DM

Deep Learning for Sentiment Analysis
A case study about Portuguese restaurant reviews

MASTER DISSERTATION

Daniel Moisés de Olival Parada
MASTER IN ELECTRICAL ENGINEERING - TELECOMMUNICATIONS

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

February | 2024

Cofinanciado por:
MADEIRA 14-20
REGIÃO AUTÓNOMA DA MADEIRA
PORTUGAL 2020
UNIÃO EUROPEIA
Fundo Social Europeu
arditi
LARSyS Laboratory of Robotics and Engineering Systems
Interactive Technologies Institute LARSyS

Deep Learning for Sentiment Analysis
A case study about Portuguese restaurant reviews

MASTER DISSERTATION

Daniel Moisés de Olival Parada
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

Electrical Engineering – Telecommunications

Deep Learning for Sentiment Analysis: a case study about Portuguese Restaurant Reviews

Daniel Moisés De Olival Parada

Master's degree dissertation

Supervisor:

PhD. Fernando Manuel Rosmaninho Morgado Ferrão Dias

Co-supervisor:

PhD. Fábio Ruben Silva Mendonça

Funchal, Portugal

February 2024

Abstract

This work investigates the usage of deep learning algorithms to perform sentiment analysis over restaurant reviews from the Zomato application, making use of natural language processing techniques to handle text data and taking advantage of the rating given by consumers to perform supervised training. This work presents two models developed from scratch to address the case study problem using recurrent neural networks and self-attention: Recurrent Encoder Classifier and Attentive Recurrent Encoder Classifier. These models were subject to two heuristic-based optimization procedures: a discrete genetic algorithm to select an optimal set of hyperparameters and optimal architecture and a grid search algorithm to optimize the text preprocessing steps. The usage of deep learning models with Portuguese data is limited; hence, the gain in performance was evaluated against classical machine learning models trained on Zomato's dataset, verifying an improvement of 3% in F1-score. The genetic algorithm yielded a relative obtainable improvement score of 4.4% and 8.3% on the recurrent and attentive recurrent encoders architectures, respectively, against their baseline configuration, with the possibility of further optimization by increasing the number of generations. The grid search algorithm slightly improved the performance of each architecture. Both had comparable results, where the Attentive Recurrent Encoder Classifier presented the best performance with 76% of F1-score, 92.5% of ROC-AUC, and 82.7% of accuracy. Tests on a Raspberry Pi application to use the model for inference demonstrated the feasibility of the proposed approach for sentiment analysis in real-world, resource-constrained environments. The results of the study demonstrate that deep learning algorithms can effectively analyze sentiment and show superior results to the traditional ML algorithms and supports the need of exploring smaller, single-task Deep Learning models in the transition of businesses to solutions based on artificial intelligence.

Keywords: natural language processing, sentiment analysis, Portuguese language, deep learning, genetic algorithm, edge computing.

Resumo

Esta dissertação investiga a utilização de algoritmos de aprendizagem profunda para realizar análise de sentimentos em avaliações de restaurantes da aplicação Zomato, fazendo uso de técnicas de processamento de linguagem natural para lidar com dados de texto e aproveitando a classificação atribuída pelos consumidores para realizar o treino supervisionado. Este trabalho apresenta dois modelos desenvolvidos de raiz usando redes neurais recorrentes e mecanismos de atenção: *Recurrent Encoder Classifier* e *Attentive Recurrent Encoder Classifier*; para abordar o caso de estudo. Estes modelos foram submetidos a dois processos de otimização baseados em heurísticas, um algoritmo genético discreto para selecionar um conjunto ótimo de híper-parâmetros e configurações arquiteturais, e um algoritmo de pesquisa de grade para otimizar as etapas de pré-processamento de texto. Dada a limitada utilização de modelos de aprendizagem profunda com dados em português, o seu desempenho foi comparado com modelos clássicos treinados nos dados da Zomato, revelando uma melhoria de 3% no F1. O algoritmo genético resultou num valor da métrica *relative obtainable improvement* de 4,4% e 8,3% para as arquiteturas com codificadores recursivos e recursivos com atenção, respetivamente, em comparação com suas configurações de referência, com a possibilidade de estender o processo de otimização aumentando o número de gerações. A pesquisa em grade melhorou ligeiramente o desempenho de cada arquitetura. Ambas as arquiteturas apresentaram resultados comparáveis, com a *Attentive Recurrent Encoder* obtendo o melhor desempenho, com 76% de pontuação F1, 92,5% de ROC-AUC e 82,7% de precisão. Testes numa aplicação com Raspberry Pi utilizando o modelo para inferência demonstraram a viabilidade da abordagem proposta para análise de sentimentos em cenários do mundo real, com recursos limitados. Os resultados indicam que os algoritmos de aprendizagem profunda podem analisar sentimentos de forma eficaz e mostrar resultados superiores aos algoritmos tradicionais, e apoiam a necessidade de explorar modelos de aprendizagem profunda pequenos e de uma única tarefa na transição das empresas para soluções baseadas em inteligência artificial.

Palavras-chave: processamento de linguagem natural, análise de sentimentos, língua portuguesa, aprendizagem profunda, algoritmo genético, dispositivo de borda.

Acknowledgment

I would like to express my gratitude to Professor Morgado Dias, my supervisor, for the opportunity to do my thesis in this area and for his invaluable guidance and advice throughout this research. Also, thanks to Professor Fábio Mendonça, my co-supervisor, for his insightful suggestions, encouragement, and continuous support during the development of this thesis.

I am deeply thankful to my fellow colleagues from the course, especially my friend Alexandre, for their unwavering friendship, encouragement, and engaging discussions that have enriched my academic journey and this thesis. I would also like to express my gratitude to Eng. Filipe Santos for his remarkable advice and willingness to help.

Finally, I appreciate the RRSO team for granting me the opportunity to actively participate in this project**,** which greatly enriched my research experience. Additionally, I would like to express my gratitude to Zomato Portugal, now Dig-In, for generously providing the essential data, without which this research would not have been feasible.

This research was done under the scope of the project **M1420-01-0247–RRSO–Restaurant Review Sentiment Output**, in collaboration with the *Instituto de Tecnologias Iterativas* (**ITI**), a research unit from the *Laboratório de Robótica e Sistemas de Engenharia* (**LARSYS**). The research was funded by **ARDITI–***Agência Regional para o Desenvolvimento da Investigação, Tecnologia e Inovação* and co-financed by the Madeira **FEDER-000055** Program–European Social Fund. This funding was motivated by the PROCiência 2020 Incentive System.

Index

List of figures

List of tables

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
AREC	Attentive Recurrent Encoder Classifier
AUC	Area Under the Curve
BC	Best Chromosome
BCE	Binary Cross Entropy
BERT	Bidirectional Encoder Representations from Transformers
BT	Back Translation
BoW	Bag-of-Words
CBOW	Continuous Bag-of-Words
CCE	Categorical Cross Entropy
CNN	Convolutional Neural Network
CSL	Cost-Sensitive Learning
CV	Cross-Validation
DL	Deep Learning
DT	Decision Tree
EDA	Exploratory Data Analysis
ELMo	Embeddings from Language Models
FFNN	Feed Forward Neural Network
FN	False Negative
FP	False Positive
GA	Genetic Algorithm
GEV	Generalized Extreme Value
GNMT	Google Neural Machine Translation
GPU	Graphics Processing Unit
GPT	Generative Pre-trained Transformer
GRU	Gated Recurrent Unit
GT	Ground Truth
GeLU	Gaussian Error Linear Unit
HMM	Hidden Markov Model
LAT	Layer-wise Attention Tracing
LLM	Large Language Model
LLaMA	Large Language Model Meta AI
LR	Logistic Regression
LSTM	Long Short-Term Memory
ML	Machine Learning
MLM	Masked Language Modeling
MLP	Multi-Layer Perceptron
NB	Naïve Bayes
NER	Named-Entity Recognition
NLP	Natural Language Processing
NSP	Next Sentence Prediction

OM	Opinion Mining
OOM	Out of Memory
OOV	Out-of-Vocabulary
OVR	One-vs-Rest
PM	Performance Metric
POS	Part-of-Speech
PPV	Positive Predictive Value
PREDATOR	Pretrained Data Augmentor
Pi4B	Raspberry Pi 4 Model B
QA	Question Answering
RBF	Radial Basis Function
REC	Recurrent Encoder Classifier
RF	Random Forest
RMSProp	Root Mean Squared Propagation
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
ROI	Relative Obtainable Improvement
RRSO	Restaurant Review Sentiment Output
ReLU	Rectified Linear Unit
SA	Sentiment Analysis
SBC	Single-Board Computer
SG	Skip-Gram
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
SeLU	Scaled Exponential Linear Unit
TF	Term Frequency
TLM	Transformer Language Model
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
TPU	Tensor Processing Unit
USB	Universal Serial Bus
VPU	Vision Processing Unit

1. Introduction

This work emerges within the scope of a research project aimed at developing Natural Language Processing (NLP) models for the automatic analysis of user reviews in the Zomato Portugal application, which is focused on the restaurant industry. The objective is to identify the sentiment polarity associated with user reviews and to determine the aspects to which these sentiments refer. This way, information can be extracted quickly and automatically, readily available to restaurant owners, eliminating the need for them to read through every customer review manually.

NLP is a complex task and an open research area that has been active lately. Hence, it is intended to explore machine learning algorithms, taking advantage of the available data from Zomato. One key advantage of using these algorithms is their ability to handle large amounts of data and make predictions based on them, typically called data-driven algorithms.

Before delving into this study, it is essential to understand the difference between three terms sometimes used interchangeably: Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL). Their relationship is depicted in Figure 1.1, including NLP, as it is the scope of this work.

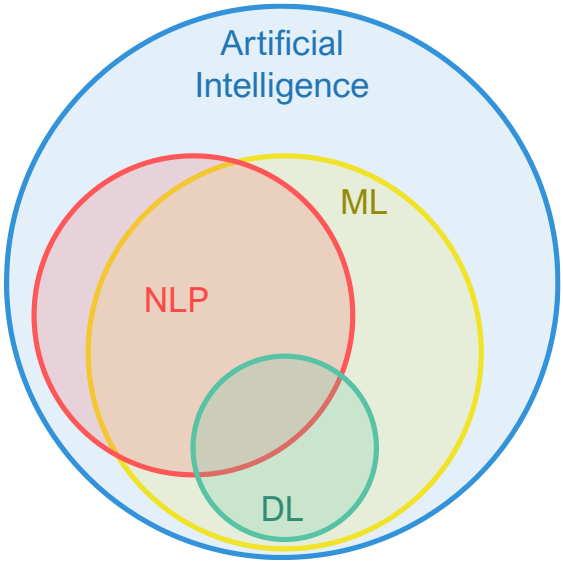


Figure 1.1 – Venn diagram of Artificial Intelligence and Natural Language Processing.

AI was born in the 1950s as an effort to automate intellectual tasks normally performed by humans. The first approaches were based on symbolic AI, where programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge. These rules were suitable for well-defined logical problems. However, they had leakages when dealing with more complex tasks such as image classification, speech recognition, and language processing, where information patterns are not well-defined, and it becomes harder to extract information manually, mainly because humans do not completely understand how our brains work and how we understand and extract meaningful information from images or written and spoken language [1].

Eventually, ML emerged as a new programming paradigm that refers to the development of algorithms and models that can learn and make predictions or decisions based on patterns found in data without being explicitly programmed for each specific task. First, ML algorithms were supervised learners, so the input data and the expected output were given to a model to find a transfer function that could be generalizable to unseen data under a process referred to as training.

Any ML algorithm aims to search for valuable representations of the input data within a predefined space of possibilities (hypothesis space), using guidance from a feedback signal [1].

Finally, DL emerged as a subarea of ML focused on models called Artificial Neural Networks (ANNs) that consist of multiple stacked layers, trained to find the best data transformations that allow better extraction of features and patterns for the intended task. ANNs are mathematical models inspired by our understanding of human brains, where each layer is capable of automatically learning hierarchical representations of data, gradually increasing the level of abstraction. This capability has led to breakthroughs in various domains, such as computer vision, natural language processing, and speech recognition [1].

A state-of-the-art study is conducted to explore viable solutions using ML with a particular emphasis on DL and text representation techniques. Moreover, the study focuses on works developed using Portuguese data since Zomato Portugal primarily serves Portuguese-speaking customers. The study may enable the implementation and optimization of models for sentiment analysis, which can be subjected to rigorous comparative analysis to understand which architectures favor text analysis the most. Moreover, a chapter is dedicated to deploying sentiment analysis models onto an edge device, enabling offline functionality, low power consumption, and cost efficiency.

The subsequent sections of this chapter present the motivations behind employing NLP techniques within the restaurant industry. It also outlines the specific objectives set forth by this work and poses research questions to address the core challenges. Finally, a description of the work structure is provided.

1.1. Motivation

The internet generates a massive amount of data daily, with a significant portion being textual information. While formal and partially structured text can be found in news articles and informative blogs, social media platforms contribute with much informal, subjective, ironic, and sarcastic text. Analyzing these data is challenging, even for humans, as the interpretation heavily depends on the context.

Businesses receive substantial information from social media and their applications, including customer feedback and reviews. However, managing this vast amount of data can be difficult, potentially compromising the quality of service provided. In the restaurant industry, for instance, popular reservation and online ordering applications such as TheFork and Zomato allow customers to leave reviews and ratings so that restaurant owners can extract valuable insights from these reviews to evaluate whether their products and services meet customer expectations. Sentiment analysis has become popular recently, as examining how consumers feel about given products and services helps businesses understand the general consumer's sentiment toward their brand.

DL models constitute the current state of the art for solving complex natural language tasks by extracting patterns from processed text [2], [3], [4]. Today, Large Language Models (LLMs) trained on vast non-domain specific datasets allow enterprises to perform natural language tasks with some ease. However, the resulting models are usually large and considerably resource-demanding, becoming extremely difficult to run locally and highly costly to train. For instance, the LLM Meta AI (LLaMA), which is free to download and use locally, needs at least 10GB of RAM and last-generation Graphics Processing Units (GPUs) [5]. Hence, it is valuable to develop models prepared for specific tasks to optimize their performance and reduce hardware requirements.

1.2. Research questions and objectives

The study aims to develop small DL models (bigger than classical ML models but smaller than LLMs) specializing in domain-specific scenarios. This is valuable for enterprises that already possess business and consumer data and want to create a system that learns from these data. Trained models on specific-domain data may understand nuances about the enterprise's consumers more specifically. The detailed objectives proposed for this study are:

1. Study the current state of the art in Sentiment Analysis (SA), with particular attention to Portuguese data.
2. Exploratory data analysis of the dataset provided by Zomato Portugal.
3. Development of SA models using state-of-the-art techniques, addressing the characteristics of the data.
4. Optimization of text cleaning process and models' hyperparameters.
5. Comparison of performance of each model developed for sentiment analysis considering a baseline model.
6. Implementation of a trained model on an edge device and simple application as proof-of-concept.

By addressing the objectives stated above, this study tries to provide answers to the following research questions:

- Is it possible to develop high-accuracy sentiment analysis models without relying on LLMs?
- Can a model's structure optimized by a heuristic algorithm provide a significant improvement over baseline models in NLP?

1.3. Thesis outline

The remainder of this document is structured as follows: Chapter 2 reviews the theory of NLP, along with the current state of the art in SA using ANNs. Chapter 3 introduces the case study regarding Zomato's dataset provided by Zomato Portugal and defines the methodology followed in the rest of the work. In chapter 4, the development of this study is explained in detail, including data preprocessing, building, and parametrization of the models for SA and methods used to evaluate the performance on a separated test dataset. In chapter 5, an introduction to hyperparameter tuning using genetic algorithms, which are used to optimize the SA models, is presented. Chapter 6 presents all results and comparisons attained, jointly with a discussion on each result. Later, in chapter 7, the best-performed SA model is subject of a proof-of-concept by deploying it on a low-cost edge device. Finally, chapter 8 provides a summary and the conclusions of the developed work, along with the detected limitations and possible solutions to be addressed in future research.

2. Natural Language Processing

NLP, also known as computational linguistics, is a subarea of AI that seeks to solve problems related to understanding human languages in speech or textual form. To do this, NLP provides tools to transform human languages into something understandable by a computer to allow interpretation and information retrieval.

NLP can be divided into two domains: core areas, which deal with language modeling to allow the representation of text into numerical pieces of information, considering semantics, syntax, morphology, word occurrences, and segmentation of meaningful components; and the application area, which deals with specific problems such as opinion extraction, aspect extraction, summarization, machine translation, question-answer problems, classification, and clustering [6]. This study focuses on the second area and utilizes some resources from the core area, such as tools for text preprocessing and text representation.

This chapter introduces NLP from a historical and practical perspective, showing the evolution of this research area and the tools typically used for solving NLP problems. This field has several challenges, but this document primarily concentrates on the SA task using DL techniques.

2.1. Historical background

During the late 1940s, after World War II, there was a significant demand for the creation of machines capable of automatically translating one language into another. At that point, the term NLP had not yet been used, and most translation efforts relied on rudimentary techniques such as dictionary lookups and hard-coded rules [7]. Between 1957 and 1970, the field of NLP split into two dominant thought currents: symbolic and stochastic. Symbolic researchers concentrated on formal languages and syntax, while stochastic researchers explored statistical and probabilistic methods for tasks like optical character identification and pattern recognition.

However, scientists realized that these NLP techniques had limitations when applied to real-world challenges, leading to the incorporation of other fields of science, bridging the gap between computer science and linguistics. In the 1980s, computational grammar theory emerged as a very active research area, encompassing meaning, knowledge representation, and handling user beliefs, intentions, emphasis, and themes. This process permitted the availability of practical resources, grammars, tools, and parsers [4], [7].

The 1990s marked the prominence of statistical language processing, driven by a more cohesive research community emphasizing empiricism and the development of probabilistic models. In the early 2000s, neural language modeling emerged, focusing on predicting the probability of the next word based on previous words using ANNs. However, these solutions were not widely adopted and initially faced a decline in interest due to challenges in training deep neural networks with multiple layers. However, around 2013, DL experienced a revival in their NLP application fueled by improved hardware, such as the GPUs that empowered the algorithms to train deep neural networks, the availability of large training datasets, and the capability of neural networks to learn intermediate representations. This transition allowed breakthroughs in various domains, including computer vision, speech recognition, and NLP [2]. Particularly, techniques from computer vision, such as Convolutional Neural Networks (CNNs), started to be used for NLP applications like machine translation, sentence classification, SA, and text summarization [4]. A notable shift occurred in the 2010s when researchers turned to DL algorithms for NLP problem-

solving while mainly focusing on information extraction and generation due to the vast availability of online information, facilitating the training of LLMs [7].

The main objectives of NLP have included interpreting, analyzing, and manipulating natural language data for the intended purpose using various algorithms, tools, and methods. However, many challenges may depend upon the natural language data under consideration, making it difficult to achieve all the objectives with a single approach. Therefore, developing word embeddings, sequence-to-sequence models, and attention mechanisms have been the main areas of study that have driven researchers to state-of-the-art results in NLP tasks [4].

Currently, NLP is mainly a data-driven field that uses statistical and probabilistic computations along with ML algorithms to try to solve the mentioned challenges, while in the past, most solutions employed probabilistic models such as Naïve Bayes (NB) and Hidden Markov Models (HMMs) [6], [8]. With the emergence of the Bidirectional Encoder Representations from Transformers (BERT) model, many solutions have turned into transfer learning approaches to produce high-accuracy models with less training data. Otter et al. [6] stated that DL models are becoming the norm in computational linguistics, with pre-training and transfer learning being the most used approaches. Other models have emerged in recent years, all based on variations of the original Transformer architecture from Vaswani et al. [9] and optimization of the training process, using large fleets of GPU and Tensor Processing Units (TPUs) and large datacenters. Models such as LLaMA [10], Falcon [11], GPT-4 [12] from the Generative Pre-trained Transformer (GPT) family, and, recently, Gemini [13] from Google, are LLMs with billions of parameters, trained on massive corpora and based either on the encoder or decoder of the Transformer. The first 2 mentioned are open-source models. These models are capable of understanding not only text sequences, but also images, video, and audio, surpassing previous benchmarks and even surpassing human evaluation in most cases [12].One of the main limitation of the LLMs is the limited context window, which is in constant improvement with each new model launched, as it is the case of Gemini 1.5, that uses contextual windows of up to a million tokens, ~3000% larger than previous GPT-4 [14].

2.2. Sentiment analysis

SA, sometimes called Opinion Mining (OM), is a subfield of NLP in active development that focuses on automating the extraction and classification of opinions, polarity, emotions, and feelings expressed in written text by developing computational methods and tools that enable valuable insights into public sentiment across various domains and applications. The sources of these texts are typically social networks, online blogs, forums, and shopping platforms. SA primarily centers on the valence of emotions, categorizing them as positive, negative, or neutral. It deals with the challenge of understanding the literal meaning of words, their contextual and semantic roles, and the author's intentions to classify the sentiments expressed. A sentiment is a feeling, emotion, or opinion expressed through thoughts, words, or actions. Yadolahhi et al. [15] defined a sentiment as an opinion or idea colored by an emotion, so both opinions and granular emotions must be investigated when analyzing the sentiments printed in text. Sentiments can be positive or negative, varying in intensity and complexity. Different models have been developed following the theoretical framework of the Dimensional Emotion Theory, supported by Plutchik, Kellerman, Shaver, and Lövheim [15], which includes emotions such as anger, disgust, fear, joy, sadness, surprise, shame, trust, and others.

One can mathematically formulate the task of SA as follows [16]: Considering a corpus $D = \{d_1, d_2, \dots, d_n\}$ of N documents (reviews or sentences). Each document is a string of characters

that could be represented as a list of features extracted from different techniques, such as term frequency, Part-of-Speech (POS) tagging, or word and sentence embedding. Considering $C = \{c_1, c_2, \dots, c_l\}$ the set of L classes that label each document with their respective polarity, or emotion. The primary aim of SA is to develop a classification function able to map a document d to its c according to the extracted features.

