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_af-
fine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): RobertaPooler(
      (dense): Linear(in_features=1024, out_features=1024, bias=True)
      (activation): Tanh()
    )
  )
  (drop): Dropout(p=0.2, inplace=False)
```

```
  (linear): Linear(in_features=1024, out_features=3, bias=True)
)
```

## 4.5. Boosting Sentiment Classifier Models

The sentiment classifier is composed, as depicted in Figure 4.8, of a pre-trained model and two layers prior to its output. The pre-trained model, whether it be BERTimbau or RoBERTa, is seamlessly integrated and referred to as the sentiment classifier.

Figure 4.9 illustrates the ensemble architecture utilizing the representation of this classifier, considering that it can be implemented with either of the pre-trained models mentioned above. This approach was pursued to ensure the easy incorporation of various pre-trained models, thereby enhancing its versatility for future endeavors.

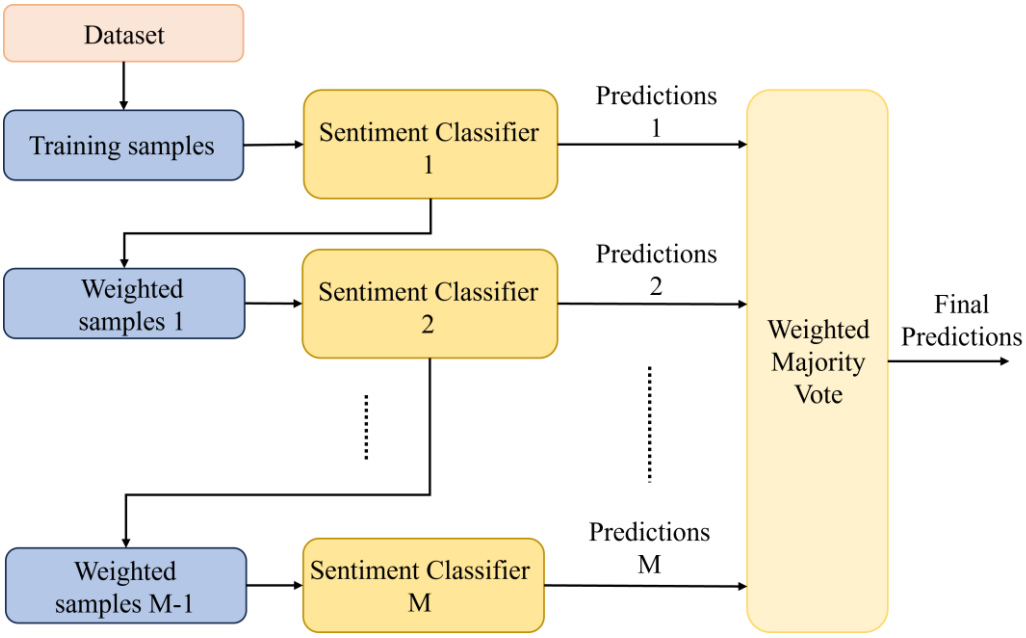



Figure 4.9 – Architecture of the ensemble applied to each sentiment classifier.

In the provided attachments, the detailed Python configurations and implementations are available. While the layer sequence is not explicitly outlined, it is possible to find the script containing the code, providing insights into the intricate connections between the layers, with a particular emphasis on the weighted voting system.

## 4.6. Implemented Model

The following framework delineates the comprehensive, step-by-step process involved in the analysis of a review and the model's interaction with it. This abstraction serves the purpose of clarifying the intricacies of the entire developed process, wherein the textual content of a review undergoes transformation into numerical data. Employing robust computational techniques, this methodology facilitates the training of the model's ability to recognize contextual nuances within the text, ultimately enabling the extraction of underlying sentiments.

Figure 4.10 has been crafted to enhance the comprehension of the review processing procedure. On the left-hand side, it provides a structural representation of the model, while on the right-hand side, it elucidates how the model perceives the textual content of a review, effecting a transformation into a format that holds heightened computational significance.

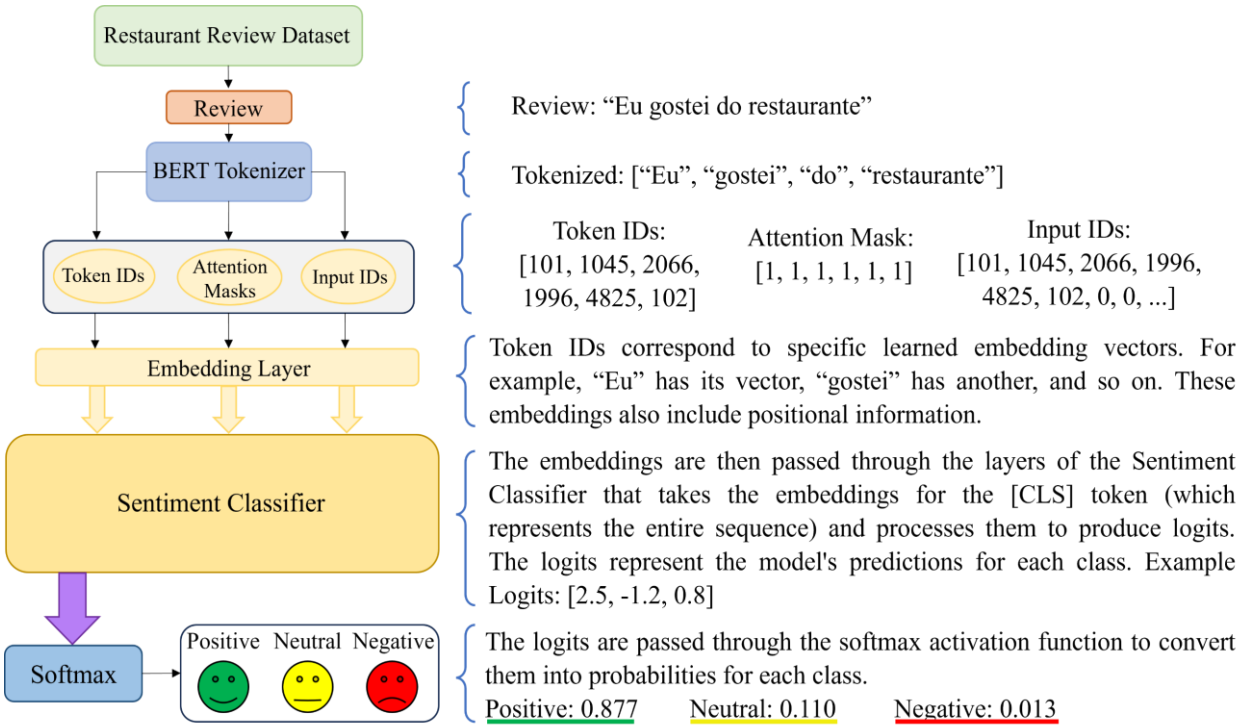



Figure 4.10 – Full model architecture developed.

These probabilities reflect the model's confidence in the input sentence belonging to each sentiment class. In this example, the highest probability is for the "positive" class, suggesting that the model predicts a positive sentiment for the input sentence "Eu gostei do restaurant."

Throughout these implementations, various Python scripts were developed, ranging from the base models to the models incorporating different ensembles and voting mechanisms. These codes are available for reference in the attached documents.

## 4.7. Key Remarks

In this study, we utilized a main dataset obtained through collaboration between Zomato's technical team and the RRSO project team. This dataset, extracted from Zomato's database and structured with MongoDB, a versatile NoSQL database, has been specifically tailored for the management of extensive and intricate data, contributing to the study's success.

To optimize our model's performance during development and training, it is imperative to possess a deep understanding of the underlying hardware parameters. All our training processes were meticulously executed using the high-performance GPU RTX 3090, renowned for its computational prowess, particularly well-suited for the demands of complex NLP tasks.

BERTimbau, thoughtfully crafted by NeuralMind, represents an adaptation of the BERT architecture specifically tailored for the intricacies of the Portuguese language. A notable feature of this model is its sensitivity to case distinctions, a critical element for preserving the grammatical nuances found in Portuguese.

Furthermore, we incorporated RoBERTa, a variant of BERT, into our model. RoBERTa's adaptation introduces essential hyperparameter adjustments and minor embedding changes while retaining the core architecture of the original BERT model. To create the sentiment classifier block, we adopted the same layers employed in the BERTimbau model, encompassing regularization and fully-connected layers, followed by the application of a Softmax function for the final output.

Our ensemble architecture enhances the versatility of the model, facilitating the integration of the classifier's representation with various pre-trained models, ensuring adaptability for diverse future applications.

Lastly, is presented a framework that provides a systematic guide to the intricate process of reviewing analysis and the model's interaction. The detailed abstraction illustrates how textual content undergoes a transformation into numerical data using advanced computational techniques. This process is crucial for training the model to recognize contextual subtleties within text, ultimately enabling the extraction of underlying sentiments.

# 5. Hardware Implementation

This chapter addresses sentiment classification in edge devices, specifically utilizing Jetson Nano, Raspberry Pi, and the RTX 3090 graphics card. SA, a popular application of NLP, involves determining text sentiment, such as positive, negative, or neutral. Performing SA at the edge enables real-time decision-making and enhances privacy by processing sensitive data locally without relying on cloud-based services.

## 5.1. Implementation using Jetson Nano

As a power-efficient and high-performance single-board computer designed for AI applications at the edge, Jetson Nano, a developer kit, offers an ideal platform for real-time sentiment classification.

### 5.1.1. Hardware

The NVIDIA Jetson Nano is a low-cost, small form-factor computer designed for AI and robotics applications. It is powered by a quad-core ARM Cortex-A57 CPU and an NVIDIA Maxwell GPU with 128 CUDA cores, providing high-performance computing capabilities for DL applications.

One of the key advantages of the Jetson Nano is its ability to perform hardware acceleration of ML models using the NVIDIA CUDA architecture. This allows for the real-time processing of large datasets and the high-speed execution of DL models, making it ideal for applications such as SA.

The Jetson Nano also supports a range of AI frameworks, including TensorFlow, PyTorch, and MXNet. She provides access to the NVIDIA JetPack Software Development Kit (SDK) and the NVIDIA Deep Learning Institute (DLI) for training and deployment of DL models [62].

### 5.1.2. Challenges Encountered

The Jetson Nano is a popular choice for edge computing applications due to its compact size, low power consumption, and high computational performance. Its GPIO and I2C interfaces are well-suited for robotics and IoT applications.

However, despite these appealing features for edge computing, it has been observed that the platform requires a specific good quality Micro-USB cable and a power supply that delivers 5 V-2 A at the developer kit's Micro-USB port. Not every power supply can output this required power. Nvidia has as a suggestion the a power supply providing 12.5 W, sometimes requiring even higher power capacities [62]. It uses the power supply of raspberry Pi that provides 5.1 V-3.0 A and 15.3 W. Additionally, it demands substantial cooling, often necessitating an additional fan. Another disadvantage of using this platform is the complexity involved in its main implementation of the setup, mainly done in the command line. For example, during the preparation to run the image of JetPack, after it is written in the microSD card, a machine with a native Linux Operating System (OS) is required, and virtual environments like VirtualBox cannot be used.

When the system boots the first time, the developer kit will guide through some initial setup, including:

- Review and accept NVIDIA Jetson software, EULA - End User License Agreement,
- Select system language, keyboard layout, and time zone,
- Create a username, password, and computer name,
- Select APP partition size, recommending the maximum size suggested.

Furthermore, difficulties were encountered in utilizing the dedicated GPU, as it is constrained to specific versions included in JetPack, rendering it outdated and incapable of using newer versions like Pytorch 1.13, used in this work.

The Jetson Nano wheels provide support for CUDA 10.2, cuDNN 8.0, and NEON. These wheels constitute an integral part of the Python ecosystem, streamlining package installations seamlessly. Their implementation facilitates quicker installation processes and enhances the overall stability of package distribution.

For PyTorch versions 1.11 and above, Python 3.7 is a requisite. Given that JetPack 4.6.1 incorporates Python 3.6, the installation of PyTorch 1.11.0 on a Jetson Nano becomes unfeasible. As of now, it appears that Nvidia has not outlined any intentions to release JetPack 5.0 for the Jetson Nano, limiting its availability to the Xavier series.

Another viable possibility would involve constructing the complete PyTorch framework from scratch, an option that entails a significantly intricate and detailed approach [63].

Therefore, for comparison with its direct competitor, specifically the Raspberry Pi, only the quad-core ARM CPU was utilized for inference, matching the capabilities of the Raspberry Pi.

Regarding the market prices, the utilized Jetson Nano typically costs between 135 € and 150 €. For this implementation, a microSD card is necessary for main storage with a minimum size of 32 GB which is not included in the package. For this work a microSD card of 64 GB of capacity was used.

## 5.2. Implementation using Raspberry PI 4 Model B

Raspberry Pi, a cost-effective and accessible single-board computer, is well-suited for lightweight inference tasks at the edge.

### 5.2.1. Hardware

The Raspberry Pi 4 Model B is a small, low-cost single-board computer powered by a quad-core ARM Cortex-A72 CPU and a Broadcom VideoCore VI GPU, which provide high-performance computing capabilities for various applications, including ML.

One of the key advantages of the Raspberry Pi 4 is its ability to perform hardware acceleration of ML models using the Google Coral Edge TPU or the Intel Neural Compute Stick 2. This allows for the real-time processing of large datasets and high-speed execution of DL models, making it suitable for applications such as SA.

The Raspberry Pi 4 also supports a range of AI frameworks, including TensorFlow, PyTorch, and Keras, and provides access to software development kits and training resources for developing and deploying DL models [64].

### 5.2.2. Challenges Encountered

This platform is well-designed to be multifunctional and adaptable, depending on the user's intended tasks. It comes with a user-friendly OS, often utilizing graphical interfaces, simplifying and expediting the configuration process, as experienced in our case.

The OS installation is straightforward and swift, thanks to the software imager, specifically designed to be as user-friendly as possible, particularly for beginners.

Regarding market prices, the used Raspberry Pi 4 can be acquired for around 80 €. It uses a quality power supply specifically for this model of Raspberry Pi, providing the platform with 5.1 V-3.0 A and 15.3 W. Related to storage, the recommendation is a microSD card with 8 GB or greater for the OS. In this work, it was used a 16 GB microSD card for OS and storage.

## 5.3. Overall Assessment

In this subchapter, we will delve into a comparative analysis of the Raspberry Pi 4 and the Jetson Nano by examining a detailed table showcasing their specifications. TABLE IV provides a convenient and concise overview of the key technical aspects of both devices, facilitating a thorough understanding of their individual strengths and capabilities.

TABLE IV – Specifications of Raspberry Pi 4 and Jetson Nano platforms.

| | **Jetson Nano** | **Raspberry Pi 4** |
|---|---|---|
| CPU | ARM Cortex-A57 | ARM Cortex-A72 |
| Type | 64 bit | 64 bit |
| Cores | 4 | 4 |
| Frequency | 102 MHz – 1.43 GHz | 600 MHz – 1.5 GHz |
| L1 cache | 32 kB | 32 kB |
| L2 cache | 2 MB | 1 MB |
| Memory | 4 GB | 4 GB |
| Memory Type | LPDDR4 | LPDDR4 |
| GPU Architecture | Maxwell | VideoCore VI 3D |
| GPU cores | 128 | - |

### 5.3.1. Operating System

Raspberry Pi utilizes Raspberry Pi OS as its official operating system, which is a Linux distribution based on Debian. Despite the Raspberry Pi 4 featuring a 64-bit CPU, its official OS remains 32-bit (armv7l). In contrast, the Jetson Nano is bundled with a 64-bit Ubuntu 18.04.4 LTS OS, utilizing the aarch64 architecture.

### 5.3.2. CPU

When considering CPU level, both the Raspberry Pi 4 and the Jetson Nano present comparable outcomes. This similarity in CPU showcases how both devices are designed to meet the demands of a variety of applications.

### 5.3.3. Memory

Jetson Nano holds an edge with an additional megabyte of L2 cache. Consequently, when it comes to memory-related aspects, the Jetson Nano emerges as the frontrunner in this comparison.

## 5.4. Key Remarks

The Raspberry Pi 4 offers a cost-effective alternative compared to the Jetson Nano. It comes equipped with built-in Wi-Fi and Bluetooth functionalities, enhancing its connectivity options right out of the box. Additionally, Raspberry benefits from a well-established community and an extensive array of tutorials that facilitate a seamless learning curve for users. However, it is worth noting that Raspberry lags the Nano in terms of memory performance, and despite its 64-bit hardware architecture, the official OS remains confined to a 32-bit system.

On the contrary, the Jetson Nano stands out with a robust GPU that holds the potential to greatly accelerate ML tasks for beginners in this field. However, for projects that do not need to run heavy ML models, Raspberry Pi 4 is likely powerful enough and cheaper.

# 6. Results and Discussion

This chapter will comprehensively present the main results derived from the proposed solution. Following the presentation, a discussion will be provided to delve deeper into the implications and findings of the results. Various scripts were developed, including those for training different models and for inference on edge computing platforms.

For a better understanding, we will provide a concise explanation of how the classifier was created using the pre-trained BERTimbau model, with the assistance of script segments, starting from the initial stages up to the pre-training phase.

As usual, the first step, shown in the SCRIPT I, involves importing the necessary libraries.

SCRIPT I – Imported libraries.

```
# Import libraries:
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from torch import nn
from torchmetrics import AUROC
from pycm import ConfusionMatrix
from transformers import logging
from time import time, strftime, gmtime
from sklearn.metrics import classification_report
from sklearn.utils.class_weight import compute_class_weight
from transformers import BertModel, BertTokenizer, get_linear_sched-
ule_with_warmup
```

Here is TABLE V listing the libraries used in the code along with their versions and a brief justification for using each one of them:

TABLE V – Primary libraries used.

| Library | Version | Justification |
|---|---|---|
| torch | (1.12.1) | Torch is a powerful deep learning framework. It is used for building and training NN in the code. |
| torchmetrics | (1.1.0) | TorchMetrics is used for computing metrics during model training, and it simplifies metric calculation and reporting. |
| numpy | (1.23.5) | NumPy is essential for numerical operations and handling arrays, commonly used for data manipulation in Python. |
| pandas | (1.5.3) | Pandas is used for data manipulation and analysis, often employed for loading, preprocessing, and analyzing datasets. |
| matplotlib | (3.7.0) | Matplotlib is a popular plotting library in Python, useful for creating visualizations and plots of data. |
| transformers (Hugging Face) | (4.24.0) | The Transformers library is widely used for natural language processing tasks, providing pre-trained models and tools. |
| PyCM | (3.9) | Is chosen for its specialized capabilities in generating confusion matrices and detailed classification metrics |
| time (gmtime, strftime) | (3.10.9) | The time module provides functions for time-related operations, which can be helpful for logging and timing code. |
| sklearn.metrics | (1.2.1) | Scikit-learn (sklearn) is a versatile library for machine learning, including tools for metrics and model evaluation. |

The selection of these libraries is guided by their effectiveness in machine learning model development, data manipulation, and reporting, ensuring that the code is suitably equipped to facilitate efficient model training and evaluation. It is relevant to highlight the existence of a suite of scripts containing functions that have been developed and implemented as integral components of our ongoing processes. These scripts, meticulously crafted to enhance various aspects of our operations, serve as valuable tools.

Scripts such as “earlystop.py,” designed to control overfitting by saving the best model checkpoint, “utils.py,” housing functions such as “train_epoch” for epoch-based training, “eval_model” for model evaluation, “get_predictions” for probability-based predictions, and “create_data_loader,” responsible for preprocessing raw text data by tokenizing it and structuring it into batches suitable for training a NN model, deserve particular mention.

Here is TABLE VI listing the locally created libraries used in the code, along with a brief justification for using each of them:

TABLE VI – Locally created libraries.

| Library | Justification |
|---|---|
| PytorchTools (utils.py) | This script contains utility functions that streamline PyTorch operations. It includes custom functions for data preprocessing, model training, or other PyTorch-related tasks. |
| EarlyStop (earlyStop.py) | The EarlyStop module is used for implementing early stopping during model training, which helps prevent overfitting and saves training time. |
| PytorchTools (getMetrics.py) | This script contains functions for obtaining and displaying training metrics, enabling detailed analysis of model performance during training. |
| PytorchTools (mySaveMetrics.py) | The mySaveMetrics module is used for custom saving of metrics or model checkpoints during training. It allows you to save specific metrics or information that are not covered by standard model checkpointing. |

Let us consider the example of one of the developed classes as illustrated in the SCRIPT II that elucidates the Python code defining a custom dataset class named "ReviewDataset." This class comprises three distinct components:

1. Constructor (__init__ method):

The constructor takes four parameters:

reviews: A list or array containing textual reviews.

targets: A list or array containing corresponding target values (e.g., ratings or labels) for each review.

tokenizer: A tokenizer object, used to convert text data into numerical representations.

max_len: An integer indicating the maximum length to which the input text should be truncated or padded.

2. __len__ method:

This method is defined to return the total number of samples in the dataset, which is equal to the number of reviews. It's used by data loaders to determine the dataset's size.

3. __getitem__ method:

This method is defined to retrieve a single data sample at a given index (item).

It starts by extracting the review text and its corresponding target value at the specified index. Then, it uses the tokenizer to encode the review text. It sets various parameters like adding special tokens, limiting the maximum length, and returning tensors in PyTorch format.

The method returns a dictionary containing the following components:

'review_text': The original review text as a string.

'input_ids': The encoded input IDs (tokenized text) as a 1D tensor.

'attention_mask': The attention mask tensor, which indicates which tokens are real and which are padding.

'targets': The target value (e.g., rating or label) as a tensor of type torch.long.

SCRIPT II –Creation PyTorch Dataset.

```
# Building blocks required to create a PyTorch dataset:
class ReviewDataset:
  def __init__(self, reviews, targets, tokenizer, max_len):
    self.reviews = reviews
    self.targets = targets
    self.tokenizer = tokenizer
    self.max_len = max_len

  def __len__(self):
    return len(self.reviews)

  def __getitem__(self, item):
    review = str(self.reviews[item])
    target = self.targets[item]

    encoding = self.tokenizer.encode_plus(review,
                                          add_special_tokens=True,
                                          max_length=self.max_len,
                                          return_token_type_ids=False,
                                          pad_to_max_length=True,
                                          return_attention_mask=True,
                                          return_tensors='pt')
    return {'review_text': review,
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'targets': torch.tensor(target, dtype=torch.long)}
```

This “ReviewDataset” class serves as a bridge between raw text data and machine learning models. It allows to easily preprocess and organize textual data into a format suitable for NLP models by using a specified tokenizer and handling the necessary encoding, padding, and conversion to PyTorch tensors.

The next step is depicted in SCRIPT III in which certain hyperparameters were defined, along with the specification of paths, the name model to import, among other aspects:

SCRIPT III – Variables and hyperparameters definition.

```
# Save name for metrics and checkpoint:
save_name = 'SentAnalysisPt' # change the name to avoid overwrite
save_metrics = True # save the best model parameters
save_model = True # save the model

# Hyperparameters:
max_len = 100
batch_size = 128
epochs = 20
patience = 10
delta = 0.0003
Dropout = 0.2
l_r = 2e-5
```

```
# Set Path for save checkpoint:
path = './NewBert/SentimentAnalysisClassPt/'
# set path for save the model:
path_model = path + 'Model_' + save_name + '/'

# Language
lang = 'Pt'
num_class = 3
data_path = './Data/FinalData' + lang + '/'
class_names = ['Negative', 'Neutral', 'Positive']

# Set the name to import the model:
model_name = 'neuralmind/bert-base-portuguese-cased'
```

Following these definitions, it is time to prepare the dataset for loading into the classifier. We proceed with the following SCRIPT IV section.

SCRIPT IV – Preparation of data to the classifier.

```
# Load data 80% of dataset for training
df_train = pd.read_csv(data_path + 'trainData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 70% of the 20% remaining of dataset for valifation
df_val = pd.read_csv(data_path + 'valData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 30% of the 20% remaining of dataset for testing
df_test = pd.read_csv(data_path + 'testData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)

# Load Tokenizer:
tokenizer = BertTokenizer.from_pretrained(model_name) # BertTokenizer

# Creating Data Loader:
train_data_loader = ut.create_data_loader(df_train, tokenizer, max_len,
batch_size)
val_data_loader = ut.create_data_loader(df_val, tokenizer, max_len,
batch_size)
test_data_loader = ut.create_data_loader(df_test, tokenizer, max_len,
batch_size)
```

We have now arrived at the creation of the classifier. In a straightforward manner, we can observe in SCRIPT V that this code defines a sentiment classification model based on BERT. It initializes the BERT model, optionally restricts parameter updates, introduces dropout for regularization, and incorporates a linear layer for classification. The forward method delineates the data flow through the model during both inference and training.

SCRIPT V – Creation of the classifier with BERTimbau model.

```
# Create a classifier that uses the BERT model
class SentimentClassifier(nn.Module):
  def __init__(self, n_classes, model_name):
    super(SentimentClassifier, self).__init__()
    self.bert = BertModel.from_pretrained(model_name, re-
turn_dict=False)
    # Freeze BERT parameters:
    if freeze_bert:
        # freeze all the parameters excluding classifier
        for param in self.bert.named_parameters():
            # param = (nome_da_layer, parametros)
            param[1].requires_grad = False
    self.drop = nn.Dropout(Dropout) # layer for regulation
    #The last_hidden_state is a sequence of hidden states of the last
layer of the model
    self.linear = nn.Linear(self.bert.config.hidden_size, n_classes)
  def forward(self, input_ids, attention_mask):
    _, pooled_output = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask)
    output = self.drop(pooled_output)
    output = self.linear(output)
    return output
```

The option to freeze the parameters of the pre-trained loaded model allows us to decide between not training the existing parameters and training all of them. Since we are dealing with a classification task that differs from what the model has encountered previously, training will be conducted for all parameters in the context of this new task.

After the creation of the classifier, training parameterization is addressed as per the following SCRIPT VI.

SCRIPT VI – Set training parametrization.

```
#%% Full model to GPU and compute weights:
model = SentimentClassifier(len(class_names),model_name)
model = model.to(device)
data = next(iter(train_data_loader))
input_ids = data['input_ids'].to(device)
attention_mask = data['attention_mask'].to(device)

# Train parameters optimizer:
# optimizer = torch.optim.AdamW(model.parameters(), lr=l_r)
optimizer = torch.optim.Adagrad(model.parameters(), lr=l_r)

# Scheduler function:
total_steps = len(train_data_loader) * epochs
warmup_steps = total_steps*0.1 # 10% de total_steps
scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=warmup_steps, num_training_steps=total_steps)
```

```
# Compute the Class Weights:
class_weights = compute_class_weight(class_weight = "balanced",
                                     classes = np.unique(df_train['rating']),
                                     y = df_train['rating'])
class_wts_dic = dict(zip(np.unique(df_train['rating']), class_weights))
class_wts = [class_wts_dic[0]]
for index in range(1,np.count_nonzero(np.unique(df_train['rat-
ing']))+1):
    class_wts.append(class_wts_dic[index])

# Loss function:
loss_fn = nn.CrossEntropyLoss(weight = torch.ten-
sor(class_wts,dtype=torch.float)).to(device)
```

Following these processes, training can then be initiated, with due attention to saving and monitoring the epochs.

The structure used to store the data of the different models is depicted in Figure 6.1. This structure is applied to all the models being represented here for one model. A script named "getMetrics.py" was created to read all the metrics when needed. These developed full scripts are available in the attached documentation.

```
/
├── NewBert/
│   ├── SentimentAnalysisClassPt/
│   │   └── Model_SentAnalysisPt/
│   │       ├── ConfusionMatrix/
│   │       │   ├── Class.pkl
│   │       │   ├── CM.pkl
│   │       │   ├── FinalAUC.pkl
│   │       │   └── Overall.pkl
│   │       ├── Training/
│   │       │   ├── ACC.pkl
│   │       │   ├── AUC.pkl
│   │       │   ├── BestEpoch.pkl
│   │       │   ├── LOSS.pkl
│   │       │   └── Time.pkl
│   │       ├── ExportFile.txt
│   │       ├── ExportFileText50.txt
│   │       ├── ExportFileText100.txt
│   │       ├── log.txt
│   │       └── modelSentAnalysisPt.pth
│   ├── bert_model.pth
│   ├── checkpoint_SentAnalysisPt.pth
│   ├── loaderSentAnaly.py
│   ├── sentAnalyPt.py
│   └── tokenizer.pth
```

Figure 6.1 – Representation of model structure display for storing metrics.

In the "NewBert" directory, all the developed models using BERTimbau are stored (Figure 6.1 only shows the "SentimentAnalysisClassPt" model). Inside it, there is a directory that holds the training and validation metrics for the respective model, along with five files. Two of these files are Python scripts (.py) related to training execution and model loading. The

remaining three files (.pth) relate to the initial weights, tokenizer, and the weights that achieved the best model performance.

Within the individual directory for each model, in this case, the "Model_SentAnalysisPt" directory, there are two folders, "ConfusionMatrix" and "Training," along with five files. The "ConfusionMatrix" folder stores evaluation parameters for the model, while the "Training" folder holds training parameters. The text files (.txt) consist of three files containing inference results within established restricted parameters and one log file for the project. The last file stores the best model, along with its corresponding weights, found during training. This file is placed within the model's directory to simplify the computation on different platforms, avoiding the need to go through the entire process of loading the model, then the weights, then the checkpoint, and only then performing inference.

## 6.1. Model Development

In this subchapter, the results of training and validation of the different classifiers are displayed.

### 6.1.1. Initial Results

An initial set of results was obtained to establish a baseline for comparison and measure the improvements made in subsequent sections. Tests were conducted using BERT pre-trained for spam classification in English, resulting in an ACC of 71%. However, the English testing data showed a highly imbalanced F1-Score in the classes.

Further testing was performed using the same model but with Portuguese data. As expected, the ACC decreased to 43%, highlighting the importance of using a pre-trained model in Portuguese to match the dataset specifications. To validate this, the full restaurant review dataset was translated to English, and the ACC returned to 70%, confirming that a Portuguese pre-trained model is more suitable, as mentioned in subchapter 3.6. For the objective of a Portuguese model, it is crucial to use a pre-trained model in Portuguese due to its compatibility with the dataset and achieving better results.

For the first train and as a starting point, using the BERTimbau model, which was pre-trained in Portuguese, 60 thousand samples were used that were previously balanced and used the following hyperparameters based on SOTA:

- max_len = 100,
- batch_size = 128,
- epochs = 200,
- patience = 10,
- delta = 0.005.

These parameters were arbitrarily selected to be aligned with default values for the problem at hand. The results show, looking at Figure 6.2 and Figure 6.3, that the model is overfitting, and as it already uses an early stop function (monitoring the loss value), the model stops at epoch 77 instead of continuing overfitting until it reaches 200 epochs that were previously defined.

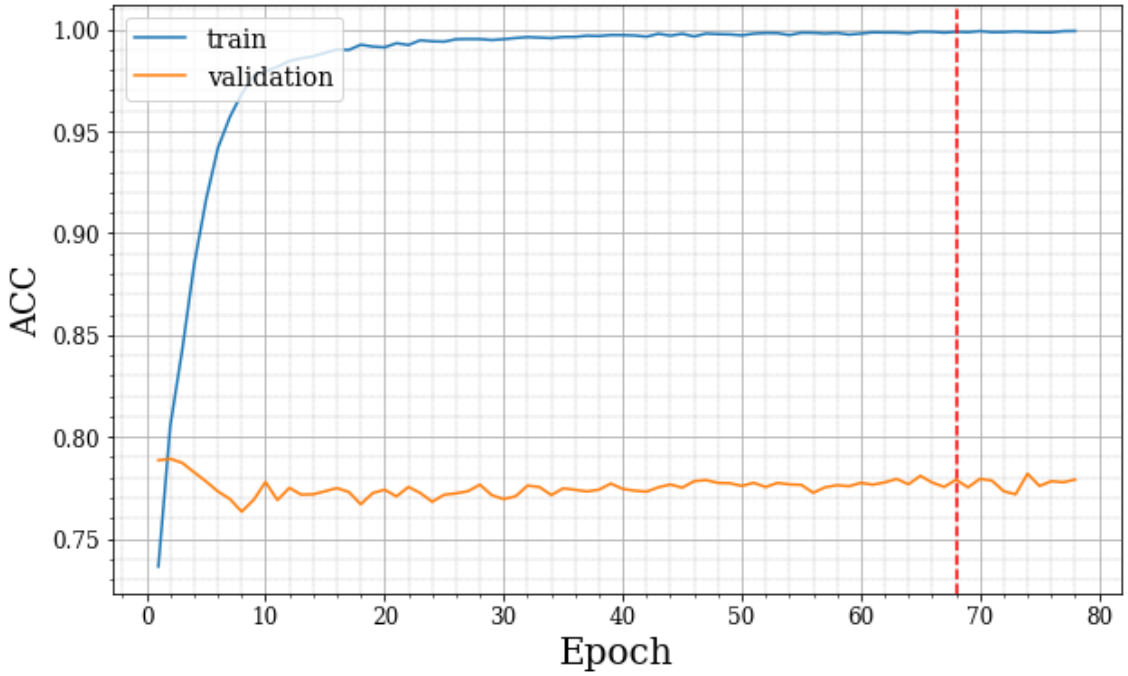



Figure 6.2 – Train history of ACC metric in base Portuguese model.

Looking at the loss graph (Figure 6.3), the model decreases the loss value only until about the two epoch and then start to increase the losses.

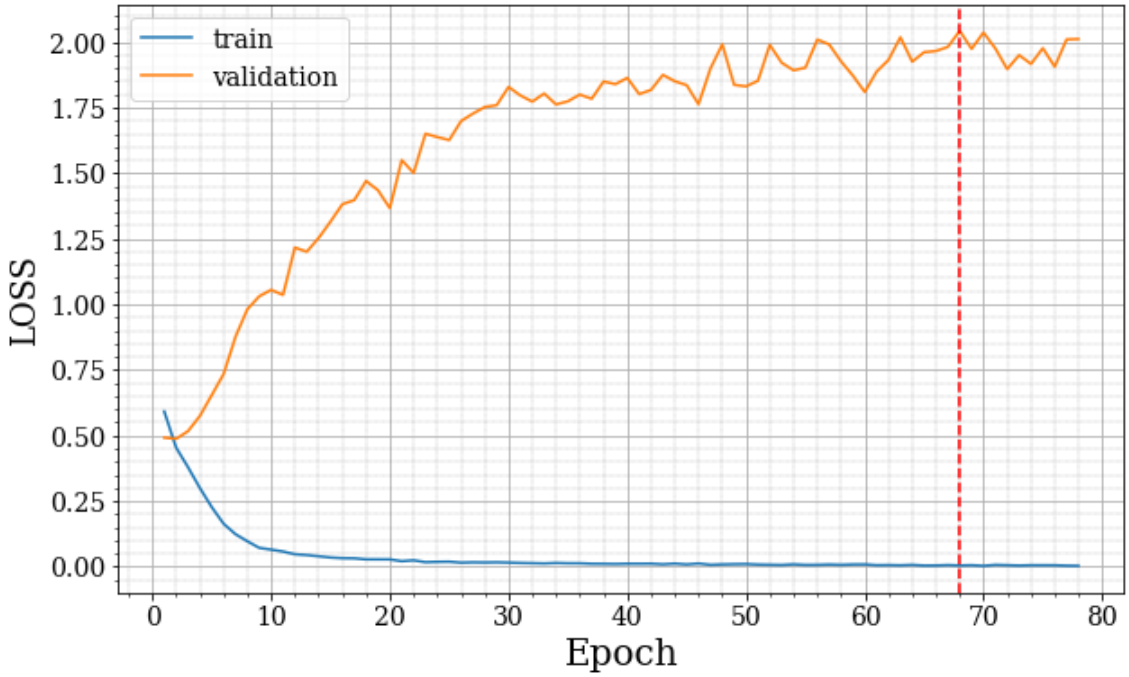



Figure 6.3 – Train history of LOSS metric in base Portuguese model.

The CM can be observed in Figure 6.4, where it has been normalized, and color coding has been applied to make it darker according to the value, making it easier to read.

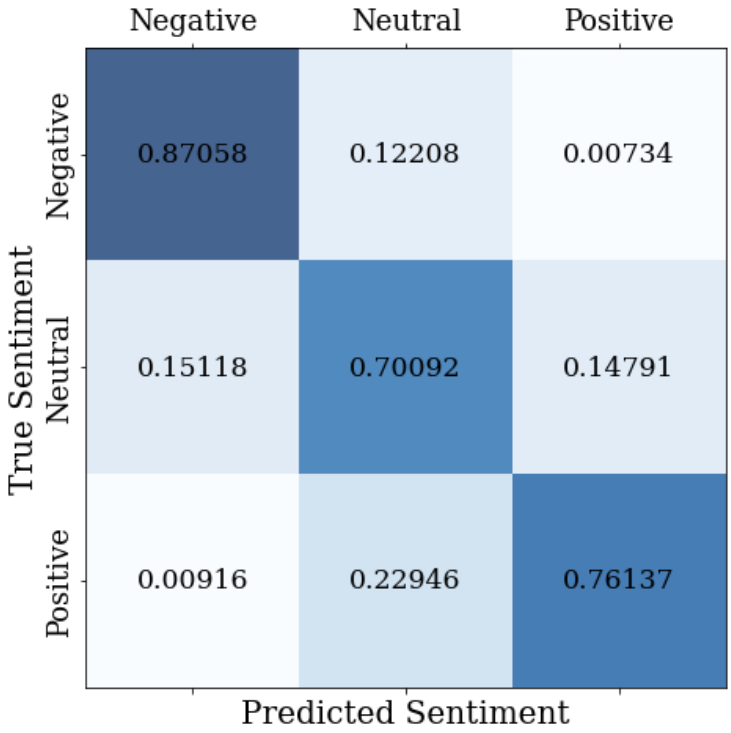



Figure 6.4 – CM of the base Portuguese model using BERTimbau.

Since this test used a sample from the dataset, and it is balanced, it is possible to verify that BERTimbau is a promising model, achieving 77% ACC without additions. Looking at TABLE VII, we see very similar F1-Score and Sensitivity values, indicating a well-balanced model.

TABLE VII – Results of the base model train using BERTimbau.

| **Architecture** | **ACC** | **Avg. F1-Score** | **Avg. Sensitivity** | **Avg. Specificity** | **Avg. Precision** |
|---|---|---|---|---|---|
| BERTimbau (Base model) | 0.777 | 0.778 | 0.777 | 0.888 | 0.780 |

Now that we have a base model to serve as a reference let us fine-tune the hyperparameters, the scheduler, and the optimizer. We will also apply inverse frequency balancing to balance the dataset and search for the best values using a dataset of 60 thousand samples.

Different learning rates were applied ($3e^{-9}$, $3e^{-6}$, $2e^{-5}$, and $1e^{-4}$) and tested two optimizers, specifically, AdamW and AdaGrad.

These tests led to the determination of the best learning rate ($2e^{-5}$) and the optimal optimizer for the developed architecture, which is AdaGrad. The TABLE VIII displays the metrics per class for the tests performed with the optimizer.

TABLE VIII – Optimizer and learning rate tests. Values per class.

| | AdamW | | | | AdaGrad | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Sensitivity | F1-Score | CM | Precision | Sensitivity | F1-Score | CM |
| Neg. | 0.79 | 0.77 | 0.78 | **0.97** | 0.70 | 0.84 | 0.76 | **0.84** |
| Neu. | 0.67 | 0.59 | 0.63 | **0.59** | 0.58 | 0.68 | 0.63 | **0.68** |
| Pos. | 0.89 | 0.92 | 0.91 | **0.92** | 0.92 | 0.85 | 0.89 | **0.84** |

The results displayed in TABLE VIII suggest that AdaGrad is the preferable choice, as it delivers better-balanced outcomes across classes despite yielding lower values in the CM.

During these tests, the model was stopped approximately at epoch 33, accounting for the ten tolerance epochs specified by the early stop function. Considering this, we decided to reduce the number of epochs to 30 and monitor the AUC ROC instead of losses using the early stop function. Additionally, for the scheduler, we implemented an initial warmup covering 10% of the total steps, as the model achieved its best performance in the initial training epochs.

### 6.1.2. BERTimbau

Applying the changes mentioned in 6.1.1 and after adding layers and tuning hyperparameters, it was possible to reach 80% ACC, and looking at the ACC charts, shown in Figure 6.5, we can see that the best epoch was 20 due to the monitorization of AUC ROC, presented in Figure 6.6, although it is visible that from epoch eight the model begins to have difficulty evolving more.

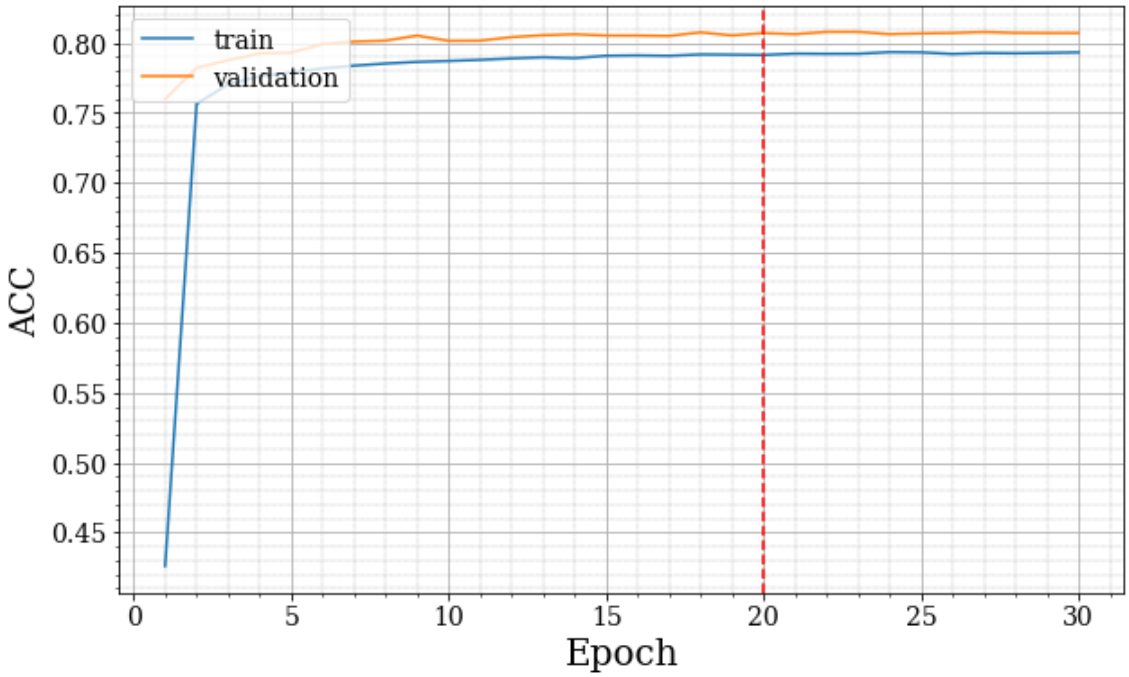



Figure 6.5 – Train history of ACC metric in SA Portuguese model, using BERT.

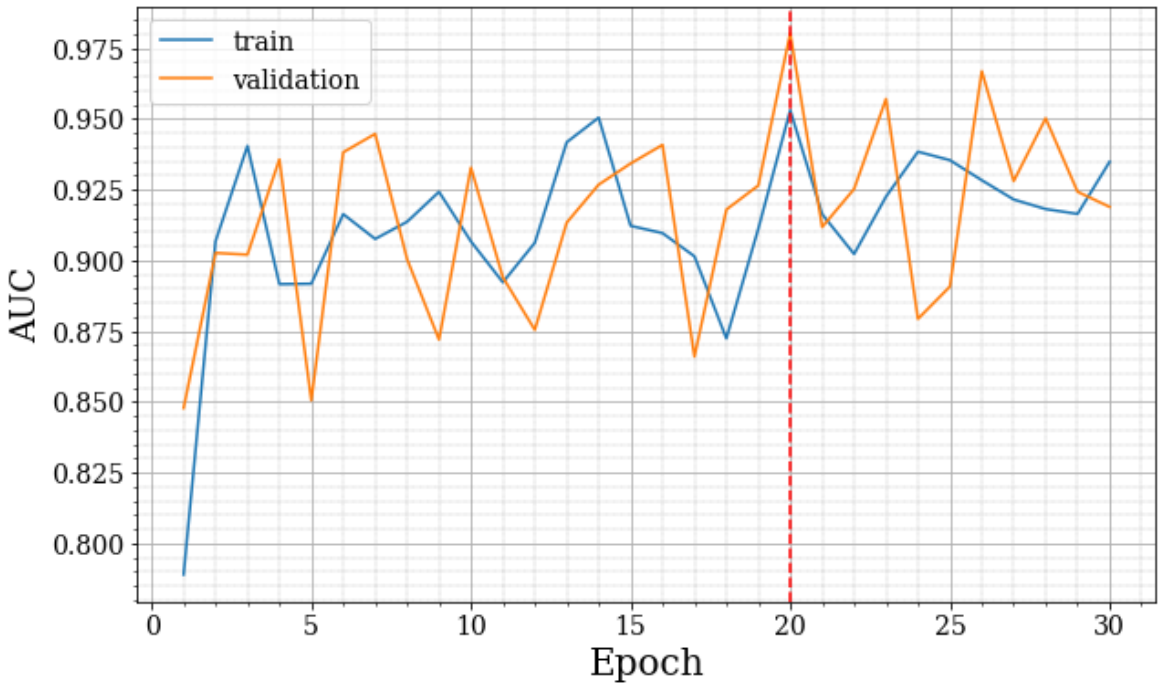



Figure 6.6 –Train history of AUC metric in SA Portuguese model, using BERT.

As for the graph that represents the losses, presented in Figure 6.7, we do not see the model with the erroneous behavior, as was the case of the base model (Figure 6.3), and from epoch 14, the loss stabilizes.

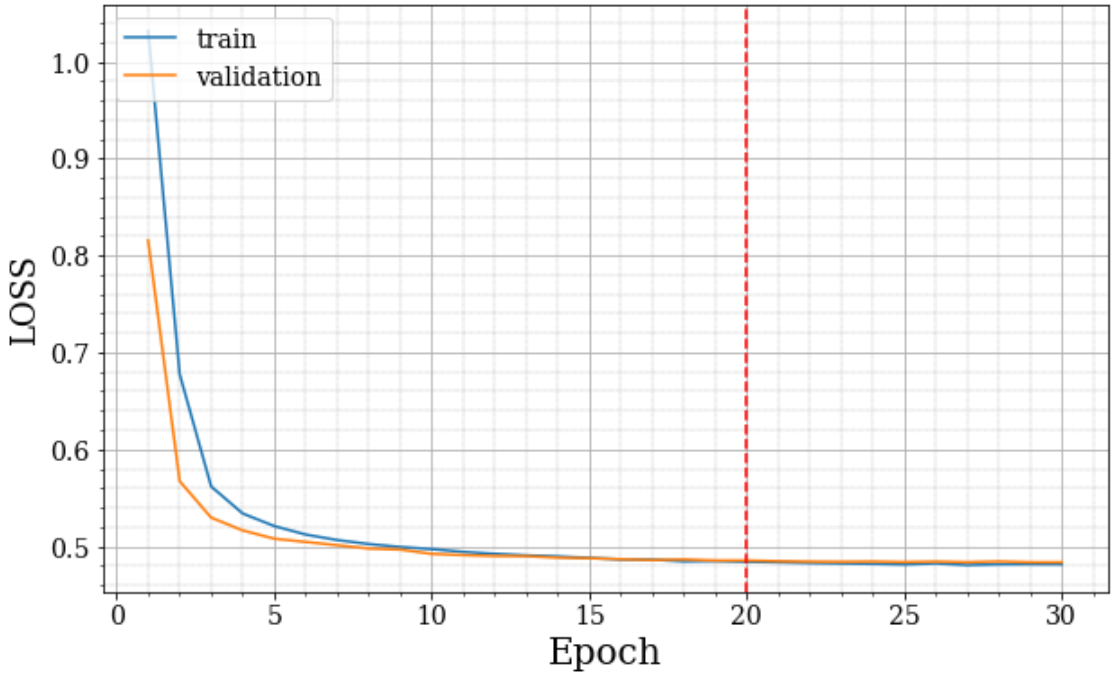



Figure 6.7 - Train history of LOSS metric in SA Portuguese model, using BERT.

The CM in Figure 6.8 has better ACC than the base model due to a more balanced class distribution, ensuring the model performs better across all classes.

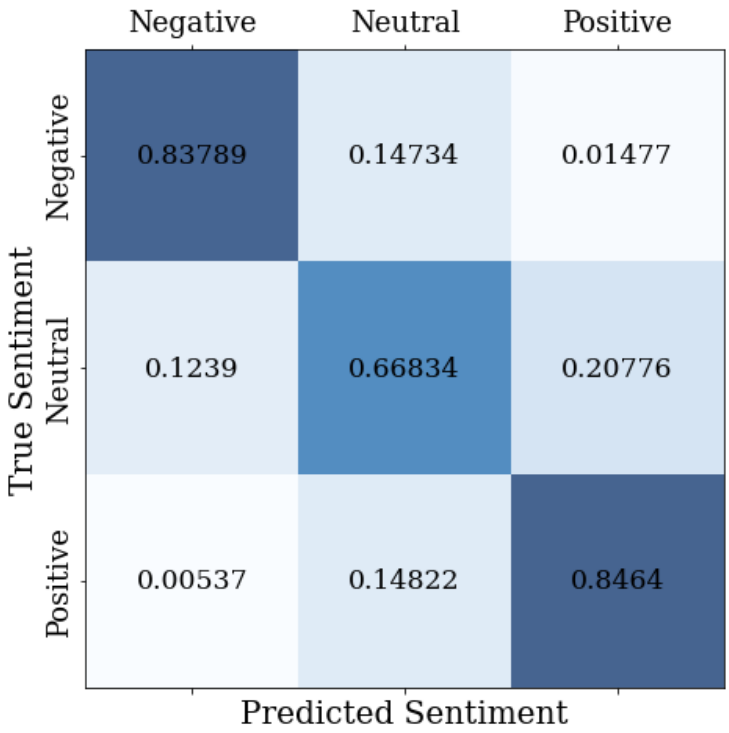



Figure 6.8 - CM of the SA Portuguese model, using BERT.

The values of the more detailed metrics are represented in TABLE IX. It is noteworthy the fact that in the base model, there is no AUC, and especially the improvement in ACC going from approximately 78% to the value of 81%, although the precision has decreased. It is also possible to confirm that there is a slight discrepancy between the base model and the larger one, which leads to the decision to discard the large model. This choice aligns with the overarching objective of implementing it on edge computers, as the larger model would demand significantly more computational power.

TABLE IX – Metrics of the different models developed using BERT.

| **Architecture** | **AUC** | **ACC** | **Avg. F1-Score** | **Avg. Sensitivity** | **Avg. Specificity** | **Avg. Precision** |
|---|---|---|---|---|---|---|
| BaseModel | - | 0.777 | 0.778 | 0.778 | 0.888 | 0.780 |
| SentAnalysisPt | 0.923 | 0.805 | 0.752 | 0.784 | 0.887 | 0.727 |
| SentAnalysisPtBertLarge | 0.923 | 0.807 | 0.759 | 0.790 | 0.889 | 0.735 |

Analyzing the metrics by class, we can see that regarding ACC, as shown in Figure 6.9, we have a slight decrease in the positive class but an improvement in the negative and neutral classes, thus making the model classification more uniform. The full metrics by class graphs for these three models are in the attachments.

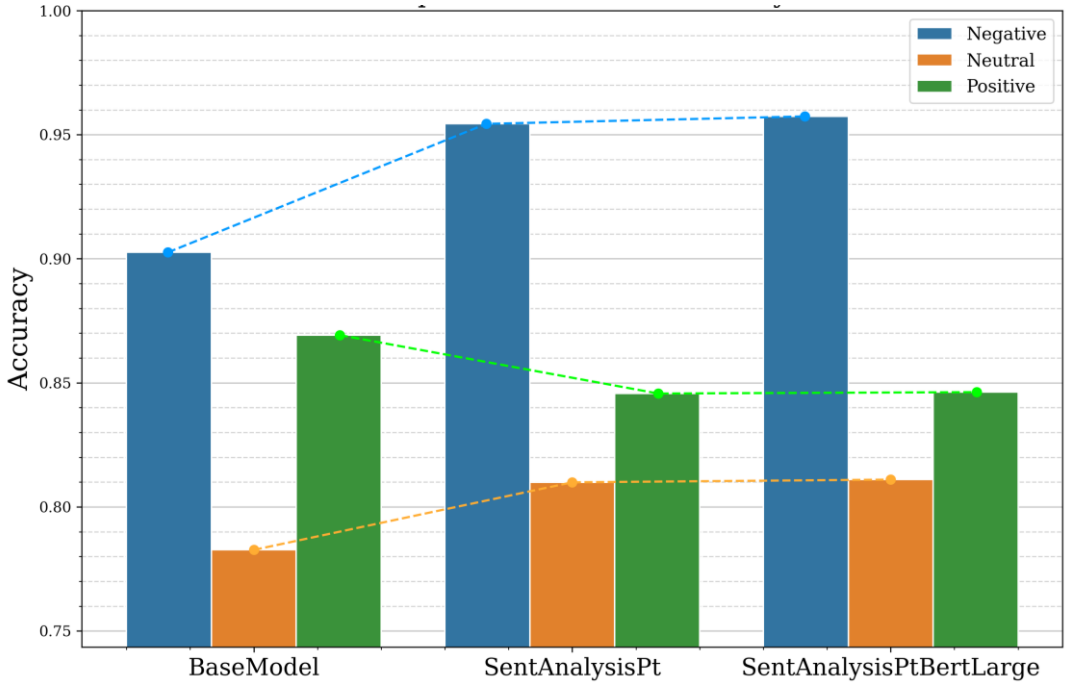



Figure 6.9 – Comparison of the ACC of the three models.

To examine the number of classifiers suitable for an ensemble, a continuous series of experiments were executed involving ensembles ranging from two to six classifiers, as shown in TABLE X. The primary goal was to ascertain the ideal number of classifiers, all while considering that the inclusion of each extra classifier leads to an expansion in the model's size.

To determine the best model from TABLE X, we need to consider which architecture performs consistently well across multiple metrics.

TABLE X – Performance metrics for different architectures, using ensemble, in a SA task.

| **Architecture** | **ACC** | **Avg. F1-Score** | **Avg. Sensitivity** | **Avg. Specificity** | **Avg. Precision** |
|---|---|---|---|---|---|
| SentAnalyPtAdaBoost_2 | **0.840** | **0.777** | **0.765** | **0.885** | 0.790 |
| SentAnalyPtAdaBoost_3 | 0.840 | 0.770 | 0.751 | 0.878 | 0.794 |
| SentAnalyPtAdaBoost_4 | 0.838 | 0.759 | 0.733 | 0.869 | 0.797 |
| SentAnalyPtAdaBoost_5 | 0.840 | 0.764 | 0.745 | 0.873 | 0.793 |
| SentAnalyPtAdaBoost_6 | 0.838 | 0.758 | 0.728 | 0.869 | 0.801 |

Considering the metrics collectively, the "SentAnalyPtAdaBoost_2" architecture seems to perform consistently well with high values in metrics like ACC, Avg. F1-Score, Avg. Sensitivity, and Avg. Specificity. It also has a balanced performance in terms of precision. The "SentAnalyPtAdaBoost_5" architecture also performs well, but "SentAnalyPtAdaBoost_2" has slightly higher values in some metrics. Therefore, based on the provided metrics, the "SentAnalyPtAdaBoost_2" architecture appears to be the best model due to its overall high performance across multiple important metrics.

While TABLE X suggests that the ensemble architecture employing two classifiers is the most suitable, TABLE XI showcases diverse architectures that underwent generalization using

the two-fold cross-validation technique. These architectures encompass the model without an ensemble, as well as the model featuring ensembles of two and three classifiers.

TABLE XI – Performance of different model approaches in a SA task using cross-validation, presenting the standard deviation in brackets.

| Architecture | AUC (σ) | ACC (σ) | F1 | TPR | TNR | PPV |
|---|---|---|---|---|---|---|
| SentAnalysisPt-CrossValidation | 0.924 (0.010) | 0.829 (0.009) | 0.748 | 0.736 | 0.869 | 0.770 |
| SentAnalysisPtAda-BoostCrossValida-tion_2classifier | 0.892 (0.003) | 0.820 (0.009) | 0.728 | 0.718 | 0.859 | 0.760 |
| SentAnalysisPtAda-BoostCrossValidation | 0.857 (0.014) | 0.820 (0.010) | 0.727 | 0.716 | 0.860 | 0.762 |

Based on the TABLE XI metrics, the "SentAnalysisPtCrossValidation" architecture appears to be the best model when generalized. It has the highest ACC (0.829) among the three.

Given the higher values in all metrics, it would be reasonable to conclude that the "SentAnalysisPtCrossValidation" architecture performs the best among the three options. Also, the standard variation in AUC and ACC are similar between the different architectures. It demonstrates that although there is a small drop in the values of the models that apply ensemble, it will not be significant to affect the chosen ensemble procedure.

Figure 6.10 displays the ACC per class, where better values are observed in the negative class, perhaps because it is underrepresented, and when the division is done by the CV, this class is even more affected. In fact, what matters most in this figure is to understand if the classes differ greatly when generalized.

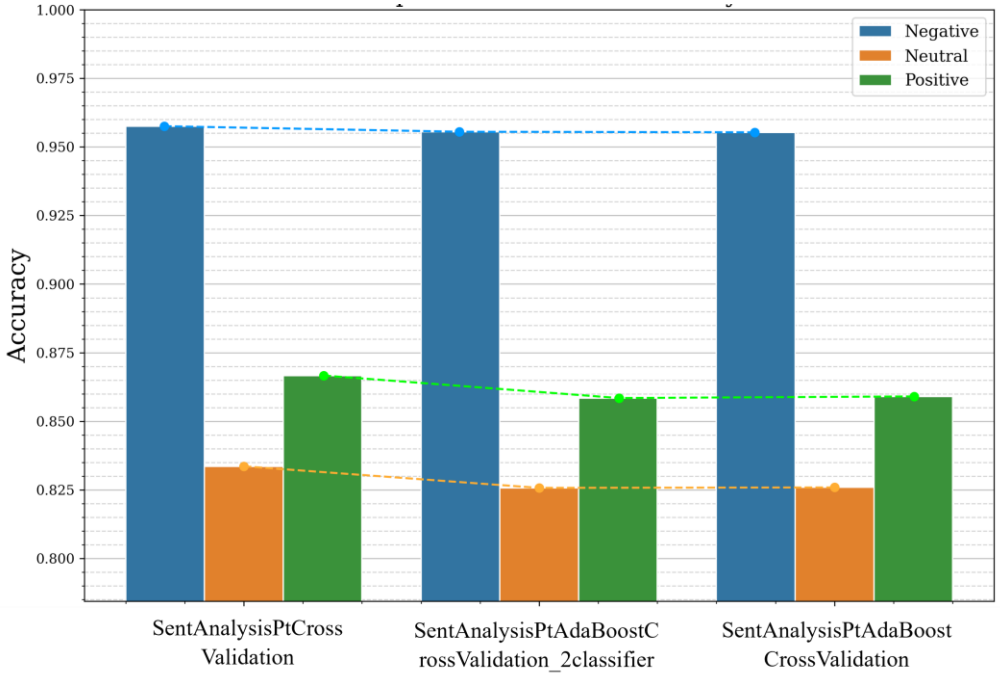



Figure 6.10 – ACC per class of the generalized models, using BERT.

It can be observed that despite employing an ensemble approach, its influence slightly impacts the performance during generalization, yet not to a significant extent warranting consideration for altering the proposed architecture.

It becomes apparent that there is greater variability in ACC within the ensemble with a higher number of classifiers, prompting us to choose the ensemble classifier comprising only two classifiers. This choice is predicated upon the context of developing a sentiment classification model, which is subsequently intended for deployment on edge computing devices. Within this context, various aspects, such as model size and associated variability in ACC, must be carefully weighed.

The decision to favour a less complex ensemble arises from the need to meticulously evaluate the model's attributes to ensure its optimal suitability for integration into edge computing environments. Unless a substantially positive difference in model ACC were to be attained, justifying the adoption of a more substantial model in terms of size.

### 6.1.3. RoBERTa

To perform TL based on the pre-trained RoBERTa model, it was necessary to change some of the hyperparameters. These included the "batch_size," which was reduced from 128 to 64 to decrease the computational demand for training, and an increase in the learning rate to $1e^{-4}$ instead of $2e^{-5}$.

Figure 6.11 shows that there were epochs with validation ACC higher than the best epoch. That occurrence is often due to the complex and dynamic nature of the training process.

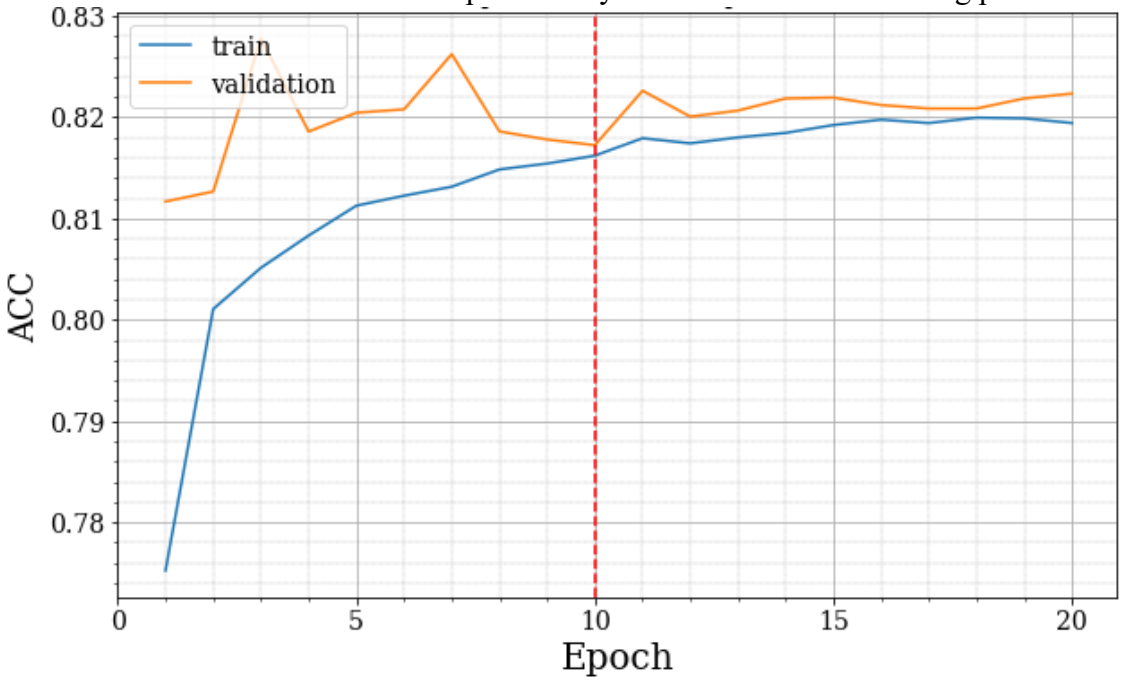



Figure 6.11 – Train history of ACC metric in SA Portuguese model, using RoBERTa.

If we look at Figure 6.12, which is the one we are monitoring for it to be more balanced among the three classes, we can see that epoch 10 is the one that achieves the highest value, providing a more balanced validation.

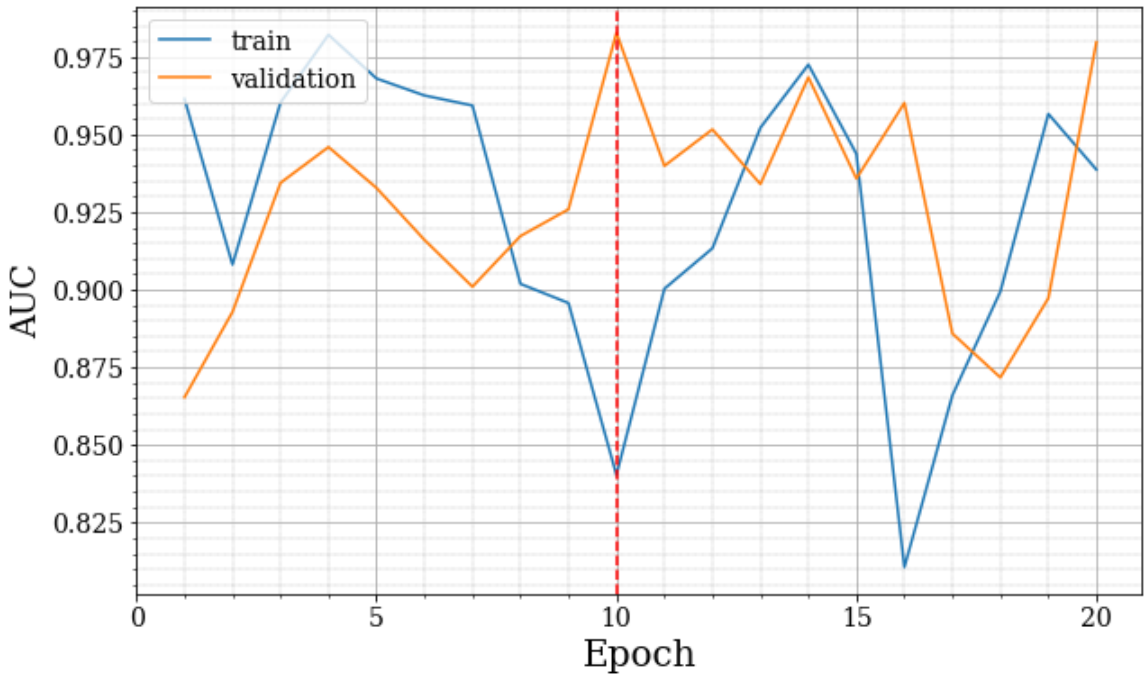



Figure 6.12 – Train history of AUC metric in SA Portuguese model, using RoBERTa.

Overall, if we observe all three graphs (Figure 6.11, Figure 6.12, Figure 6.13), there seems to be an improvement specifically in the last epoch, which did not hold true when tested with a higher learning tolerance.

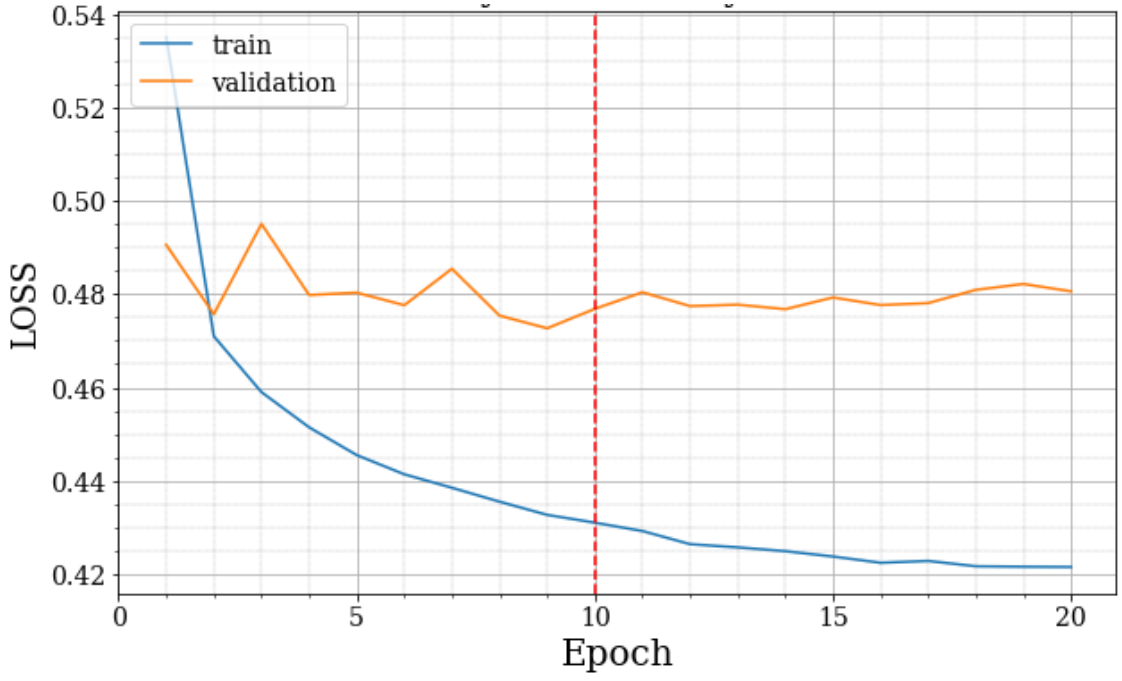



Figure 6.13 – Train history of LOSS metric in SA Portuguese model, using RoBERTa.

The CM shown in Figure 6.14 presents a better balance between classes, although the intermediate class (neutral) always ends up being disadvantaged. This is one of the characteristics of intermediate classes, as they are consistently subject to a higher classification error.

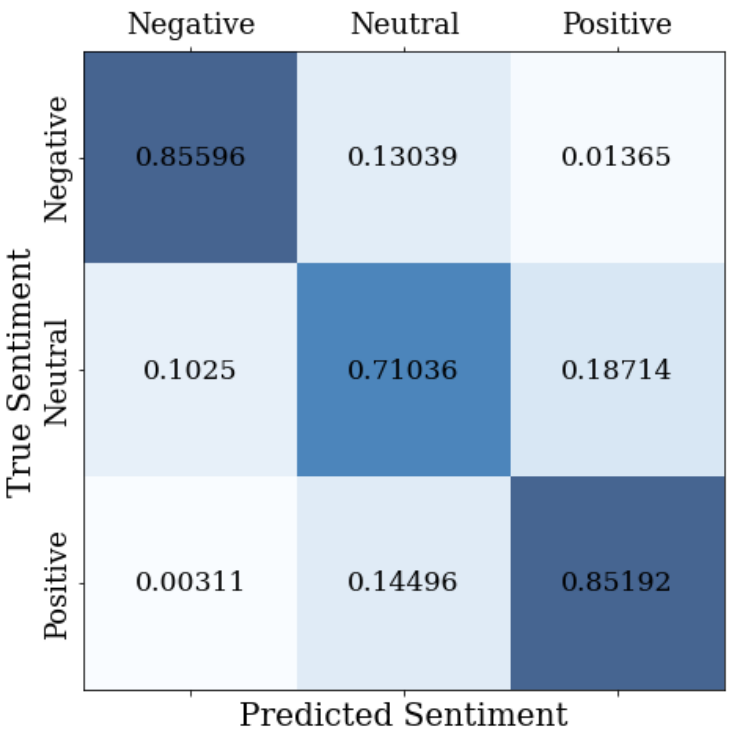



Figure 6.14 - CM of the SA Portuguese model, using RoBERTa.

TABLE XII displays the three models without an ensemble, where it is evident that the RoBERTa model exhibits higher ACC and performs better in the remaining metrics, except for precision, which ends up being lower.

TABLE XII – Comparison of developed model metrics and base model.

| **Architecture** | **AUC** | **ACC** | **Avg. F1-Score** | **Avg. Sensitivity** | **Avg. Specificity** | **Avg. Precision** |
|---|---|---|---|---|---|---|
| BaseModel | - | 0.777 | 0.778 | 0.778 | 0.888 | 0.780 |
| SentAnalysisPt | 0.923 | 0.805 | 0.752 | 0.784 | 0.887 | 0.727 |
| SentAnalysisPtRoBERTa | 0.933 | **0.820** | 0.777 | 0.806 | 0.897 | **0.756** |

The “BaseModel,” while lacking AUC information, exhibits a balanced combination of moderate ACC and average F1-Score, Sensitivity, Specificity, and Precision. The “SentAnalysisPt” architecture slightly falters in terms of F1-Score, Sensitivity, and Precision in comparison to the other variant.

In contrast, the “SentAnalysisPtRoBERTa” architecture emerges as the frontrunner among the three. With an AUC score of 0.933 and an ACC of 0.820, it consistently demonstrates superior performance across the average F1-Score, Sensitivity, and Precision. These results collectively underline the advantages of employing the RoBERTa model for SA, affirming its ability to yield enhanced overall performance.

It is important to note that this choice of architecture depends on other factors such as computational resources, training time, and interpretability of the model. Being the bigger model is not an advantage for edge computing purposes as in this work.

Following the same reasoning as for the previous model where BERTimbau was applied, the development of two models was carried out using RoBERTa with two and three classifiers, employing Adaboost. The performance metrics are displayed in TABLE XIII, showing that the ensemble with more classifiers does not improve the performance, as all the metrics except the precision are consistently higher.

TABLE XIII – Performance metrics for different architectures, using an ensemble of RoBERTa, in a SA task.

| Architecture | ACC | Avg. F1-Score | Avg. Sensitivity | Avg. Specificity | Avg. Precision |
|---|---|---|---|---|---|
| SentAnalyPtAdaBoostRoB-ERTa_2 | 0.828 | 0.765 | 0.757 | 0.881 | 0.773 |
| SentAnalyPtAdaBoostRoB-ERTa_3 | 0.805 | 0.725 | 0.697 | 0.864 | 0.790 |

Analysing TABLE XIV, we observe that the comparison between this architecture reveals distinctive performance trends. The initial architecture, referred to as "SentAnalysisPt-CrossValidationRoberta," demonstrates superior overall performance compared to the subsequent architecture, denoted as "SentAnalysisPtAda-BoostCrossValidationRoberta." Notably, this performance advantage encompasses metrics such as AUC, ACC, and average precision. Additionally, both architectures exhibit similar trends in terms of average F1-Score, sensitivity, and specificity, implying comparable behaviour in these aspects.

It is worth noting that the "SentAnalysisPtCrossValidationRoberta" architecture displays a notably higher AUC value of 0.933 in contrast to the ensemble model architecture, which achieves an AUC of 0.897. This disparity in AUC values typically suggests a greater discriminatory ability when distinguishing between different classes.

Furthermore, the "SentAnalysisPtCrossValidationRoberta" architecture also outperforms its counterpart in terms of average ACC, boasting a value of 0.840 as opposed to the ensemble model architecture's ACC of 0.829. This circumstance could be related to the fact that the base model, RoBERTa, is already a much better model, leaving us with an ensemble that does not present weak learners.

TABLE XIV – Performance of RoBERTa model approaches in an SA task with cross-validation, presenting the standard deviation in brackets.

| Architecture | AUC (σ) | ACC (σ) | F1 | TPR | TNR | PPV |
|---|---|---|---|---|---|---|
| SentAnalysisPt-CrossValidationRoberta | 0.933 (0.010) | 0.840 (0.010) | 0.765 | 0.748 | 0.878 | 0.792 |
| SentAnalysisPtAda-BoostCrossValida-tionRoberta | 0.897 (0.004) | 0.829 (0.013) | 0.745 | 0.744 | 0.873 | 0.771 |

Figure 6.15 presents class-specific accuracies, with notable levels very similar between them. The values might stem from the same situation described in Figure 6.10 in BERT architecture analysis.

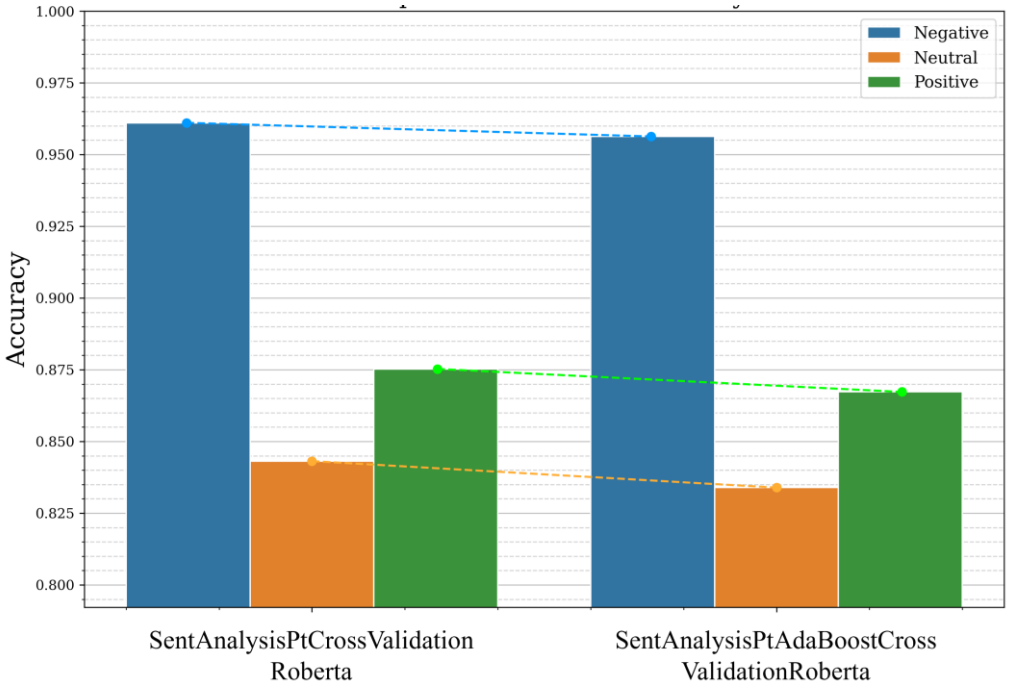



Figure 6.15 - ACC per class of the generalized models, using RoBERTa.

Based on the tests conducted and described in this chapter, it can be affirmed that developing a classifier based on the RoBERTa architecture yields better results when applied individually, as compared to the ACC values obtained from the standard BERT-based architecture. It is imperative to bear in mind the objective of this study, which revolves around implementing an efficient model on edge computing platforms through an ensemble approach. However, it is worth noting that this also exposes the limitation of using the RoBERTa architecture due to concerns related to the final model's size.

When attempting to implement an ensemble using the RoBERTa base architecture, there is a noticeable degradation in the values of almost all metrics, contrary to what might be expected. Possibly due to the model being a too strong learner, hampering the boosting procedure. Thus, using this architecture for developing a classifier of this nature is not suitable.

The earlier observation that this architecture performs better when applied individually to a classifier becomes evident when employing 2-fold cross-validation for generalization. This leads to diminished values for the ensemble model, further emphasizing its unsuitability.

### 6.1.4. Comparative Assessment of the Developed Models

Within this subchapter, we embark on a comprehensive assessment of the four primary models created. We commence this analysis with an overview provided by Figure 6.16, a waterfall chart illustrating the metric progression. Subsequently, we conduct a detailed examination of each metric, unraveling the nuances of their performance:

Regarding the AUC metric, our analysis is limited to the "SentAnalysisPt" and "SentAnalysisPtRoberta" models, with the remaining models being represented with a value of zero. Notably, both "SentAnalysisPt" and "SentAnalysisPtRoberta" exhibit AUC values exceeding 0.923, indicating exceptional performance. Higher AUC values, closer to 1, are indicative of superior model performance. These models excel in effectively distinguishing between classes based on ROC curves.

High accuracy is generally desirable, but it may not be the only metric to consider, especially for imbalanced datasets. All models have relatively high accuracy values, with

“SentAnalyPtAdaBoost_2” having the highest at 0.840. However, accuracy alone might not be sufficient for model selection.

The F1-score serves as a measure that strikes a balance between false positives and false negatives. Notably, “SentAnalyPtAdaBoost_2” exhibits a relatively elevated F1-score of 0.776, signifying a commendable equilibrium between precision and sensitivity. Additionally, “SentAnalyPtRoberta” also has a respectable F1-score of 0.777.

In terms of correctly identifying positive instances (Sensitivity), the basic models exhibit superior performance, with the “SentAnalyPtRoberta” model achieving the highest sensitivity at 0.806.

Regarding the classification of negative instances, all the models demonstrate relatively high specificity values. Notably, “SentAnalyPtRoberta” achieves the highest specificity at 0.896, followed closely by “SentAnalysisPt” at 0.887. Based on these two metrics, it is evident that when it comes to the classification of correctly identifying positive or negative instances, the models without ensemble techniques perform better.

The last metric to analyze is Precision, which measures the proportion of correctly identified positive instances out of all instances predicted as positive. “SentAnalyPtAdaBoost_2” has the highest precision at 0.789, having the best value by far from the other models, especially from the “SentAnalysisPtRoberta” that has the best F1, Sensitivity and Specificity.

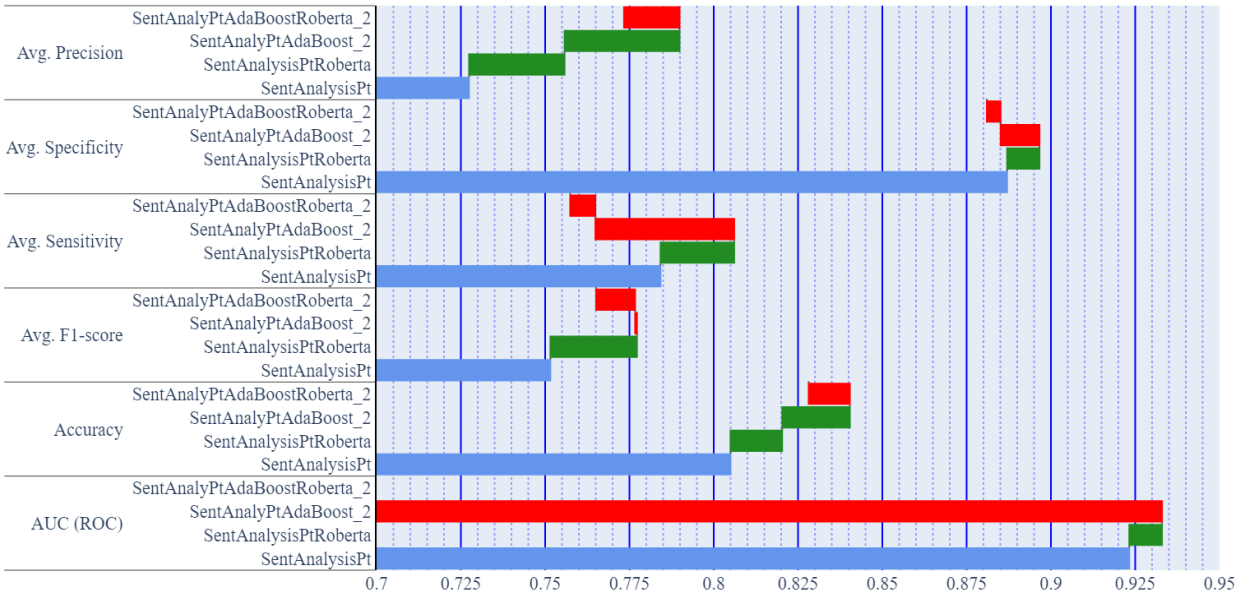



Figure 6.16 - Waterfall chart with all metrics of the four major models.

Based on the provided metrics, “SentAnalyPtAdaBoost_2” seems to be a strong candidate for inference on an edge computing device. It has good accuracy and precision values, with slightly lower F1-score than “SentAnalysisPtRoberta”.