Researchers usually separate the SA task into three levels of granularity, specifically:

- **Document-level:** Evaluation of an entire document, regardless of its length, to determine its overall sentiment polarity. The document should be on a single topic to consider this approach [17]. This method is widely used for tasks such as classifying product reviews, news articles, or social media posts as positive, negative, and sometimes neutral based on the emotions and opinions conveyed in the entire text [2]. This method provides valuable insights into the polarity of a specific feature, but it does not capture the nuances of people's likes and dislikes. However, this approach concerns most of the body of the work for this area and is considered the most straightforward sentiment analysis task in the research community [15].
- **Sentence-level:** Focuses on evaluating the sentiment expressed within individual sentences. It involves two key steps [18], specifically, subjectivity classification to determine whether a sentence expresses opinions or factual information, and polarity classification to classify subjective sentences into positive or negative. This type of analysis is valuable when documents contain a mixture of sentiments. Sentence-level sentiment analysis allows for a more granular assessment of the sentiment within a text.
- **Feature/Aspect-level:** Focuses on discovering opinion polarities regarding specific aspects or features of a product, service, or entity [17]. Unlike document or sentence-level sentiment analysis, which provides generalized sentiment assessments, aspect-level sentiment analysis delves into finer details, including the identification of the aspects, which can be done manually or using methods to extract entities (or topics) and aspects automatically, and the sentiment classification concerning each aspect [2], [18].

Additionally, researchers have included the phrase, word, and character level [8], [19]. Online text reviews can be encountered in many formats, which can conditionate the SA challenges that researchers must overcome. Books and research reviews are generally written in a formal manner, so the polarity of the phrases is more easily highlighted. Therefore, these reviews are mentioned to contain structured sentiments. On the other hand, unstructured sentiment reviews are informal texts where the writer does not necessarily follow any format constraints, and have the potential to provide more abundant and detailed opinion information than its counterpart, possibly including explicit and implicit features from a product or service. Also, a third category lies in the middle of the two before-mentioned, the semi-structured sentiments reviews, where the writer typically enumerates pros and cons using short phrases [19].

Most research focuses on document-level SA to extract unstructured sentiments, applying one of the two main approaches: Lexicon-based and ML-based, and sometimes hybrid approaches. Figure 2.1 summarizes the different alternatives to perform SA [8], [17], [18].

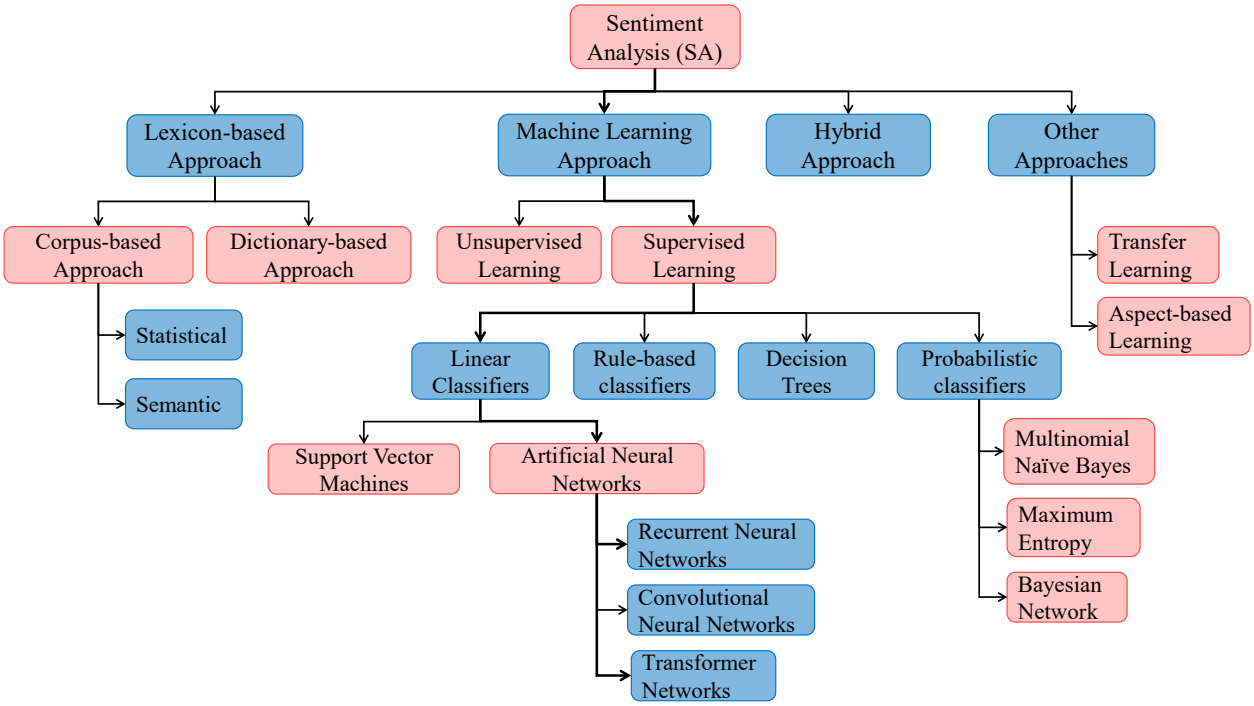


Figure 2.1 – Approaches of SA.

Until 2014, lexicon-based approaches were the most common in NLP tasks due to the lack of computational and data resources to train ML. Only shallow ANNs had been used by researchers, demonstrating inferior performance compared to classical ML methods [18]. However, recent advances in SA have been driven by using ANNs as the primary tool to achieve state-of-the-art results, using the mechanisms shown in Figure 2.1 and encouraging other approaches, such as transfer learning. The following section describes the most critical components used in SA based on ANNs and DL.

2.3. Artificial Neural Networks for SA

ANNs are a subset of ML at the core of DL algorithms, which rely on training datasets to learn and improve their performance on two primary tasks: regression and classification. The main limitation is the lack of labeled data for supervised training and the complexity of dealing with text or speech data. In the last decade, SA has surged in popularity within the field of NLP, essentially thanks to the application of DL techniques empowered by increased computational resources, the wealth of training data, and DL's ability to handle complex data, enabling state-of-the-art results. These advancements have made DL a dominant force in SA, demonstrating its versatility and effectiveness [2].

This section provides an overview of the most used architectures and layers in ANNs to perform SA and obtain state-of-the-art results, focusing on the mathematical operation of the layers and the reasoning behind their functioning. First, a brief description of a tensor is given since it constitutes the mathematical object at the core of all operations performed by DL models.

All current ANN systems use tensors as their primary data structure. A tensor is a data container, typically for numerical values, with an arbitrary number of dimensions. A tensor that contains only one value is called a scalar (0D tensor), an array of values is called a vector (1D tensor), and an array of vectors is called a matrix (2D tensor). Finally, for the general case, 3D or higher-dimensional tensors can be interpreted as cubes of numbers or cubes packed inside vectors and are simply referred to as tensors. Typically, DL models deal with tensors from 0D to 4D [20].

For NLP tasks, data is generally treated as time-series or sequential data stored in 3D tensors with an explicit time axis (the second axis, for convention), as shown in Figure 2.2.

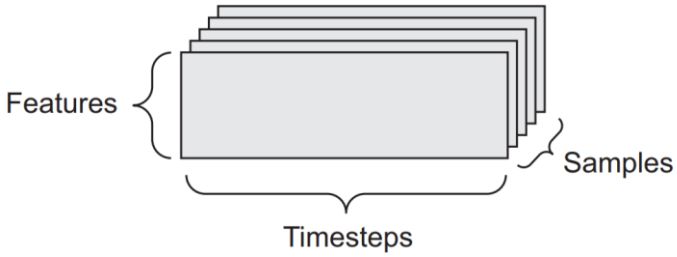


Figure 2.2 – Shape of 3D tensor representing time-series data, such as text.

All data transformations learned by DL models are possible thanks to tensor operations, such as adding, multiplying, and logical operations. These can be performed as element-wise operations. Tensor multiplication can also be computed as the inner product, or dot product, which is the most common and useful tensor operation.

2.3.1. Overview of ANNs

ANNs are mathematical models loosely inspired by the structure and functions of the biological neural networks in humans' brains. ANNs are composed of layers of artificial neurons, which are their processing units and consist of weights, or parameters, that must be trained with labeled data and an error backpropagation algorithm to update their values after several iterations. The mathematical representation of a neuron's output can be described as [21]

$$z_k = \varphi_k\left(\sum_{i=0}^{M} w_{ki}x_i + b_k\right), \tag{2.1}$$

where x_i are the input values, w_{ki} are the weight parameters of the neuron ($\mathbf{w_k}$ is a vector with the same dimension as the input vector $\mathbf{x}$), and M is the dimensionality of the input. A neuron k can also include an additional weight referred to as bias, b_k, which allows to include a fixed offset at the neuron's output, providing more flexibility [21]. Finally, the result of this operation is passed through a non-linear activation function φ to allow the neuron to model complex relationships in data.

A set of neurons can be organized to form a neural network, typically called Feed-Forward Neural Networks (FFNNs) or Multi-Layer Perceptrons (MLP). These fundamental ANN architectures employed in DL consist of multiple layers of interconnected nodes, each with its own neurons. The network is structured so that information flows in one direction, from the input to the output layer, without forming cycles or loops. The relevance of this new stacked architecture comes from the limitation of a single neuron to solve initially simple problems such as modeling the XOR function [22]. Considering (2.1), an FFNN composed of an input layer, a hidden layer, and an output layer can be described as [21]

$$y_k(\mathbf{x},\mathbf{w}) = \varphi_l\left(\sum_{j=0}^{N} w_{lj}^{(2)}\left[\varphi_k\left(\sum_{i=0}^{M} w_{ki}^{(1)}x_i + b_k^{(1)}\right)\right] + b_l^{(2)}\right), \tag{2.2}$$

where the superscripts (1) and (2) indicate the parameters of the hidden layer and output layer, respectively. M is the dimensionality of the input, and N is the number of neurons in the hidden layer. Equation (2.2) can be extrapolated for deeper networks (higher number of layers), with FFNNs with up to two layers usually denominated shallow networks and networks with more than two layers designated deep networks. The activation functions ϕ are typically chosen to be

sigmoidal functions, such as the logistic Sigmoid or the Tanh function, or linear units, such as Rectified Linear Unit (ReLU), Scaled Exponential Linear Unit (SeLU), or Gaussian Error Linear Unit (GeLU). The mathematical expressions for these functions are given, respectively, by

$$\phi_\sigma(z) = \frac{1}{1+e^{-z}}, \tag{2.3}$$

$$\phi_h(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{2.4}$$

$$\phi_R(z) = \begin{cases} z, & z > 0 \\ 0, & \mathrm{z} \leq 0 \end{cases}, \tag{2.5}$$

$$\phi_S(z) = \begin{cases} \lambda z, & z > 0 \\ \lambda\alpha(e^z - 1), & \mathrm{z} \leq 0 \end{cases}, \tag{2.6}$$

$$\phi_G(z) = z \cdot \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{z}{\sqrt{2}}\right)\right]. \tag{2.7}$$

The constants λ and α in (2.5) are predefined parameters of the function, and (2.6) requires the input $z \in Z \sim \mathcal{N}(0,1)$. A commonly used activation function for the output layer, particularly in multi-class classification problems, is the SoftMax function. Given $\mathbf{z}$ of values $z_1, z_2, \ldots, z_C$, the SoftMax function is given by

$$\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{C} z_j}, \tag{2.8}$$

for each element in $\mathbf{z}$, which converts a vector of C real numbers into a probability distribution of C possible outcomes. It is a generalization of the logistic function to multiple dimensions.

FFNNs follow the universal approximation theorem, which states, in simple terms, that regardless of what continuous function a model is trying to learn, there is always a MLP that can represent this function. However, there is no universal procedure to train the network correctly. Therefore, this theorem says that there exists a network large enough to achieve any degree of desired accuracy. However, it does not say how extensive this network will be. The training limitations are regarding the optimization algorithm, further discussed in subsection 2.3.4.

In addition to exploring optimization algorithms that enable the proper training of MLPs, specialized types of neural networks have emerged over the years for processing data of diverse kinds, particularly for NLP tasks. CNNs, Recurrent Neural Networks (RNNs), and attention mechanisms are the primary techniques employed in NLP. The former was initially developed for image data and later adapted for sequential data, while the latter two were explicitly designed for managing sequential information. In this study, RNNs and attention mechanisms are exclusively investigated, as the capacity to capture sequence dependencies using recurrent models has motivated researchers to use them over CNNs in NLP areas [3]. However, different studies contrast this idea, showing that CNNs can provide comparable performance, but it depends on the global semantics required by specific tasks [23].

As referred by Otter et al. [6] and Young et al. [3], ML approaches such as NB, Support Vector Machine (SVM), Logistic Regression (LR), Decision Tree (DT), and HMM were mainly used to target NLP problems, training on very high dimensional and sparse features. However, for more complex tasks where larger datasets are used for training, their practical applicability is limited by the curse of dimensionality. ANNs, particularly DL models, were introduced to overcome this problem, particularly with the introduction of distributed representations of NLP

data. The upcoming subsections describe the mathematical foundations used as building blocks for DL models and briefly overview the training process.

2.3.2. Recurrent Neural Networks

When dealing with sequential information, such as text, it is known that the semantical meaning of each word depends on the previous words and, in some cases, on the following words in the sentence. Therefore, it is appealing to execute computations that have "memory" over previous computations and can use this information in the current processing [3]. With this mindset, RNNs were developed by Williams and Zipser in the 1980s [24], characterized by their hidden layers that have recurrent connections that allow them to maintain a form of memory by using their output as input for the next step. RNNs are one of the most utilized methods for SA.

The most general representation of a vanilla (or standard) RNN is illustrated in Figure 2.3, which is unfolded across time to accommodate a whole sequence. The purpose of the network is that, given an input vector $\boldsymbol{x_t}$ at timestep t, the hidden state of the same timestep, $\boldsymbol{h_t}$, is computed by

$$h_t = f(U\boldsymbol{x_t} + V\boldsymbol{h_{t-1}}), \tag{2.9}$$

where U and V are the learnable parameters of the network for input-to-hidden and hidden-to-hidden links, respectively, and are shared across time. There is also a T parameter for hidden-to-output, seen in Figure 2.3, also shared across time. The fact that these are shared parameters allows us to train a single model capable of dealing with all time steps [25]. The hidden state of the RNN is typically considered to be its most crucial element due to its ability to accumulate information from previous time steps. However, these networks suffer from the vanishing and exploding gradient problem, provoking dependencies that occur over a long period to be less well captured by RNN [3], [24]. In posterior notation, all weights denoted by W encompass two matrix parameters, as seen in (2.9). Therefore, $W = [V, U]$.

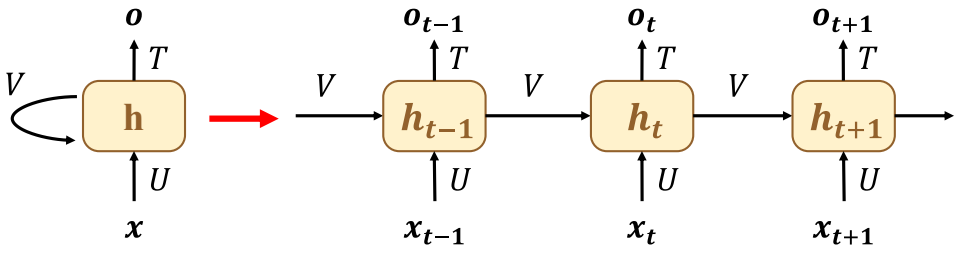


Figure 2.3 – Illustration of the vanilla RNN and its unfolded representation.

Researchers have developed many modifications to the vanilla RNN to overcome this limitation. The most relevant variants in NLP application are the Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) [3].

LSTM is a variant of the vanilla RNN that addresses the challenges of preserving and discarding information from the hidden state of a RNN cell, essential for preventing the problem of gradient vanishing. An LSTM employs three crucial gates: The input gate that updates the cell state by passing the input $\mathbf{x} = [h_{t-1}, x_t]$ into a Sigmoid function with parameters W_i and b_i, as in

$$i_t = \sigma(W_i \cdot \mathbf{x} + b_i), \tag{2.10}$$

with values between 0 and 1, denoting higher and lower relevance, respectively. Next, $\mathbf{x}$ passes into a TanH function with parameters W_c and b_c to create an intermediate vector $\widetilde{c_t}$ given by

$$\widetilde{c_t} = \tanh(W_c \cdot \mathbf{x} + b_c), \tag{2.11}$$

with values ranging from –1 to 1. Furthermore, the forget gate decides which information can be ignored by computing

$$f_t = \sigma\left(W_f \cdot \mathbf{x} + b_f\right) \tag{2.12}$$

and generating values near to zero when information from previous time steps must be forgotten and near to one when information must be kept. The output from the two gates is combined using an element-wise multiplication given by

$$c_t = f_t * c_{t-1} + i_t * \widetilde{c_t}\,, \tag{2.13}$$

where $*$ denotes the element-wise multiplication operation and c_{t-1} the cell state from the previous timestep. The current cell state, c_t, is computed by deciding how much information is forgotten (first term of the equation) and how much must be learned from the current timestep (second term) [2], [3]. Finally, the output gate defines the value of the next hidden state and determines which relevant information from the prior steps is needed by combining **x** into a Sigmoid function of parameters W_o and b_o, as in

$$o_t = \sigma(W_o \cdot \mathbf{x} + b_o)\,. \tag{2.14}$$

On the other hand, the current cell state, c_t is passed into a TanH function, and the operation

$$h_t = o_t * \tanh(c_t) \tag{2.15}$$

is computed to obtain the current hidden state. The new cell state and new hidden state are carried over to the next time step, and the process repeats for all subsequent time steps.

These gates collectively determine how much relevant context should be preserved while filtering out irrelevant context, which, unlike vanilla RNNs, enable error back-propagation through numerous time steps, facilitating the capture of dependencies over extended sequences [3].

GRU is a simpler variant of the LSTM and faster to compute, which has demonstrated comparable or superior performance in various tasks [6]. GRU comprises two gates: The reset gate, r_t, decides how much of the past information is needed to neglect and the update gate, z_t, is responsible for determining the amount of previous information that needs to be passed to the next state. GRU handles the flow of information like an LSTM without a memory unit. Thus, it exposes all of the hidden content without any control. The reset gate combines with the previous hidden state to determine the most relevant information, computing an intermediate hidden state $\widetilde{h_t}$. Finally, the update gate takes place to select the information to keep and disregard at the same time from the previous and the current hidden state [2], [26]. Like LSTM, the expressions that describe the working of GRU are the following:

$$z_t = \sigma(W_z \cdot \mathbf{x} + b_z)\,, \tag{2.16}$$

$$r_t = \sigma(W_r \cdot \mathbf{x} + b_r)\,, \tag{2.17}$$

$$\widetilde{h_t} = \tanh(W_h \cdot [r_t * h_{t-1}, x_t] + b_h)\,, \tag{2.18}$$

$$h_t = (1 - z_r) * h_{t-1} + z_t * \widetilde{h_t}\,. \tag{2.19}$$

Figure 2.4 graphically represents the LSTM and GRU cells. Throughout history, most of the choices over the RNN variant tended to be heuristic [3]. Typically, LSTM is preferred over GRU despite its increased complexity. Moreover, BiLSTM layers (two LSTM layers performing direct and inversed passes through the input sequence) have shown the greatest performance, as they are able to extract contextual information from both sides of each token.

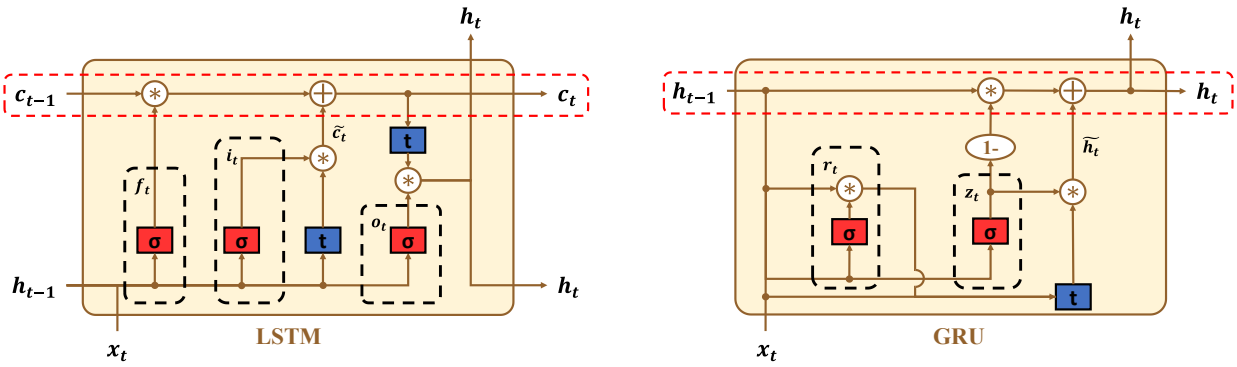


Figure 2.4 – Graphical representation of LSTM and GRU cells.

2.3.3. Attention mechanisms and Transformers

The introduction of attention mechanisms had a specific objective: to enhance the efficiency of the encoder-decoder model in machine translation by emphasizing particular segments within a sequence. The concept behind it was to enable the decoder to incorporate the most relevant segments of the input sequence. This was achieved by computing a weighted sum of all the encoded input vectors, attributing the highest weights to the most relevant vectors.

Bahdanau et al. [27] introduced the first attention mechanism in 2014 with the aim of solving the bottleneck problem produced by the inability of a fixed-length vector to encode all the relevant information, as irrelevant information introduces noise. The attention scores would automatically search for parts of a source sentence that are relevant to predicting a target word. Bahdanau's attention mechanism was intended to be used with RNNs, providing a weight for the hidden state of each timestep [2]. The procedure was the following: first, the alignment scores were calculated by

$$e_{t,i} = \mathrm{a}(s_{t-1}, h_i)\,, \tag{2.20}$$

where h_i are all the previous hidden states from the encoder and s_{t-1} are the previous decoder output. An FFNN typically defines the function a. Second, these scores are passed into a SoftMax function to compute the attention weights, $\alpha_{t,i}$, ranging from 0 (less relevant) to 1 (more relevant). Finally, the context vector is computed as the weighted average of all encoder's hidden states as

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} \cdot h_i\,. \tag{2.21}$$

In 2017, Vaswani et al. proposed a new simple network architecture called the Transformer, which revolutionized the NLP field [9]. The Transformer is based on an encoder-decoder structure, where an input sequence **x** is transformed into a sequence of continuous representations **z** that is then decoded to generate a new sequence **y**, utilizing an architecture constituted of stacked self-attention mechanisms and FFNNs. The Transformer architecture was initially developed to solve NLP's machine translation task. Models developed using this architecture are typically referred to as Transformer Language Models (TLMs).