Consider the trade-offs between model performance and computational resources available on the edge device. If there is a need to balance between performance and resource efficiency, “SentAnalyPtAdaBoost_2” might be a suitable choice. However, if computational resources are less constrained, “SentAnalysisPtRoberta” may be preferred due to their higher AUC values.

### 6.1.5. Comparative Review of the State-Of-The-Art

In this subsection, a comparison is drawn among the previously mentioned works in the state of the art.

TABLE XV reveals a distinct consistency in the utilization of base transformer models, delineating the path forward in the realm of SA and NLP tasks through applying TL techniques and post-training. The developed models have demonstrated ACC levels on par with those showcased in the state-of-the-art, even reaching top values, with the ensemble model achieving an ACC of 84%, the highest recorded. This achievement maintains comparable F1-Score and precision levels to the other approaches.

TABLE XV - Comparative Analysis of state-of-the-art.

| **Model** | **Task** | **ACC** | **F1** | **TPR** | **PPV** |
|---|---|---|---|---|---|
| TeamDeepLearn-ingBrasil [58] | Aspect Term Extraction | 0.67 | - | - | - |
| PTT5 [58] | Sentiment Orientation Extraction | 0.82 | 0.82 | 0.82 | 0.81 |
| BERTimbau [55] | Aspect Extraction 3-class | 0.80 | 0.77 | - | 0.78 |
| BERT [54] | SBERT-QA | 0.79 | - | - | - |
| RoBERTa [54] | SRoBERTa-QA | 0.78 | - | - | - |
| BERTimbauBase | Sentiment Analysis Pt | 0.81 | 0.75 | 0.78 | 0.73 |
| RoBERTaBase | Sentiment Analysis Pt | 0.82 | 0.78 | 0.81 | 0.76 |
| BERTimbauBoosted | Sentiment Analysis Pt ensemble | **0.84** | 0.78 | 0.77 | 0.79 |
| RoBERTaBoosted | Sentiment Analysis Pt ensemble | 0.83 | 0.77 | 0.76 | 0.77 |

It is noteworthy that upon comparing the ensemble models outlined in [54] with those developed within the scope of this work, a remarkable similarity emerges. Specifically, when employing an ensemble with the RoBERTa model, there appears to be a consistent decrease in ACC compared to the BERT model. This observation may suggest a potential challenge in effectively forming an ensemble with this model.

## 6.2. Inference

The rationale behind adopting this approach, which utilizes tokens instead of words, stems from a deliberate consideration of several factors. These factors can be distilled into three distinct categories:

- **50 Tokens per Review:** The utilization of a 50-token threshold per review serves as the first category, marking the lower limit of token allocation. This choice considers the need to maintain a balance between precision and computational efficiency. By examining the classification of sentiment within this token constraint, we aim to evaluate the computational resources required for a minimalistic yet functional model.
- **100 Tokens per Review:** In the second category, the threshold is extended to 100 tokens per review. This intermediate level of token allocation reflects our intention to delve deeper into the sentiment analysis process while keeping computational demands within manageable boundaries. This choice allows for an

exploration of the potential trade-off between processing time and the depth of sentiment analysis.

- **More than 100 Tokens per Review:** The final category encompasses reviews that exceed the 100-token mark. It accounts for reviews that are substantially longer and provides insights into the handling of more complex and information-rich textual content. Here, we explore the upper limits of token-based sentiment analysis and its implications for computational intensity.

Within each of these categories, three sentences are employed to illustrate the temporal resources required for sentiment classification and the overall commitment to this classification's degree of assertiveness. It is essential to emphasize that these illustrative sentences are generated without any alterations to the original reviews. This approach is specifically chosen to ensure data integrity and maintain the validity of our results.

The primary objective behind the adoption of token-based analysis over individual words in our study serves as an elegant solution to exercise precision and control over the volume of information processed by the sentiment analysis platforms. This serves to guarantee a fair and equitable basis for comparison across different platforms and models. The inherent limitation of word counting lies in its inability to provide a consistent basis for comparison due to variations introduced by punctuation. Therefore, the token-based methodology is preferred, as it not only upholds the integrity of the data but also underscores the meticulous and rigorous approach we have taken to ensure the accuracy and fairness of our comparative assessment.

All models created and previously trained with our dataset needed to be loaded in advance onto their respective platforms, which sometimes caused space issues on them. However, they had the advantage of not requiring an internet connection. This approach was also taken to avoid delays in model loading due to potential internet traffic, aligning with one of the objectives of this work: to use these platforms for edge computing, thereby avoiding the need to utilize costly Cloud Computing resources.

For the analysis of the inference with 50 tokens, the following reviews were used:

- **Negative:** "Já fui a outros Fogo de Chão muito mas muito melhores em Lisboa este aqui tem apresentações horríveis e comida parece sei lá o q.... a comida parecia ser comida quase da cantina da escola omg !!!"
- **Neutral:** "Ideal para uma refeição mais rápida, quando se tem menos tempo para almoçar/jantar. Bom quando já não falta muito tempo para começar a sessão de cinema, mas apetece petiscar algo antes de entrar ;)"
- **Positive:** "A comida é muito boa, mexilhão à espanhola muito bom, filetes fritos com bom óleo bem gostosos, as lulas à correio a meu ver estavam um pouco secas e salgadas. Preço muito convidativo."

To analyze inference using 100 tokens, we utilized the following set of reviews:

- **Negative:** "Sítio engraçado mas infelizmente recusarem-se a servir cafés e bebidas quase às 15h, com o argumento de que é hora de almoço, e com uma série de mesas vazias, é um pouco triste e até desrespeitador. Sendo o único sítio disponível foi muito chato, mas felizmente existem outros locais não muito longe, onde valeu bastante a pena ir e que não tínhamos conhecido não fosse a tristeza e a má vontade."
- **Neutral:** "Espaço extremamente agradável, o aspecto rústico com os presuntos expostos e malagueta, etc... Na minha opinião tornam o espaço muito caloroso.

Funcionários muito simpáticos e prestáveis. Em relação às iguarias servidas, tem bom aspecto mas a nível de sabor deixam algo a desejar. Tendo em conta a quantidade de sítios bons para se comer uma refeição italiana. Tiramisu divinal apesar do aspecto algo descabido na chegada a mesa."

- **Positive:** "Fui levado de surpresa a um jantar a este restaurante e não podia ter gostado mais! Começámos com umas empadinhas de perdiz e seguimos para as lascas de bacalhau e para a tagine de cordeiro, tudo muito bem servido e muito saboroso. Para terminar pedimos um cheesecake e um bolo de chocolate, dos melhores que já comi! Este é sem dúvida um restaurante que vou passar a recomendar!"

To assess inference using more than 100 tokens, we considered the subsequent collection of reviews:

- **Negative:** "Acabei agora de sair deste estabelecimento, depois de esperar mais de uma hora em vão. Nesta hora, inúmeros clientes que chegaram depois de me sentar foram servidos antes. Ao manifestar a insatisfação junto do dono do estabelecimento, este foi mal educado e mentiroso. De referir que o resto do staff foi impecável, mas este tipo de tratamento por parte do dono do estabelecimento é inaceitável."
- **Neutral:** "Espaço amplo e bem decorado que serve soluções rápidas para um almoço que não deve demorar. A salada de salmão muito mal servida, mais alface que outra coisa, o que me deixou um pouco desconsolada. O que salvou foi a sopa e o sumo que estavam bons. Fico reticente se voltarei ou não, talvez a escolha possa não ter sido a mais acertada."
- **Positive:** "Muito bom! Só tenho comido o vegetariano, de feijão preto, com picante, e gosto muito! Costumo ir buscar para comer em casa e só de uma vez veio sem queijo. Não foi nada de gravíssimo, mas referimos isso. Pediram desculpa e ofereceram um hambúrguer na próxima compra. Gostei muito da atitude e já voltei várias vezes depois disso. E voltarei mais! No Top 3 de hambúrguerias no Porto."

The inference tests were made with the models BERT and RoBERTa as follows:

- BERT: It used BERTimbau Base, BERTimbau Large, and the Boosted model, which is an ensemble of two classifiers, BERTimbau Base, as this was the best architecture found in chapter 6.1.2, TABLE X.
- RoBERTa: It used RoBERTa Base and the two classifier RoBERTa Base ensemble as Boosted, as found in chapter 6.1.3, TABLE XI.

### 6.2.1. Jetson Results and Performance Evaluation

The subsequent tables display the outcomes of inference on various architectures created on the Jetson Nano platform. They illustrate the duration required to categorize diverse reviews of varying sizes, the cumulative time needed to classify all three classes, the confidence level in the categorization, and the correctness of the categorization.

Based on TABLE XVI, the BERTimbau Base model on the Jetson Nano platform appears to perform well, with generally fast classification times, high confidence levels, and correct classifications across different review lengths and sentiment categories.

TABLE XVI – Jetson performance of developed architecture based on BERTimbau Base model.

| **BERTimbau Base** | | **Classification time (seconds)** | **Total inference time (seconds)** | **Level of confidence (%)** | **Correctness** |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 1.02 | 2.85 | 68.5 | Correct |
| | Neutral | 0.99 | | 77.7 | Correct |
| | Positive | 0.84 | | 63.5 | Correct |
| 100 tokens Review | Negative | 0.88 | 2.76 | 78.8 | Correct |
| | Neutral | 0.98 | | 73.8 | Correct |
| | Positive | 0.90 | | 97.2 | Correct |
| 100+ tokens Review | Negative | 0.86 | 2.55 | 98.2 | Correct |
| | Neutral | 0.86 | | 75.1 | Correct |
| | Positive | 0.83 | | 86.1 | Correct |

TABLE XVII shows the BERTimbau Large model on the Jetson Nano platform exhibiting variable performance. While it achieves high ACC for many cases, it comes at the cost of significantly longer classification times, especially for Negative and Neutral sentiments in longer reviews.

TABLE XVII – Jetson performance of developed architecture based on BERTimbau Large model.

| **BERTimbau Large** | | **Classification time (seconds)** | **Total inference time (seconds)** | **Level of confidence (%)** | **Correctness** |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 6.31 | 29.99 | 93.5 | Correct |
| | Neutral | 7.97 | | 77.5 | Correct |
| | Positive | 15.72 | | 50.4 | **Incorrect** |
| 100 tokens Review | Negative | 4.89 | 29.11 | 54.6 | Correct |
| | Neutral | 20.32 | | 61.1 | Correct |
| | Positive | 3.90 | | 97.2 | Correct |
| 100+ tokens Review | Negative | 20.99 | 62.18 | 97.5 | Correct |
| | Neutral | 27.33 | | 87.1 | Correct |
| | Positive | 13.86 | | 91.3 | Correct |

The BERTimbau Boosted model on the Jetson Nano platform, TABLE XVIII, shows variable performance. It achieves correct classifications in many cases, but the classification times are significantly longer, especially for Negative and Neutral sentiments in reviews with more tokens.

TABLE XVIII – Jetson performance of developed Boosted architecture based on BERTimbau Base model.

| BERTimbau Boosted | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 30.57 | 47.69 | 53.9 | Correct |
| | Neutral | 13.27 | | 63.8 | Correct |
| | Positive | 3.85 | | 58.9 | Correct |
| 100 tokens Review | Negative | 22.93 | 51.92 | 51.1 | Correct |
| | Neutral | 25.76 | | 50.0 | **Incorrect** |
| | Positive | 3.23 | | 95.8 | Correct |
| 100+ tokens Review | Negative | 41.03 | 122.60 | 71.1 | Correct |
| | Neutral | 40.92 | | 61.0 | Correct |
| | Positive | 40.65 | | 88.9 | Correct |

The RoBERTa Base model on the Jetson Nano platform consistently delivers strong performance across different review lengths and sentiment categories. TABLE XIX shows that it provides both fast classification times and high confidence levels while maintaining correct classifications, making it a robust choice for our SA processing tasks, just with the flip side of being an extremely heavy architecture for edge computing devices.

TABLE XIX – Jetson performance of developed architecture based on RoBERTa Base model.

| RoBERTa Base | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 2.15 | 3.84 | 98.4 | Correct |
| | Neutral | 0.86 | | 90.9 | Correct |
| | Positive | 0.83 | | 52.5 | Correct |
| 100 tokens Review | Negative | 2.21 | 3.99 | 65.8 | Correct |
| | Neutral | 0.96 | | 75.9 | Correct |
| | Positive | 0.82 | | 99.5 | Correct |
| 100+ tokens Review | Negative | 2.21 | 4.27 | 99.6 | Correct |
| | Neutral | 1.23 | | 95.1 | Correct |
| | Positive | 0.83 | | 95.6 | Correct |

RoBERTa Boosted model on the Jetson Nano platform offers high correctness in its classifications, as depicted in TABLE XX, but it comes at the cost of extremely long classification times, especially for shorter reviews and Neutral sentiment. These lengthy processing times may limit its practicality for real-time or high-throughput applications as our SA task.

TABLE XX – Jetson performance of developed Boosted architecture based on RoBERTa Base model.

| RoBERTa Boosted | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 282.66 | 1429.98 | 63.8 | Correct |
| | Neutral | 567.21 | | 50.8 | Correct |
| | Positive | 580.11 | | 57.4 | Correct |
| 100 tokens Review | Negative | 383.12 | 1151.44 | 52.1 | Correct |
| | Neutral | 421.70 | | 64.9 | Correct |
| | Positive | 346.62 | | 99.5 | Correct |
| 100+ tokens Review | Negative | 354.97 | 1306.18 | 77.2 | Correct |
| | Neutral | 416.57 | | 68.1 | Correct |
| | Positive | 534.64 | | 90.6 | Correct |

### 6.2.2. Raspberry PI Results and Performance Evaluation

The results of the inference for various architectures created on the Raspberry Pi platform are displayed in the subsequent tables. These tables demonstrate the duration required to classify reviews of varying sizes, the cumulative classification time for the three classes, the confidence level of the classification, and the correctness of the classification.

TABLE XXI shows that the BERTimbau Base model on the Raspberry Pi demonstrated commendable performance across different review lengths, with generally high ACC and confidence levels in its classifications.

TABLE XXI – Raspberry Pi performance of developed architecture based on BERTimbau Base model.

| BERTimbau Base | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 1.46 | 3.79 | 68.5 | Correct |
| | Neutral | 1.34 | | 77.7 | Correct |
| | Positive | 0.99 | | 63.5 | Correct |
| 100 tokens Review | Negative | 1.61 | 4.52 | 78.8 | Correct |
| | Neutral | 1.48 | | 73.8 | Correct |
| | Positive | 1.43 | | 97.2 | Correct |
| 100+ tokens Review | Negative | 1.79 | 4.27 | 98.2 | Correct |
| | Neutral | 1.48 | | 75.1 | Correct |
| | Positive | 1.00 | | 86.1 | Correct |

TABLE XXII exhibits the BERTimbau Large model on the Raspberry Pi, showing varying performance based on review length. It performed well with high confidence and correctness for negative and positive reviews in long tokens, but it struggled with negative reviews in the 100-token category. Additionally, for 50-token positive reviews, it had lower confidence and was incorrect.

TABLE XXII – Raspberry Pi performance of developed architecture based on BERTimbau Large model.

| BERTimbau Large | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 3.79 | 11.44 | 93.5 | Correct |
| | Neutral | 3.88 | | 77.5 | Correct |
| | Positive | 3.77 | | 50.4 | **Incorrect** |
| 100 tokens Review | Negative | 5.66 | 16.84 | 54.6 | Correct |
| | Neutral | 5.68 | | 61.1 | Correct |
| | Positive | 5.50 | | 97.2 | Correct |
| 100+ tokens Review | Negative | 4.11 | 17.90 | 97.5 | Correct |
| | Neutral | 9.94 | | 87.1 | Correct |
| | Positive | 3.85 | | 91.3 | Correct |

BERTimbau Boosted model on the Raspberry Pi, TABLE XXIII, exhibited varying performance depending on the review length and sentiment. It generally performed well with high correctness for negative and positive reviews but struggled with neutral reviews in the 100-token category, where it had an incorrect classification.

TABLE XXIII – Raspberry Pi performance of developed Boosted architecture based on BERTimbau Base model.

| BERTimbau Boosted | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 4.75 | 16.96 | 53.9 | Correct |
| | Neutral | 7.69 | | 63.8 | Correct |
| | Positive | 4.52 | | 58.9 | Correct |
| 100 tokens Review | Negative | 5.68 | 17.84 | 51.1 | Correct |
| | Neutral | 7.46 | | 50.0 | **Incorrect** |
| | Positive | 4.70 | | 95.8 | Correct |
| 100+ tokens Review | Negative | 3.84 | 11.98 | 71.1 | Correct |
| | Neutral | 4.77 | | 61.0 | Correct |
| | Positive | 3.38 | | 88.9 | Correct |

TABLE XXIV shows the RoBERTa Base model on the Raspberry Pi demonstrated consistently strong performance across different review lengths and sentiment categories, with high correctness and confidence levels in its classifications. It presents good performance and confidence in longer reviews.

TABLE XXIV – Raspberry Pi performance of developed architecture based on RoBERTa Base model.

| RoBERTa Base | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 1.98 | 4.94 | 98.4 | Correct |
| | Neutral | 1.11 | | 90.9 | Correct |
| | Positive | 1.85 | | 52.5 | Correct |
| 100 tokens Review | Negative | 1.76 | 4.14 | 65.8 | Correct |
| | Neutral | 1.39 | | 75.9 | Correct |
| | Positive | 0.99 | | 99.5 | Correct |
| 100+ tokens Review | Negative | 1.13 | 3.91 | 99.6 | Correct |
| | Neutral | 1.80 | | 95.1 | Correct |
| | Positive | 0.98 | | 95.6 | Correct |

RoBERTa Boosted model on the Raspberry Pi demonstrated accurate classifications but at the cost of significantly longer processing times compared to other models, particularly for neutral and negative reviews, as exposed in TABLE XXV.

TABLE XXV – Raspberry Pi performance of developed Boosted architecture based on RoBERTa Base model.

| RoBERTa Boosted | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 62.45 | 186.31 | 63.8 | Correct |
| | Neutral | 64.87 | | 50.1 | Correct |
| | Positive | 58.99 | | 57.4 | Correct |
| 100 tokens Review | Negative | 61.68 | 184.86 | 52.1 | Correct |
| | Neutral | 70.99 | | 64.9 | Correct |
| | Positive | 52.18 | | 99.5 | Correct |
| 100+ tokens Review | Negative | 88.33 | 194.53 | 77.2 | Correct |
| | Neutral | 53.82 | | 68.1 | Correct |
| | Positive | 52.38 | | 90.6 | Correct |

### 6.2.3. Nvidia RTX 3090 Results and Performance Evaluation

The subsequent tables display the outcomes of the inference conducted on various architectures created on the RTX 3090 platform. These tables depict the time required for classifying reviews of varying sizes, the cumulative time taken for classifying all three classes, the degree of confidence in the classifications, and the correctness of the classifications.

Looking at TABLE XXVI, it seems that the BERTimbau Base model performs well across different review lengths, maintaining a good balance between classification time, confidence, and correctness.

TABLE XXVI – RTX 3090 performance of developed architecture based on BERTimbau Base model.

| BERTimbau Base | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 0.72 | 2.20 | 68.5 | Correct |
| | Neutral | 0.75 | | 77.7 | Correct |
| | Positive | 0.73 | | 63.5 | Correct |
| 100 tokens Review | Negative | 0.66 | 2.03 | 78.8 | Correct |
| | Neutral | 0.68 | | 73.8 | Correct |
| | Positive | 0.69 | | 97.2 | Correct |
| 100+ tokens Review | Negative | 0.65 | 2.09 | 98.2 | Correct |
| | Neutral | 0.72 | | 75.1 | Correct |
| | Positive | 0.72 | | 86.1 | Correct |

BERTimbau Large model performs well in terms of classification time and correctness, with some variability in confidence levels, especially in the case of positive reviews in the 50-token category. Interestingly, TABLE XXVII proves it is faster in classification but less assertive.

This fact prompts us to seek the reason for such an occurrence in the inference process. To do so, we examined the training parameters, and it may be related to the use of a smaller batch size with the larger models, reducing it to 64 instead of the 128 used in the BERTimbau Base model. This change was imposed by GPU memory constraints that made training the model impossible.

However, this change has resulted in slower training. In the case under analysis, training the BERTimbau Base took approximately 11 hours and 55 minutes, with the best epoch occurring at the 13th epoch. On the other hand, the BERTimbau Large model took 21 hours and 2 minutes to train and reached its peak performance in the 3rd epoch.

Regarding inference, a smaller batch size might lead to quicker inference times for the BERTimbau Large model. BERTimbau Large is a larger model with more parameters compared to BERTimbau Base. Smaller batch sizes can be processed faster because they require less memory and computation. So, even though BERTimbau Large is a more complex model, its smaller batch size might allow it to fit comfortably in available GPU memory and be processed efficiently.

TABLE XXVII – RTX 3090 performance of developed architecture based on BERTimbau Large model.

| BERTimbau Large | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 0.20 | 0.61 | 93.5 | Correct |
| | Neutral | 0.20 | | 77.5 | Correct |
| | Positive | 0.21 | | 50.4 | **Incorrect** |
| 100 tokens Review | Negative | 0.19 | 0.58 | 54.6 | Correct |
| | Neutral | 0.19 | | 61.1 | Correct |
| | Positive | 0.20 | | 97.2 | Correct |
| 100+ tokens Review | Negative | 0.19 | 0.60 | 97.5 | Correct |
| | Neutral | 0.21 | | 87.1 | Correct |
| | Positive | 0.21 | | 91.3 | Correct |

TABLE XXVIII shows the BERTimbau Boosted model that points to variability in classification times and correctness across different review lengths and sentiment classes. While correctness is mostly maintained, improvements could be made in terms of classification time and ACC, especially for neutral reviews in the 100-token category.

TABLE XXVIII – RTX 3090 performance of developed Boosted architecture based on BERTimbau Base model.

| BERTimbau Boosted | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 0.79 | 1.88 | 53.9 | Correct |
| | Neutral | 0.55 | | 63.8 | Correct |
| | Positive | 0.55 | | 58.9 | Correct |
| 100 tokens Review | Negative | 1.29 | 2.86 | 51.1 | Correct |
| | Neutral | 0.79 | | 50.0 | **Incorrect** |
| | Positive | 0.78 | | 95.8 | Correct |
| 100+ tokens Review | Negative | 0.91 | 2.43 | 71.1 | Correct |
| | Neutral | 0.76 | | 61.0 | Correct |
| | Positive | 0.76 | | 88.9 | Correct |

RoBERTa Base model demonstrates, in TABLE XXIX, impressive performance with fast classification times, high confidence levels, and strong correctness across varying review lengths and sentiment categories.

TABLE XXIX – RTX 3090 performance of developed architecture based on RoBERTa Base model.

| RoBERTa Base | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 0.70 | 2.13 | 98.4 | Correct |
| | Neutral | 0.74 | | 90.9 | Correct |
| | Positive | 0.69 | | 52.5 | Correct |
| 100 tokens Review | Negative | 0.65 | 2.02 | 65.8 | Correct |
| | Neutral | 0.71 | | 75.9 | Correct |
| | Positive | 0.66 | | 99.5 | Correct |
| 100+ tokens Review | Negative | 0.70 | 2.17 | 99.6 | Correct |
| | Neutral | 0.75 | | 95.1 | Correct |
| | Positive | 0.72 | | 95.6 | Correct |

TABLE XXX shows the RoBERTa Boosted model that demonstrates slower classification times and varying levels of confidence across different review lengths and sentiment classes. Maintaining high correctness, especially for negative reviews, could be a point for improvement.

TABLE XXX – RTX 3090 performance of developed Boosted architecture based on RoBERTa Base model.

| RoBERTa Boosted | | Classification time (seconds) | Total inference time (seconds) | Level of confidence (%) | Correctness |
|---|---|---|---|---|---|
| 50 tokens Review | Negative | 1.40 | 3.76 | 63.8 | Correct |
| | Neutral | 1.19 | | 50.8 | Correct |
| | Positive | 1.16 | | 57.4 | Correct |
| 100 tokens Review | Negative | 1.34 | 3.64 | 52.1 | Correct |
| | Neutral | 1.14 | | 64.9 | Correct |
| | Positive | 1.16 | | 99.5 | Correct |
| 100+ tokens Review | Negative | 1.35 | 3.65 | 77.2 | Correct |
| | Neutral | 1.14 | | 68.1 | Correct |
| | Positive | 1.16 | | 90.6 | Correct |

### 6.2.4. Performance

An analysis was conducted comparing the performance of the developed models and architectures, taking into consideration the inference platform. The RTX 3090 platform, being a GPU, will serve as a simulation of a cloud computing system to gain insight into the advantages or disadvantages of implementing such complex models on edge computing platforms.

TABLE XXXI displays the average elapsed times for the classification of the three classes in reviews of three different sizes.

TABLE XXXI – Performance to complexity of developed models.

| Model | | Average platform time (seconds) | | |
|---|---|---|---|---|
| **Architecture** | **Parameters** | **Jetson Nano** | **Raspberry Pi** | **RTX 3090** |
| BERTBase | 110 M | 2.720 | 4.193 | 2.107 |
| BERTBoost | 220 M | 74.070 | 15.593 | 2.390 |
| BERTLarge | 335 M | 40.427 | 15.393 | 0.597 |
| RoBERTaBase | 355 M | 4.033 | 4.330 | 2.107 |
| RoBERTaBoost | 710 M | 1,295.867 | 188.567 | 3.683 |

Based on TABLE XXXI, we can conduct a comprehensive analysis of computational efficiency across various models and platforms, exploring the model complexity, comparing hardware platforms, examining performance relative to model intricacy, assessing the impact of hardware, and considering energy efficiency.

BERTBase and RoBERTaBase exhibit relatively lower parameters, while BERTBoost and RoBERTaBoost are notably parameter-rich. Significantly divergent computational performances emerge when scrutinizing the platforms. The RTX 3090 stands out as the most powerful, followed by the Jetson Nano and Raspberry Pi.

A closer look at the models' average times per platform, shown in Figure 6.17, reveals that computational efficiency extends beyond mere model complexity. Surprisingly, BERTLarge, with fewer parameters than BERTBoost and RoBERTaBoost, demonstrates notably swifter performance. This highlights the pivotal role of model optimization and design in computational efficiency.

The influence of hardware on computational efficiency is obvious. The RTX 3090 consistently outperforms the Jetson Nano and Raspberry Pi, efficiently handling even the most intricate models, such as BERTBoost and RoBERTaBoost. The Jetson Nano and Raspberry Pi, while less powerful in computational terms, manage well with less complex models like BERTBase and RoBERTaBase.

Though the RTX 3090 excels in computational performance, it comes at the cost of significantly higher power consumption than the energy-efficient Jetson Nano and Raspberry Pi. Particularly in edge computing applications, where power resources are constrained, energy efficiency becomes a critical factor to weigh.

Choosing the appropriate model and platform hinges on the specific demands of the task at hand. In resource-limited environments where computational power and energy efficiency are paramount, opting for BERTBase or RoBERTaBase on the Jetson Nano or Raspberry Pi may prove advantageous. In scenarios like cloud computing, where computational prowess is paramount, the RTX 3090 paired with models like BERTBoost or RoBERTaBoost may become indispensable.

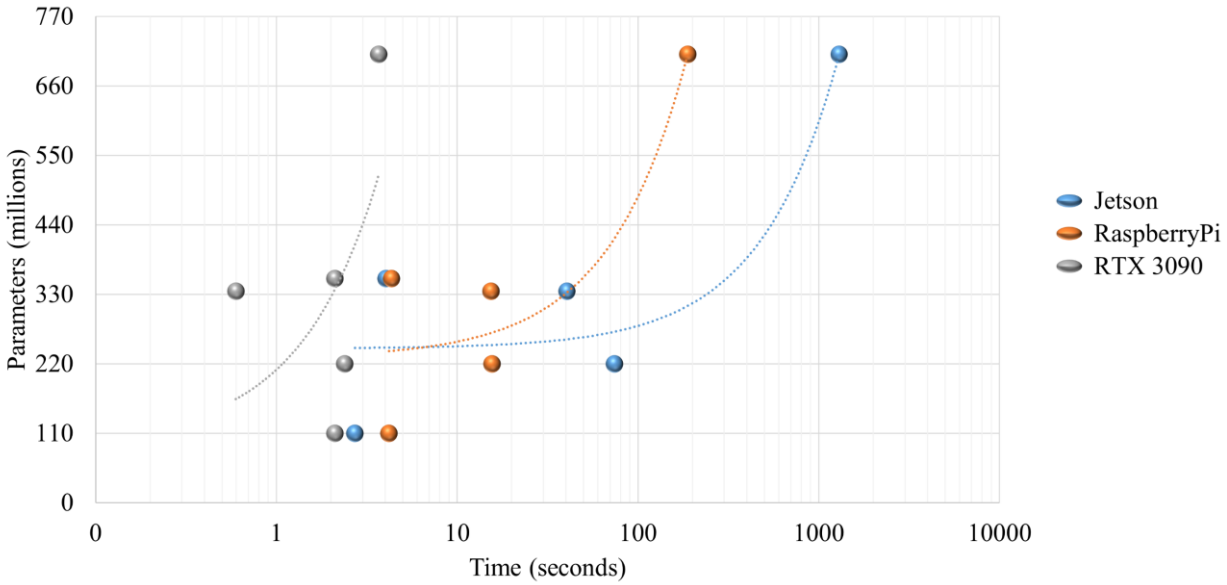



Figure 6.17 - Performance to complexity ratio in terms of parameters and time.

Figure 6.17 illustrates the performance-to-complexity ratio, showing the trendline of various platforms. It becomes evident that, on a global scale, as the complexity of the developed model and architecture increases, in this case, the application of an ensemble system with boosting and weighted voting, the platforms face significant challenges.

## 6.3. Key Remarks

The base versions of both BERTimbau and RoBERTa models appear to strike a commendable balance between speed and ACC on the Jetson Nano platform. They deliver rapid inference times coupled with accurate classifications and reasonably high confidence levels. Nevertheless, when it comes to the enhanced iterations of these models, while they do achieve elevated confidence levels, they come at the expense of considerably longer classification times, rendering them less suitable for real-time applications.

On the Raspberry Pi platform, the RoBERTa Base model consistently outperforms the other models. It consistently exhibits high correctness and confidence levels across various review lengths and sentiments. The BERTimbau Base model also demonstrates reasonably good performance but encounters challenges with 100-token neutral reviews. Meanwhile, BERTimbau Boosted showcases commendable correctness, albeit at a slower processing pace, whereas RoBERTa Boosted provides accurate classifications but notably extended processing times for certain categories.

Within the realm of RTX 3090, the RoBERTa Base model stands out as a robust performer, excelling in classification speed, confidence, and correctness. BERTimbau Base and BERTimbau Large offer competitive performances, displaying slightly varying confidence and correctness levels. On the other hand, RoBERTa Boosted, and BERTimbau Boosted yield decent results but entail slower classification times and exhibit slight variability in correctness and confidence levels, particularly for negative and neutral sentiments.

# 7. Conclusion

The advent of NLP related to the classification of sentiments led to advancements in this field. One of the key aspects of this work is the almost nonexistence of sentiment classification models in Portuguese. Creating a model based on TL, initially trained on other tasks, and then fine-tuned and retrained to understand the nuances of language used in reviews, often in an unconventional manner, is a monumental challenge.

## 7.1. Overview of the Work

In this study, we conducted a comprehensive analysis of SA in Portuguese restaurant reviews using TL with Transformer models in an edge computing environment, allowing IoT devices to process data without the need of communication with a data center.

The performance of Transformer models, particularly RoBERTa Base and BERTimbau Base, proved to be effective in providing accurate sentiment classifications in Portuguese restaurant reviews. These models offered a balanced trade-off between speed and ACC, making them suitable for real-time analysis.

On the other hand, the deployment of these models on edge computing devices, such as the Jetson Nano and Raspberry Pi, demonstrated their viability for resource-constrained environments. This opens opportunities for real-time SA applications, even with limited processing power.

The choice between base and boosted versions of models should be driven by specific requirements, considering the trade-off between inference speed and model confidence. For real-time applications, base models are recommended, while boosted models may be suitable for applications where higher confidence levels are crucial, even if it comes at the cost of longer processing times.

Regarding the literature review, our findings align with the latest advancements in terms of ACC, Sensitivity, and Specificity. These metrics serve as the standard for evaluating ML models of this nature. Utilizing genuine restaurant reviews created an atmosphere that encouraged diverse writing styles, often leaning toward irony. This aspect added an extra layer of complexity to our work.

On the software front, Python's and Pytorch's libraries significantly eased the implementation of our proposed models. These resources presented us with user-friendly functions to build and optimize the ANN and execute other essential processes for our AI system's development.

Throughout the solution's development, we noticed that testing the method demanded meticulous planning, primarily due to the algorithm's extensive training time. This extended duration arises from the network's intricacy, the volume of training data involved, and the hardware used. In contrast, testing code in other developmental areas typically doesn't entail such prolonged waiting periods, potentially slowing down the overall system development.

In summary, we successfully achieved our project objectives. The ensemble algorithm was successfully implemented and demonstrated commendable performance in terms of sensitivity, specificity, accuracy, and AUC. Altogether, this project made significant contributions to the relevant fields of knowledge and holds promise for future real-world applications, particularly in the realm of SA in Portuguese.

This research has been published in the Electronics journal, being available since the end of January 2024 [65].

## 7.2. Limitations of the Work

While our study has provided valuable insights into SA in Portuguese restaurant reviews using Transformer models in an edge computing setting, it is essential to acknowledge its limitations in terms of data size, language specificity, and edge device variability.

The availability of a larger and more diverse dataset could potentially improve the model's generalization to a wider range of reviews and sentiments.

This work focused exclusively on Portuguese restaurant reviews. Expanding the analysis to multiple languages or domains could enhance the model's versatility.

The performance of edge computing models can vary depending on the specific hardware and software configurations of the edge device. Future studies should consider different edge devices and their impact on model performance.

## 7.3. Future Work

Looking ahead, several avenues for future research and application emerge from this work, for example, exploring more advanced NN architectures beyond BERT and RoBERTa to improve the performance and efficiency of SA models.

Investigating ways to incorporate domain-specific knowledge or contextual information to enhance the SA model's understanding of specialized domains. Adapting the models to different domains within the restaurant industry, such as fast food, fine dining, or cafes, could improve their ACC for specific contexts.

Extending the research to develop models and techniques capable of performing SA on multiple languages, addressing the challenges of language diversity. Extending SA to multiple languages would increase the applicability of this work in global contexts. Investigating the transferability of models trained in Portuguese to other languages could be a promising direction.

Given the processing capacity of quad-core edge devices like the Jetson Nano or Raspberry Pi, future work could explore the integration of AI into automation devices used in the Programmable Logic Controller (PLC) industry. This could enhance real-time monitoring of industrial processes, enabling timely adjustments based on SA or reading information of data streams from sensors and machinery.

Developing real-time feedback systems in industrial automation powered by SA or reading information on edge devices together with edge computing in machine vision and Field Programmable Gate Arrays (FPGAs) for embedded vision could provide immediate insights into process quality and efficiency, facilitating proactive maintenance and optimization.

## 7.4. Final Remarks

In conclusion, this work not only establishes a foundation for SA in Portuguese restaurant reviews within an edge computing context but also holds the potential for significant contributions to the PLC industry and edge computing in machine vision by enhancing automation and decision-making processes, specifically in the building optimization industry.

# 8. References


[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, 'BERT: Pre-training of deep bidirectional transformers for language understanding', in *Proceedings of the 2019 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. doi: 10.18653/v1/N19-1423.

[2] Y. Liu *et al.*, 'RoBERTa: A Robustly Optimized BERT Pretraining Approach'. arXiv, Jul. 26, 2019. doi: 10.48550/arXiv.1907.11692.

[3] T. Brown *et al.*, 'Language Models are Few-Shot Learners', in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2020, pp. 1877–1901. Accessed: Jun. 01, 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] 'What Is Deep Learning? | How It Works, Techniques & Applications'. Accessed: Jun. 09, 2023. [Online]. Available: https://www.mathworks.com/discovery/deep-learning.html

[5] A. Mathew, A. Arul, and S. Sivakumari, 'Deep Learning Techniques: An Overview', 2021, pp. 599–608. doi: 10.1007/978-981-15-3383-9_54.

[6] M. Adnan, R. Sarno, and K. R. Sungkono, 'Sentiment Analysis of Restaurant Review with Classification Approach in the Decision Tree-J48 Algorithm', in *2019 International Seminar on Application for Technology of Information and Communication (iSemantic)*, Sep. 2019, pp. 121–126. doi: 10.1109/ISEMANTIC.2019.8884282.

[7] K. Zahoor, N. Z. Bawany, and S. Hamid, 'Sentiment Analysis and Classification of Restaurant Reviews using Machine Learning', in *2020 21st International Arab Conference on Information Technology (ACIT)*, Nov. 2020, pp. 1–6. doi: 10.1109/ACIT50332.2020.9300098.

[8] D. R. Patil, D. Shukla, A. Kumar, Y. Rajanak, and D. Y. Pratap Singh, 'Machine Learning for Sentiment Analysis and Classification of Restaurant Reviews', in *2022 3rd International Conference on Computing, Analytics and Networks (ICAN)*, Nov. 2022, pp. 1–5. doi: 10.1109/ICAN56228.2022.10007390.

[9] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, 'Improving Language Understanding by Generative Pre-Training', *Comput. Sci.*, 2018, [Online]. Available: https://api.semanticscholar.org/CorpusID:49313245

[10] D. Khurana, A. Koli, K. Khatter, and S. Singh, 'Natural Language Processing: State of The Art, Current Trends and Challenges', *Multimed. Tools Appl.*, vol. 82, pp. 3713–3744, Jul. 2022, doi: 10.1007/s11042-022-13428-4.

[11] F. Souza, R. Nogueira, and R. Lotufo, 'BERTimbau: Pretrained BERT Models for Brazilian Portuguese', 2020, pp. 403–417. doi: 10.1007/978-3-030-61377-8_28.

[12] J. Rodrigues *et al.*, 'Advancing Neural Encoding of Portuguese with Transformer Albertina PT-*'. arXiv, 2023. doi: 10.48550/arXiv.2305.06721.

[13] S. Haykin, *Redes Neurais: Princípios e Prática*, 2 nd, ed. Bookman Editora, 2001.

[14] J. Monteiro, 'Acquisition and processing of EEG signal for neurologic disorders associated with memory', Instituto Superior de Engenharia do Porto, 2016. doi: 10.13140/RG.2.2.25614.77121.

[15] S. Razavi and B. A. Tolson, 'A New Formulation for Feedforward Neural Networks', *IEEE Trans. Neural Netw.*, vol. 22, no. 10, pp. 1588–1598, Oct. 2011, doi: 10.1109/TNN.2011.2163169.

[16] S. Raschka, *Python Machine Learning*. Packt Publishing Ltd, 2015.

[17] G. Bonaccorso, *Machine Learning Algorithms*. Packt Publishing Ltd, 2017.

[18] K. Hornik, M. Stinchcombe, and H. White, 'Multilayer feedforward networks are universal approximators', *Neural Netw.*, vol. 2, no. 5, pp. 359–366, Jan. 1989, doi: 10.1016/0893-6080(89)90020-8.

[19] S. I. Gallant, *Neural Network Learning and Expert Systems*. MIT Press, 1993.
[20] N. Zhang, D. Lei, and J. F. Zhao, 'An Improved Adagrad Gradient Descent Optimization Algorithm', in *2018 Chinese Automation Congress (CAC)*, Nov. 2018, pp. 2359–2362. doi: 10.1109/CAC.2018.8623271.
[21] Y. Yu, L. Zhang, L. Chen, and Z. Qin, 'Adversarial Samples Generation Based on RMSProp', in *2021 IEEE 6th International Conference on Signal and Image Processing (ICSIP)*, Oct. 2021, pp. 1134–1138. doi: 10.1109/ICSIP52628.2021.9688946.
[22] D. M. Hawkins, 'The Problem of Overfitting', *J. Chem. Inf. Comput. Sci.*, vol. 44, no. 1, pp. 1–12, Jan. 2004, doi: 10.1021/ci0342472.
[23] H. K. Jabbar and R. Z. Khan, 'Methods to Avoid Over-Fitting and Under-Fitting in Supervised Machine Learning (Comparative Study)', in *Computer Science, Communication and Instrumentation Devices*, Research Publishing Services, 2014, pp. 163–172. doi: 10.3850/978-981-09-5247-1_017.
[24] L. Prechelt, 'Automatic early stopping using cross validation: quantifying the criteria', *Neural Netw.*, vol. 11, no. 4, pp. 761–767, Jun. 1998, doi: 10.1016/S0893-6080(98)00010-0.
[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 'Dropout: A Simple Way to Prevent Neural Networks from Overfitting', *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014, [Online]. Available: https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
[26] 'sklearn.model_selection.StratifiedKFold', scikit-learn. Accessed: Jun. 06, 2023. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
[27] Bharathi, 'Latest Guide on Confusion Matrix for Multi-Class Classification', Analytics Vidhya. Accessed: Jul. 22, 2023. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/confusion-matrix-for-multi-class-classification/
[28] J. Brownlee, 'One-vs-Rest and One-vs-One for Multi-Class Classification', MachineLearningMastery.com. Accessed: Jul. 22, 2023. [Online]. Available: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/
[29] D. Ballabio, F. Grisoni, and R. Todeschini, 'Multivariate comparison of classification performance measures', *Chemom. Intell. Lab. Syst.*, vol. 174, pp. 33–44, Mar. 2018, doi: 10.1016/j.chemolab.2017.12.004.
[30] L. Cuadros-Rodríguez, E. Pérez-Castaño, and C. Ruiz-Samblás, 'Quality performance metrics in multivariate classification methods for qualitative analysis', *TrAC Trends Anal. Chem.*, vol. 80, pp. 612–624, Jun. 2016, doi: 10.1016/j.trac.2016.04.021.
[31] D. Jurafsky and J. H. Martin, *Speech and Language Processing, An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 3 ed, Draft. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf
[32] A. Vaswani *et al.*, 'Attention Is All You Need', in *31st Conference on Neural Information Processing Systems*, Curran Associates, Inc., 2017. doi: 10.48550/arXiv.1706.03762.
[33] J. Vig, 'A Multiscale Visualization of Attention in the Transformer Model', in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 37–42. doi: 10.18653/v1/P19-3007.
[34] B. Chiu and S. Baker, 'Word embeddings for biomedical natural language processing: A survey', *Lang. Linguist. Compass*, vol. 14, no. 12, Dec. 2020, doi: 10.1111/lnc3.12402.
[35] Z. Liu, 'Effective Transfer Learning for Low-Resource Natural Language Understanding'. arXiv, Aug. 19, 2022. Accessed: Sep. 16, 2022. [Online]. Available: http://arxiv.org/abs/2208.09180

[36] M. Iman, H. R. Arabnia, and K. Rasheed, ‘A Review of Deep Transfer Learning and Recent Advancements’, *Technologies*, vol. 11, no. 2, Art. no. 2, Apr. 2023, doi: 10.3390/technologies11020040.
[37] A. J. Ferreira and M. A. T. Figueiredo, ‘Boosting Algorithms: A Review of Methods, Theory, and Applications’, in *Ensemble Machine Learning*, C. Zhang and Y. Ma, Eds., Boston, MA: Springer US, 2012, pp. 35–85. doi: 10.1007/978-1-4419-9326-7_2.
[38] Y. Freund and R. E. Schapire, ‘Experiments with a New Boosting Algorithm’, 2001, [Online]. Available: https://www.researchgate.net/publication/237536819_Experiments_with_a_New_Boosting_Algorithm
[39] Y. Freund and R. E. Schapire, ‘A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting’, *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, Aug. 1997, doi: 10.1006/jcss.1997.1504.
[40] Y. Freund and R. E. Schapire, ‘Game theory, on-line prediction and boosting’, in *Proceedings of the ninth annual conference on Computational learning theory - COLT ’96*, Desenzano del Garda, Italy: ACM Press, 1996, pp. 325–332. doi: 10.1145/238061.238163.
[41] J. Zhu, S. Rosset, H. Zou, and T. Hastie, ‘Multi-class AdaBoost’, *Stat. Interface*, vol. 2, Feb. 2006, doi: 10.4310/SII.2009.v2.n3.a8.
[42] ‘EnsembleVoteClassifier: A majority voting classifier - mlxtend’. Accessed: Jun. 05, 2023. [Online]. Available: http://rasbt.github.io/mlxtend/user_guide/classifier/EnsembleVoteClassifier/
[43] B. Liu, *Sentiment Analysis and Opinion Mining*, 1st ed. in Synthesis Lectures on Human Language Technologies. Springer Nature, 2022. [Online]. Available: https://link.springer.com/book/10.1007/978-3-031-02145-9
[44] H. He and E. A. Garcia, ‘Learning from Imbalanced Data’, *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009, doi: 10.1109/TKDE.2008.239.
[45] T. Raeder, G. Forman, and N. V. Chawla, ‘Learning from Imbalanced Data: Evaluation Matters’, in *Data Mining: Foundations and Intelligent Paradigms: Volume 1: Clustering, Association and Classification*, D. E. Holmes and L. C. Jain, Eds., in Intelligent Systems Reference Library. , Berlin, Heidelberg: Springer, 2012, pp. 315–331. doi: 10.1007/978-3-642-23166-7_12.
[46] L. F. S. Britto, L. A. S. Pessoa, and S. C. C. Agostinho, ‘Cross-Domain Sentiment Analysis in Portuguese using BERT’, in *Anais do Encontro Nacional de Inteligência Artificial e Computacional (ENIAC)*, SBC, Nov. 2022, pp. 61–72. doi: 10.5753/eniac.2022.227217.
[47] Y. Wu *et al.*, ‘Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation’. arXiv, Oct. 08, 2016. doi: 10.48550/arXiv.1609.08144.
[48] P. Izsak, M. Berchansky, and O. Levy, ‘How to Train BERT with an Academic Budget’. arXiv, Sep. 09, 2021. doi: 10.48550/arXiv.2104.07705.
[49] M. Mosbach, M. Andriushchenko, and D. Klakow, ‘On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines’. arXiv, Mar. 25, 2021. doi: 10.48550/arXiv.2006.04884.
[50] T. Fawcett, ‘An introduction to ROC analysis’, *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006, doi: 10.1016/j.patrec.2005.10.010.
[51] A. Kurniasih and L. P. Manik, ‘On the Role of Text Preprocessing in BERT Embedding-based DNNs for Classifying Informal Texts’, *Int. J. Adv. Comput. Sci. Appl.*, vol. 13, p. 927, Jun. 2022, doi: 10.14569/IJACSA.2022.01306109.
[52] J. Sarzynska-Wawer *et al.*, ‘Detecting formal thought disorder by deep contextualized word representations’, *Psychiatry Res.*, vol. 304, p. 114135, Oct. 2021, doi: 10.1016/j.psychres.2021.114135.
[53] J. Howard and S. Ruder, ‘Universal Language Model Fine-tuning for Text Classification’. arXiv, May 23, 2018. doi: 10.48550/arXiv.1801.06146.

[54] A. Moura, P. Lima, F. Mendonça, S. S. Mostafa, and F. Morgado-Dias, 'On the Use of Transformer-Based Models for Intent Detection Using Clustering Algorithms', *Appl. Sci.*, vol. 13, no. 8, Art. no. 8, Jan. 2023, doi: 10.3390/app13085178.
[55] É. P. Lopes, L. Freitas, G. Gomes, G. Lemos, L. O. Hammes, and U. B. Corrêa, 'Exploring BERT for Aspect-based Sentiment Analysis in Portuguese Language', *Int. FLAIRS Conf. Proc.*, vol. 35, May 2022, doi: 10.32473/flairs.v35i.130601.
[56] F. D. Souza and J. B. de O. e S. Filho, 'Embedding generation for text classification of Brazilian Portuguese user reviews: from bag-of-words to transformers', *Neural Comput. Appl.*, Dec. 2022, doi: 10.1007/s00521-022-08068-6.
[57] L. I. Kuncheva, *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley & Sons, 2014.
[58] J. R. S. Gomes *et al.*, 'Deep Learning Brasil at ABSAPT 2022: Portuguese Transformer Ensemble Approaches', *CEUR-WS*, Sep. 2022, [Online]. Available: https://ceur-ws.org/Vol-3202/absapt-paper1.pdf
[59] F. L. dos Santos and M. Ladeira, 'The Role of Text Pre-processing in Opinion Mining on a Social Media Language Dataset', in *2014 Brazilian Conference on Intelligent Systems*, Sao Paulo, Brazil: IEEE, Oct. 2014, pp. 50–54. doi: 10.1109/BRACIS.2014.20.
[60] 'neuralmind (NeuralMind Inteligência Artificial)'. Accessed: Jul. 24, 2023. [Online]. Available: https://huggingface.co/neuralmind
[61] 'thegoodfellas/tgf-xlm-roberta-base-pt-br · Hugging Face'. Accessed: Jul. 25, 2023. [Online]. Available: https://huggingface.co/thegoodfellas/tgf-xlm-roberta-base-pt-br
[62] 'Jetson Nano Developer Kit', NVIDIA Developer. Accessed: Jul. 17, 2023. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit
[63] Q-engineering, 'PyTorch wheels for Jetson Nano, TX2, (AGX) Xavier'. Aug. 07, 2023. Accessed: Aug. 07, 2023. [Online]. Available: https://github.com/Qengineering/PyTorch-Jetson-Nano
[64] R. P. Ltd, 'Raspberry Pi', Raspberry Pi. Accessed: Jul. 17, 2023. [Online]. Available: https://www.raspberrypi.com/
[65] A. Branco, D. Parada, M. Silva, F. Mendonça, S. S. Mostafa, and F. Morgado-Dias, 'Sentiment Analysis in Portuguese Restaurant Reviews: Application of Transformer Models in Edge Computing', *Electronics*, vol. 13, no. 3, Art. no. 3, Jan. 2024, doi: 10.3390/electronics13030589.

# Attachments

## A. Model Creation Scripts

### I. Script for SentAnalysisPt