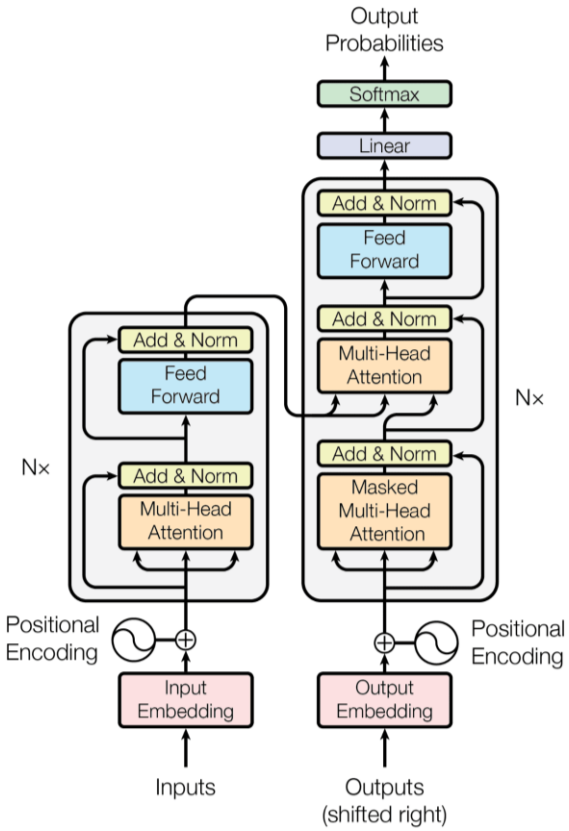


Figure 2.5 – Architecture of the Transformer, introduced by Vaswani et al. [22].

An attention function can be defined as a process that maps a query vector (Q) and a collection of key-value vector pairs (K-V) as input and produces an output. This output comes from calculating a weighted sum of V, with each weight determined by a compatibility function between Q and its corresponding K. Vaswani et al. [9] introduced two distinct adaptations of the self-attention mechanism:

The scaled dot-product attention is an adaptation, where the dot product of Q and K vectors of dimension d_k, is computed, and the result is normalized by dividing by $\sqrt{d_k}$ to control the magnitude of the resulting weights. Afterward, these values are passed into a SoftMax function and multiplied by the V vector, with dimension d_v. The performed operation is the following:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V\,. \tag{2.22}$$

This mechanism allows the model to give more attention to relevant parts of the input sequence while dampening the importance of less relevant elements.

Multi-head attention is an adaptation that involves using multiple sets of attention mechanisms, each known as a "head". Vaswani et al. [9] found it beneficial to linearly project the Q, K, and V vectors h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. Each head independently and parallelly learns different attention patterns that are then concatenated and projected back to the original d_{model} dimension, providing richer context and improving its ability to handle complex dependencies in sequences. The computation of multi-head attention is given by

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h)W_o\,, \tag{2.23}$$

where each head is computed by applying (2.22) to the projections of Q, K, and V vectors, with parameters W_i^Q, W_i^K, and W_i^V, respectively, and i being the corresponding head. The parameters W_o performs the projection from the head domain to the model domain.

2.3.4. Training process

When training a ANN, the aim is to determine the network parameters, denoted as **w**, that can effectively transform a vector **x** of input variables to a vector **y** of output, or target, variables. These parameters are sought in a way that minimizes an associated error function, denoted as $\mathrm{E}(\mathbf{w})$ [21]. Figure 2.6 provides a geometrical view of the error function as a surface sitting over a weight space, where $\nabla\mathrm{E}$ specifies the direction of the greatest rate of increase of the error function when a slight variation is applied to **w**.

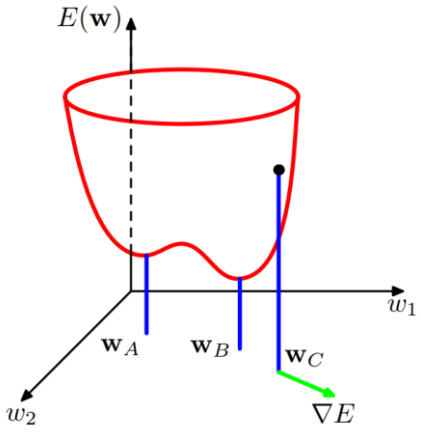


Figure 2.6 – Geometrical representation of the error function on a weight space. Point w_A is a local minimum. Point w_B is the global minimum. Any point w_C is assigned to a local gradient of the error, $\nabla\mathrm{E}$ [15].

The smallest value of the error function occurs at a point in **w** space where the gradient of the error function vanishes, thus

$$\nabla\mathrm{E}(\mathbf{w}) = 0\,. \tag{2.24}$$

This point cannot be obtained analytically, so the optimization of the error function is iterative, involving setting an initial value $\mathbf{w}^{(0)}$ and moving through the weight space in a succession of little steps. It is possible to evaluate the gradient of an error function efficiently by means of the backpropagation procedure to choose the weight update vector to comprise a small step in the direction of the negative gradient. The general expression to update a ANN's weights is given by

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla\mathrm{E}\left(\mathbf{w}^{(\tau)}\right), \tag{2.25}$$

where τ is the iteration step and the parameter $\eta > 0$ is known as the learning rate.

The error backpropagation is a technique to compute the gradient of an error function (∇E) with respect to the parameters of a ANN based on the chain rule of differentiation, allowing the definition of the derivative of a composite function in terms of the derivatives of its constituent functions. The process is divided into two: a forward pass to compute the activations of all hidden and output units and a backward pass to propagate the errors from the output units to the hidden units. The errors are defined as the derivatives of the error function with respect to the summed inputs of each unit. The result is a vector of partial derivatives that can be used to update the weights using a gradient descent optimization algorithm [21].

The selection of an optimization algorithm is not a straightforward task. There is no consensus among researchers about what the best algorithm is in each situation. Currently, the

most popular optimization algorithms actively in use include Stochastic Gradient Descent (SGD), SGD with momentum, Root Mean Squared Propagation (RMSProp), RMSProp with momentum, adaptative learning rate method AdaDelta, and adaptative momentum estimation method Adam. The choice of the most appropriate algorithm highly depends on the user's experience [28].

Moving forward, there is a natural thumb rule to choose the output unit activation function of any ANN and a matching error function according to the type of problem to be solved. For the multiclass classification scenario (of concern in this study), it is typical to use the Categorical Cross Entropy (CCE) loss function and a SoftMax activation function in the last layer of the ANN [21], [29]. CCE can be defined as

$$\mathrm{E}(y, \hat{y}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} w_c \times \left[y_{n,c} \ln\left(\hat{y}_{n,c}\right)\right], \tag{2.26}$$

where C is the number of classes in the classification problem, and w_c is an optional weight attributed to each class, depending on the distribution of the classes of the training data. When these weights are not defined, they default to one. The CCE can be computed at the logits level (the actual classes, encoded as integer values) or the probabilities level, where $\hat{y}$ is an array of probabilities, and each class corresponds to each array position. Equation (2.26) applies to the latter case. A relevant case of the CCE function is the Binary Cross Entropy (BCE) function, which is used for binary classification problems and can be obtained straightforwardly based on (2.26).

2.4. Feature extraction methods

Feature extraction methods are fundamental in NLP since text data must be converted into something an ML algorithm can interpret. Many techniques have been developed to find an efficient and consistent way to represent text in numerical form mathematically. The standard ML pipeline for SA includes text cleaning or text preprocessing, tokenization, and vectorization, which are explained in this section. It is essential to have a dataset, typically referred to as a corpus. A corpus is a collection of N authentic documents, d, organized into a dataset that can be used to train ML algorithms. Each document is formed by sentences, words, and characters that constitute units of information for each document.

As mentioned, text data can often present unstructured sentiments when extracted from reviews, forums, or blogs. Therefore, it is expected to find phenomena like ambiguity, sarcasm, or irony expressed by emojis, punctuation marks, and slang. Depending on the downstream task, these elements can be helpful or introduce redundancy during the classification process, so a feature selection process must be conducted. Text preprocessing, or text cleaning, is a common step that deals with this and is typically defined empirically, being the most common tasks are blank space removal, punctuation removal, stopwords removal, stemming or lemmatization, and standardization of the characters. The text analysis can be done in terms of single words (unigrams) up to n consecutive words (n-grams), where normally combined approaches are shown to be helpful. This process limits the vocabulary size of the model and improves the generalization of word occurrences by identifying the most relevant features from the training corpus [8], [18].

The feature extraction process is mainly characterized by two elements: an embedding matrix that maximizes the interpretability and similarity between different symbols and a functional composition model that reduces the embedding representation, typically employing a dimensionality reduction method, recurrent neural networks, or attention mechanisms [30].

2.4.1. Fixed-vector representation

The two approaches for text representation are symbolic and distributed representation. Symbolic representations are a natural approach where sounds are transformed into letters or ideograms, and their combination can form words that humans can interpret. On the other hand, distributed representations are transformations applied to the symbolic ones to put them in metric spaces where characteristics and similarities among examples can be learned by ANNs or classical ML models, with the trade-off of reduced human interpretability [30]. The extraction of valuable information from text data can be done in several ways, but the most common ones are:

- **Local distributed representation, or one-hot encoding:** converts symbols into a binary vector that a ML algorithm can interpret, providing a simple encoding in a metric space. Given a set of S symbols, one-hot encoding maps the *i*-th symbol in S to the i-th base unit vector e_i in $\mathbb{R}^n$, where n is the cardinality of S [30].
- **Bag of words (BoW):** simple, flexible, and easy-to-implement approach describing word occurrence within a document. The term "bag" is due to any information about the order or structure of words in the document being discarded, so the documents are represented as a set of unrelated words. The most important aspects involving the usage of BoW are the definition of a vocabulary based on a known corpus and the definition of a score that measures the presence of known words: binary BoW, which records the presence or absence of words; Term Frequency (TF) that counts the existence of each word in a document and creates a vector of integers to represent each document; or Term Frequency – Inverse Document Frequency (TF-IDF) that is another counting method by considers the frequency of the symbols in the whole corpus provides a weighted scheme that measures the importance of any symbol in a given document [2], [8], [18], computed as

$$(TF - IDF)_{w,d} = tf_{w,d} \times \log\left(\frac{N}{df_w}\right) \tag{2.27}$$

  where $tf_{w,d}$ denotes the frequency of a word w in document d, N is the total number of documents in the corpus, and df_w is the number of documents in the corpus where the word w appears.

  The primary challenge associated with BoW approaches lies in their dimensionality and sparsity. As the corpus's vocabulary expands, the model's dimensionality also increases, directly affecting the input representation's sparsity, i.e., the input matrix contains numerous zeros that do not convey meaningful information but consume significant memory and computational resources. Word embedding techniques were proposed as an innovative approach for creating distributed representations of text, generating dense, low-dimensional vectors that efficiently encode the semantic and syntactic properties of words to tackle the shortcomings of BoW representations [2]. Word embeddings are needed in most NLP models to pass the input features. Word embeddings are vectors of continuous real values that transform a high-dimensional sparse vector space (one-hot encoding space) into a lower-dimensional dense vector space where each dimension of the embedding vector represents a latent feature of a word [2]. As a result, they efficiently provide more information. The most widely used methods to train word embedding are:

- **Word2Vec:** is a computationally efficient neural network prediction model that learns word embeddings from a corpus, developed by Mikolov et al. [31]. It is a two-layer neural network (an input layer, a fully connected hidden layer, and an output layer) that

takes tokens as input to learn neural embeddings. The size of the input layer is equal to the vocabulary size, V, of the corpus, and each word is represented as a one-hot vector. The hidden layer corresponds to the dimension, D, of the output word vectors, which is a hyperparameter pre-defined by the user. This layer's weights constitute the embedding matrix of size $V \times D$. Higher D captures more word contextual/semantic information based on the local context defined by the words inside the specified window [31], [32] but results in a larger word representation matrix, which is more computationally expensive. The output layer performs the average of the input context word ($c(w_t)$, where w_t is the target word) vectors multiplied by the hidden layer weights, using a SoftMax function that estimates a probability distribution over all words in the vocabulary. For further details on neural networks, refer to section 2.3.

Figure 2.7 exemplifies the two main approaches of Word2Vec, Continuous Bag-of-Words (CBOW) and Skip-Gram (SG) models, which are shallow window-based approaches where a window size of N words is defined. The value of N defines the range of words to be included as the context of a root word.

1. **CBOW:** Given the 2×N surrounding words (N words to the right and N to the left of a target word in a document), the model aims to predict this target word by computing the conditional probability of the target word. This model is more effective for smaller datasets. The context vector is defined as the average of the 2×N words within the window, calculated as [32]:

$$c(w_t) = \frac{1}{|c(w_t)|}\left(\sum_{\boldsymbol{c_i} \in c(w_t)} \boldsymbol{c_i}\right)^{\mathsf{T}}, \tag{2.28}$$

   where $c(w_t)$ is the set of context words that surround the target and $\boldsymbol{c_i}$ are the individual word vectors within the window.

2. **SG:** Given the target word, predict the 2×N surrounding words by computing the exact opposite conditional probability. This model leads to better embeddings on larger datasets. The difference with respect to CBOW is how the context vector is defined, which in this case is given by [32]

$$c(w_t) = (\boldsymbol{c_i})^{\mathsf{T}}. \tag{2.29}$$

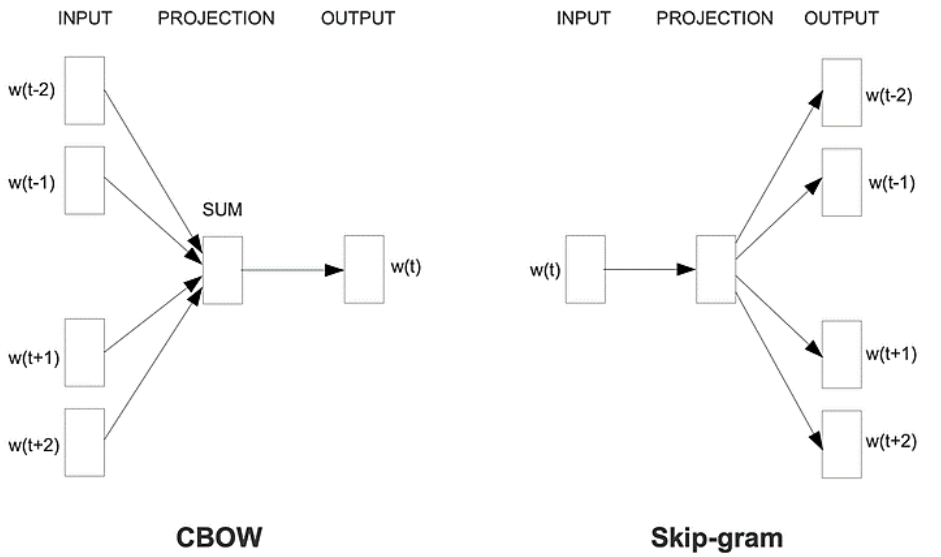


Figure 2.7 – Model's architecture of CBOW and SG. In this example, the window size is N=2 [26].

- **FastText:** is a variation of the SG model from Word2Vec, developed by Bojanowski et al. [18], that aims to learn word representations considering their morphological structure. This model introduces word and character level information to improve embeddings, where each word/token is treated as a collection of character n-grams, with each n-gram mapped to a dense vector. The word representation is then computed by summing these n-gram vectors [32]. This mechanism enables the generation of embeddings even for words not encountered during training. Unlike the Word2Vec model, where the scoring function s is defined as the dot product between word w_t and context vectors w_c, FastText defines it as the sum of the dot product between all n-gram vectors appearing in w_t and the context vector w_c, allowing to learn reliable representation for rare words [33].
- **Glove:** introduced by Pennington et al. [34], operates similarly to Word2Vec but includes matrix factorization of the word co-occurrence matrix to grasp inter-sentence (global) word statistics. This enables Glove to capture word relationships and meanings within a broader context, enhancing its ability to understand the nuances of language [32].

2.4.2. Contextualized word embeddings

One notable drawback of the discussed word embedding techniques occurs when a single word has multiple meanings depending on its context (polysemy), which cannot be solved by single vector representation. Another limitation is that these methods do not preserve the word order. Consequently, deep contextualized word embeddings were developed to tackle these issues. The most relevant algorithms in the literature are Embeddings from Language Models (ELMo) and BERT.

ELMO was introduced by Peters et al. [35]. It is a word representation method that considers the context and morphological structures of individual words at each state in text, allowing the same word to have different embeddings depending on their surrounding words. The model computes the word vectors using multiple BiLSTM layers with character-level embeddings at the input that capture the character n-gram features that lead to a more robust representation. These character embeddings are passed through a convolutional layer and a max-pool layer to get a fixed-

length representation of the entire word. Figure 2.8 illustrates the unwrapped architecture of the vanilla ELMo model, where the function F is given by [32]

$$F(t) = s_0 \times X_t + s_1 \times H_{1,t} + s_2 \times H_{2,t} \tag{2.30}$$

where s_i are the weights on the word and hidden representations from the language model. The output of F constitutes a matrix of size T×D (where T is the length of the input sentence and D is the dimensionality of the BiLSTM layers) containing the final embedding representation of each word w_t, noting that the embedding is computed as a linear combination of the character-level representation, the hidden state of the first BiLSTM layer, and the hidden state of the last BiLSTM layer. A scaling factor γ can be included in this formula to allow ELMo to be fine-tuned on a downstream task, such as SA. Peters et al. [35] argue that the two-layer architecture allows to extract syntactic information from the lower layer and semantic information from the higher layer.

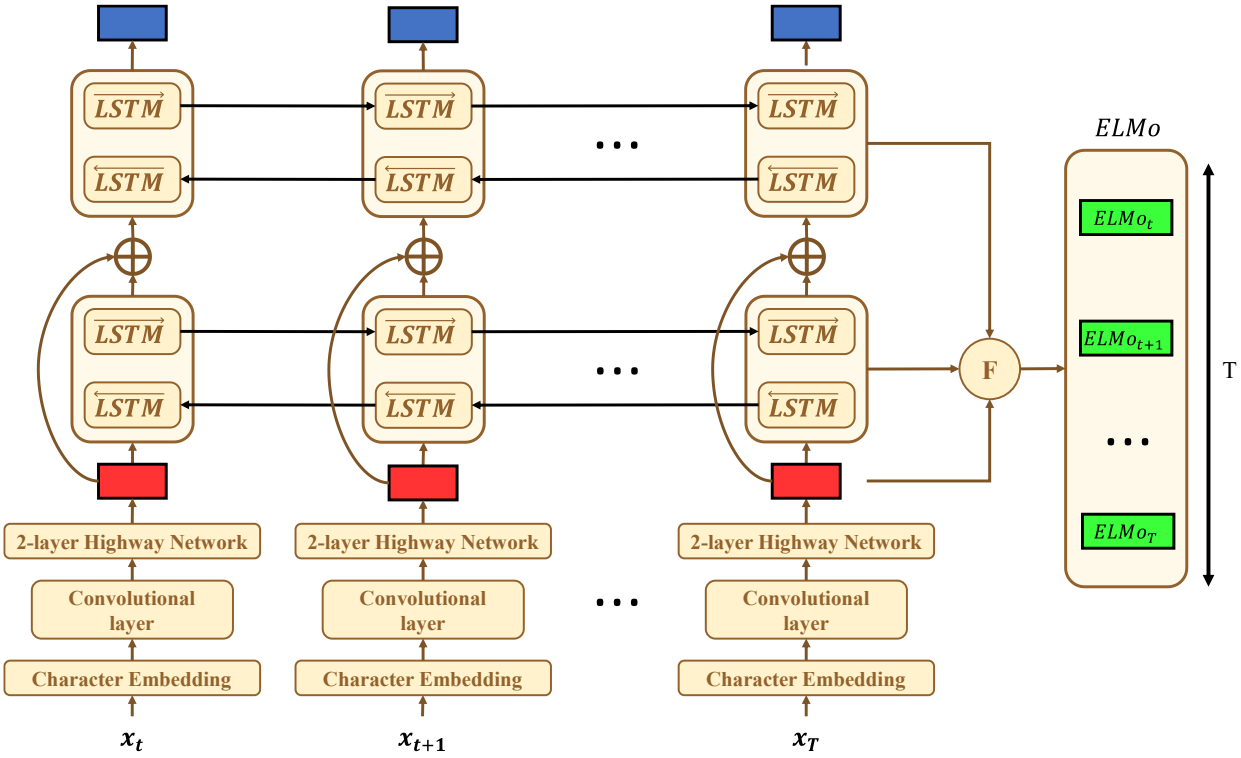


Figure 2.8 – Unwrapped representation of ELMo's Vanilla architecture, considering an input length of T words/tokens.

BERT is a contextualized neural embedding model that learns word embeddings by leveraging the contextual relationships between words (or sub-words) in text, utilizing the architecture of the transformer encoder proposed by Vaswani et al. [9]. The input layer transforms the input sequence of words into input embeddings formed by a token embedding term, a sequence embedding term, and a position embedding term. Then, these first embeddings are processed by the transformer encoder, which applies multi-head self-attention and residual connection followed by normalization layers to explore the optimization space and stabilize the learning process, respectively. The encoder block is replicated 12 times and stacked together; the multi-head uses 12 heads. The output is a set of extracted feature embeddings, which are then used as features for predicting according to the training task: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP) [36]. Figure 2.9 represents the architecture of the BERT model, where F is a linear combination of any of the thirteen output embeddings (input embedding and 12 encoder outputs), such as the sum of all encoder outputs, the sum or concatenation of the last four outputs, or only the last encoder output [37].