```
# Import libraries:
import torch
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from torch import nn
from torchmetrics import AUROC
from pycm import ConfusionMatrix
from transformers import logging
from collections import defaultdict
from time import time, strftime, gmtime
from sklearn.metrics import classification_report
from sklearn.utils.class_weight import compute_class_weight
from transformers import BertModel, BertTokenizer, get_linear_sched-
ule_with_warmup
# Libraries created localy:
from PytorchTools import utils as ut
from EarlyStop import earlyStop as es
from PytorchTools.getMetrics import show_train_history
from PytorchTools.mySaveMetrics import selectMetrics, save, create_dir

# To ignore deprecated warnings:
warnings.filterwarnings("ignore")
# BERT warning --> this warning means that during your training, you're
not using the pooler in order to compute the loss
logging.set_verbosity_error()

# set CUDA
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Constants:
freeze_bert = False # freeze BERT model to train only the classifier

# Save name for metrics and checkpoint:
save_name = 'SentAnalysisPt' # change the name to avoid overwrite
save_metrics = True # save the best model parameters
save_model = True # save the model

# Hyperparameters:
max_len = 100
batch_size = 128
epochs = 20
```

```
patience = 10
delta = 0.0003
Dropout = 0.2
l_r = 2e-5

# Set Path for save checkpoint:
path = './NewBert/SentimentAnalysisClassPt/'
# set path for save the model:
path_model = path + 'Model_' + save_name + '/'

# Language
lang = 'Pt'
num_class = 3
data_path = './Data/FinalData' + lang + '/'
class_names = ['Negative', 'Neutral', 'Positive']

# Set the name to import the model:
model_name = 'neuralmind/bert-base-portuguese-cased'

#%% Load Dataset:
# Load data 80% of dataset for training
df_train = pd.read_csv(data_path + 'trainData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 70% of the 20% remaining of dataset for valifation
df_val = pd.read_csv(data_path + 'valData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 30% of the 20% remaining of dataset for testing
df_test = pd.read_csv(data_path + 'testData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)

# Load Tokenizer:
tokenizer = BertTokenizer.from_pretrained(model_name) # BertTokenizer

# Creating Data Loader:
train_data_loader = ut.create_data_loader(df_train, tokenizer, max_len,
batch_size)
val_data_loader = ut.create_data_loader(df_val, tokenizer, max_len,
batch_size)
test_data_loader = ut.create_data_loader(df_test, tokenizer, max_len,
batch_size)

#%% Classifier using BERT:
# Create a classifier that uses the BERT model
class SentimentClassifier(nn.Module):
  def __init__(self, n_classes, model_name):
    super(SentimentClassifier, self).__init__()
    self.bert = BertModel.from_pretrained(model_name, re-
turn_dict=False)
    # Freeze BERT parameters:
```

```
    if freeze_bert:
        # freeze all the parameters excluding classifier
        for param in self.bert.named_parameters():
            # param = (nome_da_layer, parametros)
            param[1].requires_grad = False
    self.drop = nn.Dropout(Dropout) # layer for regulation
    #The last_hidden_state is a sequence of hidden states of the last
layer of the model
    self.linear = nn.Linear(self.bert.config.hidden_size, n_classes)
  def forward(self, input_ids, attention_mask):
    _, pooled_output = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask)
    output = self.drop(pooled_output)
    output = self.linear(output)
    return output

#%% Full model to GPU and compute weights:
model = SentimentClassifier(len(class_names),model_name)
model = model.to(device)

data = next(iter(train_data_loader))
input_ids = data['input_ids'].to(device)
attention_mask = data['attention_mask'].to(device)

# Train parameters optimizer:
optimizer = torch.optim.Adagrad(model.parameters(), lr=l_r)

# Scheduler function:
total_steps = len(train_data_loader) * epochs
warmup_steps = total_steps*0.1 # 10% de total_steps
scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=warmup_steps,
                                            num_training_steps=to-
tal_steps)

# Compute the Class Weights:
class_weights = compute_class_weight(class_weight = "balanced",
                                     classes = np.unique(df_train['ra-
ting']),
                                     y = df_train['rating'])
class_wts_dic = dict(zip(np.unique(df_train['rating']), class_weights))
class_wts = [class_wts_dic[0]]
for index in range(1,np.count_nonzero(np.unique(df_train['rat-
ing']))+1):
    class_wts.append(class_wts_dic[index])

# Loss function:
```

```
loss_fn = nn.CrossEntropyLoss(weight = torch.ten-
sor(class_wts,dtype=torch.float)).to(device)

#%% Early stopping:
early_stopping = es.EarlyStopping(verbose=True, patience=patience, op-
timizer=optimizer,
                                  path=path, path_name=save_name,
delta=delta)

history = defaultdict(list)
best_accuracy = 0
t = time()
print("\nTraining started...")

# log for final visualization
log = ''
log += " Log of classifier: " + save_name + "\n"
for epoch in range(epochs):
    print(f'Epoch {epoch + 1}/{epochs}')
    # Train function:
    train_acc, train_loss, train_auc =
ut.train_epoch(model,train_data_loader,loss_fn,optimizer,
                                                    device,sched-
uler,len(df_train))
    print(f'Train:  loss {train_loss:.3f}   accuracy {train_acc:.3f}')

    # Validation function:
    val_acc, val_loss, val_auc =
ut.eval_model(model,val_data_loader,loss_fn,
                                              device,len(df_val))

    print(f'Val:    loss {val_loss:.3f}   accuracy {val_acc:.3f}')
    print()

    # Save metrics histoty:
    history['train_acc'].append(train_acc.item())
    history['train_loss'].append(train_loss)
    history['train_auc'].append(train_auc.item())
    history['val_acc'].append(val_acc.item())
    history['val_loss'].append(val_loss)
    history['val_auc'].append(val_auc.item())

    # follow the accuracy metric:
    if val_acc > best_accuracy:
        # torch.save(model.state_dict(), './Mod-
els/best_model_state.bin')
        best_accuracy = val_acc
        print(f'\n Best Accuracy: {best_accuracy.item():.3f} ')
```

```
    # early stopping
    early_stopping(val_auc.item(), model)
    if early_stopping.early_stop:
       print("\n Stop at epoch:", epoch+1, "\n")
       break

trainTime = time()-t
print(f"Training finished...(Elapsed time: {strftime('%H:%M:%S',
gmtime(trainTime))})")

# Plot training Losses:
loss_hist = (history['train_loss'].copy(), history['val_loss'].copy())
show_train_history(loss_hist, 'loss', early_stopping.best_epoch,
save_name)
# Plot training Accuracy
acc_hist = (history['train_acc'].copy(), history['val_acc'].copy())
show_train_history(acc_hist, 'acc', early_stopping.best_epoch,
save_name)
# Plot training AUC
auc_hist = (history['train_auc'].copy(), history['val_auc'].copy())
show_train_history(auc_hist, 'auc', early_stopping.best_epoch,
save_name)

# Load best model parameters:
load_path= path + 'checkpoint_' + save_name +'.pth'
checkpoint = torch.load(load_path)
model.load_state_dict(checkpoint['model_state_dict'])

# Save the best model
if save_model:
    model_path = create_dir(path[:-1], f"Model_{save_name}")
    torch.save({'epoch': epoch,
           'model_state_dict': model.state_dict(),
           'acc': val_acc}, path_model + 'model' + save_name + '.pth')
    print("\nFull Model Save...")

log += f" Training time: {strftime('%H:%M:%S', gmtime(trainTime))}\n"
log += f" Best epoch: {early_stopping.best_epoch}\n"

#%% Predictions:
print("\nTesting model...\n")
y_review_texts, y_pred, y_pred_probs, y_test = ut.get_predic-
tions(model, test_data_loader)

# Classification report:
classTest = classification_report(y_test, y_pred, tar-
get_names=class_names)
# print(classTest) # visualização teste
```

```
cmTest = ConfusionMatrix(actual_vector= y_test.tolist(), predict_vector
= y_pred.tolist())
# print (f'\n === Confusion Matrix ===\n  \n {cmTest}')  # visualização
teste

# Confusion Matrix:
cmTest.plot(cmap=plt.cm.Blues, number_label=True,
            normalized=True,    # Using normalized because dataset is
imbalanced
            plot_lib="seaborn",
            title="Confusion Matrix")
plt.rcParams['font.family'] = 'DeJavu Serif'
plt.rcParams['font.serif'] = ['Times New Roman']
plt.ylabel('True Sentiment')
plt.xlabel('Predicted Sentiment')
plt.xticks(np.arange(3)+0.5, class_names)
plt.yticks(np.arange(3)+0.5, class_names)
plt.show()

# compute AUC_ROC:
auroc = AUROC(num_classes=num_class)
ovr_AUC = auroc(y_pred_probs, y_test).item()

#%% Store metrics:
# Names of metrics to store from PyCM
LabelsOverall = [k for k,v in cmTest.overall_stat.items() if isin-
stance(v,(float,int)) ]
LabelsClass   = [k for k,v in cmTest.class_stat.items() if isin-
stance(v[0],(float,int))]

# Extract metrics from cm object
rawOverall = selectMetrics(cmTest.overall_stat,LabelsOverall) +
[ovr_AUC]
rawClass   = np.asarray([list(d.values()) for d in select-
Metrics(cmTest.class_stat,LabelsClass)])

# Convert into dictionary
mOverall = {k:v for k,v in zip( LabelsOverall, rawOverall )}
mClass = {k:{j:v for j,v in enumerate(varray)} for k,varray in zip( La-
belsClass, rawClass )}

log += f" Accuracy: {rawOverall[0]:.3f}\n"
log += f" AUC value: {ovr_AUC:.3f}\n"
log += f" F1 Macro: {mOverall['F1 Macro']:.3f}\n"
log += f"\n Classification report \n{classTest}\n"

# Print Log:
print(log,sep='\n')
with open(f'{path_model}log.txt','a') as log_file:
```

```
    log_file.write(log)

# Save the model
if save_metrics:
    save(save_name, path=path,
        trainBestEpoch = [early_stopping.best_epoch],
        trainTime = [trainTime],
        trainLOSS = loss_hist,
        trainACC = acc_hist,
        trainAUC = auc_hist,
        pycmFinalAUC = ovr_AUC,
        pycmOverall = mOverall,
        pycmClass = mClass,
        pycmCM = cmTest.to_array(normalized=True)) # confussion matrix

#============================== END CODE ==============================
```

## II. Script for SentAnalyPtRoberta

```
# Import libraries:
import torch
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from torch import nn
from torchmetrics import AUROC
from pycm import ConfusionMatrix
from transformers import logging
from collections import defaultdict
from time import time, strftime, gmtime
from sklearn.metrics import classification_report
from sklearn.utils.class_weight import compute_class_weight
from transformers import AutoTokenizer, AutoModel, get_linear_sched-
ule_with_warmup
# Libraries created localy:
from PytorchTools import utils as ut
from EarlyStop import earlyStop as es
from PytorchTools.getMetrics import show_train_history
from PytorchTools.mySaveMetrics import selectMetrics, save, create_dir

# To ignore deprecated warnings:
warnings.filterwarnings("ignore")
# BERT warning --> this warning means that during your training, you're
not using the pooler in order to compute the loss
logging.set_verbosity_error()

# set CUDA
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Constants:
freeze_bert = False # freeze BERT model to train only the classifier

# Save name for metrics and checkpoint:
save_name = 'SentAnalysisPtRoberta' # change the name to avoid over-
write
save_metrics = True # save the best model parameters
save_model = True # save the model

# Hyperparameters:
max_len = 100
batch_size = 64
epochs = 20
patience = 10
delta = 0.0003
Dropout = 0.2
```

```
# l_r = 2e-5
l_r = 1e-4

# Set Path for save checkpoint:
path = './NewRoberta/SentimentAnalysisClassPtRoberta/'
# set path for save the model:
path_model = path + 'Model_' + save_name + '/'

# Language
lang = 'Pt'
num_class = 3
data_path = './Data/FinalData' + lang + '/'
class_names = ['Negative', 'Neutral', 'Positive']

# Set the name to import the model:
model_name = "thegoodfellas/tgf-xlm-roberta-base-pt-br"


#%% Load Dataset:
# Load data 80% of dataset for training
df_train = pd.read_csv(data_path + 'trainData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 70% of the 20% remaining of dataset for valifation
df_val = pd.read_csv(data_path + 'valData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 30% of the 20% remaining of dataset for testing
df_test = pd.read_csv(data_path + 'testData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)

# Load Tokenizer:
# tokenizer = AutoTokenizer.from_pretrained("thegoodfellas/tgf-xlm-rob-
erta-base-pt-br")
tokenizer = AutoTokenizer.from_pretrained(model_name)

# model = AutoModelForMaskedLM.from_pretrained("thegoodfellas/tgf-xlm-
roberta-base-pt-br")

# Creating Data Loader:
train_data_loader = ut.create_data_loader(df_train, tokenizer, max_len,
batch_size)
val_data_loader = ut.create_data_loader(df_val, tokenizer, max_len,
batch_size)
test_data_loader = ut.create_data_loader(df_test, tokenizer, max_len,
batch_size)

#%% Classifier using Roberta:
class CustomRobertaModel(nn.Module):
    def __init__(self, n_classes, model_name):
        super(CustomRobertaModel, self).__init__()
```

```
        # self.num_labels = num_labels  # get number of labels
        self.roberta = AutoModel.from_pretrained(model_name, re-
turn_dict=False)
        # self.roberta = RobertaModel.from_pretrained(model_name, re-
turn_dict=False)
            # config.roberta_model_name)  # get pre-trained model
        # set up percentage of drop
        self.drop = nn.Dropout(Dropout)
        # defining final output layer
        self.linear = nn.Linear(self.roberta.config.hidden_size,
n_classes)

    def forward(self, input_ids, attention_mask):
        _, pooled_output = self.roberta(input_ids=input_ids, atten-
tion_mask=attention_mask)
        output = self.drop(pooled_output)
        output = self.linear(output)
        return output

#%% Full model to GPU and compute weights:
model = CustomRobertaModel(len(class_names),model_name)
model = model.to(device)

data = next(iter(train_data_loader))
input_ids = data['input_ids'].to(device)
attention_mask = data['attention_mask'].to(device)

# Train parameters optimizer:
optimizer = torch.optim.Adagrad(model.parameters(), lr=l_r)

# Scheduler function:
total_steps = len(train_data_loader) * epochs
# warmup_steps = total_steps*0.1 # 10% de total_steps
warmup_steps = 1000
scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=warmup_steps, num_training_steps=total_steps)

# Compute the Class Weights:
class_weights = compute_class_weight(class_weight = "balanced",
classes = np.unique(df_train['rating']), y = df_train['rating'])
class_wts_dic = dict(zip(np.unique(df_train['rating']), class_weights))
class_wts = [class_wts_dic[0]]
for index in range(1,np.count_nonzero(np.unique(df_train['rat-
ing']))+1):
    class_wts.append(class_wts_dic[index])

# Loss function:
loss_fn = nn.CrossEntropyLoss(weight = torch.ten-
sor(class_wts,dtype=torch.float)).to(device)
```

```
#%% Early stopping:
early_stopping = es.EarlyStopping(verbose=True, patience=patience, op-
timizer=optimizer, path=path, path_name=save_name, delta=delta)

history = defaultdict(list)
best_accuracy = 0
t = time()
print("\nTraining started...")