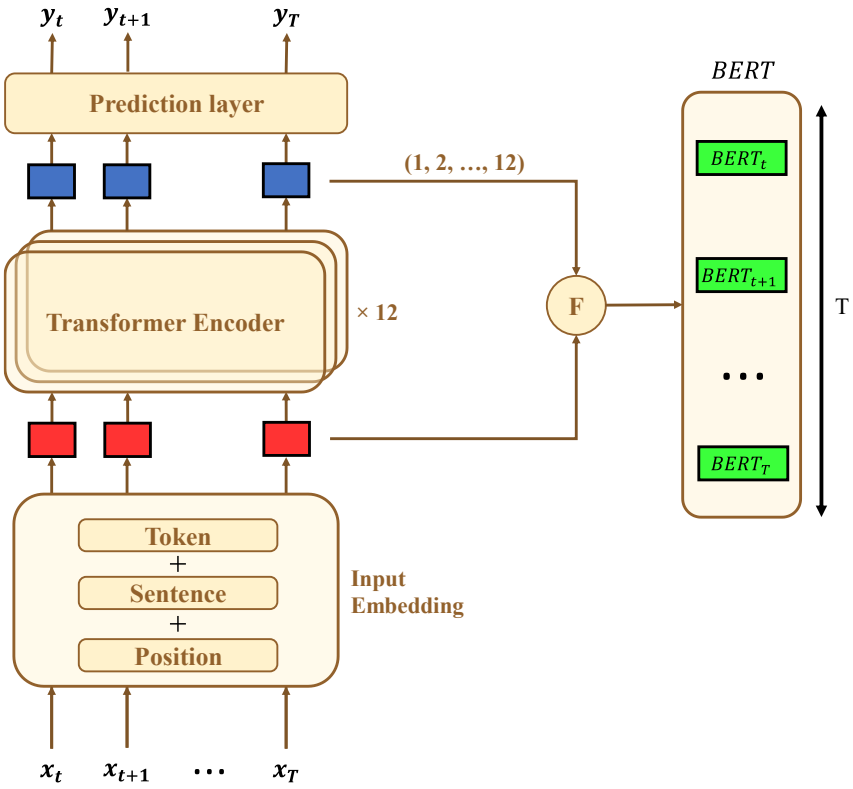


Figure 2.9 – Representation of BERT architecture, considering an input length of T words/tokens.

Both ELMo and BERT are categorized as language models that can be pre-trained in a semi-supervised way. This means they can train their layers on extensive corpora without relying on labeled datasets. These models revolutionized the NLP field, providing resources to perform transfer learning and fine-tuning with relative ease. Both models significantly outperform traditional word embeddings like Word2Vec and GloVe, capturing nuances in word meaning and sentence context [32]. Through pre-training, ELMo and BERT can more accurately represent polysemous words in various contexts and are more informative about the text's higher-level semantics [8]. Additionally, BERT can reduce locality bias due to the semi-supervised pre-training on MLM and NSP tasks [32], [36].

To summarize, word-level embeddings such as Word2Vec and GloVe are known for their quick training and ease of use compared to FastText, ELMo, and BERT. However, they have limitations; they do not consider morphological details, struggle with unseen words (unlike FastText), and do not handle the polysemous nature of words as effectively as ELMo and BERT. On the other hand, ELMo and BERT excel in capturing these nuances but come with the drawback of being computationally intensive and time-consuming during training.

2.5. State of the art

The field of NLP has been popularized lately with the use of DL models that are able to map relationships between words, phrases, and ideas. The most common applications are machine translation, text generation, and text classification, along with NLP tasks such as Named-Entity Recognition (NER), POS-tagging, and Question Answering (QA). This state-of-the-art study aims to discuss the latest advances regarding text representation and SA, explore the pros and cons of using each approach, and explore solutions made from scratch that have been competitive with state-of-the-art models.

The most relevant surveys in the literature regarding broad NLP applications were published by Young et al.[3] in 2018, Ferrone and Zanzotto [30] in 2020, Otter et al. [6] in 2021, and Khurana et al. [4] in 2022. They compared various DL models and provided a walk-through of their evolution, overviewing their architectures and describing current state-of-the-art trends and challenges.

Young et al. [3] claimed to be the first paper to comprehensively cover the most popular methods in NLP. They described the concept of distributed representation and popular models such as convolutional, recurrent, and recursive neural networks and provided a brief review of SA research. They found that SA was mainly evaluated on the Stanford Sentiment TreeBank (for the English language only) using BiLSTM over vanilla RNN layers and CNN layers with max-pooling operations [3]. Otter et al. [6] provided a brief overview of the most relevant techniques in NLP tasks, highlighting that SA is becoming increasingly popular in using DL techniques, being the current state-of-the-art model an ensemble model that includes CNNs and LSTMs. Khurana et al. [4], in conformity with Otter et al. [6], stated that ANNs are preferred by researchers in most NLP tasks, with the primary task being the text representation in a vectorized dense form. LSTM and BiLSTM-based models are the latest approaches preferred over CNNs.

Additionally, the development of pre-trained models such as BERT, a TLM architecture, by Devlin et al. [36] allowed fine-tuning for tasks such as SA or QA using Transfer Learning techniques [4]. A limitation of BERT is that it was pre-trained primarily on English datasets. Some researchers have made efforts to train this model using large datasets from other languages, emphasizing the Portuguese variations (as concerned by this study) BERTimbau [38] and Albertina [39]. The lack of studies in languages different than English was also highlighted in Khurana et al.'s paper and is proposed as future work.

Ferrone and Zanzotto [30] provided a link between symbolic representations and distributed representations with the aim of providing the basis to delve into interpreting how symbols are represented inside neural networks. They offer two relevant definitions: human-interpretable representation, where each dimension has a specific meaning, and decodable representation, where even if it is not directly interpretable, the representation can be decoded into an interpretable, symbolic representation.

Current state-of-the-art models lack interpretability in most cases, as they are built over distributional and composite representations [30]. Otter et al. [6] mentioned that lately, NLP is primarily a data-driven field using statistical and probabilistic computations along with ML, with neural DL models being more consistently used over classical approaches such as Naïve Bayes, Support Vector Machines, or Hidden Markov models. Nevertheless, they are still commonly used as benchmarks against newly proposed DL approaches. Recently, Wankhade et al. [8] presented a survey dedicated to SA applications and challenges, studying the pros and cons of classical and DL techniques, revealing that DL models are preferred due to the reduced need for feature engineering and scalability for domains different from the training dataset, with the drawback of being computationally costly, needing long training sessions, and, in many times, requiring huge datasets. The most relevant architectures in recent studies are constituted by LSTMs, BiLSTMs, CNNs, and TLMs, which benefit from high accuracy by mapping long-term dependencies and focusing on the most relevant terms of the input sequence.

Many researchers have built implementations considering the DL components discussed above, showing proficiency for their particular scenario. A relatively old example is the article from Zhou et al. [40] that developed two models, BiLSTM-2DPooling and BiLSTM2DCNN, and tested them on six text classification tasks, revealing that the latter model, a combination of

BiLSTM, 2D convolutional filters, and 2D max-pooling operation, achieving excellent performance on 4 out of 6 tasks (two were SA tasks). Later, Rehman et al. [41] proposed a Hybrid CNN-LSTM model composed of 20 CNN layers and an LSTM decoder that outperformed classical ML approaches, CNN, and LSTM models individually when evaluated on the same dataset.

With the advent of TLM-based models [9], researchers started integrating attention mechanisms to improve NLP performance. Xia et al. [42] proposed a Self-AT-LSTM model with character-based input embeddings utilizing a CNN with max-pooling to solve the Out-of-Vocabulary (OOV) problem, two BiLSTM layers, a Multi-head Self-Attention layer as used by Vaswani et al. [9], and a decoder constituted by an LSTM layer and a MLP. They empirically demonstrated that attention models can identify hierarchical information between words that are too far away from each other, either when working in cooperation with RNNs or CNNs for feature extraction. In the same year, Wu et al. [43] utilized a full attention model composed of a Multi-head Self-Attention encoder followed by a local attention layer that takes only the output from the last Self-Attention layer and a MLP classifier. They proposed a Layer-wise Attention Tracing (LAT) method to trace the attention 'paid' to input tokens. They demonstrated that the accumulated attention scores tend to favor words with greater semantic meaning, which appeals to SA.

2.6. Key remarks

Researchers have been more interested lately in the SA task, with the development of LLMs such as BERT that can be fine-tuned. Parallelly, many researchers have developed models from scratch using CNNs, LSTMs, attention mechanisms, or hybrid approaches, attaining state-of-the-art results.

The most relevant challenges encountered in SA are the text representation and level of classification. While many techniques exist that can convey enough information about the tokens, such as BoW, word embeddings, and contextualized embeddings, specific datasets may contain nuances that are harder to fulfill using some of these techniques. The preferred approach is the usage of word and contextualized embeddings, typically using LSTM or BiLSTM neural models over TLM-based models due to the relative ease of training. Generally, large amounts of data are needed to train TLM-based models, which is the main reason to work with RNNs. Regarding the classification level, most studies have focused on the document level as it is the simplest approach. However, researchers state that document and sentence levels do not provide the necessary detail needed to mine opinions on all aspects of the entity, which is desirable in many applications.

Researchers have noticed a lack of studies on languages other than English, which represents a relevant research opportunity that has already been handled in some studies. Some studies have proposed model architectures that can extract enough information from text reviews. The studies presented in the state of the art were English-based models. Section 3.3 discusses developments in the Portuguese language (a concern in this study), which highly vary in terms of adopted solutions, as it is a less developed language in the NLP area.

3. Case study: Zomato

The development of this work was motivated by the Restaurant Review Sentiment Output (RRSO) project in partnership with Zomato Portugal, a food service enterprise that provides an application that allows its clients (restaurants) to manage reservations, takeaway orders, payments, and centralized access to customer reviews. On the other hand, customers can utilize the application to provide ratings and comments as reviews.

The data collected by the company since its inception has been made available for the accomplishment of this project, which is a crucial step in facilitating the implementation of models. Data collection is one of the most important and challenging steps in the field of ML. In this chapter, an in-depth analysis is conducted on the provided dataset to extract relevant information crucial for implementing models for text representation and SA. Additionally, an examination of studies in the restaurant area and applications on Portuguese SA was conducted to understand the most relevant aspect to consider when building the models.

3.1. Dataset

Initially, the data provided consisted of two separate CSV files that contained information about restaurants and consumer reviews. These files comprise approximately one million entries. However, nearly half of these entries lacked any text reviews and were excluded from this study.

With the help of Zomato's technical team and RRSO project team, the dataset was structured using MongoDB, a NoSQL database where information is stored as files instead of tables. The dataset consists of a total of 536,833 entries gathered from April 1, 2014, to September 2, 2022, and has five attributes:

- *review_id*: internal reference of the review in Zomato's system.
- *res_id*: internal reference of a restaurant registered in Zomato's system.
- *res_name*: name of the restaurant.
- *raw_text*: informal expressions of consumers' opinions about their restaurant experiences, where about 513k were written in Portuguese and 23k were written in English. The latter set was translated to Portuguese using the Google Translate Application Programming Interface (API), considering that this method is not 100% effective. In some cases, consumers may write their reviews in both languages.
- *rating*: represents the consumer's assessment of the restaurant in terms of five stars (1.0 to 5.0 in increments of 0.5).

Before further analysis, the dataset was divided into three subsets, following the standard practice of the ML pipeline: 70% of the data was used for training, while the remaining 30% was used for testing. Additionally, the training set was divided into training and validation sets, the last one being used to early assess an unbiased evaluation of the model and tune its hyperparameters without leaking information from the test dataset [29], ending with the partitions depicted in Table 3.1. All data analysis and final deployment of the models were done considering these partitions to avoid any bias. Intermediate developments, such as model optimization and evaluation, were performed using Cross-Validation (CV) techniques. Therefore, these partitions were not considered. These partitions were stratified based on the column *rating* to guarantee that every partition maintained the same distribution.

Table 3.1 – Dataset split into training, testing, and validation partitions.

	Train set	Test set	Validation set
Number of samples (in thousands)	319.4	161.0	56.3
Percentage	60.0	30.0	10.0

3.2. Exploratory Data Analysis

Before utilizing the available data in an ML model, it is essential to undergo a systematic procedure to comprehend the information contained within the data or identify how new features can be extracted to enhance models' performance by providing preprocessed data.

An Exploratory Data Analysis (EDA) was conducted in two stages. Firstly, the *raw_text* column, which constitutes the corpus of Zomato's dataset, was analyzed. Secondly, the *rating* column was examined to gain insights into the distribution of ratings. This column is a viable candidate for consideration as a label for our text, enabling supervised learning. The rest of the columns were eventually used to assist in handling the dataset and were not directly used to train any model.

3.2.1. Corpus analysis

Zomato's dataset has a large corpus of reviews provided by consumers without any length restriction, so it was expected to find documents with a wide range of lengths. The visualization of textual data is normally a challenging task due to the enormous number of words found in a corpus, including misspelled words. A first glance at the corpus allowed us to identify formatting problems regarding the collection and storage of text data. These issues were subsequently corrected:

- **Special HTML characters**: in some reviews, accents and special characters in words like "Já" or "emoção" were represented as "Já" and "emoção", ultimately losing their meaning. A hard-coded table was used to reverse this format.
- **End of line special symbol**: in some reviews, the string "$%" appears between two phrases of the same review. An empty character replaced it.
- **User ID tag**: in some reviews, a tag indicating the user ID appears in the format "{uid:123456789}". The regular expression pattern "{uid:[0-9]+}" was used to catch this event.

From the training dataset, it was possible to extract 20.5 million words, forming a vocabulary of 135,193 unique words, excluding punctuation. A word is accounted as any sequence of characters delimitated by blank spaces. This is a relatively extensive dictionary to work with. However, approximately 44% of these words appear only once in the entire corpus and can be disregarded as they will not have sufficient opportunities to be trained by any ML model. In the same way, even words that appear a total of $n > 1$ times can be disregarded. For instance, $n = 5$ corresponds to 70.5% of the words that appear in the corpus, and their removal results in a new vocabulary of 39,899 unique words. Figure 3.1 shows how the distribution of words has an exponential tendency, where the top frequent words are "e", "de", "a", "que", "o", and more, which do not convey any significant information for sentiment analysis. These words are defined as stopwords and are typically eliminated for most NLP tasks.

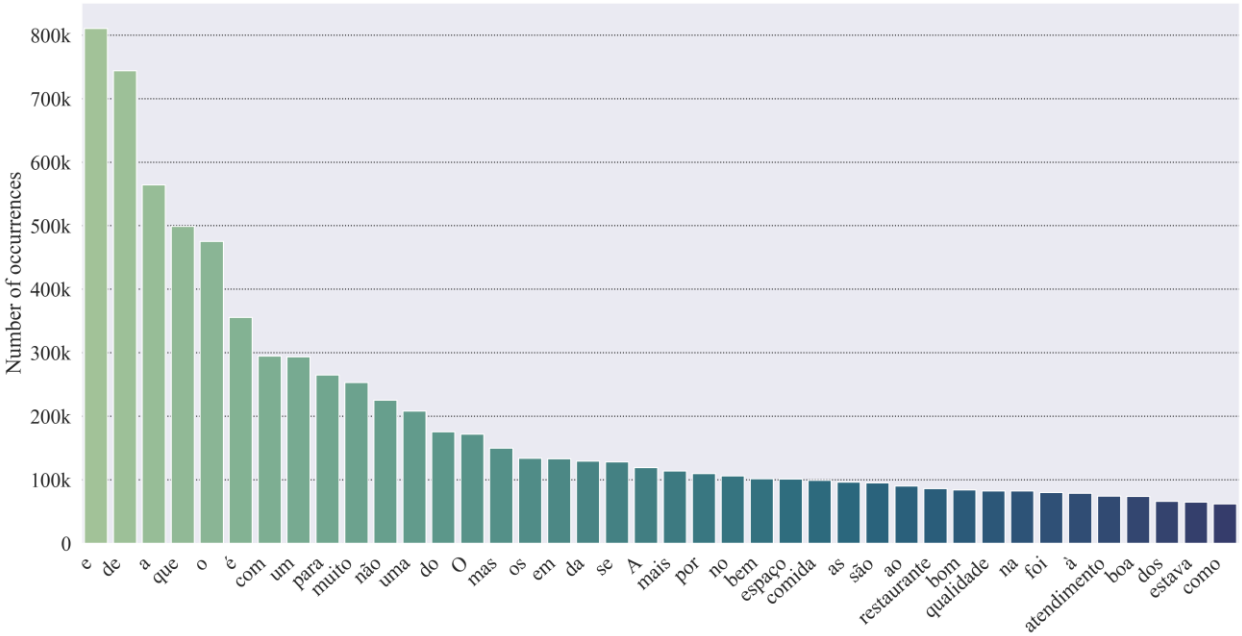


Figure 3.1 – Words distribution (top 40 most frequent words, mainly stopwords).

As there are 20.5 million words in the training dataset, one can analyze the statistics of the number of words per review to better understand the length of the documents in the corpus. Figure 3.2 exemplifies the possible shapes of data distribution. Two cases were considered, one using all the words extracted from the corpus and another using the same corpus but with stopwords being removed (consult some reviews examples in Appendix A):

1) From the original corpus, the average length of the reviews is 64 words and a median of 46 words. This suggests a skewness towards longer reviews (negative skew, Figure 3.2), with a tail value of 160 words at a 95% confidence interval.
2) From the reduced corpus, the average length of the reviews is 36 words and a median of 26 words. Again, this suggests a skewness towards longer reviews (negative skew, Figure 3.2), with a tail value of 85 words at a 95% confidence interval.

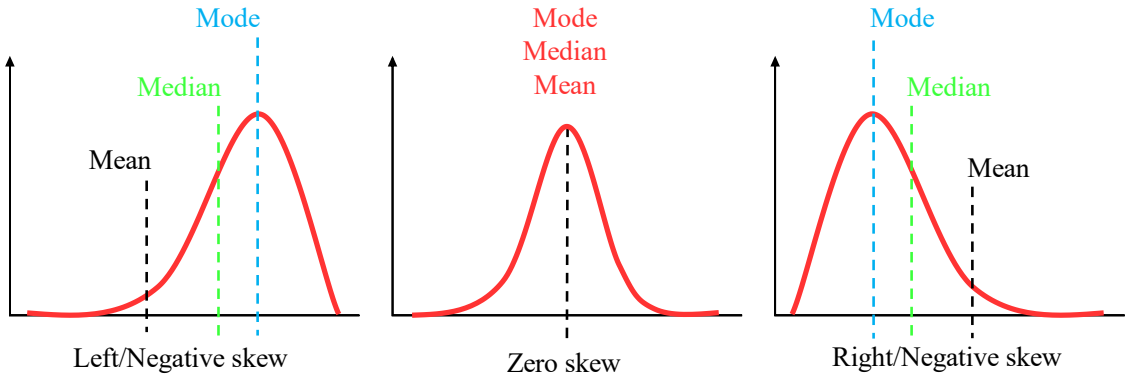


Figure 3.2 – Representation of the skewness of a data distribution based on the mode, median, and mean values.

The confidence interval was computed by fitting the data distribution to a generalized extreme value (GEV) distribution. The reduced corpus (without stopwords) was mainly considered to reduce the models' complexity and their generalizability capacity on unseen data from the same domain.

3.2.2. Restaurant ratings

Star ratings have been used extensively by former researchers to perform text classification. The rating distribution of the training set is depicted in Figure 3.3, where it is noticeable that most text reviews are labeled with more than 2.5 stars. The imbalance in the rating is evident, revealing a highly biased distribution of data toward a positive experience. It can be noticed that half-star ratings are less commonly given by consumers and maintain the same increasing behavior as full-star ratings.

As was typically done in previous work [15], [44], [45], these ratings were merged to perform polarity classification using three classes. The nine original classes were equally divided into three groups: 1.0 to 2.0 was assigned to the negative class (label 0), 2.5 to 3.5 was assigned to the mixed class (label 1), and 4.0 to 5.0 was assigned to the positive class. The intermediate class was named mixed rather than neutral due to the origin of the data since these cases show a combination of positive and negative sentiments. Figure 3.3 also shows the new distribution of the data.

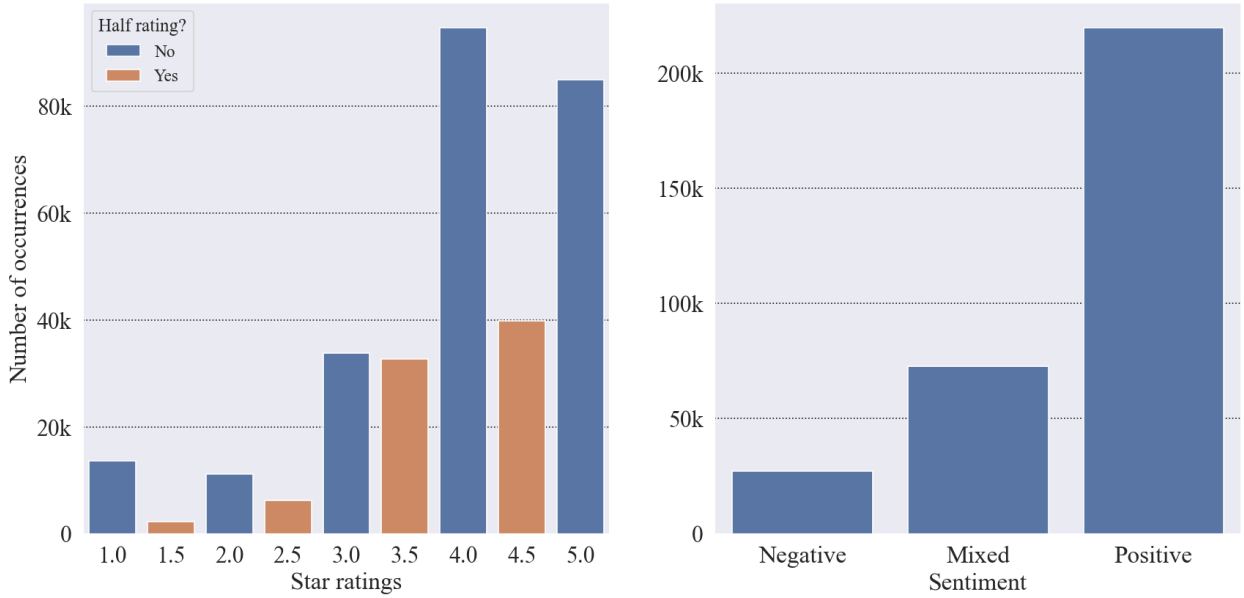


Figure 3.3 – Ratings distribution: (a) original star ratings (1.0 to 5.0); and (b) three-class merged ratings.

The ratio between the majority (4 stars) and minority (1.5 stars) groups is significantly large, and the practical difference between, for instance, reviews with 1 and 1.5 stars is hard to tell. That is the reason why splitting the ratings into three classes is appealing. Table 3.2 shows the percentage of reviews in each class and the ratio concerning the majority class (positive label).

Table 3.2 – Distribution of the ratings and ratios against the majority class.

	Negative	**Mixed**	**Positive**
Percentage	8.47%	22.76%	68.77%
Ratio to the majority class	1:8	1:3	1:1

3.3. Related work

The intended task is to develop solutions for NLP problems related to SA using Portuguese datasets. While there have been advancements in this area, it is essential to explore whether any prior research has specifically addressed this problem in restaurant reviews. This investigation is valuable as it allows comparisons and the utilization of previous findings to enhance the

performance of the proposed models in this study. The following subsections show the recent developments in this area.

3.3.1. Recent developments in Portuguese language

Despite Portuguese being among the top 5 most spoken languages in the world, with approximately 171 million internet users (representing 3.7% of all users) [46], limited research has been conducted on this language. Nevertheless, in recent years, there has been an increased number of publications addressing NLP problems in the Portuguese language, primarily focusing on Brazilian Portuguese and, to a lesser extent, European Portuguese. The main approaches focus on using multi-lingual models or translating Portuguese text into English to use tools developed for this language, often producing more favorable models [16]. The lack of linguistic resources is the main reason for this, leading to most studies focusing on ensuring the availability of resources, including dictionaries and databases. On the other hand, to establish baselines for future developments, most studies rely on classical ML classifiers, such as NB, SVM, LR, DT, and Random Forest (RF) classifiers.

Many authors have put their efforts into enhancing the linguistic resources of the Portuguese language. Souza and Filho [45] reunited five known Brazilian datasets and analyzed them in terms of size, number of n-grams, and length of reviews. They concluded that these Portuguese datasets are less rich in vocabulary than English datasets. Therefore, complex algorithms may not provide the same performance gain as faced with English data. Also, Brum and Nunes [47] introduced the Brazilian dataset TweetSentBR, which was manually labeled as positive, negative, and neutral by seven native speakers of Brazilian Portuguese; and Gomes et al. [48] introduced the dataset BRTweetSentCorpus manually labeled as positive, negative, ambiguous, and non-opinionated. This process is very time-consuming and is not reasonable on larger datasets. Many studies are based on consumer reviews along with a numerical value indicating a rating evaluation between 1 and 5 stars. This information is readily available on websites and can be used to surpass the problem of manual labeling. Some works [44], [45] considered a partition into three classes described such as: a rating of 1 to 2 stars is assigned to negative sentiment, a rating of 3 stars is assigned to neutral (and sometimes disregarded for binary classification), and 4 to 5 stars is assigned to positive. Moreover, dos Santos and Ladeira [44] presented an empirical foundation for this heuristic, applying a t-student test to verify the comparability of the star rating with a manual classification performed by a human. The result indicated that the star rating is equivalent to manual classification, allowing reliable usage for training an ML algorithm.

The central studies that introduce and characterize the landscape and developments on SA for the Portuguese language were driven by Pereira [16] and Souza et al. [49]. The former did a systematic mapping review up to 2014 and discovered that almost all language resources available are from Portugal and Brazil variations, almost 70% from Brazilian Portuguese. The studies that were encountered mainly investigated sentiment classification at the document level and used LR, NB, and SVM models with at least one type of text preprocessing task. The latter made a survey of SA in Portuguese, which states the need for new proposals of models for the Portuguese language to tackle the language specificities better. The survey analyzes the whole spectrum of solutions, including lexicon-based approaches, classical ML approaches, and DL approaches. The typical implementation can be divided into three steps: text preprocessing, text representation or feature extraction, and development and evaluation of an SA model.

Several approaches have been employed to perform text preprocessing. De Oliveira and Merschmann [50] conducted an in-depth analysis to understand the relationship between a

classifier and the text preprocessing steps considered. They considered POS-tagging, stemming, lowercase conversion, and n-grams extraction, and tested all possible combinations, concluding that there is no combination of preprocessing tasks that is always the best regardless of the classifier used, so an empirical evaluation must be conducted when selecting the text preprocessing steps.

A similar but simplest analysis was done by dos Santos and Ladeira [44], performing text standardization (removal of accents, special characters, punctuation, and numbers), stemming, stopwords removal, n-grams extraction, and spelling corrections, suggesting that text preprocessing techniques do not result in significant improvement of classification performance on their domain-specific dataset, although only four preprocessing scenarios were considered. They compared results using SVM, ANN, and NB classifiers without further characterizing the models. In some cases, preprocessing tasks are highly dependent on the specific scenario, such as Brum and Nunes [47], in which text preprocessing consists of replacing numbers, names, and links with unique identifiers, POS-tagging, and the usage of a character repetition trimming mechanism as an alternative to spell correctors on twitter texts. Classical ML models were used to assess performance, but no comparisons were made to understand the effect of preprocessing. Souza and Filho [45] organized and analyzed a set of known Portuguese datasets, applying a predefined sequence of text preprocessing to facilitate the analysis of former researchers, including text standardization but maintaining numerical characters. Again, this work does not include an analysis showing the performance improvement when employing text preprocessing techniques.

Text representation is a fundamental step that enables automatic feature selection and feature engineering. In this context, researchers have mainly used two approaches: BoW and contextualized word embeddings. Souza and Filho [45] compared Word2Vec, GloVe, and FastText vectors trained from the NILC word embeddings repository [51] on a downstream SA task and found that the embedding technique and its dimensionality have a profound impact on the model performance. On their evaluation, FastText, with a vector size of 300, outperformed on every dataset. They also evaluated the usage of a TF-IDF scheme for text representation, reporting a superior performance than word vectors. However, they argued that the poor aggregation method of word embedding (average of all word vectors) might have led to information loss. Their analysis also highlights the importance of tuning the vocabulary size, showing that a range from 5000 to 10000 n-grams typically allows a stable performance.

Later, Souza & Filho [52] did a similar analysis considering FastText-300 as the BoW approach (motivated by their previous work) and LSTM, CNN, and TLM models for embedding generation, suggesting that fine-tuned TLM models with BERTimbau weights provide the best classification performance, slightly better than LSTM and CNN. On the other hand, using BoW provides the worst performance, but it is still attractive due to its implementation simplicity. Other authors have not addressed the representation problem specifically, as their SA solutions typically rely on using BoW techniques. Brum and Nunes [47] and Cardoso et al. [53] carried out experiments using binary BoW (presence/absence), while dos Santos and Ladeira [44] and de Oliveira and Merschmann [50] employed the TF-IDF representation. Their primary motivation when selecting the representation technique was, again, the simplicity of its implementation.

The authors mentioned above represent the main developments in recent years on Portuguese SA and have presented solutions to this problem considering different approaches for the predictive model. Pereira [16] found that the main supervised ML techniques are NB classifiers, while DL approaches are mainly based on CNNs. Classical ML approaches such as NB, LR, and SVM have been predominantly utilized as classifiers, while recent works incorporate DL models

like CNN, LSTM, and TLM. One promising approach was the one presented by Cardoso et al. [53], which uses an ensemble to combine DT, NB, SVM, and LR to attain a performance boost in binary SA, with average values of 82% on both F1-score and accuracy after ten folds. Souza and Filho [45] evaluated their embeddings using LR, RF, and Light-GBM binary classification models, adopting the ROC-AUC as the classification metric, where Light-GBM showed the best performance when adopting word vector representation (average of 91.4% among the five datasets), and LR when adopting TF-IDF (average of 94.8% among the datasets). In a second paper, Souza and Filho [52] used an LR model to assess the classification performance when using embedding generated by fine-tuned BERTimbau, attaining an average of 97.3% of ROC-AUC among the same five datasets.

Britto et al. [54] did a similar analysis to Souza and Filho, exploring other classical ML approaches such as LR, NB, RF, and SVM and comparing them to BERTimbau. Their conclusions suggest that the BERT model outperforms in every data domain, but the imbalance of the classes produces a detriment on minority classes, which is easily seen by the F1-score metric. The best performance registered for three-class classification (polarity with neutral class) was 74%. Brum & Nunes [47] also tackled the three-class scenario, attaining around 59.7% of F1-score when using NB and SVM classifiers. Although the work developed by de Oliveira and Merschmann [50] does not aim to improve the state of the art, from their analysis, it can be understood that SVM usually performs better than MLP, regardless of the preprocessing procedure, obtaining an average F1-score of 77.3% among three datasets labeled with positive, negative, and neutral. They even highlight the importance of considering the SVM classifier in the SA task using Portuguese written texts. It is worth referring to one of the works that used DL techniques. Adán-Coello and Neto [55] optimized a CNN binary classification model using a grid search algorithm and attained an average performance of 82% of accuracy in five folds.

Typically, previous researchers' datasets used for SA show a highly skewed distribution of labels. The same is verified for the dataset provided by Zomato in this project (Figure 3.3). The imbalanced dataset problem in NLP tasks has been addressed by Abonizio et al. [56] via a comparative analysis where four data augmentation mechanisms were used, discovering that the contribution of the Back Translation (BT) augmentation method is remarkable for all classifiers, and the Pretrained Data Augmentor (PREDATOR) method demonstrates comparable effectiveness in imbalanced scenarios. They also compared the usefulness of each augmentation method on different classifiers and verified that every SA model benefited from data augmentation (either DL or classical ML), with SVM the least benefited. This work also suggests that, in some SA cases, the selected classification model has the most significant impact, disregarding the augmentation method used. There have been other attempts of BT for languages of groups that are more remote from English, as investigated by Amjad et al. [57], which were not successful due to the machine translation quality. None of the research described in this subsection indicated the usage of class weight techniques, which is an attractive approach since it does not directly manipulate the training data.

3.3.2. Applications in the restaurant industry

It is difficult to find extended work on Zomato’s dataset in the literature as the data is not directly openly available. In general, EDAs are conducted over large Zomato datasets but without the firm presence of text data. The analysis typically includes the volume of reviews, restaurant and cuisine types, and prices.

Wibawa et al. [58] did an EDA on Zomato's dataset scraped from their website, which was limited to data from India and Indonesia, so syntactical and grammatical rules are very different from the Portuguese language. However, a good number of reviews are written in English. They base their analysis on a big data approach, extracting insights regarding the most common types of restaurants and cuisine in Indonesia and the relationship between rating and the average meal cost, finding a positive correlation between the cost of eating and the rating given. A similar analysis was done by Gupta et al. [59]. Moreover, SA was introduced using dictionary-based methods, concluding that there is a linear relationship between the rating provided by the user and the score calculated by subtracting the number of negative words from the number of positive words within a review. In the work of Jagdale and Deshmukh [60], the "Zomato Restaurants Data" dataset [61] was used to perform an EDA and build sentiment analysis models on non-textual data, therefore excluding NLP techniques.

Only one Portuguese dataset of the restaurant reviews domain was found [62], with a reduced size and aimed at aspect-based SA. Still, the Brazilian Portuguese datasets organized by Souza and Filho [45] can be used to perform cross-domain validation to assess the model's generalization ability.

3.4. Methodology

The RRSO project intended to develop an automatic algorithm able to extract the sentiments from a collection of reviews associated with a restaurant to provide a comprehensive overview of the overall perception of the service the consumers experienced without needing to read through all the individual reviews. A DL approach was preferred for SA, as emphasized in section 2.5, using several NLP tools for text preprocessing and representation.

Several tests were addressed at different levels of the SA problem to attain a reliable model, following the workflow depicted in Figure 3.4:

1) The data processing was already partially introduced in section 3.2 and encompasses the definition of the labels for classification tasks and the text representation techniques based on the EDA of Zomato's corpus. The text was cleaned before being used in further steps and included several processes applied sequentially to the input text. The sequence of preprocessing steps was selected empirically after a joint evaluation of the preprocessing techniques based on the performance of the downstream SA classifier. Two text representation approaches, motivated by the previous works, were also presented. The first is TF-IDF, a widely used technique due to its simplicity, and the second is an Embedding Layer that serves as an interface to convert the integer-represented input into a dense vector representation. During the classifier's training, the parameters of this layer are learned, with the main concern being the initialization of its parameters, which can be addressed by employing random initializers or pre-trained vector representations for each input token. The latter method was evaluated using two algorithms, namely FastText and Word2Vec.
2) As the dataset is skewed towards a negative sentiment (ratings are mostly above 2.5 stars), two data balancing techniques were investigated empirically to find the best possible solution for this scenario: a paraphrasing technique using BT combined with under-sampling of the majority class and cost-sensitive learning. This step incorporates the *k*-fold CV technique to ensure results are robust and not biased by the data partitions so that the original dataset is split into three: training, testing, and validation partition in a stratified mode, i.e., the original distribution of the classes is maintained on every

subset. The balancing procedure is strictly performed on the training set, avoiding any data leakage on validation and test sets.

3) The training phase is performed in two steps for each fold set. First, the vocabulary with a limited size is extracted from the training partition to define the tokens the model can recognize, and the token representation technique selected is applied to all data partitions to create the features that the model will consume. Finally, the SA model with a selected set of hyperparameters is trained using training and validation partitions. The models developed were based on RRNs and self-attention mechanisms, showing a typical architecture of encoder-decoder, which is the most suitable approach for text classification referred to in literature when high performance is the main objective. Specifically, the architecture is altered in a way to substitute the decoder with a classification layer.
4) Subsequently, the resulting checkpoint model is evaluated using the testing samples. In this stage, the relevant metrics are compared with a baseline model to find insights about how a model needs to be updated to result in a better outcome. This manual optimization procedure is further improved in this work by means of a genetic algorithm that aims to find the best set of hyperparameters that maximizes the model's performance and a grid search algorithm to optimize the preprocessing steps to run on the input sequence.
5) The final stage in this work involves deploying one of the most optimized SA models on an edge device, along with a simple application that captures a person's speech and displays the sentiment conveyed in the phrase based on its contents. The recorded phrase is converted into text before the analysis; thus, the performance is conditioned not only by the SA model but also by the speech-to-text tool used.

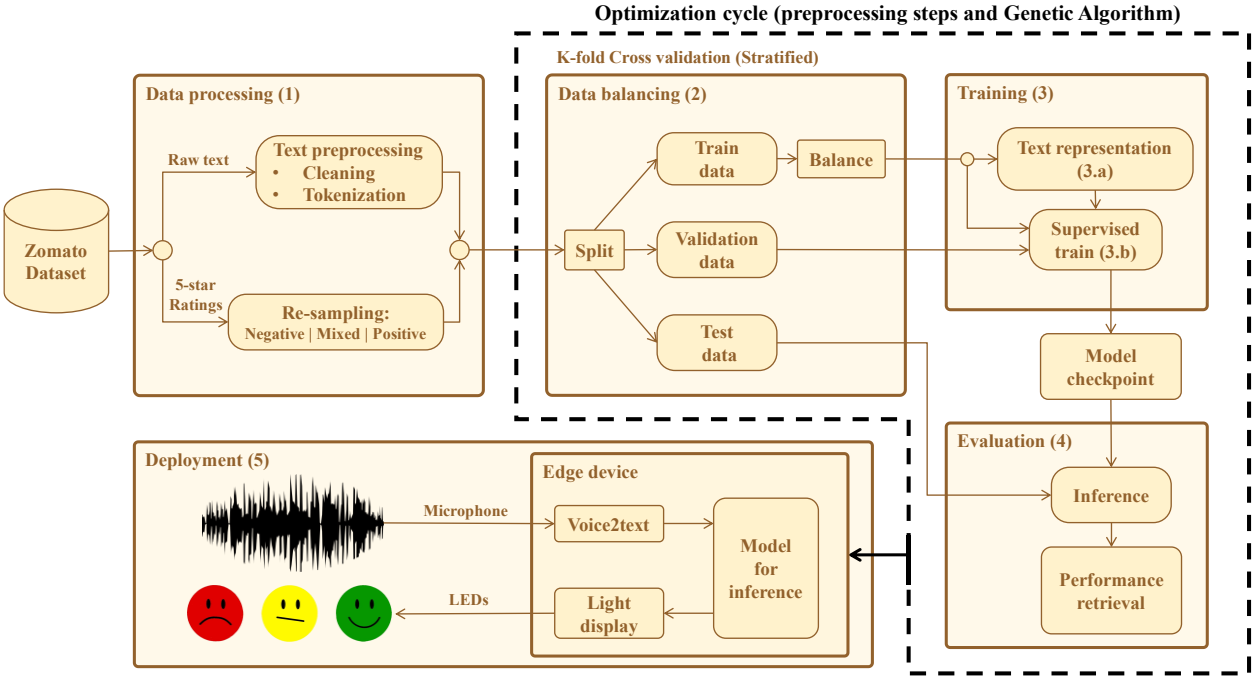


Figure 3.4 – Workflow proposed for developing DL models for SA and deployment on an edge device.

Added to the concepts presented in chapter 2, the workflow proposed obligates to define the specific methods and tools needed for the development of this work. The following subsections provide the definition of BT and cost-sensitive learning for class balancing along with the methodology proposed in this work, an introduction to tokenization techniques typically used by researchers and explored in this study, a description of how an encoder-decoder architecture is

suitable for text classification, and the definition of the CV techniques used during the evaluation of the models.

3.4.1. Handling imbalanced data

Class imbalance is a problem that commonly occurs in ML classification scenarios and refers to a significantly high difference between the number of samples among the classes. In other words, a skew in the data provokes the existence of a minority class. Most ML algorithms for classification assume that there is an equal number of examples for each observed class. Therefore, the effectiveness of the training process may be affected by any imbalance. The most typical situation is to have fewer samples in the class with more interest. Zomato's dataset presents a very imbalanced distribution of ratings, with a lack of negative samples with respect to the positive samples. As the aim of restaurant applications is to detect if the products and services can be changed to improve the overall sentiment of the consumers, identifying the negative class is of great relevance. When adopting the three-class scenario, the ratios shown in Table 3.2 indicate the majority class is eight times bigger than the minority class, which denotes a moderate degree of class imbalance [63]. In this study, two methods that could handle the imbalanced data problem are compared to understand what is more convenient for this Portuguese NLP scenario:

- **Cost-Sensitive Learning (CSL):** This is a subfield of ML that consists of developing models that might have uneven cost or weight for each output class during the training process. The aim of CSL is to minimize the penalty associated with a misclassification on the training dataset instead of minimizing the error or cost function. Several approaches can be taken to define these weight or penalty values, such as data resampling, modification of existing ML algorithms, and the usage of ensemble classification to relabel samples and minimize the cost of misclassification [64]. In this work, the approach followed is the modification of the cost or loss function used during the training of the ANNs. The modification consists of assigning fixed penalty weights to each class, allowing to change the impact the cost function has in the gradient based on the class that is being misclassified. The minority classes have a more significant impact on the gradient when misclassified, while the majority have less impact. Therefore, the parameters that benefit the learning of the majority class converge slowly [65]. These weight or penalty values can be easily integrated into the training process by using the parameter *class_weights* provided by the *Keras* fit function [66]. Class weights are typically computed based on false positive and false negative values on binary classification problems. As this work deals with a three-class scenario, a static approach was used, setting observation weights based on the number of samples of each class in the training dataset as

$$W = \frac{N}{C} \times [w_1 \quad \dots \quad w_c], \tag{3.1}$$

where N is the number of samples in the training data, and C is the number of classes (in this case is equal to 3). The weights w_c are then defined as the inverse class frequency given by

$$w_c = \frac{1}{\sum_{d \in \boldsymbol{D}} \text{is\_equal}(d, c)}, \tag{3.2}$$

where $\boldsymbol{D}$ is a vector that represents the labels of the training data. The denominator represents the number of samples that belong to the class c. This formula assigns higher weights to classes with fewer samples.

- **Back Translation (BT):** This is one of the most widespread methods for augmenting datasets for NLP tasks. In the taxonomy of text data augmentation methods proposed by Abonizio et al. [56], BT is a sentence-manipulation approach that provides rephrased variations from original data samples by translating an input sentence into an intermediate language and translating it back to the original language. By doing this, the process creates new samples with slight variations while maintaining the same overall meaning, thus, it is not necessary to assign new labels to the new samples. However, the effectiveness of BT is directly subjected to the performance of the translation mechanism used. In this work, the translation model used was the Google Neural Machine Translation (GNMT) system, introduced in Google Translate service in 2016 [67] through the deep-translator Python package [68]. To attain a more balanced dataset, the minority data classes were over-sampled using the new augmented samples, while the majority class was under-sampled to a comparable number of data points with the expectative of mitigating the skewed distribution of data to fall, at the very least, in the category of mild class imbalance.

As the usage of CSL was motivated by an imbalanced learning problem, the models trained with this technique were referred to as "CSL <model name>". Additionally, when referring to a model trained with augmented samples, it is mentioned as "BTOS <model name>" after BT over-sampled. Finally, when referring to a model trained without any balancing method, it is referred to as "CIL <model name>" after Cost-Insensitive Learning (CIL).

3.4.2. Tokenization techniques

Text tokenization plays a crucial role in NLP to break down text data into smaller pieces of information that can be later transformed into numerical representations. A token is a sequence of characters representing a linguistically significant and methodologically useful unit of information in the text. Tokens can be words, subwords, punctuation marks, numbers, symbols, or even emojis. The choice of a tokenization method is closely related to the text cleaning process since it can significantly reduce the number of tokens known by the model, i.e., its vocabulary, to make it more memory-efficient while increasing the number of OOV tokens. Many tokenization techniques can be used to split text data into smaller pieces of information. This work evaluates the effectiveness of two of them by addressing the SA model performance: Blankspace tokenizer and Wordpiece tokenizer.

The blankspace tokenizer is a simple yet effective technique that operates by splitting the input text into individual words based on the presence of whitespace characters. It treats any sequence of consecutive non-whitespace characters as a single token and provides rule-based mechanisms to better separate tokens, such as identifying punctuation marks to separate them from words or identifying hyphenated words to be treated as one. In particular, the implementation for this study was based only on the presence of whitespace characters, inheriting properties from the prior text cleaning process.

The tokenizer is built over the training data, extracting a fixed vocabulary to be used to encode the input and defining a fixed size for the tokenized representation. Figure 3.5 illustrates the operation of the Blankspace tokenizer with two examples, noting that special tokens may also be introduced to tackle OOV tokens ('[UNK]') and to pad the sequence of tokens to a fixed size ('[PAD]'). This tokenizer was implemented using the TextVectorization layer from *Keras* [66]. The maximum length of the tokenized representation was decided based on the analysis of subsection 3.2.1, setting the value to 100 to guarantee a margin. Finally, the vocabulary size was

defined during the training process since it is directly related to the training data of the word embedding algorithm.

'Restaurante com atendimento simpático, num espaço agradável.'	Restaurante com atendimento simpático , num espaço agradável [PAD] [PAD]
'A comida é típica portuguesa.'	A comida é [UNK] portuguesa . [PAD] [PAD] [PAD] [PAD]

Figure 3.5 – Operation of blankspace tokenizer with two examples.