# log for  final visualization
log = ''
log += " Log of classifier: " + save_name + "\n"
for epoch in range(epochs):
    print(f'Epoch {epoch + 1}/{epochs}')
    # Train function:
    train_acc, train_loss, train_auc =
ut.train_epoch(model,train_data_loader,loss_fn,optimizer,device,sched-
uler,len(df_train))
    print(f'Train:  loss {train_loss:.3f}   accuracy {train_acc:.3f}')
    # Validation function:
    val_acc, val_loss, val_auc =
ut.eval_model(model,val_data_loader,loss_fn, device,len(df_val))
    print(f'Val:    loss {val_loss:.3f}   accuracy {val_acc:.3f}')
    print()
    # Save metrics histoty:
    history['train_acc'].append(train_acc.item())
    history['train_loss'].append(train_loss)
    history['train_auc'].append(train_auc.item())
    history['val_acc'].append(val_acc.item())
    history['val_loss'].append(val_loss)
    history['val_auc'].append(val_auc.item())

    # follow the accuracy metric:
    if val_acc > best_accuracy:
        # torch.save(model.state_dict(), './Mod-
els/best_model_state.bin')
        best_accuracy = val_acc
        print(f'\n Best Accuracy: {best_accuracy.item():.3f} ')

    # early stopping
    early_stopping(val_auc.item(), model)
    if early_stopping.early_stop:
       print("\n Stop at epoch:", epoch+1, "\n")
       break

trainTime = time()-t
print(f"Training finished...(Elapsed time: {strftime('%H:%M:%S',
gmtime(trainTime))})")
```

```
# Plot training Losses:
loss_hist = (history['train_loss'].copy(), history['val_loss'].copy())
show_train_history(loss_hist, 'loss', early_stopping.best_epoch,
save_name)
# Plot training Accuracy
acc_hist = (history['train_acc'].copy(), history['val_acc'].copy())
show_train_history(acc_hist, 'acc', early_stopping.best_epoch,
save_name)
# Plot training AUC
auc_hist = (history['train_auc'].copy(), history['val_auc'].copy())
show_train_history(auc_hist, 'auc', early_stopping.best_epoch,
save_name)

# Load best model parameters:
load_path= path + 'checkpoint_' + save_name +'.pth'
checkpoint = torch.load(load_path)
model.load_state_dict(checkpoint['model_state_dict'])

# Save the best model
if save_model:
    model_path = create_dir(path[:-1], f"Model_{save_name}")
    torch.save({'epoch': epoch,
           'model_state_dict': model.state_dict(),
           'acc': val_acc}, path_model + 'model' + save_name + '.pth')
    print("\nFull Model Save...")

log += f" Training time: {strftime('%H:%M:%S', gmtime(trainTime))}\n"
log += f" Best epoch: {early_stopping.best_epoch}\n"

#%% Predictions:
print("\nTesting model...\n")
y_review_texts, y_pred, y_pred_probs, y_test = ut.get_predic-
tions(model, test_data_loader)

# Classification report:
classTest = classification_report(y_test, y_pred, tar-
get_names=class_names)
# print(classTest) # visualização teste

cmTest = ConfusionMatrix(actual_vector= y_test.tolist(), predict_vector
= y_pred.tolist())
# print (f'\n === Confusion Matrix ===\n  \n {cmTest}')  # visualização
teste

# Confusion Matrix:
cmTest.plot(cmap=plt.cm.Blues, number_label=True,
            normalized=True,    # Using normalized because dataset in
imbalanced
```

```
            plot_lib="seaborn",
            title="Confusion Matrix")
plt.rcParams['font.family'] = 'DeJavu Serif'
plt.rcParams['font.serif'] = ['Times New Roman']
plt.ylabel('True Sentiment')
plt.xlabel('Predicted Sentiment')
plt.xticks(np.arange(3)+0.5, class_names)
plt.yticks(np.arange(3)+0.5, class_names)
plt.show()

# compute AUC_ROC:
auroc = AUROC(num_classes=num_class)
ovr_AUC = auroc(y_pred_probs, y_test).item()

#%% Store metrics:
# Names of metrics to store from PyCM
LabelsOverall = [k for k,v in cmTest.overall_stat.items() if isin-
stance(v,(float,int)) ]
LabelsClass   = [k for k,v in cmTest.class_stat.items() if isin-
stance(v[0],(float,int))]

# Extract metrics from cm object
rawOverall = selectMetrics(cmTest.overall_stat,LabelsOverall) +
[ovr_AUC]
rawClass   = np.asarray([list(d.values()) for d in select-
Metrics(cmTest.class_stat,LabelsClass)])

# Convert into dictionary
mOverall = {k:v for k,v in zip( LabelsOverall, rawOverall )}
mClass = {k:{j:v for j,v in enumerate(varray)} for k,varray in zip( La-
belsClass, rawClass )}

log += f" Accuracy: {rawOverall[0]:.3f}\n"
log += f" AUC value: {ovr_AUC:.3f}\n"
log += f" F1 Macro: {mOverall['F1 Macro']:.3f}\n"
log += f"\n Classification report \n{classTest}\n"

# Print Log:
print(log,sep='\n')
with open(f'{path_model}log.txt','a') as log_file:
    log_file.write(log)

# Save the model
if save_metrics:
    save(save_name, path=path,
        trainBestEpoch = [early_stopping.best_epoch],
        trainTime = [trainTime],
        trainLOSS = loss_hist,
        trainACC = acc_hist,
```

```
        trainAUC = auc_hist,
        pycmFinalAUC = ovr_AUC,
        pycmOverall = mOverall,
        pycmClass = mClass,
        pycmCM = cmTest.to_array(normalized=True)) # confussion matrix

#============================== END CODE ==============================
```

## III. Script for SentAnalyPtAdaboost (alike SentAnalyPtAdaboostRoberta)

```
"""
Note: this script uses adaboost from paper Multi-class adaboost -> zhu
et.al 2006  -> SAMME

    SAMME - Stagewise Additive Modeling using a Multi-class Exponential
loss function
"""
# Import libraries:
import torch
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import defaultdict
from torch import nn
from torchmetrics import AUROC
from pycm import ConfusionMatrix
from transformers import logging
from time import time, strftime, gmtime
from sklearn.metrics import classification_report
from transformers import BertModel, BertTokenizer, get_linear_sched-
ule_with_warmup

# Libraries created localy:
from EarlyStop import earlyStopAdaboost as es_ada
from PytorchTools import utilsAdaboost as uta
from PytorchTools.getMetrics import show_train_history
from PytorchTools.mySaveMetrics import selectMetrics, save, create_dir

# To ignore deprecated warnings:
warnings.filterwarnings("ignore")
# BERT warning --> this warning means that during your training, you're
not
# using the pooler in order to compute the loss
logging.set_verbosity_error()

# set CUDA
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Constants:
freeze_bert = False # freeze BERT model to train only the classifier

# Number of weak classifiers:
NumberIterations = 2  # number of weak classifiers

# Save name for metrics and checkpoint:
save_name = 'SentAnalyPtAdaBoost_DANIEL_' + str(NumberIterations) #
change the name to avoid overwrite
```

```
save_metrics = True # save the best model parameters
save_model = True # save the model

# Hyperparameters:
max_len = 100
batch_size = 128
epochs = 20
patience = 10  # need to increase from 10 to --->
delta = 0.0003   # need to decrease to have 4 decimals
Dropout = 0.2
l_r = 2e-5

# Set Path for save checkpoint:
path = './NewBert/SentimentAnalysisClassPtAdaBoost/'
# set path for save the model:
path_model = path + 'Model_' + save_name + '/'

# Language
lang = 'Pt'
num_class = 3
data_path = './Data/FinalData' + lang + '/'
class_names = ['Negative', 'Neutral', 'Positive']

# Set the name to import the model:
model_name = 'neuralmind/bert-base-portuguese-cased'

#%% Load Dataset:
# Load data 80% of dataset for training
df_train = pd.read_csv(data_path + '/trainData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 70% of the 20% remaining of dataset for valifation
df_val = pd.read_csv(data_path + '/valData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)
# Load 30% of the 20% remaining of dataset for testing
df_test = pd.read_csv(data_path + '/testData_' + str(num_class) +
'Class_'+lang +'.csv', index_col=0)

# Load Tokenizer:
tokenizer = BertTokenizer.from_pretrained(model_name) # BertTokenizer

# Creating Data Loader:
train_data_loader = uta.create_data_loader(df_train, tokenizer,
max_len, batch_size)
val_data_loader = uta.create_data_loader(df_val, tokenizer, max_len,
batch_size)
test_data_loader = uta.create_data_loader(df_test, tokenizer, max_len,
batch_size)
```

```
# df_train.shape, df_val.shape, df_test.shape # show the shape of the
datasets

#%% AdaBoost Functions:
# this is the implementation os SAMME - Stagewise Additive Modeling us-
ing a Multi-class Exponential loss function

def compute_error(y, y_pred, w_i):
    return (sum(w_i * (np.not_equal(y, y_pred)).astype(int)))/sum(w_i)

def compute_alpha(error, num_class):  # this is for multi class (named
SAMME — Stagewise Additive Modeling using a Multi-class Exponential
loss function)
    return np.log((1 - error) / error) + np.log(num_class-1)

def update_weights(w_i, alpha, y, y_pred):
    wi = w_i * np.exp(alpha * (np.not_equal(y, y_pred)).astype(int))
    weights = wi / wi.sum()  # Renormalize
    return weights

#%% AdaBoost
# Clear before calling
alphas = []
training_errors = []
w_i_update = []
y_pred_adaBoost = []
y_pred_vote_final = []
y_test_vote_final = []

# for having the probabilities of train and test to ensemble voting
y_pred_adaBoost_prob = []
y_test_vote_pred_final_prob =[]
classifier_acc = []
ovr_AUC_wsv = []

# Set weights for current boosting iteration
w_i = np.ones(len(df_train)) * 1 / len(df_train)
adaTime = np.zeros((NumberIterations))
best_epoch = np.zeros((NumberIterations))

# log for  final visualization
log = ['']*NumberIterations

# Iterate over NumberIterations weak classifiers
for i, m in enumerate(range(0, NumberIterations)): # not using m
    log[i] += f"Classifier #{i} " + save_name + "\n"
    # Create a classifier that uses the BERT model
    class SentimentClassifier(nn.Module):
      def __init__(self, n_classes, model_name):
```

```
        super(SentimentClassifier, self).__init__()
        self.bert = BertModel.from_pretrained(model_name, re-
turn_dict=False)
        # Freeze BERT parameters:
        if freeze_bert:
            # freeze all the parameters excluding classifier
            for param in self.bert.named_parameters():
                # param = (nome_da_layer, parametros)
                param[1].requires_grad = False
        self.drop = nn.Dropout(Dropout) # layer for regulation
        #The last_hidden_state is a sequence of hidden states of the
last layer of the model
        self.linear = nn.Linear(self.bert.config.hidden_size, n_clas-
ses)
      def forward(self, input_ids, attention_mask):
        _, pooled_output = self.bert(
          input_ids=input_ids,
          attention_mask=attention_mask)
        output = self.drop(pooled_output)
        output = self.linear(output)
        return output

    # Fit weak classifier and predict labels
    model = SentimentClassifier(len(class_names),model_name)
    model = model.to(device)

    data = next(iter(train_data_loader))
    input_ids = data['input_ids'].to(device)
    attention_mask = data['attention_mask'].to(device)

    # Train parameters optimizer:
    optimizer = torch.optim.Adagrad(model.parameters(), lr=l_r)

    # Scheduler function: # possivel retirar (colocar depois na função
de treino)
    # total_steps = len(train_data_loader) * epochs
    # warmup_steps = total_steps*0.1 # 10% de total_steps
    # scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=warmup_steps,
    # num_training_steps=total_steps)

    # Loss function:
    loss_fn = nn.CrossEntropyLoss(reduction='none').to(device)

    # Early Stopping:
    early_stopping = es_ada.EarlyStopping(verbose=True, patience=pa-
tience, optimizer=optimizer, path=path,
path_name=save_name+'Ada'+str(i), delta=delta)
```

```
    history = defaultdict(list)
    best_accuracy = 0
    t = time()
    print("\nTraining started...")

    for epoch in range(epochs):
        print(f'BERT {i}: Epoch {epoch + 1}/{epochs}')
        # Train function: #!!! colocar o sheduler a entrar no treino
        train_acc, train_loss, train_auc = uta.train_epoch(model,
train_data_loader, loss_fn, optimizer, device, len(df_train), w_i,
batch_size)
        print(f'Train:  loss {train_loss:.3f}   accuracy
{train_acc:.3f}')
        # Validation function:
        val_acc, val_loss, val_auc = uta.eval_model(model,
val_data_loader, loss_fn, device, len(df_val))
        print(f'Val:    loss {val_loss:.3f}   accuracy {val_acc:.3f}')
        # Save metrics histoty:
        history['train_acc'].append(train_acc.item())
        history['train_loss'].append(train_loss)
        history['train_auc'].append(train_auc.item())
        history['val_acc'].append(val_acc.item())
        history['val_loss'].append(val_loss)
        history['val_auc'].append(val_auc.item())

        if val_acc > best_accuracy:
            best_accuracy = val_acc
            print(f'\n Best Accuracy: {best_accuracy.item():.3f} ')

        # Early stopping
        early_stopping(val_auc.item(),val_acc, model)
        if early_stopping.early_stop:
           print("\n Stop at epoch:", epoch+1, "\n")
           break

    adaTime[i] = time()-t
    best_epoch[i] = early_stopping.best_epoch
    print(f"Training finished...(Elapsed time: {strftime('%H:%M:%S',
gmtime(adaTime[i]))})")

    # Plot training Losses:
    loss_hist = (history['train_loss'].copy(), his-
tory['val_loss'].copy())
    show_train_history(loss_hist, 'loss', early_stopping.best_epoch,
save_name)
    # Plot training Accuracy
    acc_hist = (history['train_acc'].copy(), history['val_acc'].copy())
    show_train_history(acc_hist, 'acc', early_stopping.best_epoch,
save_name)
```

```
    # Plot training AUC
    auc_hist = (history['train_auc'].copy(), history['val_auc'].copy())
    show_train_history(auc_hist, 'auc', early_stopping.best_epoch,
save_name)

    # Load best model parameters:
    load_path = path + 'checkpoint_' + save_name + 'Ada' + str(i)
+'.pth'
    checkpoint = torch.load(load_path)
    model.load_state_dict(checkpoint['model_state_dict'])

    classifier_acc.append(checkpoint['acc'].detach().cpu().item())

    # Save the classifier model
    if save_model:
        model_path = create_dir(path[:-1], f"Model_{save_name}")
        torch.save({'epoch': epoch,
               'model_state_dict': model.state_dict(),
               'acc': val_acc}, path_model + 'model' + save_name +
'Ada' + str(i) + '.pth')
        print("\nClassifier Ada "+ str(i) + " Model Save...")

    print("\nPredicting...")
    # Predictions for Boosting:
    y_review_texts_ada, y_pred_ada, y_pred_probs, y_train =
uta.get_predictions(model, train_data_loader)
    y_pred_adaBoost.append(y_pred_ada.numpy())
    y_pred_adaBoost_prob.append(y_pred_probs.numpy())

    log[i] += f" Training time: {strftime('%H:%M:%S',
gmtime(adaTime[i]))}\n"
    log[i] += f" Best epoch: {best_epoch[i]}\n"
    log[i] += f" Accuracy: {classifier_acc[i]}\n"

    print("\nComputing error...")
    # Compute error
    error_m = compute_error(y_train.numpy(), y_pred_ada.numpy(),
torch.from_numpy(w_i))
    training_errors.append(error_m)
    # Compute alpha
    alpha_m = compute_alpha(error_m, num_class)
    alphas.append(alpha_m)
    # Update weights
    w_i = update_weights(w_i, alpha_m.numpy(), y_train.numpy(),
y_pred_ada.numpy())
    w_i_update.append(w_i)
    print("Updating weights...\n")

    # Predictions for voting
```

```
    y_review_texts_vote, y_pred_vote, y_pred_probs_vote, y_test_vote =
uta.get_predictions(model, test_data_loader)
    y_pred_vote_final.append(y_pred_vote.numpy())
    y_test_vote_final.append(y_test_vote.numpy())
    y_test_vote_pred_final_prob.append(y_pred_probs_vote.numpy())

    # compute AUC_ROC of each classifier:
    auroc_wsv = AUROC(num_classes=num_class)
    ovr_AUC_wsv.append(auroc_wsv(y_pred_probs_vote,
y_test_vote).item())
    log[i] += f" AUC value: {ovr_AUC_wsv[i]}\n"

#%% weighted soft voting:
# normalize the accuracy values to obtain the weights per classifier
weights = [acc / sum(classifier_acc) for acc in classifier_acc]

# calculate the weighted average of the classifier predictions
ensemble_final_pred_wsv = np.average(y_test_vote_pred_final_prob,
axis=0, weights=weights)

# the final ensemble prediction is the class with the highest probabil-
ity
final_pred_wsv = np.argmax(ensemble_final_pred_wsv, axis=1)
# print(f'final predictions: {final_pred_wsv}')

#%% Final Predictions:
# Classification report:
classTest_wsv = classification_report(y_test_vote, final_pred_wsv, tar-
get_names=class_names)
# print(classTest_wsv) # visualização teste

cmTest_wsv = ConfusionMatrix(actual_vector= y_test_vote.tolist(), pre-
dict_vector = final_pred_wsv.tolist())
# print (f'\n === Confusion Matrix ===\n  \n {cmTest_wsv}')  # visuali-
zação teste

# Confusion Matrix:
cmTest_wsv.plot(cmap=plt.cm.Blues, number_label=True,
            normalized=True,    # Using normalized because dataset in
imbalanced
            plot_lib="seaborn",
            title="Confusion Matrix")
plt.ylabel('True Sentiment')
plt.xlabel('Predicted Sentiment')
plt.xticks(np.arange(3)+0.5, class_names)
plt.yticks(np.arange(3)+0.5, class_names)
plt.show()
```

```
#%% Store metrics:
# Names of metrics to store from PyCM
LabelsOverall = [k for k,v in cmTest_wsv.overall_stat.items() if isin-
stance(v,(float,int)) ]
LabelsClass   = [k for k,v in cmTest_wsv.class_stat.items() if isin-
stance(v[0],(float,int))]

# Extract metrics from cm object
rawOverall = selectMetrics(cmTest_wsv.overall_stat,LabelsOverall)
rawClass   = np.asarray([list(d.values()) for d in select-
Metrics(cmTest_wsv.class_stat,LabelsClass)])
ACC_id = 0 # index of overall ACC (checked)
FinalACC = rawOverall[ACC_id]

log[i] += f"\nFinal Accuracy: {FinalACC}\n"

# Convert into dictionary
mOverall = {k:v for k,v in zip( LabelsOverall, rawOverall )}
mClass = {k:{j:v for j,v in enumerate(varray)} for k,varray in zip( La-
belsClass, rawClass )}

log[i] += f"F1 Macro: {mOverall['F1 Macro']:.3f}\n"
log[i] += f"\nClassification report: \n{classTest_wsv}\n"

FinaltrainTime = np.mean(adaTime)
FinalOverall = {k:v for k,v in zip( LabelsOverall+['AUC'], rawOverall
)}
FinalClass = {k:{j:v for j,v in enumerate(varray)} for k,varray in zip(
LabelsClass, rawClass )}
FinalVariationACC = np.std(classifier_acc)
FinalClassifierACC = classifier_acc
# Print Log:
for f in range(NumberIterations):
    print(log[f],sep='\n')
    with open(f'{path_model}log_'+ save_name +'.txt','a') as log_file:
        log_file.write(log[f])

# print( f" Classification report TEST\n{classTest_wsv}\n")

# Save the model:
if save_metrics:
    save(save_name, path=path,
         trainBestEpoch = [early_stopping.best_epoch],
         trainTime = [FinaltrainTime],
         trainLOSS = loss_hist,
         trainACC = acc_hist,
         trainAUC = auc_hist,
         pycmClassifierACC = FinalClassifierACC,
         pycmSTDACC = FinalVariationACC,
```

```
        pycmOverall = mOverall,
        pycmClass = mClass,
        pycmCM = cmTest_wsv.to_array(normalized=True)) # confussion
matrix

#============================== END CODE ==============================
```

## B. Inference Scripts

### I. Script for SentAnalysisPt

```
# Import libraries:
print("\nLoading libraries...")
import os
import sys
import torch
import warnings
import numpy as np
import pandas as pd
from torch import nn
import torch.nn.functional as F
from datetime import datetime
from transformers import logging
from transformers import BertModel, BertTokenizer

# To ignore deprecated warnings:
warnings.filterwarnings("ignore")
# BERT warning --> this warning means that during your training, you're
not using the pooler in order to compute the loss
logging.set_verbosity_error()

# set CUDA
device = torch.device("cpu")

# Class names:
class_names = ['Negative', 'Neutral', 'Positive']

# Load name:
load_name = 'SentAnalysisPt' # change the name to avoid overwrite
# Set Path:
path = './BERT/SentimentAnalysisClassPt/'
# set path for load the model:
path_model = path + 'Model_' + load_name + '/'

# Save to .txt file:
save_txt = True # important if store in .txt file

# Classifier using BERT pre-trained model:
class SentimentClassifier(nn.Module):
    def __init__(self, n_classes, model_name='neuralmind/bert-base-por-
tuguese-cased'):
        super(SentimentClassifier, self).__init__()
        self.bert = BertModel.from_pretrained(model_name, re-
turn_dict=False)
        self.drop = nn.Dropout(0.2) # layer for regulation
```

```
        #The last_hidden_state is a sequence of hidden states of the
last layer of the model
        self.linear = nn.Linear(self.bert.config.hidden_size, n_clas-
ses)

    def forward(self, input_ids, attention_mask):
        _, pooled_output = self.bert(
          input_ids=input_ids,
          attention_mask=attention_mask)
        output = self.drop(pooled_output)
        output = self.linear(output)
        return output

# Loading the model and model tokenizer:
model = SentimentClassifier(3, model_name='neuralmind/bert-base-portu-
guese-cased') # Load the model
tokenizer = BertTokenizer.from_pretrained('neuralmind/bert-base-portu-
guese-cased') # Load Tokenizer

# Load path of model:
path_temp = path_model + 'model' + load_name + '.pth'
# Load the model stored on Model folder
load_model = torch.load(path_temp, map_location=torch.device(device))
model.load_state_dict(load_model['model_state_dict']) # Load parameters

# model to GPU
model = model.to(device)
model = model.eval()

#%% load text for classification:
print("\nLoading Text...")
text = [# Negative with 65 words
        "Acabei agora de sair deste estabelecimento, depois de esperar
mais de uma hora em vão. Nesta hora, inúmeros "\
        "clientes que chegaram depois de me sentar foram servidos an-
tes. Ao manifestar a insatisfação junto do dono do "\
        "estabelecimento, este foi mal educado e mentiroso. De referir
que o resto do staff foi impecável, mas este tipo "\
        "de tratamento por parte do dono do estabelecimento é inaceitá-
vel.",

        # Neutral with 63 words
        "Espaço amplo e bem decorado que serve soluções rápidas para um
almoço que não deve demorar. A salada de salmão muito "\
        "mal servida, mais alface que outra coisa, o que me deixou um
pouco desconsolada. O que salvou foi a sopa e o sumo que "\
        "estavam bons. Fico reticente se voltarei ou não, talvez a es-
colha possa não ter sido a mais acertada.",
```

```
        # Positive with 68 words
        "Muito bom! Só tenho comido o vegetariano, de feijão preto, com
picante, e gosto muito! "\
        "Costumo ir buscar para comer em casa e só de uma vez veio sem
queijo. Não foi nada de gravíssimo, "\
        "mas referimos isso. Pediram desculpa e ofereceram um
hambúrguer na próxima compra. Gostei "\
        "muito da atitude e já voltei várias vezes depois disso. E vol-
tarei mais! No Top 3 de hambúrguerias no Porto."
        ]

text50 = [# Negative 50 tokens
    "Já fui a outros Fogo de Chão muito mas muito melhores em Lisboa
este aqui tem apresentações horríveis e comida "\
    "parece sei lá o q.... a comida parecia ser comida quase da cantina
da escola omg !!!",

    # Neutral 50 tokens
    "Ideal para uma refeição mais rápida, quando se tem menos tempo
para almoçar/jantar. Bom quando já não falta "\
    "muito tempo para começar a sessão de cinema, mas apetece petiscar
algo antes de entrar ;)",

    # Positive 50 tokens
    "A comida é muito boa, mexilhão à espanhola muito bom, filetes fri-
tos com bom óleo bem gostosos, as lulas à "\
    "correio a meu ver estavam um pouco secas e salgadas. Preço muito
convidativo."
    ]

text100 = [# Negative 100 tokens
    "Sítio engraçado mas infelizmente recusarem-se a servir cafés e be-
bidas quase às 15h, com o argumento de que é hora de "\
    "almoço, e com uma série de mesas vazias, é um pouco triste e até
desrespeitador. Sendo o único sítio disponível foi "\
    "muito chato, mas felizmente existem outros locais não muito longe,
onde valeu bastante a pena ir e que não tínhamos "\
    "conhecido não fosse a tristeza e a má vontade.",

    # Neutral 100 tokens
    "Espaço extremamente agradável, o aspecto rústico com os presuntos
expostos e malagueta, etc... Na minha opinião tornam"\
    " o espaço muito caloroso. Funcionários muito simpáticos e prestá-
veis. Em relação às iguarias servidas, tem bom aspecto "\
    "mas a nível de sabor deixam algo a desejar. Tendo em conta a quan-
tidade de sítios bons para se comer uma refeição "\
    "italiana. Tiramisu divinal apesar do aspecto algo descabido na
chegada a mesa.",
```

```
    # Positive 100 tokens
    "Fui levado de surpresa a um jantar a este restaurante e não podia
ter gostado mais!  Começámos com umas empadinhas "\
    "de perdiz e seguimos para as lascas de bacalhau e para a tagine de
cordeiro, tudo muito bem servido e muito saboroso."\
    " Para terminar pedimos um cheesecake e um bolo de chocolate, dos
melhores que já comi!  Este é sem dúvida um "\
    "restaurante que vou passar a recomendar!"
    ]

#%% Classify the reviews::
time_count = []
TotalInferenceTime = datetime.now() - datetime.now()
# for test in df['text']: #!!! ATENTION HERE
print("\nClassify Text...")
sample = 'Text'
for test in text:
    # Count the enlapsed time:
    t = datetime.now()
    encoded_review = tokenizer(test, max_length=100,add_special_to-
kens=True,return_token_type_ids=False,truncation=True, pad-
ding='max_length',return_attention_mask=True,return_tensors='pt')
    input_ids = encoded_review['input_ids']
    attention_mask = encoded_review['attention_mask']
    output = model(input_ids, attention_mask)
    _, prediction = torch.max(output, dim=1)
    probs = F.softmax(output, dim=1).detach().numpy()[0]

    InferenceTime = datetime.now()-t
    time_count.append(InferenceTime)
    TotalInferenceTime += InferenceTime

    # print in Console
    print("\n============================================================")
    print(f'Portuguese review: \n {test}')
    print(f'\nInferenceTime: {InferenceTime.seconds}.{round(Infer-
enceTime.microseconds/1000)} seconds')
    print(f'\nForecast:\n Sentiment {class_names[prediction]} with
{np.max(probs)*100:.4}% weight\n')
    print(pd.DataFrame(probs, class_names, columns=['Probabilities']))
    print("============================================================\n")

    # Print on .txt file
    if save_txt:
        with open(path_model + 'Export_'+ sample +'_file.txt', 'a') as
f:
            f.write('Inference Results: ')
            # this allow to save in a .txt file
```

```
            print("\n================================================",
file=f)
            print(f'Portuguese review: \n {test}', file=f)
            print(f'\nInferenceTime: {InferenceTime.seconds}.{round(In-
ferenceTime.microseconds/1000)} seconds', file=f)
            print(f'\nForecast:\n Sentiment {class_names[prediction]}
with {np.max(probs)*100:.4}% weight\n', file=f)
            print(pd.DataFrame(probs, class_names, columns=['Probabili-
ties']), file=f)
            print("================================================\n",
file=f)
# print(f"\nTotal Inference time: {TotalInferenceTime.seconds}s ({To-
talInferenceTime.microseconds} microseconds)")
print(f"\nTotal Inference time: {TotalInferenceTime.seconds}.{round(To-
talInferenceTime.microseconds/1000)} seconds")
if save_txt:
    with open(path_model + 'Export_'+ sample +'_file.txt', 'a') as
f:
        f.write(f"\nTotal Inference time: {TotalInferenceTime.sec-
onds}.{round(TotalInferenceTime.microseconds/1000)} seconds")

#============================== END CODE ==============================
```

## II. Script for SentAnalyPtAdaboost (Delivered to Zomato Team)

```
# Import libraries
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '1' # hide INFO messages
import torch
import numpy as np
import pandas as pd
from torch import nn
import torch.nn.functional as F
from datetime import datetime

# set CUDA
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Constatns
ACC = 0.84038
classifier = [0.8396616616350361, 0.5879427059089518]
classes = ['Negative', 'Neutral', 'Positive']

# Ensemble
def ensemble_prediction(accuracy, probability):
    """
    Calculates the ensemble prediction based on the weighted average of
classifier predictions.
    Args:
        accuracy (list): List of accuracy values for each classifier.
        probability (list): List of probability arrays containing the
predictions from each classifier.
    Returns:
        int: The final ensemble prediction.
        ndarray: The ensemble prediction probabilities.
    """
    # Normalize the accuracy values to obtain the weights per classi-
fier
    weights = [acc / sum(accuracy) for acc in accuracy]
    # Calculate the weighted average of the classifier predictions
    ensemble_pred = np.average(probability, axis=0, weights=weights)
    # The final ensemble prediction is the class with the highest prob-
ability
    prediction = np.argmax(ensemble_pred, axis=0)
    return prediction, ensemble_pred

# Load classifier weights
def load_model(path, i, model, device):
    """
    Loads the weights of a specific classifier.
    Args:
```

```
        path (str): The path to the folder containing the classifier
weights.
        i (int): The index of the classifier.
        model (nn.Module): The model object to load the weights into.
        device (torch.device): The device to use for loading the
weights.
    Returns:
        nn.Module: The model object with the loaded weights.
    """
    # Load path of model:
    path_temp = path + 'modelboost' + str(i) + '.pth'
    # Load the model stored on Model folder
    load_model = torch.load(path_temp, map_location=torch.device(de-
vice))
    model.load_state_dict(load_model['model_state_dict'])  # Load pa-
rameters
    model = model.eval()
    return model

# Encoder
def encode_review(model, tokenizer, text):
    """
    Encodes a review using the specified model and tokenizer.
    Args:
        model: The sentiment classification model.
        tokenizer: The tokenizer used for encoding the review.
        text (str): The review text to encode.
    Returns:
        int: The predicted sentiment class.
        ndarray: The predicted sentiment probabilities.
    """
    encoded_review = tokenizer(text, max_length=100, add_special_to-
kens=True,
                               return_token_type_ids=False, trunca-
tion=True,
                               padding='max_length', return_atten-
tion_mask=True,
                               return_tensors='pt')
    input_ids = encoded_review['input_ids']
    attention_mask = encoded_review['attention_mask']
    output = model(input_ids, attention_mask)
    _, prediction = torch.max(output, dim=1)
    probs = F.softmax(output, dim=1).detach().numpy()[0]
    return prediction.item(), probs