The wordpiece tokenizer is a sub-word approach initially developed for Japanese/Korean tokenization and was successfully adopted by GNMT [67] and BERT [36]. It is a data-driven approach that learns how to break down words into chunks based on statistical patterns in the training data, with the aim of encountering "wordpieces" that can minimize the number of OOV words. These smaller pieces of information can be used as parts of larger words, effectively handling a significant portion of unknown words. Figure 3.6 illustrates the operation of the Wordpiece tokenizer with two examples, noting again that special tokens can be introduced, including '[UNK]', but less frequently. As noticed in the examples, Wordpiece is a mid-term solution between character and word-based approaches, providing flexibility and efficiency [67].

This tokenizer was implemented using the BertWordPieceTokenizer model from Hugging Face [69] for learning the tokens and WordPieceTokenizer from *Keras NLP* [70] for the model's usage. Again, the maximum length was set to 100 tokens, and the vocabulary size is defined during the training process.

'Restaurante com atendimento simpático, num espaço agradável.'	Restaur ##ante com atend ##i ##mento simpático , num espaço agrad ##ável
'A comida é típica portuguesa.'	A com ##ida é típic ##a portug ##uesa . [PAD] [PAD] [PAD]

Figure 3.6 – Operation of wordpiece tokenizer with two examples.

Both methods perform a direct transformation into integer representation, where each token is converted into a numerical index that can be transformed back to string tokens using a Python dictionary. Inspired by BERT, the WordPieceTokenizer was only tested when attention mechanisms were used. In this implementation, both tokenization techniques were used to define a vocabulary of words determined by the Word2Vec implementation of the *Gensim* Python library.

3.4.3. Encoder-Decoder architecture

The encoder-decoder architecture is the most researched area in the era of modern NLP. It was initially proposed for solving sequence-to-sequence tasks where the input and the output were both sequences that could be of the same or different lengths. This architecture consists of two parts: an encoder and a decoder. A RNN is used as the encoder to map the input sequence into a fixed-length hidden representation, denominated context vector, that summarizes the most relevant and minimal information about the input sequence. Simultaneously, a RNN is used as the decoder trained to generate a sequence that is conditioned by this context vector. In practical terms, the last hidden state of the encoder is referred to as the context, and the decoder is initialized with this

vector rather than a vector of zeros, as it is typically done [25]. The applications of this architecture in NLP usually involve machine translation, speech recognition, and QA.

A representation of the operating principle of the encoder-decoder is presented in Figure 3.7, where $X = (x_1, x_2, \dots, x_{T_x})$ is the input sequence, with length T_x, C is the fixed-size representation of the input with dimension D, and $Y = (y_1, y_2, \dots, y_{T_y})$ is the output sequence, with length T_y. Notice that the input of the encoder is first passed through the Embedding Layer, which converts word indexes into a fixed-size vector with dimension F.

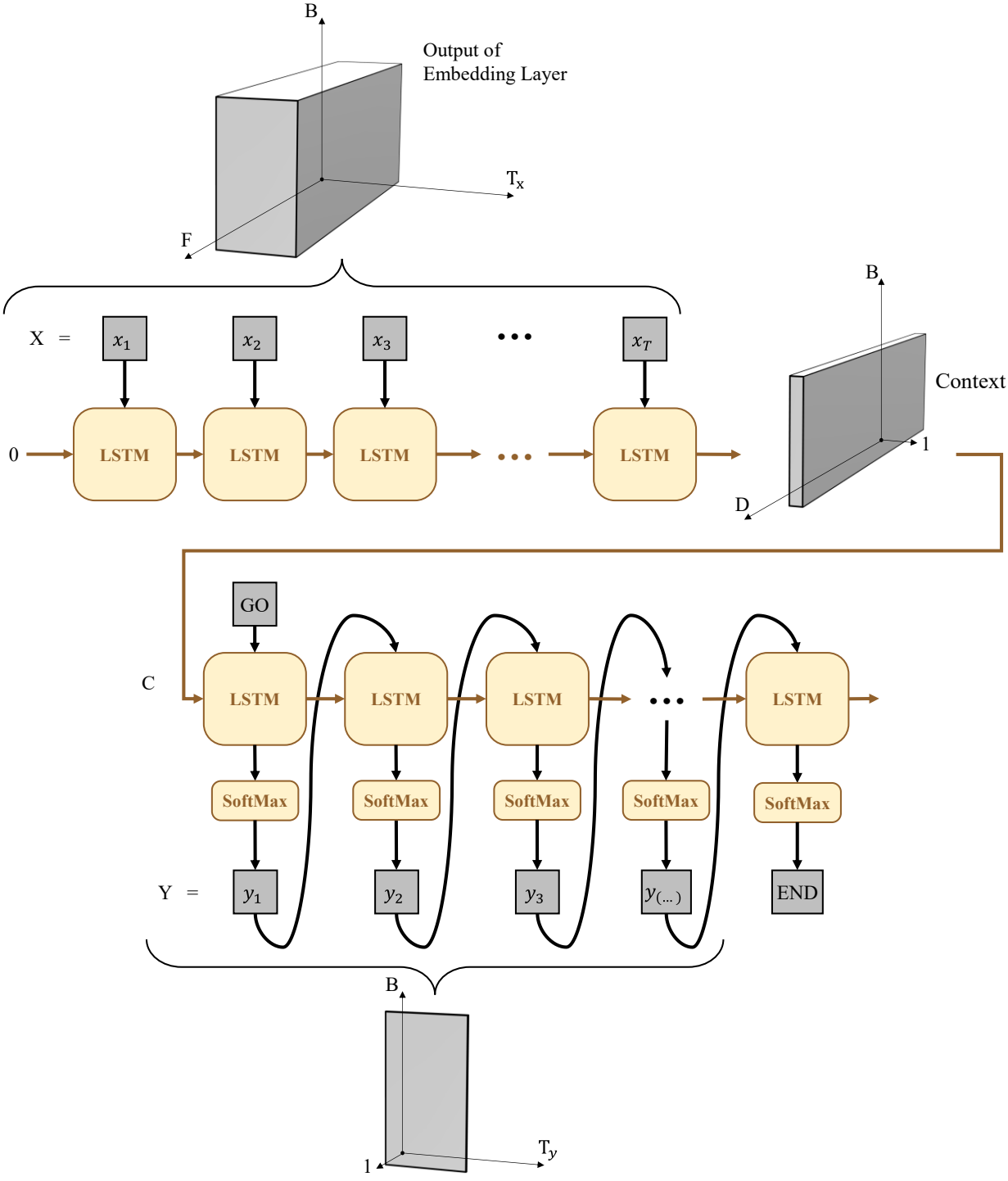


Figure 3.7 – Working principle of the encoder-decoder architecture.

This architecture can be further modified to handle other kinds of problems, such as text classification. As referred to, theoretically, the context vector already carries all the information conveyed by the input sequence, with the limitation that the dimensionality of this vector might not be large enough to represent the whole input, making this hyperparameter relevant for fine-tuning. Moreover, the decoder is not relevant for classification tasks and can be replaced by a classification layer that processes the context to produce the output of the model [25].

Based on the head of the encoder-decoder architecture, this study proposes the implementation of two solutions for the Portuguese SA problem: a Recurrent Encoder Classifier (REC) that uses only recurrent mechanisms to encounter and summarize contextual information from the input sequence of tokens that represents a restaurant review, and an Attentive Recurrent Encoder Classifier (AREC), that combines recurrence and attention to refine the contextual information and additionally, explores other configurations for the classification layer. These models were subjected to the two stages of optimization depicted in Figure 3.8, and the performance at the end of each stage was evaluated to understand the gain of improvement.

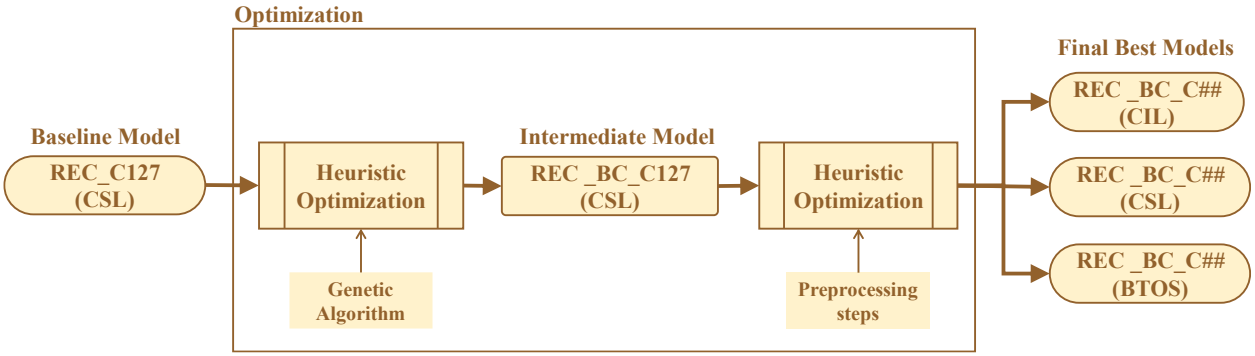


Figure 3.8 – Process of optimization of the model's architecture from baseline model to three final models.

Initial tests showed that the baseline model trained under a CIL approach became biased toward the majority class (positive sentiment). Therefore, the reasoning for performing all the optimization processes under a CSL approach was to make sure the model was not overtraining the majority class.

3.4.4. Evaluation and Cross-Validation

To effectively evaluate the performance of the developed DL models, it was necessary to define a set of evaluation metrics that are most suitable for the given data scenario. It was imperative to select metrics that can robustly characterize imbalance classification problems.

After evaluating the model with the test dataset, results can be represented using a confusion matrix (or contingency table), which is a square matrix that summarizes the experimental performance of a classifier. The statistical measures for each class were estimated using the One-vs-Rest (OVR) approach, which reduces the multi-class scenario into multiple binary scenarios, building the three confusion matrices illustrated in Figure 3.9 (note that there is only one confusion matrix and three interpretations of its values). The components of an OVR three-class confusion matrix are:

- **True Positive (TP):** the number of samples that were correctly classified. In this case, the assigned or predicted sentiment matches the actual sentiment.
- **True Negative (TN):** the number of samples that correctly verify the absence of a given sentiment. Note that this component is not able to identify the misclassifications among the 'rest' of the sentiments.

- **False Positive (FP):** the number of samples that were incorrectly assigned to a given sentiment. This component cannot distinguish among the labels assigned to the 'rest' of the sentiments.
- **False Negative (FN):** the number of samples that are labeled with a given sentiment but were misclassified with another. Again, this component cannot distinguish among the labels assigned to the 'rest' of the sentiments.

Actual \ Predictions	0	1	2
0	TP	FN	FN
1	FP	TN	TN
2	FP	TN	TN

Negative vs (Mixed & Positive)

Actual \ Predictions	0	1	2
0	TN	FP	TN
1	FN	TP	FN
2	TN	FP	TN

Mixed vs (Negative & Positive)

Actual \ Predictions	0	1	2
0	TN	TN	FP
1	TN	TN	FP
2	FN	FN	TP

Positive vs (Negative & Mixed)

Figure 3.9 – Confusion matrices resulted from the OVR analysis approach. Zero is assigned to Negative, one to Mixed, and two to Positive sentiments.

The following measures were extracted from the confusion matrices to evaluate the imbalance classifiers, computed using the PyCM Python library [71], which specializes in multi-class confusion matrices. To avoid any dependency on the training data, the metrics were computed as the macro average across the three classes. The definitions were based on Ballabio et al. [72].

- **Sensitivity**, Recall, or True Positive Rate (TPR) measures the proportion of positives that were correctly identified as such and represents the ability of the classifier to correctly identify the sample's class. In terms of the confusion matrix, the sensitivity can be expressed as

$$TPR = \frac{1}{C}\sum_{c=1}^{C}\frac{TP_c}{TP_c + FN_c}, \tag{3.3}$$

where C is the number of classes and, therefore, the summation among the classes provides the macro average. The same applies to the remaining metrics.

- **Precision**, or Positive Predictive Value (PPV), is the proportion of positive assignments that correspond to the actual presence of the condition and reflects the classifier's ability to avoid wrong predictions. Precision can be computed as

$$PPV = \frac{1}{C}\sum_{c=1}^{C}\frac{TP_c}{TP_c + FP_c}. \tag{3.4}$$

Precision and Recall are inversely proportional to each other; therefore, it is impossible to increase both Precision and Recall at the same time.

- **F1-score** is the harmonic average of the precision and recall. It is a well-suited measure of accuracy when dealing with imbalanced datasets, generally used to manage the trade-off between recall and precision. This measure can be computed in terms of the metrics mentioned earlier or in terms of the confusion matrix as

$$F1 = \frac{1}{C}\sum_{c=1}^{C}\frac{2TP_c}{2TP_c + FP_c + FN_c}. \tag{3.5}$$

- **Specificity**, or True Negative Rate (TNR), measures the proportion of negatives that are correctly identified as such and represents the ability of a classifier to reject samples of other classes. Specificity can be computed as

$$TNR = \frac{1}{C}\sum_{c=1}^{C}\frac{TN_c}{TN_c + FP_c}. \tag{3.6}$$

- **Area under the ROC curve:** Receiver Operating Characteristics (ROC) curves are graphical tools typically used for binary classification that represent the relationship between the sensitivity of the positive class (TPR) and the specificity of the positive class ($1 - TNR$). Figure 3.10 illustrates an ROC curve, where the top left corner represents the ideal classifier with maximum sensitivity and specificity, and the diagonal line indicates the performance of a random classifier. The ROC curve is insensitive to the classes' distribution.

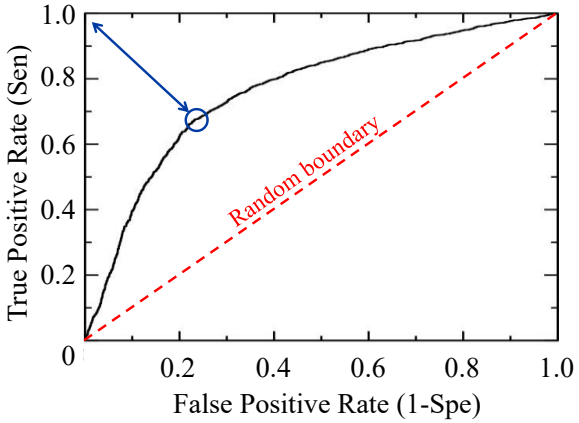


Figure 3.10 – ROC curve representation and readability.

To simplify the graphical analysis, it is common to calculate the area under the ROC curve (AUC), which measures the ability of the model to avoid errors during classification and is equivalent to the probability that the classifier will rank a randomly chosen agreement higher than a randomly chosen error [73]. PyCM computes the ROC-AUC with the approximation:

$$AUC \approx \frac{1}{C}\sum_{c=1}^{C}\frac{TNR_c + TPR_c}{2}. \tag{3.7}$$

- **Accuracy**, or overall success rate, is the most widely used metric in classification problems and corresponds to the ratio of correct classifications done over the total number of samples without accounting for any information on the classification performance of single classes. This metric is not computed as a macro average but as

$$ACC_o = \frac{1}{N}\sum_{c=1}^{C} TP_c\,, \tag{3.8}$$

where C is the number of classes, TP_c are the true positive predictions for the class c, and N is the number of instances tested. This metric is not the preferred one in imbalanced scenarios because it is highly susceptible to being biased by the majority class.

These metrics can be used to retrieve information about the model being developed, but it is also relevant to have a reliable methodology to extract this information and get a generalized evaluation of the model's performance, looking to minimize the dependency on a single-held-out test dataset. To address this, the CV technique can be employed.

CV is a statistical method that provides a more stable evaluation of a model's generalization performance compared to using only a held-out test dataset. It consists of repeatedly dividing the dataset and training multiple models with each partition, which may end with slightly different performances. The most common approach is *k*-fold CV, where *k* is a user-specified number that defines the number of non-overlapping partitions the dataset is split [74]. As illustrated in Figure 3.11, two *k* values were used during this work. A 2-fold CV scheme was used during the optimization procedures referred to in Figure 3.8 as they are considerably time-consuming, particularly the grid search, and a 5-fold CV scheme was used to retrieve the performance of the final optimized models (also referred to in Figure 3.8).

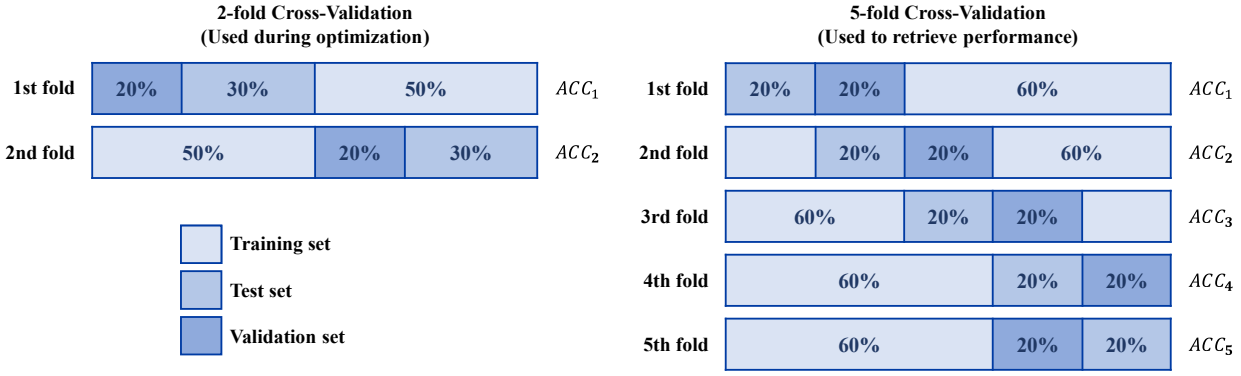


Figure 3.11 – Illustration of the 2-fold and 5-fold CV procedure and its usage in this work. Also includes the validation partition, extracted from the prior test partition in the 2-fold scenario and from the prior training partition in the 5-fold scenario.

To illustrate the process, consider the case of a 5-fold CV. First, the dataset is partitioned into five parts of approximately equal size, named folds. Then, the last $k-1$ folds are used for training, and the first fold is used for testing. This process is repeated iteratively until the network has been tested with each of the folds. At the end of each iteration, the performance is retrieved using the metrics mentioned above. The stored metrics from each fold are used to compute the average performance and its standard deviation to attain a robust evaluation of the model.

In addition to the CV evaluation, the partitions referred to in Table 3.1 were used to train and evaluate the models to be deployed. This includes the model used for the low-cost hardware inference application (see chapter 7) and the model delivered to the RRSO project.

3.5. Key remarks

Related works have shown that European Portuguese datasets are not readily available. Thus, the RRSO project presents a unique opportunity to train a DL model using this valuable data. This dataset can capture nuances specific to the Portuguese language in the European region, which may differ from those in other regions. Furthermore, data containing consumer opinions, specifically from restaurants, are often scarce and frequently limited to the English language. No other studies were found to use a dataset from Zomato to implement any NLP task.

Previous studies revealed a preference for BoW algorithms for text representation due to their simplicity of implementation. The primary focus in these works is typically on improving the classification performance. However, limited attention is given to fine-tuning the hyperparameters

of the models, often relying on heuristics instead. Additionally, the predominant trend in SA is to rely on lexicon-based and classical ML algorithms, potentially due to limited access to large datasets. DL approaches have primarily been focused on the English language, and there is still a scarcity of works in other languages. While recurrent and convolutional networks are commonly employed for English data, their exploration in Portuguese is relatively limited.

The lack of DL models being developed for Portuguese SA scenarios motivates the implementation of these models from scratch based on RNN and attention mechanisms. To the extent of the author's knowledge, there have been no implementations in the literature of Portuguese SA models based on self-attention and recurrent mechanisms built from scratch, which could potentially yield superior results for specific domains rather than using pre-trained models such as BERTimbau. The encoder-decoder architecture is the most used configuration when dealing with NLP, and therefore, that same path was followed to attain high performance in Portuguese SA.

Despite the efforts of some authors, achieving a fair comparison between studies remains challenging. The metrics commonly employed include accuracy, F1-score, and ROC-AUC, computed as the average or weighted average based on the number of samples per class in the training dataset. Additionally, validation techniques vary from cross-domain validation to 5-fold or 10-fold CV. Based on this, the present work aims to provide several metrics to ensure the independence of the dataset and easy comparison with future works. It is worth noting that AUC-ROC is particularly relevant to assessing models' performance, given the observed class imbalance.

4. Development of DL models for text classification

This chapter provides a comprehensive explanation of the processes undertaken to build the SA models. The challenges and obstacles encountered are explicitly outlined, along with a detailed rationale for the decisions made. These decisions might be grounded on previous works, heuristics, or empirical comparisons conducted on the specific problem.

Within this chapter, emphasis is given to the development of two SA models: one featuring only recurrent mechanisms to address contextual information and capture long-term dependencies and the other featuring a combination of recurrent and self-attention mechanisms to exploit the excellent capabilities shown by state-of-the-art models such as BERT and possibly enhance the performance of previous research in the Portuguese language. Concepts of text preprocessing and data balancing were of extreme importance for the development of robust models. Therefore, separate sections are presented to address these challenges. Each subsequent section deals with each step in Figure 3.4. Before implementing the DL models, a baseline model using classical ML techniques was trained on Zomato's data to fairly compare the performance gains.

4.1. Data balancing

The data balancing process was carried out after the conversion of the ratings into sentiment tags, working on the distribution depicted in Figure 3.3 (b) to address the potential issue of data imbalance. This section presents the implementation of the two balancing techniques proposed for solving this issue, looking for a more balanced model. A CIL scenario was also evaluated to work as a baseline model to compare the gain in performance of the data balancing techniques.

The CSL solution was done by combining (3.1) and (3.2), where $\boldsymbol{D}$ is an array containing only the labels of the training set to avoid any information leakage from the other sets. The weights were defined as a Python dictionary and were considered during training through the *class_weights* parameter of the fit *Keras* method, which applies these weights during the computation of the cost function. As the CV process was done in a stratified manner, the resultant weights are primarily invariant. The array of weights obtained was $W = [3.9338,\ 1.4648,\ 0.4847]$.

The BTOS solution required the elaboration of an algorithm to perform the augmentation of the training data. The first step involved the creation of the synthetic text samples: the entire dataset was divided into 17 chunks with a maximum of 10k samples each, and the samples labeled as positive were filtered out (those above 3.5 stars) to end up only with the samples from the minority classes, which are intended to be augmented. Each chunk was stored as a CSV file, making sure to maintain the *review_id* column to be able to index the correct data according to the partitions generated during the CV evaluation. Table 4.1 provides details on the specifications of each chunk and the time required to perform the BT process. The intermediate language was randomly selected from a heuristically predefined set of languages. As shown in the table, the BT was a time-consuming process, which explains why it was performed prior to the actual augmentation algorithm.

Table 4.1 – Specifications of all chunks of the dataset used to synthesize new text data.

File number	Intermediate language	Ratings	Time
0	Spanish	1.0	7h30
1	French	1.0, 1.5, 2.0	7h50
2	Italian	2.0, 2.5	9h25
3	Russian	2.5, 3.0	6h38
4	English	3.0	5h10
5	French	3.0	7h32
6	Russian	3.0, 3.5	5h40
7	German	3.5	4h58
8	English	3.5	5h32
9	Italian	3.5	7h08
10	English	1.0, 1.5, 2.0	4h57
11	German	2.0, 2.5, 3.0	7h00
12	Dutch	3.0	7h18
13	French	3.0, 3.5	5h13
14	Italian	3.5	6h13
15	German	1.0, 1.5, 2.0, 2.5, 3.0	7h11
16	Spanish	3.0, 3.5	5h13
Total time			4 days, 15h30

Once all required synthetic data were generated, these new samples were used to augment the training dataset. Considering $C = \{c_1, c_2, c_3\}$ the set of classes corresponding to negative, mixed, and positive, and R the set of reviews contained in the dataset, a hybrid-sampling process was performed to turn the moderate class imbalance into a mild class imbalance. Algorithm 1 describes the process followed to perform the data augmentation.

Algorithm 1 Hybrid-sampling (BTOS + Under-sampling)

Input: *trainData* [‘*review_id*’,‘*reviews’*,‘*rating’*]

Internal constants: *augBT* [‘*review_id*’,‘*reviews’*,‘*rating’*]

Output: $trainData_{Aug}$ [‘*review_id*’,‘*reviews’*,‘*rating’*]

1: Over-sampling minority classes (Negative and Mixed)

1.1. Select the augmented samples based on *review_id*

$augBT_{Filtered}$ ← *augBT*.query(‘*All review_id present in trainData*’)

1.2. *aux* ← concatenate(*trainData*, $augBT_{Filtered}$)

2: Under-sampling majority class (Positive)

2.1. Select samples to keep in *trainDataAug*

minority ← *aux*.query(‘*rating < 2*’)

N ← length(*aux*[‘*rating = 1*’])

Majority ← *aux*.query(‘*rating = 2*’).random_sample(*N*)

2.2. Join selected samples into one table

$trainData_{Aug}$ ← concatenate(*minority, Majority*)

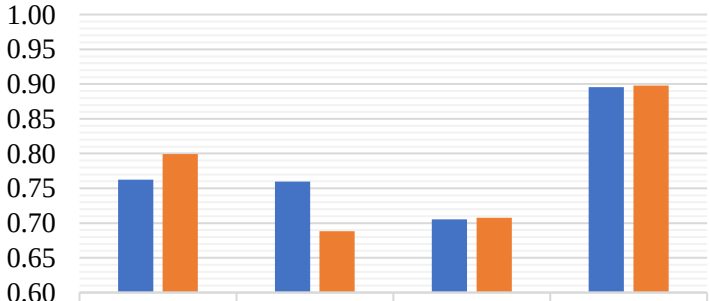
Note that the BTOS process by itself led to a minority/majority ratio of $1/4$, which already falls in the category of mild class imbalance. However, to become even more balanced, under-sampling the majority class (making it the same size as the middle class) allowed a ratio of $1/3$. These values are approximations since they may vary due to the randomness of the *k*-fold CV process.

4.2. Classical ML baseline

Before developing DL-based SA models, two widely used methods were employed to establish baseline models on Zomato's dataset. Although the literature presents various results that can be used for comparison purposes, using these models trained on our data allows a fair comparison. For this purpose, NB and SVM models were chosen, according to section 3.3, using the *Sci-kit Learn* implementations [75].

A multinomial NB classifier was built using the *Sci-kit Learn* Python library [76] using its default parametrization. The baseline NB model was not subject to CV due to a recurrent OOM exception during training that was not possible to overcome, so the results were assessed using the held-out test dataset by means of the partitions depicted in Table 3.1 and joining the train and validation sets since this model does not require the validation set during training. The text cleaning combination number 127 (default cleaning) was used to preprocess the input text, and a TF-IDF scheme was used to vectorize the reviews into words, with a maximum of 6,000 words set by heuristics. Additionally, an SVM model was built using *Sci-kit Learn* [76], considering a Radial Basis Function (RBF) kernel, a regularization parameter $C = 10$, a $\gamma = 1$, while the rest of hyperparameters were set to default [50].

Likewise, the baseline SVM was not subject to CV (due to the same reasons as before) and was trained and evaluated using the partitions of Table 3.1, merging the train and validation sets and applying the default cleaning. Vectorization was also addressed using the TF-IDF scheme. In this case, the number of train samples was limited to 10k due to the complexity of the input, which led to significant training times and kernel restarts. Figure 4.1 displays the principal metrics gathered from the evaluation of the two models on the test dataset, revealing that SVM produces higher values except for sensitivity, which, along with high accuracy, might suggest an overfitting bias to the positive class.



	Accuracy	Sensitivity	F1-score	ROC-AUC
Naïve Bayes	0.7626	0.7598	0.7054	0.8957
SVM	0.7992	0.6884	0.7077	0.8979

Figure 4.1 – Evaluation of the performance of classical ML algorithms to set a baseline.

The NB classifier was less sensitive to the unbalancing issue. In contrast, this problem was harder to tackle for the SVM classifier, typically boosting accuracy by misclassifying the minority classes, even with the usage of the class weights parameter. The models could have been tuned to

achieve a better performance, but that is not the scope of this study. The values of the Multinomial NB classifier registered in Figure 4.1 will be referred to as the ML baseline in the rest of this work.

4.3. Text representation

How text is transformed into a numerical representation has proved to be a critical concern when building an NLP model. This section defines the preprocessing steps and the methods adopted for performing tokenization and vectorization of the input text. All models developed were based on a token representation approach, where each token constitutes an input timestep. The techniques referred to in this section were jointly tested with the performance of the SA classifiers. Therefore, selecting the best text representation techniques is treated as a hyperparameter to be tuned.

4.3.1. Text cleaning

Text cleaning is the first step that can be considered when dealing with text representation, which allows for extracting more precise features that a DL network can learn from. It may result in a reduction of the number of tokens that the network can learn, leading to a reduction of the complexity of the model and a larger number of OOV tokens. Therefore, a compromise must be found to ensure good performance and a high level of generalizability. In this subsection, a set of text-cleaning steps is defined based on heuristics and the contributions of related works [44], [45], [47], [50]. A joint evaluation of the preprocessing steps was addressed considering the following list of tasks:

- **Convert to English alphabet:** Removes any kind of word accents, helping to reduce the vocabulary size but may also reduce the diversity of known words, which can be translated as a loss of information. For example, “à” is turned into “a” and “ç” is converted into “c”. The Python library *unidecode* [77] was used for this, also translating different characters, such as those encountered in Russian, Japanese, or Hindi, to standard characters.
- **Convert to lowercase:** Transforms characters from uppercase to lowercase. If characters are already in lowercase, it does nothing. This task is prevalent in literature and helps to reduce the vocabulary size but may hinder the intentionality of a phrase. For instance, “muito bom” does not convey the same intentionality as “MUITO BOM” (this phrase can be translated to “very good).
- **Remotion of emojis:** Removes any emoji found in text. Emoticons, such as “:-)” are not considered. This method also reduces the vocabulary size at the cost of losing some information. The Python library *emoji* [78] was used for it, with the option of replacing all emojis with the empty character.
- **Remotion of stopwords:** Removes stopwords present in a text review. Stopwords were defined as all words that do not convey any meaningful information with respect to SA, such as “{e, de, a, que, o, com, …}” (see Figure 3.1), as well as words with less than three characters. In this case, a lexicon-based approach was followed to remove the stopwords by creating a custom list of stopwords based on the predefined Portuguese list from the *NLTK* Python package [79]. The custom-made list can be examined in Appendix A.
- **Rule-based bi-grams:** Joins two consecutive tokens if the first one belongs to a preselected set of tokens. The list of selected tokens was hard-coded, including “não”, “muito”, and “sem”, which may significantly alter the meaning of the subsequent token. Algorithm 2 shows how this mechanism was designed.

Algorithm 2 Rule-based bi-gram creation

Input: text (string of characters)
Internal constants: prefixes (list of strings → predefined set of tokens to create bi-grams)
Output: result (string of characters with bi-grams created, if any)

1: Separate each word into a separate string. Punctuation marks are separated from full words.
tokenized ← split_into_words(*text*)

2: Initialize control variable *'jump* = False' and result *'result* = []'.

3: Iterates over each token and checks if the condition to create a bi-gram is met.
For each index of *tokenized* list:

3.1. if *jump* = True:
Skip For loop iteration
jump ← False

3.2. Select the current and next token.
bigram = *tokenized* [*i* : *i+2*]
assert length(*bigram*) = 2

3.3. Create a bi-gram if the condition is met.
if (*bigram*[0] in *prefixes*) && (*bigram* [1] in alphabet):
result ← concatenate(*result*, (*bigram*[0] + '_' + *bigram*[1]))
jump ← True
else:
result ← concatenate(*result*, *bigram*[0])

- **Remotion of numbers:** Removes all numerical expressions from text. The Python build-int function isdigit() of the *str* class was used for it.
- **Remotion of punctuation marks:** Removes all punctuation marks, such as "!", "-" or "?". The Python build-int function isalnum() of the *str* class was used for it. The character "_" was kept since it is the special character used for joining bi-grams.
- **Format blank spaces:** Converts multiple blank spaces and newline characters ('\n') into a single space character.

These steps were defined to be executed in the same order that appears in this list. Figure 4.2 shows the sequence of preprocessing steps and represents all possible combinations. Note that the step "Format blank spaces" is not considered here since it was considered a mandatory step. Considering the seven preprocessing steps, it is necessary to test $2^7 - 1 = 127$ different combinations for each SA model developed. This was addressed using a grid search algorithm, where the performance of each combination is stored, and the selection of the best one is made mainly based on sensitivity, F1-score, ROC-AUC, and accuracy.

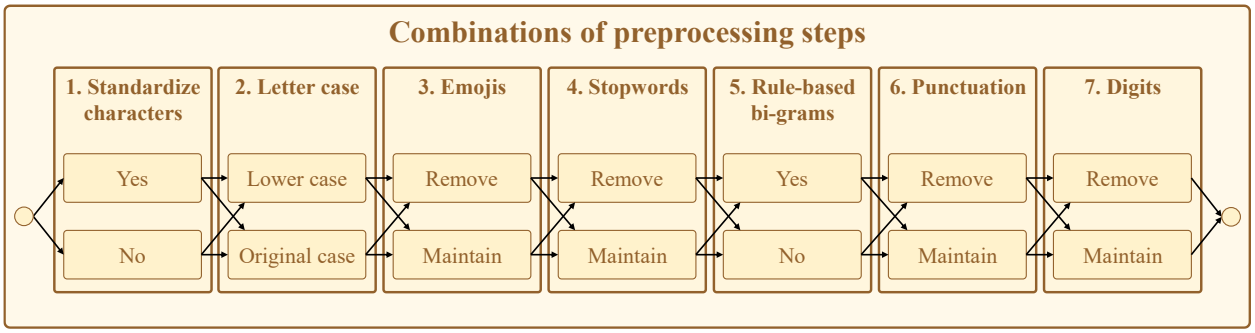


Figure 4.2 – Predefined preprocessing steps to be applied sequentially in the input pipeline. Adapted from [45].

As testing all combinations for all SA models developed is a highly time-consuming task, both empirical tuning and automatic optimization (see chapter 5) of the SA models were done using a default combination of preprocessing steps, which performs all steps represented in Figure 4.2 (corresponds to combination 127). To assess the gain in performance of a model when using text cleaning techniques, the REC baseline model (further discussed in section 4.4) underwent a 2-fold CV with and without any preprocessing step, and the results were made available in Appendix B. The model that includes all preprocessing steps showed an increase of F1-score of 4.85% with respect to the model without any cleaning, also significantly increasing sensitivity and accuracy, which shows a non-biased enhancement by the distribution of the classes. Each set of preprocessing steps is identified by a numerical ID, as depicted in Appendix B, obtained by the concatenation of all the combinations from $\binom{7}{0}$ to $\binom{7}{7}$.

4.3.2. Vectorization

Once the text has been cleaned from redundant and irrelevant information, the subsequent step involves transforming the text into a numerical representation that the ANN can consume. This process was split into two: text tokenization and token vectorization. The tokenization process is generally conducted before cleaning steps; however, the decision to be done at this point was due to its interdependence with the vectorization process, as implementing the tokenization and vectorization methods was done taking advantage of the tools provided by *Keras*. Two text tokenization techniques, namely Blankspace and Wordpiece tokenizer, were evaluated in this study. The selection was done through a genetic algorithm exclusively for the AREC model. Up to that point, it is acknowledged that a tokenization technique has already been chosen, as the process remains transparent to this choice.

As a preliminary test to understand what the best vectorization technique in this scenario is, three techniques were considered and subjected to a qualitative and quantitative evaluation. First, TF-IDF representation was used along with linear dense layers to extract features. Then, Word2Vec and FastText models were used to perform contextualized BoW vectorization.

To address the effectiveness of TF-IDF vectorization, the baseline model present in section 4.4 had its input modified to handle this representation. The TF-IDF algorithm was implemented using the scikit-learn Python library with its default parametrization and a vocabulary size of $N = 6000$ words/tokens. This value was chosen heuristically considering the hardware's capacity to handle the input dimensionality, as the TF-IDF matrix is a sparse representation that can be demanding on memory. Additionally, a linear layer with 512 neurons and a ReLU activation function was employed to reduce the dimensionality of the input, followed by a reshaping process to ensure compatibility with the input of the LSTM layer. The BoW technique does not consider the position of each token; therefore, the input dimension is no longer treated as time steps but as vocabulary size, implying $T = N$. This brings the problem that the RNN of the encoder will not find sequential information from its input.

The performance of this model is presented later in chapter 6 and shows to be inferior to subsequent models. Moreover, the TF-IDF representation proved to be inefficient, leading to Out-of-Memory (OOM) exceptions after a couple of iterations, and the training process was slower compared to later models due to the sparsity of the input. While it is acknowledged that the way the input was passed to the recurrent encoder may have acted as a bottleneck to information flow, no further architectural optimizations were conducted. Therefore, this technique was no longer considered in the rest of this work.

The *Gensim* Python library was used to implement the Word2Vec and FastText vectorization methods and define the vocabulary based on the training data. As the dataset is considerably large, and there is greater interest in semantics rather than syntactic information, the SG algorithm was preferred for training, as suggested by Mikolov et al. [31]. A Python script, available in Appendix C, was used to test multiple combinations considering the following hyperparameters:

- Window size: 2, 4, 8, 10, or 15.
- Vector size: 256, 300, 512, 768.
- Number of epochs: 5, 10, 15, 20, 50, 100.

The remaining hyperparameters were kept the same for both Word2Vec and FastText, with the lower word frequency threshold equal to 3, the initial learning rate $\alpha = 0.03$, and the negative sampling equal to 10 noise words. The rest were kept as default.

The experiments conducted in this study focused on the training partition specified in Table 3.1. Initially, the corpus underwent a cleaning process, including all the preprocessing steps outlined in Figure 4.2. The evaluation procedure involved examining the cosine similarity values between a set of keywords and their top five most similar words found within the embedding model. The chosen keywords were ‘comida’,’espaço’,’atendimento’, ‘agradável’, and ‘péssimo’, which offer a subjective assessment of the model's ability to recognize the most pertinent words or tokens in the context of these common, significant words for SA on Zomato’s dataset.

In general, the tests indicated that variations in the outcome were not significantly different between the two vectorization methods when evaluated under the same parametrization. The comparison was mostly subjective and not easily replicable, as there is not a concise mechanism to evaluate how well the embedding initialization would affect the SA downstream task, and the primary intent was to facilitate the training of the input embeddings by starting from values different than random. While FastText provided clusters of words that are similar in morphological structure (same characters and order of characters), Word2Vec offered semantically similar words, suggesting this model might be better suited for SA, as these semantic relationships are relevant when determining the intent behind a user review. Both models showed better performance at recognizing related works when trained for only five epochs.

Finally, the chosen values for the rest of this work were derived from the top five results of these experiments, being the window size equal to four words, five training epochs, and a vector size of $F = 300$ (although this parameter is variable depending on the network’s architecture). Appendix B provides the cosine similarity values between the keywords and the five most similar words within the embedding model with this parametrization. The joint selection of text cleaning steps and the tokenization mechanism must undergo a compromise between the model's complexity and the number of OOV tokens that can be tolerated without significant loss in performance during training and inference.

4.3.3. Input embedding

The steps above were necessary to create the first stage of the DL models. In the *Keras* framework, the Embedding layer can be used, parametrized with the vocabulary size (N) as the input dimension and the number of features (F) as the output dimension, as an interface to transform the tokenized text into its vector representation, during the computation of the graphs and, therefore, taking advantage of the GPU computing capabilities.

In practical terms, the embedding layer is a look-up table, or matrix, with dimension $N \times F$, which will be multiplied by the input. The input to the embedding layer is a tensor with dimension

T, which represents the maximum number of tokens considered to represent a meaningful review (subsection 3.2.1), and where each position is an integer that represents an existing token in the vocabulary. Internally, each integer is converted into a one-hot encoding representation and multiplied by the embedding matrix. Figure 4.3 exemplifies the function of the input stage with smaller values for the sake of simplicity. Actual values utilized in this step were $N = 11{,}560$, $F = 300$, and $T = 100$, and F may suffer changes later in this work due to the optimization process. Note that the vocabulary size was not mindlessly defined but was determined heuristically using a quick search algorithm where a baseline model was trained multiple times, and the range of values between 5,000 and 30,000 was tested, followed by a comparison based on accuracy and ROC-AUC.

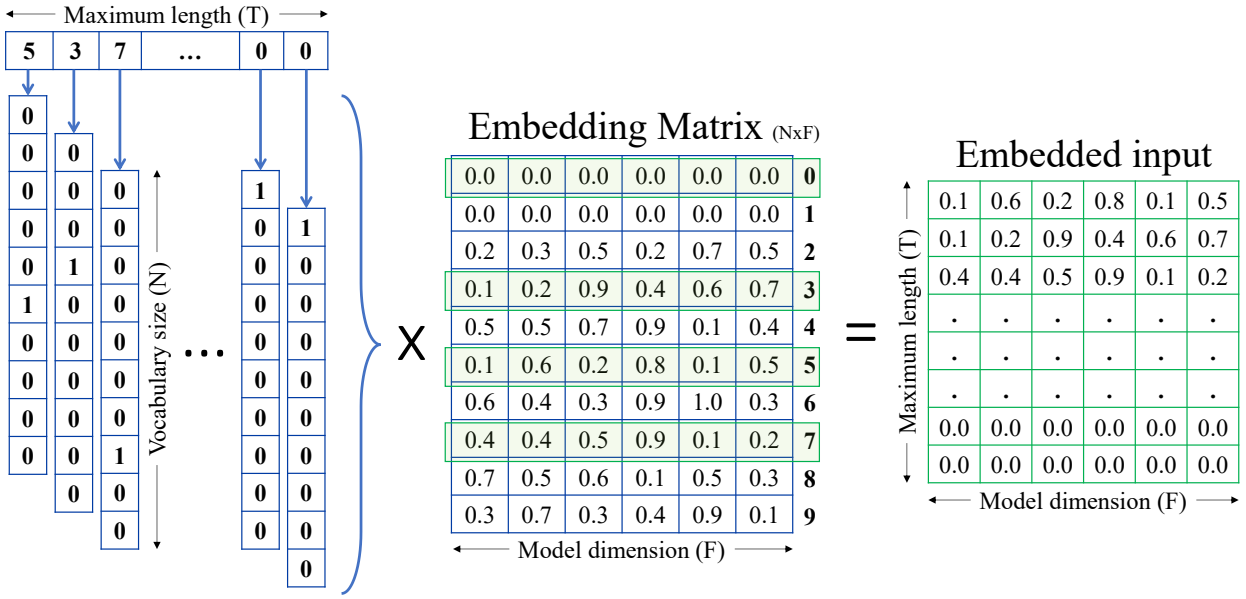


Figure 4.3 – Example to illustrate the logic behind the embedding layer. Considering the parametrization: T=8, N=10, F=6.

The embedding layer can be randomly initialized using a normal or uniform distribution. However, in this work, experimental tests have revealed that using Word2Vec pre-trained vectors results in better performance on the downstream classification task than randomized initialization methods. When Word2Vec pre-trained vectors are used, the embedding matrix already conveys some semantical and contextual information, accelerating the training process and improving the model's performance. Note that these vectors are now treated as trainable parameters within the model, allowing for fine-tuning the pre-trained vectors from the Word2Vec model. Unlike previous work, no reduction or aggregation method was applied after obtaining the input embeddings, which generally computes the average or weighted average using IDF weights [52].

Both REC and AREC architectures (discussed in the following sections) were optimized, maintaining a common input embedding layer with the same hyperparameters. Additionally, AREC uses position embedding before multi-head self-attention to ensure the model keeps track of the position of each token during the parallel processing.

4.4. Recurrent Encoder Classifier

This section explains the proposed REC model's architecture, including the reasoning behind each choice and preliminary tests conducted to eliminate solutions that do not meet the performance expectations. This type of DL model is designed to process sequential data such as time series, text, or speech, powered by recurrent mechanisms that process the time series

sequentially and create an internal representation that conveys all the relevant information and context used to make a prediction.