# Main Classifier
class SentimentClassifier(nn.Module):
    def __init__(self, n_classes, base_model):
        """
```

```
        Sentiment classifier model based on a pre-trained base model.
        Args:
            n_classes (int): The number of sentiment classes.
            base_model: The pre-trained base model.
        Attributes:
            bert: The base model.
            drop: Dropout layer for regularization.
            linear: Linear layer for classification.
        """
        super(SentimentClassifier, self).__init__()
        self.bert = base_model
        self.drop = nn.Dropout(0.2) # layer for regulation
        self.linear = nn.Linear(self.bert.config.hidden_size, n_clas-
ses)

    def forward(self, input_ids, attention_mask):
        """
        Forward pass of the classifier model.
        Args:
            input_ids: The input IDs of the encoded review.
            attention_mask: The attention mask of the encoded review.
        Returns:
            torch.Tensor: The output tensor.
        """
        _, pooled_output = self.bert(
          input_ids=input_ids,
          attention_mask=attention_mask)
        output = self.drop(pooled_output)
        output = self.linear(output)
        return output

# Sentiment classification
def classify_sentiment(reviews, model, tokenizer):
    """
    Perform sentiment classification on a list of reviews using the
given model and tokenizer.
    Args:
        reviews (list): List of reviews to classify.
        model: The sentiment classification model.
        tokenizer: The tokenizer used for encoding the reviews.
    Returns:
        final_predictions (list): Final prediction for each review
        final_probabilities (list of numpy array): Confidence of the
model
    """
    final_predictions = []
    final_probabilities = []
    time_count = []
    TotalInferenceTime = datetime.now() - datetime.now()
```

```
    for review in reviews:
        predictions = []
        probabilities = []
                # Count the enlapsed time:
        t = datetime.now()

        for i in range(len(classifier)):
            load_model('Send_to_Zomato/', i, model, device)
            prediction, probs = encode_review(model, tokenizer, review)
            predictions.append(prediction) # Predictions value
            probabilities.append(probs) # predictions probabilities

        final_pred, ensemble_pred = ensemble_prediction(classifier,
probabilities)
        InferenceTime = datetime.now()-t
        time_count.append(InferenceTime)
        TotalInferenceTime += InferenceTime
        final_predictions.append(final_pred)
        final_probabilities.append(np.max(ensemble_pred))
        print_results(review, final_pred, ensemble_pred, InferenceTime)
    return final_predictions, final_probabilities

# Print in Console
def print_results(reviews, predictions, probabilities, InferenceTime):
    """
    Prints the results of sentiment classification.
    """
    print("\n============================================================")
    print(f'Portuguese review:\n{reviews}')
    print(f'\nForecast:\nSentiment: {classes[predictions]} with
{np.max(probabilities)*100:.4}% weight\n')
    print(pd.DataFrame(probabilities, classes, columns=['Probabili-
ties']))
    print(f'\nInferenceTime: {InferenceTime.seconds}.{round(Infer-
enceTime.microseconds/1000)} seconds')
    print("============================================================\n")

#%% Classify the reviews:
if __name__ == '__main__':

    # Initialize the sentiment analyzer
    bert = torch.load('Send_to_Zomato/bert_model.pth')
    tokenizer = torch.load('Send_to_Zomato/bert_tokenizer.pth') # Load
Tokenizer
    model = SentimentClassifier(3, bert) # Load the model

    # Classify the reviews (Example)
    reviews = [# Negative
```

```
            "Acabei agora de sair deste estabelecimento, depois de es-
perar mais de uma hora em vão. Nesta hora, inúmeros "\
            "clientes que chegaram depois de me sentar foram servidos
antes. Ao manifestar a insatisfação junto do dono do "\
            "estabelecimento, este foi mal educado e mentiroso. De re-
ferir que o resto do staff foi impecável, mas este tipo "\
            "de tratamento por parte do dono do estabelecimento é ina-
ceitável.",

            # Neutral
            "Espaço amplo e bem decorado que serve soluções rápidas
para um almoço que não deve demorar. A salada de salmão muito "\
            "mal servida, mais alface que outra coisa, o que me deixou
um pouco desconsolada. O que salvou foi a sopa e o sumo que "\
            "estavam bons. Fico reticente se voltarei ou não, talvez a
escolha possa não ter sido a mais acertada.",

            # Positive
            "Muito bom! Só tenho comido o vegetariano, de feijão preto,
com picante, e gosto muito! "\
            "Costumo ir buscar para comer em casa e só de uma vez veio
sem queijo. Não foi nada de gravíssimo, "\
            "mas referimos isso. Pediram desculpa e ofereceram um
hambúrguer na próxima compra. Gostei "\
            "muito da atitude e já voltei várias vezes depois disso. E
voltarei mais! No Top 3 de hambúrguerias no Porto."
            ]
    # Classify sentiment
    prediction, confidence = classify_sentiment(reviews, model, to-
kenizer)

#============================== END CODE ==============================
```

## III. Script to manage all metrics

```
# Import libraries
import pickle
import pandas as pd
import numpy as np
from time import strftime, gmtime
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import plotly.io as pio
import seaborn as sns
pio.renderers.default='browser'
plt.rcParams['font.family'] = 'DeJavu Serif'
plt.rcParams['font.serif'] = ['Times New Roman']

#%% Functions
def show_train_history(data: tuple[list], metric: str,
                       best_epoch: int=0, model_name: str='Unknown'):
    """
    Plots the metrics monitored during training.
    Parameters:
        data (tuple of lists): must be a tuple of size=2, where
         position 0 correspond to the metric evaluated on train set,
         and position 1 correspond to the metric evaluated on
         validation set.
        metric (str): the name of the metric contained in data.
         Used for plotting.
        be (int): best epoch of the training of the model. Correspond
         to the best model returned after training.
        model_name (str): name of the model being tested. Used
        for plotting.
    """
    train_metric = 0
    validation_metric = 1
    x_axis=np.arange(len(data[0]))+1

    plt.figure(figsize=(10,6))
    plt.plot(x_axis, data[train_metric])
    plt.plot(x_axis, data[validation_metric])
    if best_epoch != 0:
        plt.axvline(x=best_epoch,color='r',ls='--')
    plt.title(f'Train History of {model_name}',fontsize=20)
    plt.ylabel(metric.upper(), fontsize=18)
    plt.xlabel('Epoch', fontsize=20)
    plt.legend(['train', 'validation'], loc='upper left', fontsize=14)
    plt.grid(True)
    plt.minorticks_on()  # Add minor ticks
    plt.grid(which='minor', linestyle='--', linewidth='0.25',
color='gray')  # Set style for minor grid lines
```

```
    # Manually set x-axis ticks
    x_ticks = np.arange(0, len(data[0]) + 1, step=5)  # Adjust the step
as needed
    plt.xticks(x_ticks, labels=x_ticks, fontsize=14)
    plt.yticks(fontsize=14)
    plt.show()


def getMetrics(path, model_name):
    """
    Load to memory all metrics stored by saveModels.py
    Parameters:
        path (str): directory where the model data is stored.
        model_name (str): custom model's name to identify
         it inside directory path.
    Return:
        Returns tuple with all extracted information,
        position-wise:
            ( 0) Training accuracy
            ( 1) Training AUC
            ( 2) Training Loss
            ( 3) Best epoch
            ( 4) Time to train
            ( 5) Standard deviation
            (-3) Confusion Matrix
            (-2) Overall metrics (pycm + sklearn)
            (-1) Class metrics (pycm + sklearn)
    """
    # if path[:2] != './': path = './'+path
    if path[-1] == '/': path = path[:-1]
    full_path = path + f"/Model_{model_name}/"
    # Load training evaluation
    path_temp = full_path + 'Training/'
    try:
        with open(path_temp + "ACC.pkl",'rb') as f:
            acc = pickle.load(f)
            acc_val = pickle.load(f)
    except:
        acc = None
        acc_val = None
    try:
        with open(path_temp + "AUC.pkl",'rb') as f:
            auc = pickle.load(f)
            auc_val = pickle.load(f)
    except:
        auc = None
        auc_val = None
    try:
        with open(path_temp + "LOSS.pkl",'rb') as f:
```

```
            loss = pickle.load(f)
            loss_val = pickle.load(f)
    except:
        loss = None
        loss_val = None
    try:
        with open(path_temp + "BestEpoch.pkl",'rb') as f:
            bestEpoch = pickle.load(f)
    except:
        bestEpoch = 0
    try:
        with open(path_temp + "Time.pkl",'rb') as f:
            trainTime = pickle.load(f)
    except:
        trainTime = 0


    # Load PYCM metrics
    path_temp = full_path + 'ConfusionMatrix/'

    with open(path_temp + "Overall.pkl",'rb') as f:
        Overall = pickle.load(f)
    with open(path_temp + "Class.pkl",'rb') as f:
        Class = pickle.load(f)
    with open(path_temp + "CM.pkl",'rb') as f:
        CM = pickle.load(f)

    try:
        with open(path_temp + "FinalAUC.pkl",'rb') as f:
            finalAuc = pickle.load(f)
    except:
        pass
    try:
        with open(path_temp + "AUC.pkl",'rb') as f:
            finalAuc = np.mean(pickle.load(f))
    except:
        pass
    try:
        finalAuc
    except:
        finalAuc = 0
    try:
        variation = {}
        try:
            with open(path_temp + "STDAUC.pkl",'rb') as f:
                variation['AUC'] = pickle.load(f)
        except: pass
            # variation_acc = None
        try:
```

```
            with open(path_temp + "STDACC.pkl",'rb') as f:
                variation['ACC'] = pickle.load(f)
        except: pass
            # variation_auc = None
    except: variation = {}
    return [acc,acc_val], [auc,auc_val], [loss,loss_val], bestEpoch,
trainTime, variation, finalAuc, \
            CM, Overall, Class


def selectMetrics(metrics: dict, select: list[str]):
    return [metrics[v] for v in select]


def renameMetrics(select: list[str]):
    rename = {"AUC": "AUC (ROC)",
              "Overall ACC": "Accuracy",
              "ACC": "Accuracy",
              "TPR": "Sensitivity",
              "TPR Macro": "Avg. Sensitivity",
              "TNR": "Specificity",
              "TNR Macro": "Avg. Specificity",
              "PPV": "Precision",
              "PPV Macro": "Avg. Precision",
              "F1": "F1-score",
              "F1 Macro":  "Avg. F1-score"
        }
    return [rename.get(v) or v for v in select]


def renameModels(names: list[str]) -> list:
    renamed_models = []
    for n in names:
        if n[-4:] == 'fold':
            renamed_models.append(n[:-6])
        else:
            title = "Single train"
            renamed_models = names.copy()
            break
    else:
        title = f"Cross-Validation with {n[-5:]}"
    return renamed_models, title


def plotHistory(metrics: list, model_name: str, plot: bool=False):
    acc, auc, loss, bestEpoch = metrics
    if plot:
        show_train_history(acc,'acc', bestEpoch, model_name)
        show_train_history(auc,'auc', bestEpoch, model_name)
```

```
        show_train_history(loss,'loss', bestEpoch, model_name)


def plotTrain(path: str, metrics: list, model_name: str, be: np.array,
plot: bool=False):
    """metrics: for compatibility to plotHistory()"""
    def _show_train_history(data, metric, ax, fn, be):
        """
        ax: object where to plot
        fn: fold number
        """
        train_metric = 0
        validation_metric = 1
        x_axis=np.arange(len(data[train_metric]))+1

        ax.plot(x_axis, data[train_metric], 'b',
                alpha=0.5, linewidth=2.5)
        ax.plot(x_axis, data[validation_metric], 'r',
                alpha=0.8, linewidth=2.5)
        if be != 0: ax.axvline(x=be,color='g',ls='--',
                               linewidth=1.8)
        ax.set_title(f'Train History (fold {fn})')
        ax.set_ylabel(metric.upper())
        ax.set_xlabel('Epoch')
        ax.legend(['train', 'validation', 'best epoch'], loc='center
right')
        ax.grid(True)

    if plot:
        try:
            path_name = f'{path}/Model_{model_name}/Training/Train-
ingCurves'
            with open(path_name, 'rb') as f:
                curves = pickle.load(f) # curves is a dict
        except FileNotFoundError:
            plotHistory(metrics, model_name, plot)
            return None

        N = len(list(curves.values())[0])
        if N == 5: # (5 folds)
            fig, axs = plt.subplots(3,5, figsize=(24,11),sharex=True,
dpi=300)
            fig.suptitle(model_name[:-6])
            for ax,(k,v) in zip(axs, curves.items()):
                # if k in ['acc', 'ACC']: continue # Don't show accu-
racy
                _show_train_history(v[0], k, ax=ax[0], fn=1, be=be if
isinstance(be,int) else be[0])
```

```
                _show_train_history(v[1], k, ax=ax[1], fn=2, be=be if
isinstance(be,int) else be[1])
                _show_train_history(v[2], k, ax=ax[2], fn=3, be=be if
isinstance(be,int) else be[2])
                _show_train_history(v[3], k, ax=ax[3], fn=4, be=be if
isinstance(be,int) else be[3])
                _show_train_history(v[4], k, ax=ax[4], fn=5, be=be if
isinstance(be,int) else be[4])
        elif N == 2: # (2 folds)
            fig, axs = plt.subplots(3,2, figsize=(16,11),sharex=True,
dpi=300)
            fig.suptitle(model_name[:-6])
            for ax,(k,v) in zip(axs, curves.items()):
                # if k in ['acc', 'ACC']: continue # Don't show accu-
racy
                _show_train_history(v[0], k, ax=ax[0], fn=1, be=be if
isinstance(be,int) else be[0])
                _show_train_history(v[1], k, ax=ax[1], fn=2, be=be if
isinstance(be,int) else be[1])
        elif N == 1: # (single fold)
            fig, axs = plt.subplots(3,1, figsize=(6,10),sharex=True,
dpi=300)
            fig.suptitle(model_name)
            for ax,(k,v) in zip(axs, curves.items()):
                # if k in ['acc', 'ACC']: continue # Don't show accu-
racy
                _show_train_history(v[0], k, ax=ax, fn=1, be=be if
isinstance(be,int) else be[0])
        fig.tight_layout()
        plt.show()
        return curves
    else: return None


def plotOverallMetric(metrics: np.ndarray, Models: list, labels:
list[str], plot=False):
    Models, title = Models
    # Waterfall plot
    df = pd.DataFrame(metrics, columns=labels)
    df_model = pd.DataFrame(Models,columns=['Model'])
    df = pd.concat([df_model,df],axis=1)
    def relative_to_prev(arr):
        metric = arr.copy()
        for i in range(len(metric)):
            base = arr[i-1]
            metric[i] = arr[i] if i==0 else arr[i]-base
        return metric
    if plot:
        fig = go.Figure()
```

```
        M = len(Models)
        L = len(labels)

        formated_labels = np.asarray([[l]*M for l in labels]).reshape(-
1).tolist()
        formated_metrics = np.zeros(shape=metrics.T.shape)
        for i,m in enumerate(metrics.T):
            formated_metrics[i] = relative_to_prev(m)
        formated_metrics = formated_metrics.reshape(-1) # flatten

        fig.add_trace(go.Waterfall(
            y = [formated_labels, L*Models],
            x = formated_metrics,
            measure = L*(['absolute']+(M-1)*['relative']),
            orientation = "h",
            decreasing = {"marker":{"color":"red"}},
            increasing = {"marker":{"color":"forestgreen"}},
            totals = {"marker":{"color":"cornflowerblue"}},
        ))

        fig.update_layout(
            title=title,
            waterfallgroupgap = 0.03,
            waterfallgap = 0.03,
            font_family="Times New Roman",
            xaxis = dict(tickmode = 'linear',
                         tick0 = 0.70,
                         dtick = 0.02
        ))

        fig.update_xaxes(title_font_family="Times New Roman",
                         range=[0.70,0.95],
                         showgrid=True, gridwidth=2, gridcolor='Blue',
                         minor_griddash="dot",
                         )
        fig.update_yaxes(title_font_family="Times New Roman",
                         showline=True, linewidth=2, linecolor='black')
        fig.show()
    return df


def plotCM(conf_mat: np.ndarray,
           conf_mat_abs: np.ndarray,
           model_name: str, which: bool=True, plot: bool=False):
    conf_mat = conf_mat if which else conf_mat_abs
    if plot:
        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        ax.matshow(conf_mat, cmap=plt.cm.Blues, alpha=0.75)
        for i in range(conf_mat.shape[0]):
```

```
            for j in range(conf_mat.shape[1]):
                ax.text(x=j, y=i,s=conf_mat[i, j].round(5),
                        va='center', ha='center', size='xx-large')
        plt.xlabel('Predicted Sentiment', fontsize=20)
        plt.ylabel('True Sentiment', fontsize=20)
        plt.xticks(ticks=[0,1,2],
                   labels=['Negative','Neutral','Positive'],
                   fontsize=18)
        plt.yticks(ticks=[0,1,2],
                   labels=['Negative','Neutral','Positive'],
                   rotation=90, fontsize=18,va='center')
        plt.title(f'Confusion Matrix of {model_name}\n', fontsize=20)
        plt.show()


def plotClassMetric(data: np.ndarray, Models: list,
                    metric: str, plot: bool=False):
    Models, title = Models # this title thing is to handle the fold ID
    N = len(Models)
    if plot:
        values = list(np.asarray(data).T.flat)
        to_plot = pd.DataFrame({'model_name':3*Models, # this 3 is for
each sentiment
                                'Sentiment':N*['Negative'] +
                                            N*['Neutral'] +
                                            N*['Positive'],
                                metric:values,
                                })

        fig, ax = plt.subplots(figsize=(12, 8),dpi=300)

        sns.barplot(data=to_plot, x='model_name', y=metric,
                    hue='Sentiment', ax=ax, edgecolor="white")
        shifts = {'Negative':[-0.27,'#0099ff'],
                  'Neutral':[0.0,'#ffad33'],
                  'Positive':[0.27,'#00ff00'],
                  }
        rotations = [0, 0, 0, 0, 25, 25, 30]
        for sent in ['Negative','Neutral','Positive']:
            s,c = shifts[sent]
            ax.plot([i+s for i in range(N)], # fix for different that 3
models
                    to_plot.query("Sentiment == @sent")[metric],
                    lw=1.5, marker='o',linestyle='dashed', color=c)
        ax.set_title(f'Comparison of metric: {metric}', fontsize=18)
        ax.set_ylabel(metric, fontsize=18)
        ax.set_xlabel(' ')
        ax.set_ylim(min(values)*0.95, 1.00)
        plt.xticks(rotation=rotations[N-1], fontsize=16)
```

```
        # plt.xticks(rotation=30, fontsize=16)
        ax.set_axisbelow(True)
        plt.grid(which='major', axis='y', alpha=0.5,
                 color='gray', linestyle='solid')
        plt.grid(which='minor', axis='y', alpha=1.0,
                 color='lightgray', linestyle='dashed')
        plt.minorticks_on()
        ax.grid(True, axis='y')
        # plt.setp(ax.lines[:-3], linewidth=2, color='white')
        # Set the legend size using the prop argument
        ax.legend(loc='upper right', fontsize=12, prop={'size': 12})
    return pd.DataFrame(data,columns=['Negative','Neutral','Positive'],
                        index=Models)


def main(Models: list[str], metrics_path: str,                          #
Set models
         plot_Hist: bool=False,                                         #
Plot learning curves
         plot_CM: bool=False, whichCM: bool=True,                       #
Plot confusion matrix
         plot_Waterfall: bool=False, plot_Waterfallw: bool=False,       #
Plot Waterfall of average metrics
         plot_Classes: bool=False,                                      #
Plot Barplots of class metrics
         selectOverall: list[str]=['AUC','Overall ACC','F1 Macro','TPR
Macro','TNR Macro','PPV Macro'],
         selectClass: list[str]=['AUC','ACC','F1','TPR','TNR','PPV']):
    info = ''
    metrics = {}
    plt_Overall = np.zeros((len(Models),len(selectOverall)))
    plt_wOverall = np.zeros((len(Models),len(selectOverall)))
    plt_Class   = np.zeros((len(Models),len(selectClass),3))
    std_Overall = []
    confMat = {}
    for i, model in enumerate(Models):
        info += f"\n[Model {model}]\n"
        # Plot training behaviour:
##########################################################################
        current = metrics[model] = getMetrics(metrics_path, model)
        plotTrain(metrics_path, current[:4], model, current[3],
plot_Hist)
        info += f"\tTraining time: {strftime('%H:%M:%S', gmtime(cur-
rent[4]))}\n"

        # Load variables to plot Overall and Class metrics: (plots are
later, but I need the variables here)
        _overall = current[-2]
        _class = current[-1]
```

```
        _overall['AUC'] = current[6]

        # Define the variable with the standard deviations:
########################################################################
        _std = current[5] # (dictionary)
        keys = {'Overall ACC':'ACC','AUC':'AUC'}
        for k,v in keys.items():
            m = _overall[k]
            s = _std.get(v) or 0.0
            info += f"\tAverage {v:>3}: {m:.4f} ± {s:.4f}\n"
        std_Overall.append(list(_std.values()))

        # Define variable to plot Confusion Matrix:
########################################################################
        _CM = current[-3]
        support = _class['P'] # Support, for weigthed average
        _CMabs = np.zeros(shape=(3,3),dtype='int')
        for j in range(3):
            actual = _CM[j]
            w = support[j]
            _CMabs[j] = np.asarray(actual*w).round(0).astype('int')
        confMat[model] = _CMabs
        plotCM(_CM, _CMabs, model, whichCM, plot_CM)

        # Define variable to plot the overall metrics
########################################################################
        current_Overall = selectMetrics(_overall,selectOverall) # dict
to array
        plt_Overall[i] = np.asarray(current_Overall) # array into ma-
trix

        # Define variable to plot the Class metrics and Weighted Over-
all metrics: ############################################################
        current_Class = [list(d.values()) for d in select-
Metrics(_class,selectClass)] # dict to array
        plt_Class[i] = np.asarray(current_Class) # array into matrix
        # try:
        weights = list(support.values()) # Support, for weigthed aver-
age
        current_wOverall = np.average(cur-
rent_Class,axis=1,weights=weights) # average
        plt_wOverall[i] = np.asarray(current_wOverall) # array into ma-
trix
        # except Exception as e:
        #     print("current_Class = ",current_Class,sep='\n',end='\n')
        #     print("support = ",support,sep='\n',end='\n')
        #     raise e

    # Plot the overall metrics
```

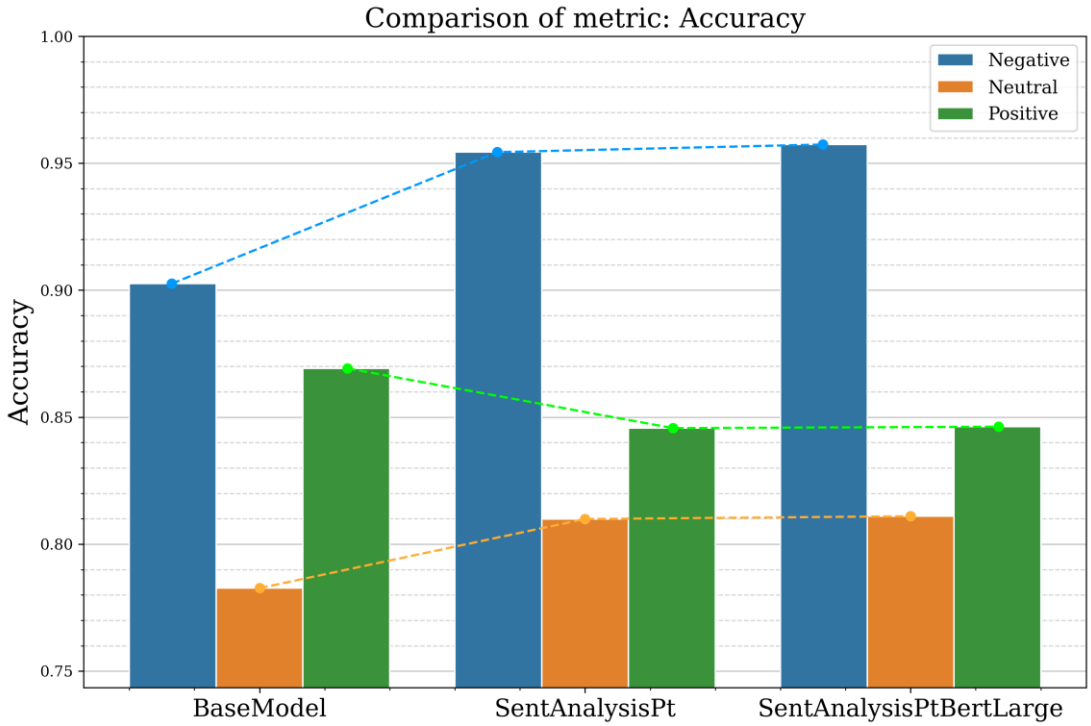
```
    Overall_table = plotOverallMetric(plt_Overall, renameModels(Mod-
els), renameMetrics(selectOverall), plot=plot_Waterfall) # Plot for
thesis
    wOverall_table = plotOverallMetric(plt_wOverall, renameModels(Mod-
els), renameMetrics(selectOverall), plot=plot_Waterfallw) # Plot for
thesis

    # Plot metrics per class
########################################################################
    Class_tables = {}
    for m in range(len(selectClass)):
        Class_tables[selectClass[m]] = plotClass-
Metric(plt_Class[:,m,:],
                                                        renameMod-
els(Models),
                                                        renameMet-
rics([selectClass[m]])[0],
                                                        plot=plot_Clas-
ses)

    # Modify the Overall_table to include the standard deviation, if
any: ###################################################################
    try:
        j=0
        std_Overall = np.asarray(std_Overall)
        for i in range(2,11):
            c = i-1
            if i%2 != 0:
                continue
            Overall_table.insert(i, Overall_table.columns[c] + ' (σ)',
                                 std_Overall[:,j], allow_dupli-
cates=True)
            wOverall_table.insert(i, wOverall_table.columns[c] + '
(σ)',
                                  std_Overall[:,j], allow_dupli-
cates=True)
            j+=1
    except:
        print("STD exception")
        pass
    # Print all information gathered to the console
########################################################################
    print(info)
########################################################################
    return metrics, Overall_table, wOverall_table, Class_tables,
confMat

#%%
if __name__ == '__main__':
```

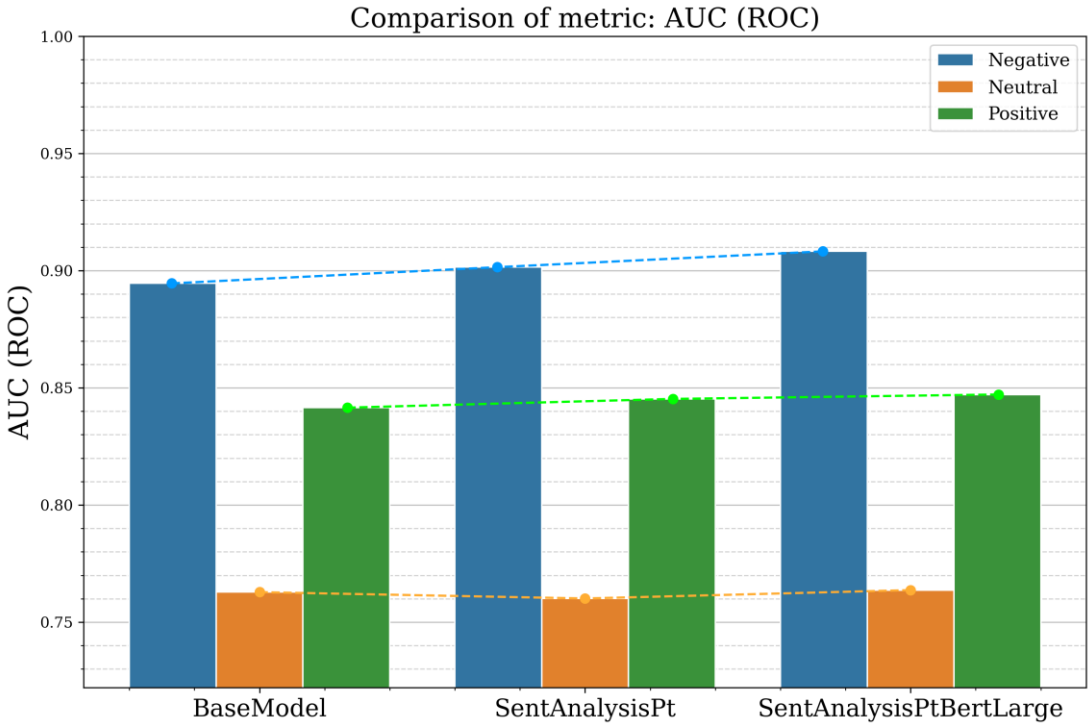
```
    # Define control variables:
#=========================================================================
    metrics_path = [
                    # './BERT/SentimentAnalysisClassPt',
                    # './BERT/SentimentAnalysisClassPtCrossValidation',
                    # './BERT/SentimentAnalysisClassPtAdaBoost',
                    #
'./BERT/SentimentAnalysisClassPtAdaBoostCrossValidation',
                    'NewModels'
                    ]
    Models = (
    [
# ====================== Models using BERT ============================
        # 'BaseModel',
        'SentAnalysisPt',
        # 'SentAnalysisPtBertLarge',

# ====================== BERT with Boost ==============================
        'SentAnalyPtAdaBoost_2',
        # 'SentAnalyPtAdaBoost_3',
        # 'SentAnalyPtAdaBoost_4',
        # 'SentAnalyPtAdaBoost_5',
        # 'SentAnalyPtAdaBoost_6',

# ========================= BERT Cross validation ====================
        # 'SentAnalysisPtCrossValidation',
        # 'SentAnalysisPtAdaBoostCrossValidation_2classifier', # 2
classifiers ensemble
        # 'SentAnalysisPtAdaBoostCrossValidation', # 3 classifiers en-
semble


# ====================== Models using RoBERTa =========================
        'SentAnalysisPtRoberta',


# ====================== RoBERTa with Boost ===========================
        'SentAnalyPtAdaBoostRoberta_2', # 2 classifiers ensemble
        # 'SentAnalyPtAdaBoostRoberta_3', # 3 classifiers ensemble

# ====================== BERT Cross validation ========================
        # 'SentAnalysisPtCrossValidationRoberta',
        # 'SentAnalysisPtAdaBoostCrossValidationRoberta',

     ],[],[],[])

#%%
    select = 0
#=========================================================================
```

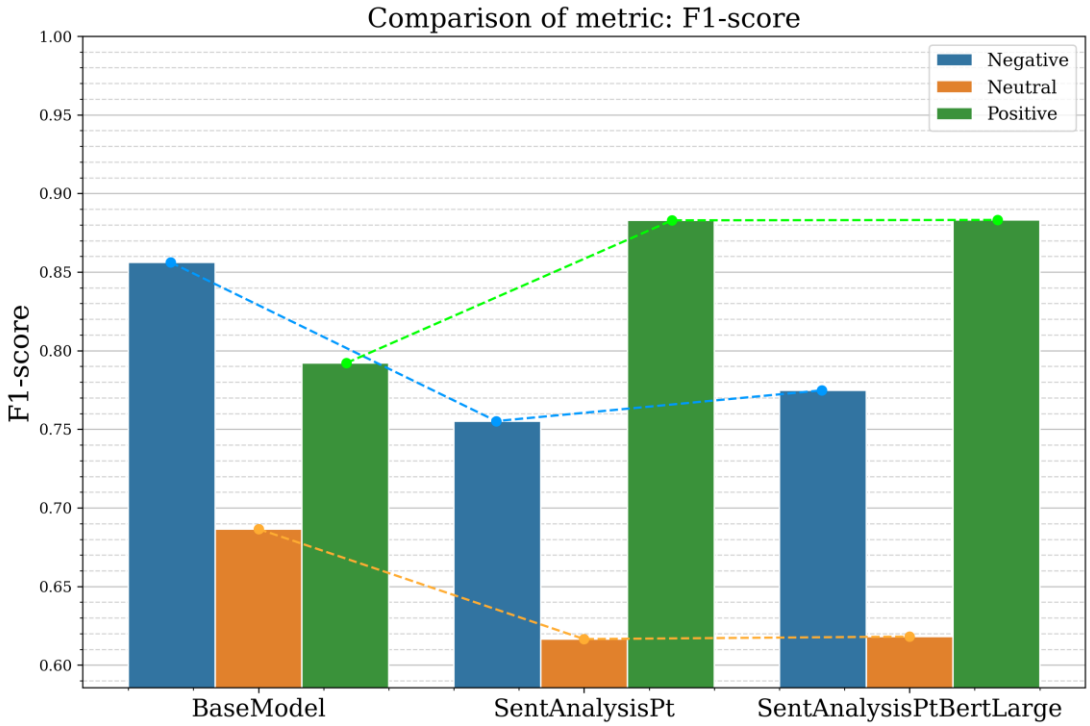
```
    # Selects path from where to load the models
    metrics_path = metrics_path[select]
    Models = Models[select]

    # Run process to extract and plot metrics
    All_metrics, Overall_metrics, wOverall_metrics, \
        Class_metrics, confMat = main(Models,
                                      metrics_path,
                                      plot_Hist=False,
                                      plot_CM=False, whichCM=True, #
True is normalized
                                      plot_Waterfall=False,
                                      plot_Waterfallw=False,
                                      plot_Classes=True,
                                    # selectOverall=['AUC','Overall
ACC','F1 Macro'],
                                    # selectClass=['AUC','ACC','F1']
                                      )
    Overall_metrics_summary = Overall_metrics.describe().T
    wOverall_metrics_summary = wOverall_metrics.describe().T
    for k,v in Class_metrics.items():
        print(f"\n {k:>3} {60*'='}")
        print(v)

    # Only if 2 model are read (first is baseline and second is new)
    # if len(Models) == 2: # Calculate ROI
    #     A = Models[0][:-6]
    #     B = Models[-1][:-6]
    #     title = Models[0][-5:]
    #     metric = 'AUC (ROC)'
    #     mA = Overall_metrics[Overall_metrics.Model == A][met-
ric].item()
    #     mB = Overall_metrics[Overall_metrics.Model == B][met-
ric].item()
    #     ROI = (mB-mA)/(1-mA)
    #     print(f"\nROI between {A} and {B} ({title}):")
    #     print(f"\t{metric} of {A}: {mA*100:.2f}%")
    #     print(f"\t{metric} of {B}: {mB*100:.2f}%")
    #     print(f"\tROI: {'+' if ROI>=0 else ''}{ROI*100:.2f}%")

#============================== END CODE ==============================
```

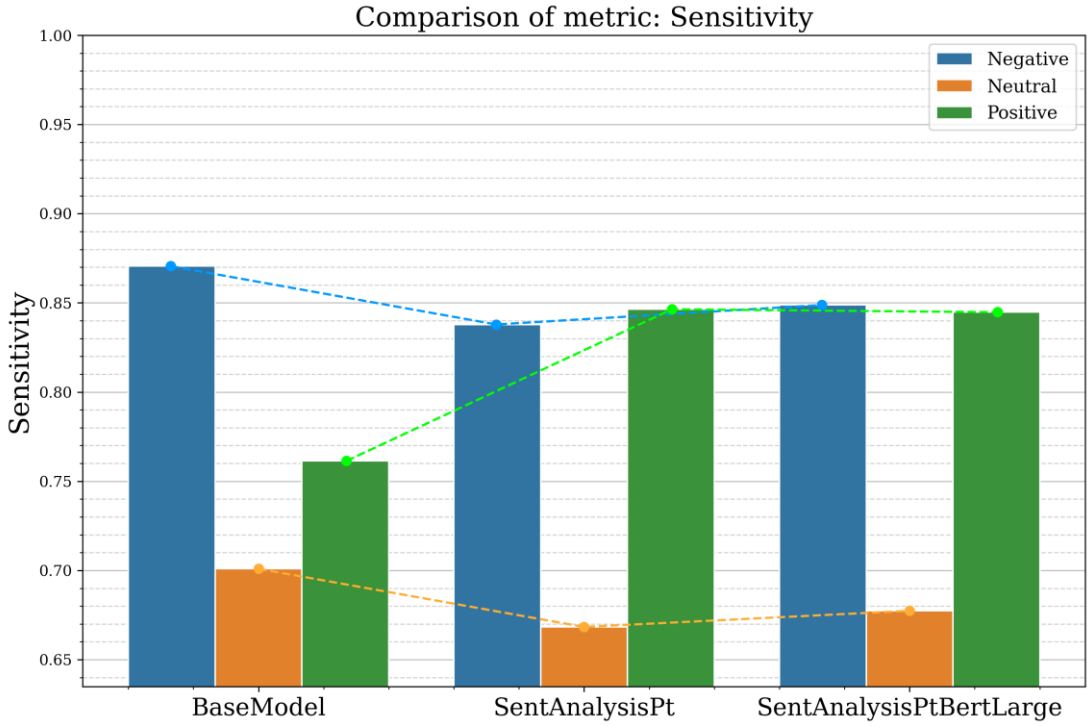
# C. Comparison metrics

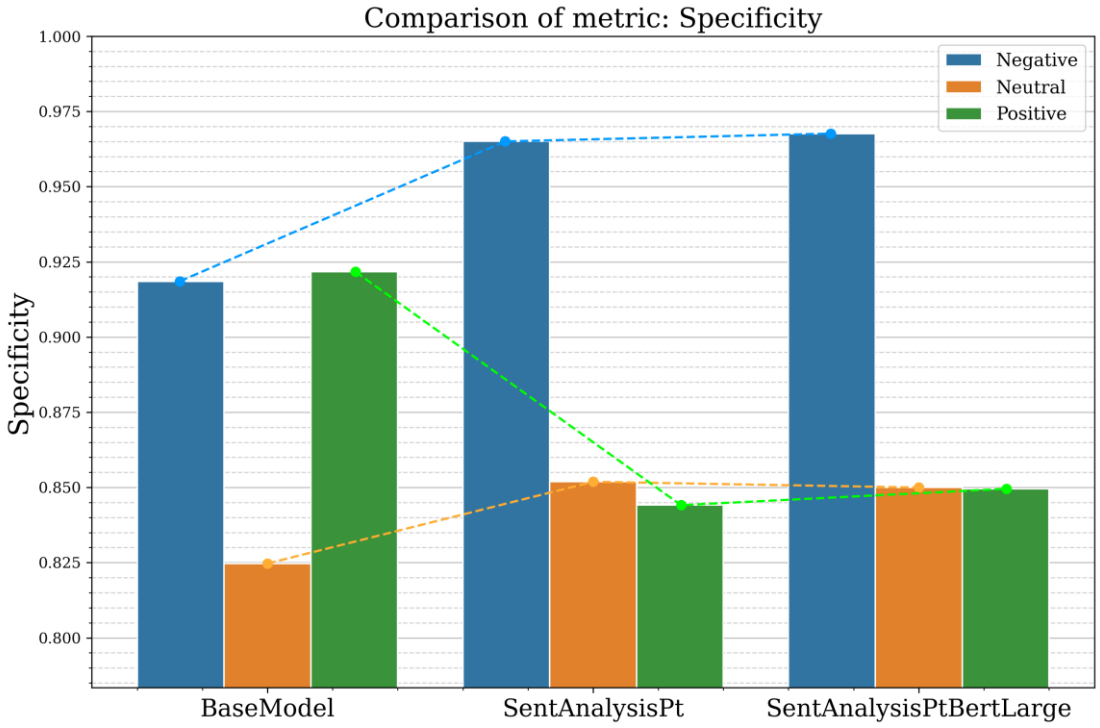
## I. SentAnalysisPt metrics per class

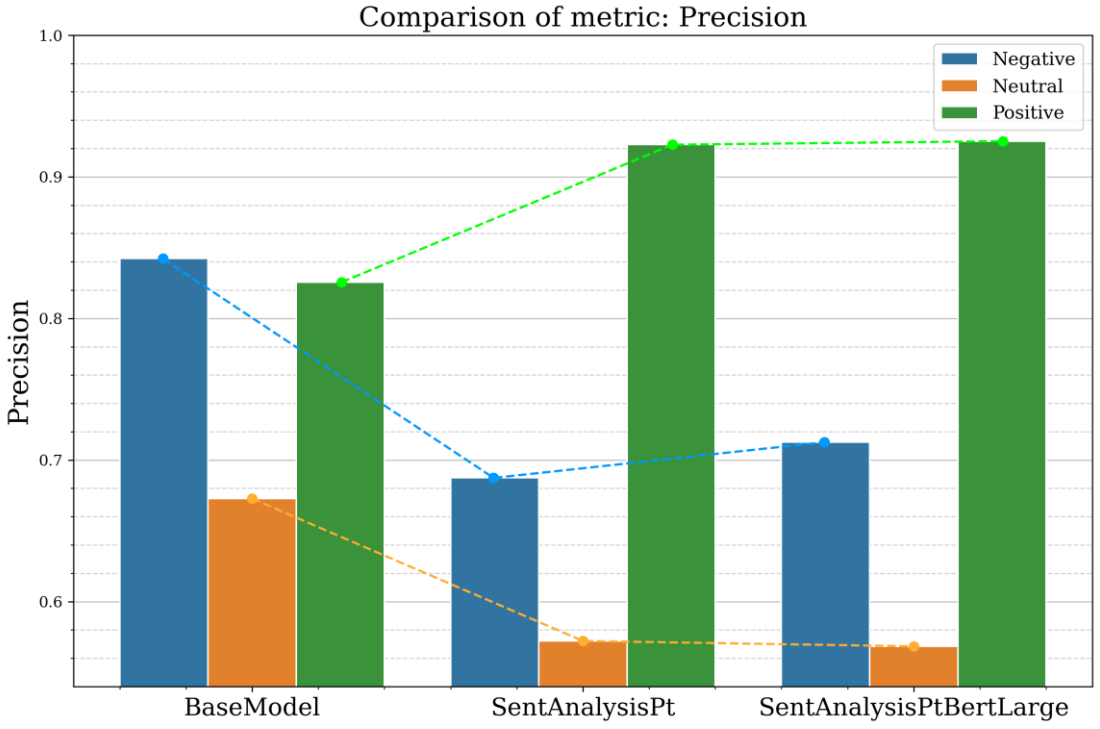


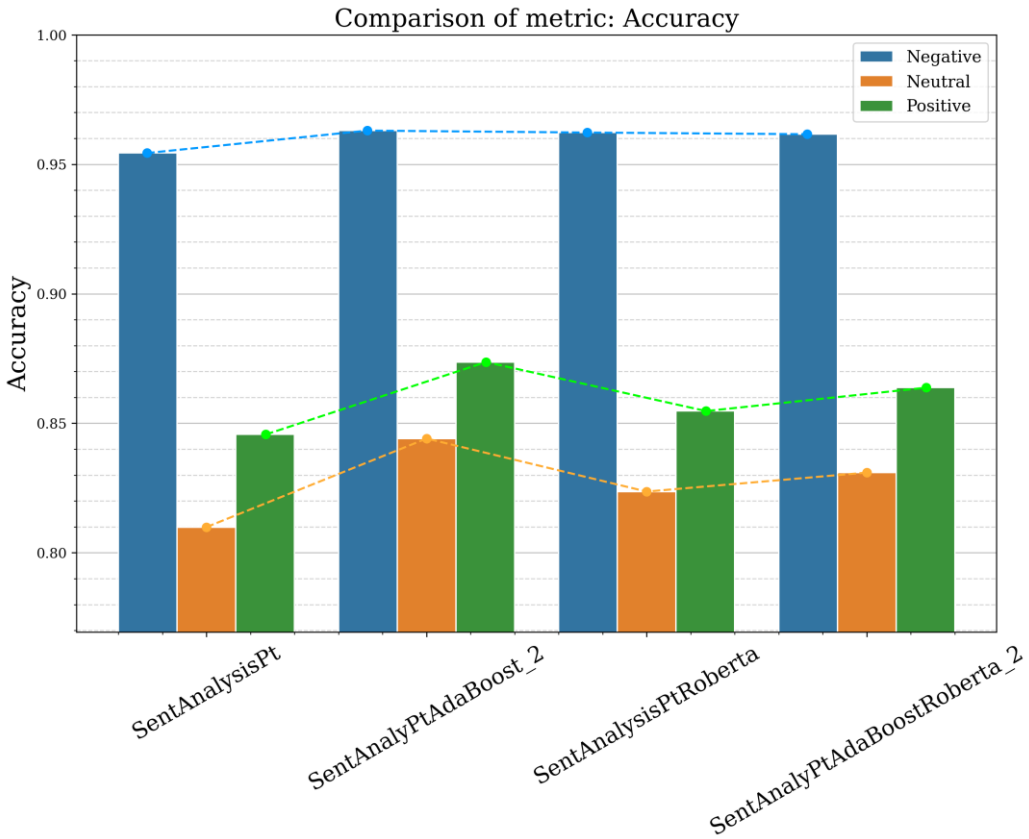

Comparison of metric: F1-score
F1-score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
Negative
Neutral
Positive
BaseModel
SentAnalysisPt
SentAnalysisPtBertLarge

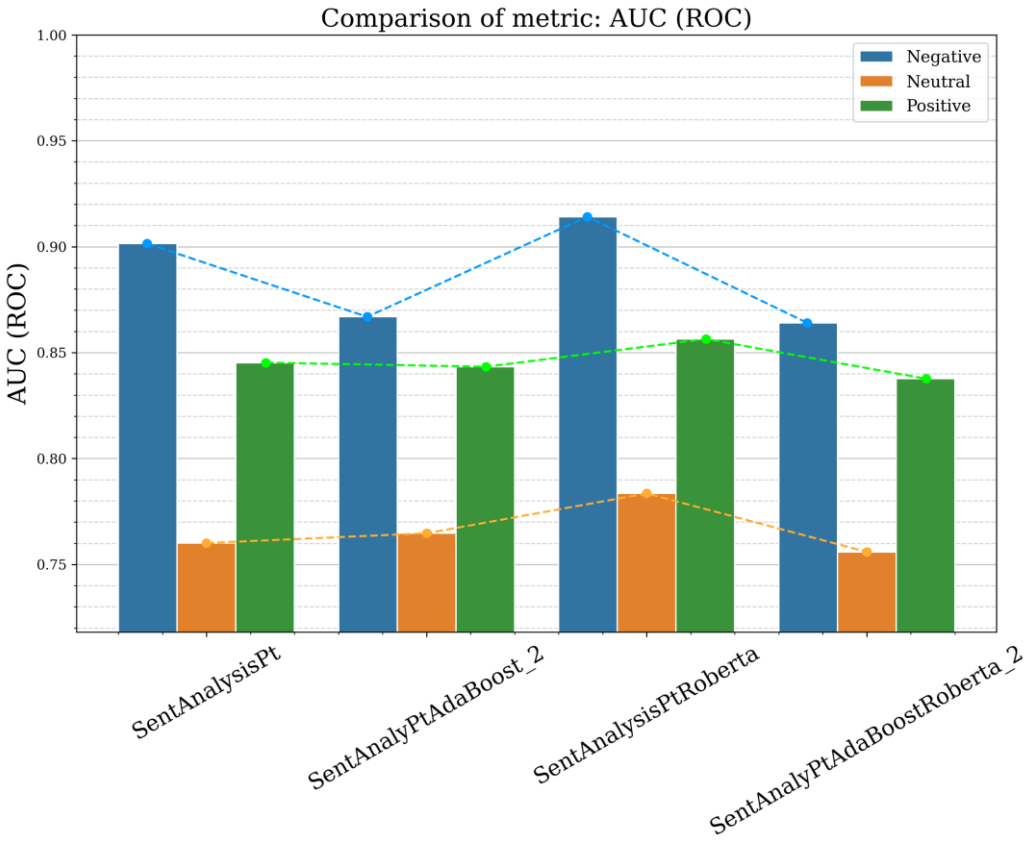

Comparison of metric: Sensitivity
Sensitivity
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
Negative
Neutral
Positive
BaseModel
SentAnalysisPt
SentAnalysisPtBertLarge

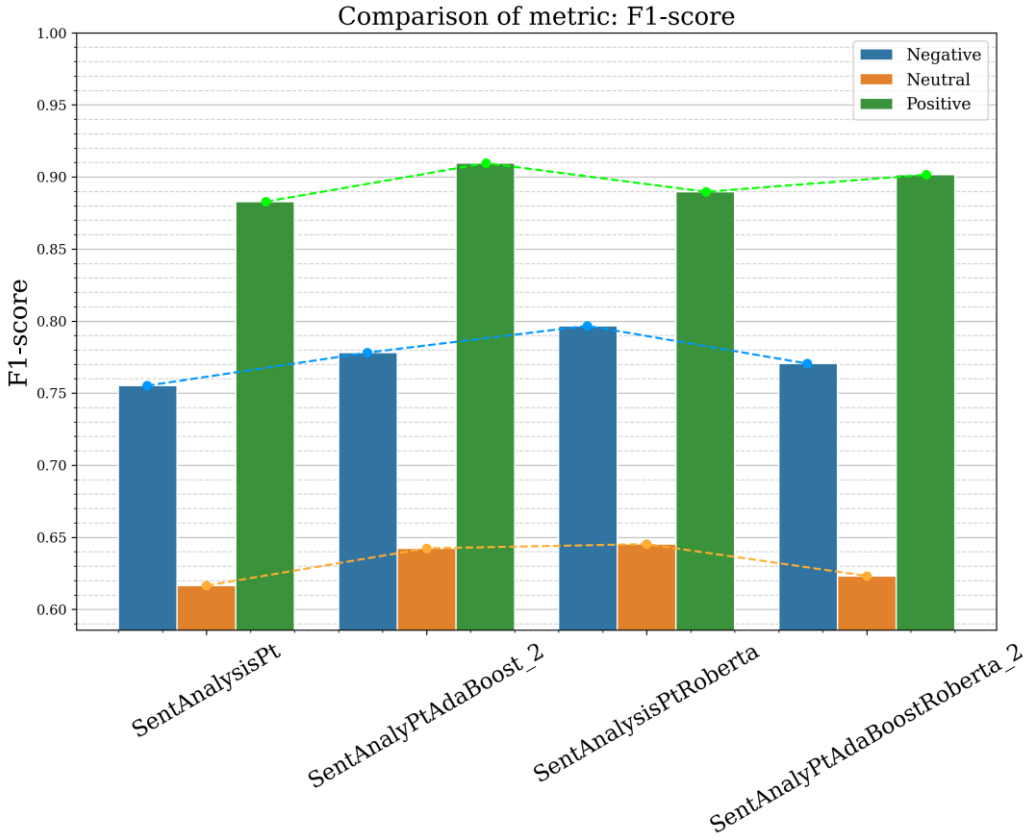

Comparison of metric: Specificity
Specificity
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
Negative
Neutral
Positive
BaseModel
SentAnalysisPt
SentAnalysisPtBertLarge

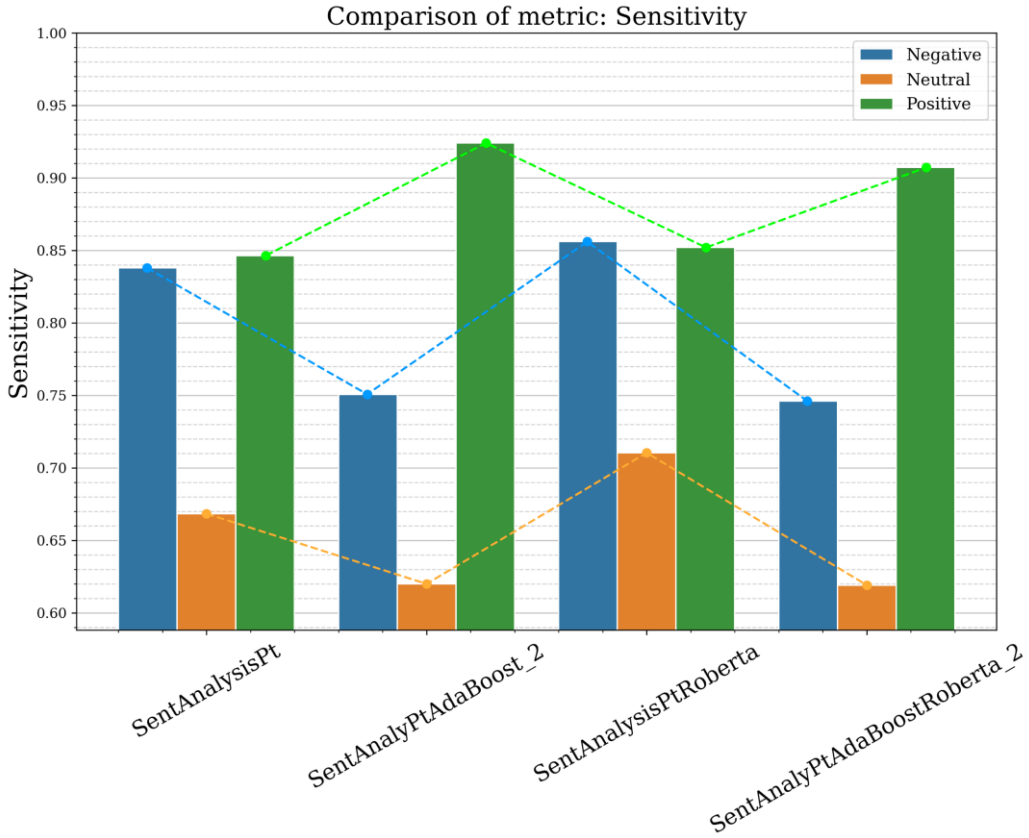

Comparison of metric: Precision
Precision
1.0
0.9
0.8
0.7
0.6
Negative
Neutral
Positive
BaseModel
SentAnalysisPt
SentAnalysisPtBertLarge

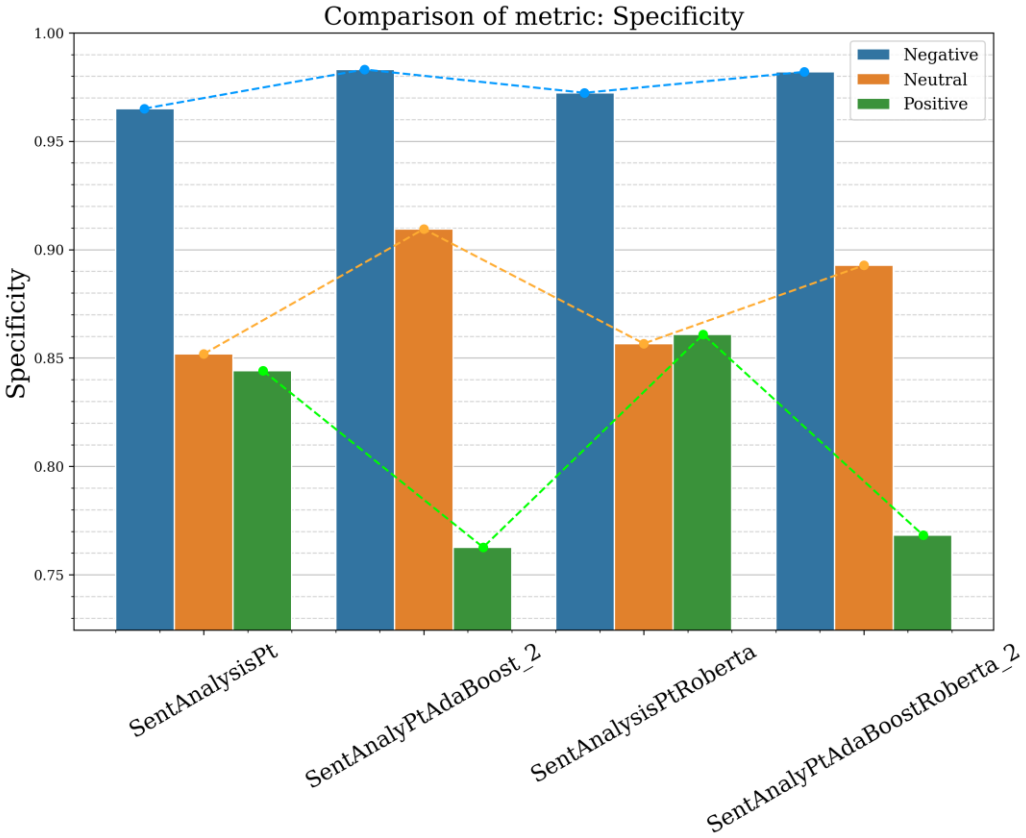
II. All four models metrics per class

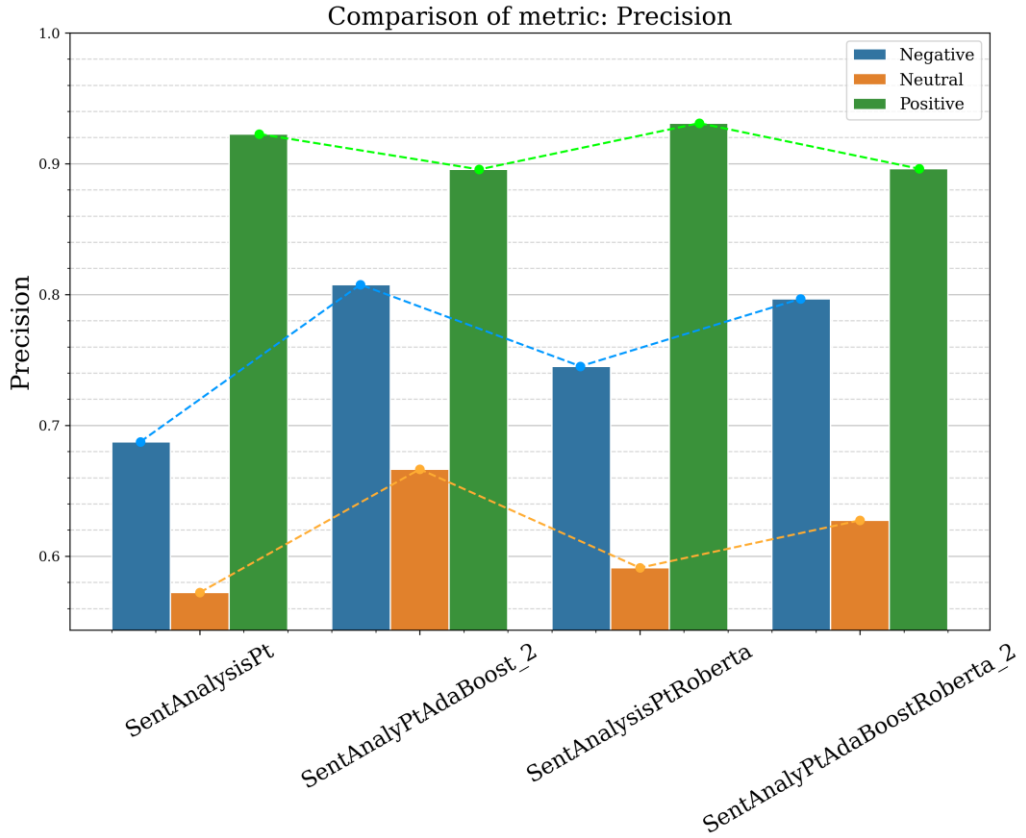



Comparison of metric: F1-score
F1-score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
Negative
Neutral
Positive
SentAnalysisPt
SentAnalyPtAdaBoost_2
SentAnalysisPtRoberta
SentAnalyPtAdaBoostRoberta_2


Comparison of metric: Sensitivity
Sensitivity
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
Negative
Neutral
Positive
SentAnalysisPt
SentAnalyPtAdaBoost_2
SentAnalysisPtRoberta
SentAnalyPtAdaBoostRoberta_2


Comparison of metric: Specificity
Specificity
1.00
0.95
0.90
0.85
0.80
0.75
Negative
Neutral
Positive
SentAnalysisPt
SentAnalyPtAdaBoost_2
SentAnalysisPtRoberta
SentAnalyPtAdaBoostRoberta_2


Comparison of metric: Precision
Precision
1.0
0.9
0.8
0.7
0.6
Negative
Neutral
Positive
SentAnalysisPt
SentAnalyPtAdaBoost_2
SentAnalysisPtRoberta
SentAnalyPtAdaBoostRoberta_2