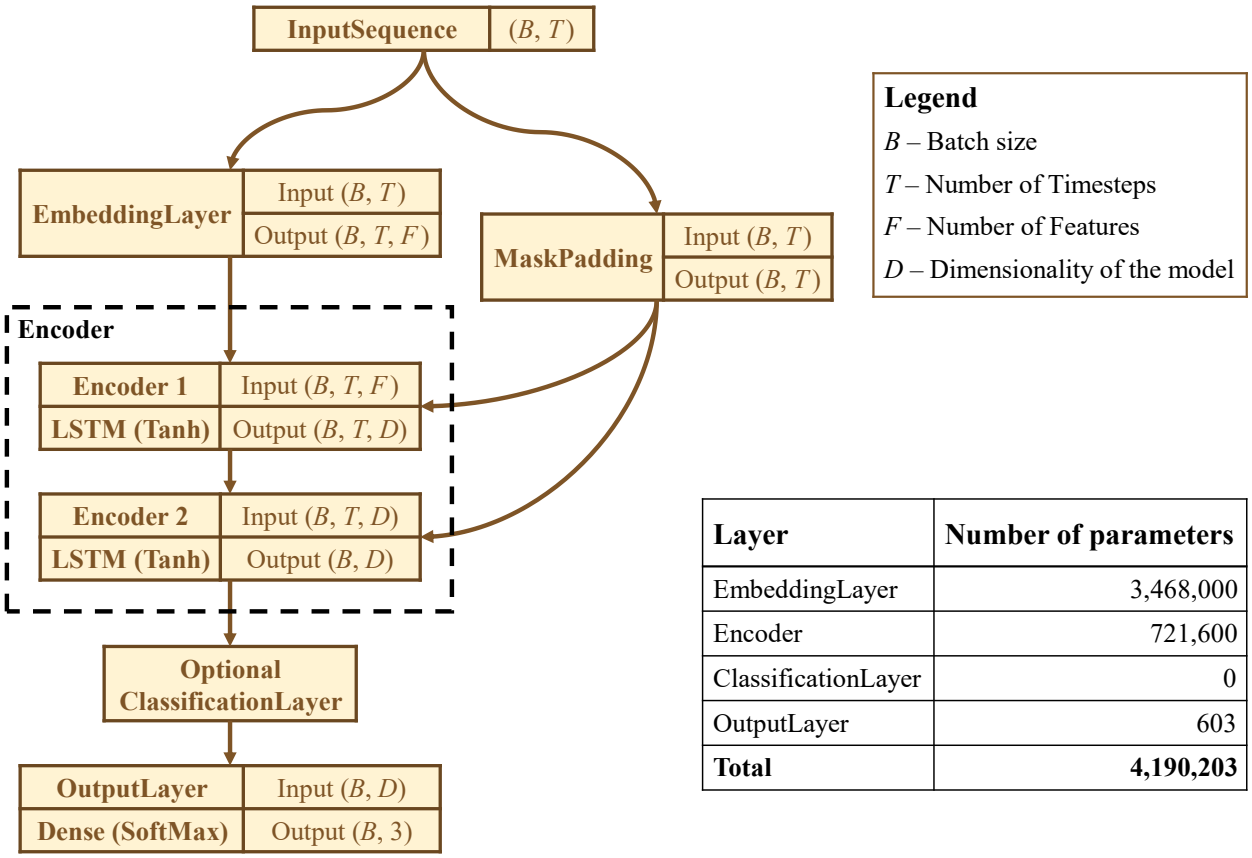
The definition of the REC model was divided into two phases. The first defines the baseline model based on empirical tuning, where several models were tested using a 2-fold CV scheme to gain insights about what the best parameters could be, while the second one employs an automated algorithm to perform this search in a faster way, also using a 2-fold CV scheme, based on a predefined set of parameters (discrete search). The decision to perform a 2-fold instead of a 5-fold CV was motivated by time restraints. The first phase was addressed in this chapter, while the second was addressed in chapter 5.

Various network configurations were explored to find a proper baseline model, taking inspiration from previous studies. The performance of the models in this phase was inspected, considering accuracy, ROC-AUC, and sensitivity. Accuracy was considered to maintain consistency with prior research, while the latter two were considered to assess the model's performance without being influenced by the class imbalance. The decisions made for each model stage are listed below. The number of comparisons is substantially large, and it takes several hours to have a consistent result. Therefore, although the process was quantitative, no performance values were retrieved during this phase, as it is not the scope of this study to conduct a profound comparison. However, this proves in practice the need to use some automated optimization procedure to save time and try to avoid bias during the selection of the parameters.

- **Recurrent layers for the encoder:** LSTMs and GRUs are the most used layers for contextual encoding. As expected, preliminary tests showed that the LSTM layer outperformed the GRU layers. However, the GRU was able to train faster. The number of recurrent units was set to twice the input sequence length ($D = 2 \times T = 200$). Additionally, regularization techniques were explored in this stage, which is also a very popular technique when using these layers to avoid overfitting and better explore the optimization space. Specifically, L1 and L2 penalties were applied to update its parameters, noting a slight improvement in the performance. Therefore, the baseline model ended up making use of the L1 penalty.
- **Number of LSTM layers:** After several tests ranging from one to four stacked layers, the number of LSTM layers was defined as two layers (like ELMo [35]) that perform only an unidirectional, forward pass through the input sequence. Note that RNN layers consist of many parameters, and as the number of layers is increased, the time needed for training also increases, making it difficult to assess by brute force.
- **Classification layer:** Limited to a MLP network, the selection was empirical and based on the number of linear layers, number of neurons, and activation function. Tests showed that using more than one linear layer most of the time resulted in the worst performance. On the other hand, several tests were done to determine the number of neurons and the activation function, which was revealed to be a relevant hyperparameter. However, the best performance was attained when this layer was excluded,
- **Output layer:** The encoded representation is passed directly to the output layer, which consists of a linear layer with three neurons, one for each sentiment. The neurons utilized a SoftMax activation function to convert the output into a probability distribution.

The selection of RNNs for the encoder architecture was motivated by their ability to handle sequential data and their past success in various NLP tasks. Adding multiple layers allowed for a more complex model that could learn more intricate relations between the input and output data.

On the other hand, it is interesting to note that the best performance was attained without a classification layer. However, this layer is further explored in chapter 5. Figure 4.4 displays the specific architecture developed for this baseline model, including the input Embedding Layer discussed in the previous section. It also includes the number of trainable parameters.



Layer	Number of parameters
EmbeddingLayer	3,468,000
Encoder	721,600
ClassificationLayer	0
OutputLayer	603
Total	**4,190,203**

Figure 4.4 – Architecture of the baseline REC model and number of parameters per layer.

4.5. Attentive Recurrent Encoder Classifier

This section explains the steps performed to develop the architecture of the proposed AREC model, including the preliminary tests conducted to eliminate solutions that do not meet the expectations. This type of DL model is designed to process sequential data, just as the REC model, with the difference that weights are computed and applied to each timestep before or after the sequential processing of the RNN, allowing the model to pay more attention to specific tokens than others.

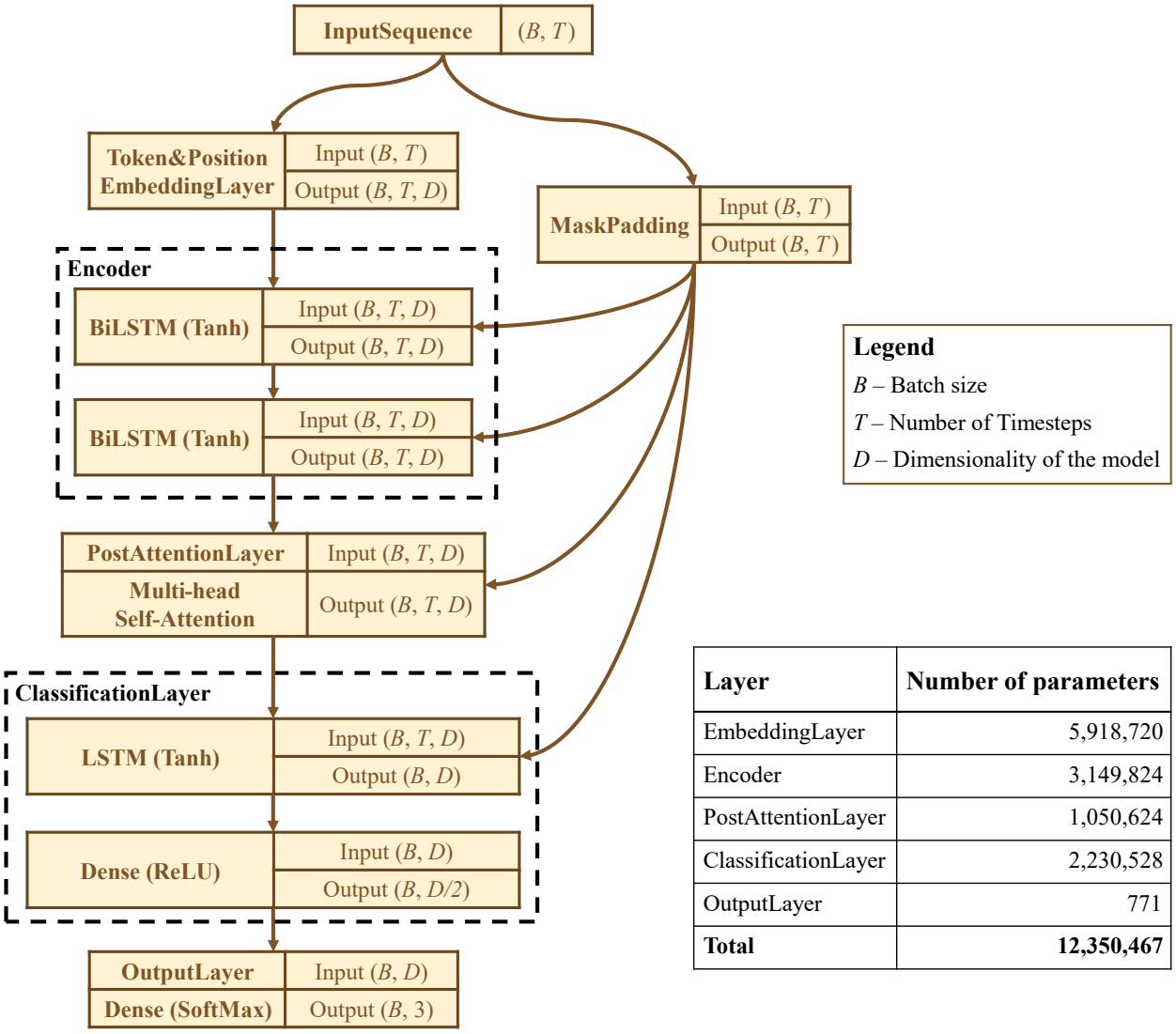
The definition of the AREC model was also divided into the same two phases as REC. The empirical definition of the baseline model was treated in this section. Some decisions made for the REC architecture were extrapolated to AREC due to its similarity at the level of the recurrent encoder. Moreover, the most inspiration was taken from the paper of Xia et al. [42], maintaining the same hyperparameters and architectural configuration and replacing the CNN-based input embedding layer with a trainable embedding layer initialized using the Word2Vec algorithm. Accuracy, ROC-AUC, and sensitivity were considered to select the best performance. A description of each architectural decision made to build the model is listed below.

- **Recurrent layers for the encoder:** Only BiLSTM layers were considered for the encoder, motivated by tests with REC. The number of recurrent units was set to $D = 512$. Additionally, to regularize the training process, an L2 regularization penalty was used with a weight of 0.0001 and a 50% dropout to avoid overfitting [42]. Two stacked

BiLSTM layers were used, maintaining the input dimensionality to be compatible with the self-attention mechanism. The optimization process also considered the possibility of using a unidirectional layer.

- **Attention layer:** A multi-head self-attention layer was defined to emphasize the most relevant tokens of the input. Empirical tests were conducted to determine the best position to apply attention: before, after, or before and after the recurrent encoder. As Xia et al. [42] did, post-attention resulted in the best performance, although the other configurations were considered for further optimization. This layer was parametrized with $H = 8$ attention heads and internal projection of $D_k = 64$ ($D = H \times D_k$), with the Tanh activation function to introduce non-linearities during the weights' computation.
- **Classification layer:** Different approaches were explored to process the context vector resulting from previous layers. An LSTM layer with D recurrent units followed by a linear of $0.5 \times D$ neurons and ReLU activation function was employed to decode the context vector. The recurrent classification layer was motivated by the work of Xia et al. [42]. Other mechanisms were also considered, including simple MLP and pooling operations [40], [80], and these were further explored during the next optimization steps.
- **Output layer:** The processed encoded representation is passed to the output layer, which is identical to the one used in the REC architecture.

Figure 4.5 displays the specific architecture developed for the baseline model, including the input Embedding Layer discussed in the previous section, which, in this case, adds a positional encoding vector to the original token embeddings, maintaining the dimensionality. This model is three times bigger than the REC baseline in terms of trainable parameters. Chapter 5 discusses how this model was optimized, including the encoding and attention block replication, which was motivated by the Transformer architecture [9], [36]. Note that dimensions *F* and *D* are the same in this architecture. Therefore, only *D* is referred to in the illustration.



Layer	Number of parameters
EmbeddingLayer	5,918,720
Encoder	3,149,824
PostAttentionLayer	1,050,624
ClassificationLayer	2,230,528
OutputLayer	771
Total	**12,350,467**

Figure 4.5 – Architecture of the baseline AREC model and number of parameters per layer.

4.6. Training configuration

The training conditions of the ANNs can significantly impact the learning process. The main networks' training process (as shown in Figure 3.8) followed the pipeline depicted in Figure 3.4. This section provides information about how the training was performed. However, the optimization process was disregarded at this stage since it will be discussed in chapter 5, which provides valuable insights into the optimization techniques employed to enhance the performance of the developed networks.

All models developed, including during the optimization processes, were trained on an NVIDIA GeForce RTX 3090 GPU with a dedicated memory of 24 GB. The framework used to develop the models was *TensorFlow* 2.9.1 in Python, alongside *Keras*. This high-level neural network library runs on top of *TensorFlow* and provides several resources to facilitate the usage of state-of-the-art models and components, such as LSTM and multi-head self-attention mechanism.

A script was developed to perform the training of the models in a loop according to the number of folds selected for evaluation. The script deals with the whole pipeline, stores the metrics of each fold in memory, and finally stores the results in a customized file structure to standardize

the comparison of the models. The hyperparameters that control the training process through the compile *Keras* function were the following:

- **Optimizer:** this option defines the algorithm dedicated to updating the parameters of the ANN based on the gradient and the cost function. After various tests using the REC baseline model, it was empirically decided that the RMSprop algorithm provides a better performance against Adam. As these are the most common algorithms employed, the selection was limited to these. The learning rate was also defined as 0.0001.
- **Loss function:** this option defines the cost function to be evaluated, which defines the process of learning. Equation (2.26) was selected as it is the most used in these scenarios. The cost was computed at the probabilities level, and the label smoothing parameter provided by *Keras* was set to 0.05 for regularization as it improved the performance.
- **Metrics:** this option defines the metrics to be computed during the training for both the training and validation set and allows the drawing of the learning curve for more learning indicators than loss function. In this scenario, accuracy and ROC-AUC were both computed.

And the hyperparameters that customize the training process through the fit *Keras* function were the following:

- **Number of epochs:** this parameter controls the number of times the training data will be used to update the ANN's parameters and the number of times the model will be evaluated on the validation data. The values were set to 100 epochs, although it was observed that the models never train that many times due to the early stopping mechanism.
- **Batch size:** is the number of samples that the GPU simultaneously processes during training, so it is mainly dependent on the available hardware. In this case, the optimal batch size that avoids OOM exceptions was determined by heuristics and set to 128. The training process can be efficiently executed by selecting an appropriate batch size without exhausting the available memory resources.

Regularization techniques were employed during training to ensure the best performance was attained at the end of the training process. As referred to in section 4.4, the REC baseline model was regularized by including L_1 loss penalty when updating the weights of the LSTM layer, with a coefficient of 0.01, but it was preferred to avoid its usage in later networks, assuming the optimization process would be able to handle the training difficulties, disregarding the need for this technique. The loss function also employed label smoothing, which was avoided in former networks.

Common to all models developed, the training process was controlled using dropout layers and two callbacks: a learning rate scheduler to ensure a warmup interval followed by a learning rate decay to avoid losing the first knowledge acquired since the learning curves showed an early peak during the training that produces the network to have a fictitious outstanding performance after 1 or 2 epochs; and early stopping that monitors the ROC-AUC of the validation set during training and stops the training process if the model is no longer training in the validation set. The minimum variation the early stopping callback considered was 0.0005, and its patience was 30 epochs.

Furthermore, after each model is trained, their performance is computed from the testing dataset. Figure 4.6 illustrates the file structure used to store the performance of each trained model, which allowed a straightforward evaluation and facilitated the comparison between multiple

models simultaneously. A script was employed for this, where model names are provided in a list, and a set of Boolean flags determine the kind of analysis to perform. The script can be consulted in Appendix C. The main interest during the comparison was to visualize the waterfall chart, which depicts the improvement or deterioration of a model after each optimization phase. The waterfall chart employed in this work must be read from bottom to top, where each metric constitutes an independent chart. The relative baseline is denoted in blue. Any increase from the previous bar is highlighted in green, while any decrease is highlighted in red. Chapter 6 compares several models using this chart hierarchically. Hence, models that show the worst performance in the former charts are no longer shown in the latter ones.

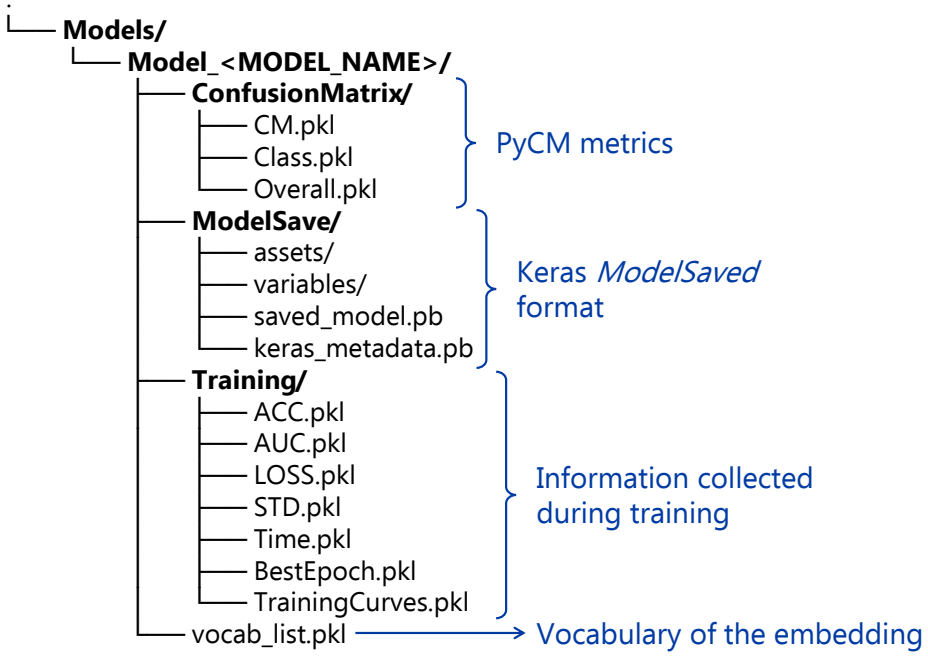


Figure 4.6 – Illustration of the file structure used to store each model's metrics and training process.

4.7. Key remarks

The training of a classical ML baseline model (in this case, NB and SVM using TF-IDF for text representation) was addressed in this chapter because it was relevant to set a baseline for the Portuguese implementation of SA for a three-class classification problem. Previous research presented results, but setting a confident baseline was challenging because some studies used weighted metrics averages considering their data distribution, introducing bias. The baseline results presented were used again in chapter 6 to compare the overall improvement from classical to DL techniques.

The text representation has shown to be a relevant step in literature. This chapter explored text cleaning and text representation techniques, laying the groundwork for subsequent optimization processes. Preliminary tests showed that dense embedding vectors produce superior results when compared to more classical approaches, such as BoW scored with TF-IDF, confirming findings in the literature. Among the embedding methods, Word2Vec showed to be better at capturing semantic information among words, although the evaluation process needs to be enhanced as it is mainly subjective and might be biased. Consequently, Word2Vec was chosen for initializing the weights of the input embedding matrix. The empirical test improved the SA task when the embeddings were initialized with Word2Vec rather than a uniform random distribution.

Furthermore, the REC and AREC baseline models exhibited a significant difference in the number of trainable parameters, directly affecting the training time. It is expected to have a consistent gain of performance thanks to the attention mechanisms, but, as previous works have suggested, the effectiveness of a model is highly related to the data domain, the text preprocessing techniques, and the complexity of the reviews.

Configuring the training process included defining the CSL and BTOS approaches and the hyperparameters that modify the training process, such as the optimizer, the cost function, training callbacks, and regularization techniques. All of these were decided empirically and, most of the time, motivated by the methodology of previous works. The model's architecture was optimized using the CSL approach since preliminary tests using the baseline REC model indicated that this technique boosted the model's performance. Note that this does not indicate that CSL is the best solution, and the *a priori* assumption is that the optimization process will not produce significantly different results when using CSL, BTOS, or CIL.

5. Hyperparameter tuning using Genetic Algorithms

The large number of hyperparameters and the wide range of different architectural configurations that a DL model can motivate the use of algorithms that automatically search for an optimized model configuration. This is particularly evident in the case of the REC and AREC models, which have a vast number of potential configurations considering the dimensionality of the LSTM layers, the size of the hidden linear layers, the dropout values for training regularization, or the position in which certain operations are performed within the model, such as the self-attention mechanism. The selection of the right parametrization lacks an analytical calculation method.

Manual optimization involves iteratively adjusting and fine-tuning various hyperparameters and configurations of a model. This process gives researchers some degree of insight into how the model responds to the hyperparameters, with the drawback of being hardly reproducible and demanding considerable investment of time and effort, which can quickly become a burden, especially when added to the relatively slow training of the models [81]. Moreover, this last characteristic remains a limitation even when transitioning to an automated optimization process, as the model will continue to train at the same relatively slow pace.

With these aspects in mind, this chapter explains the procedure for automatically optimizing the baseline models proposed in chapter 4, aiming to achieve significant performance gains. The chapter begins with an overview of alternative automatic optimization algorithms to provide context. Evolutionary algorithms are then introduced as the chosen approach for optimization. The selected algorithm for the optimization process is further explained and focused on the requirements of this work, explaining how it was used and how it was differently parametrized for REC and AREC architectures.

5.1. Hyperparameter search methods

The configuration of the model's hyperparameters must be set according to the nuances of the dataset. A search space needs to be defined according to the characteristics of the desired architecture for the model. A point within the search space can be visualized as a vector with a specific set of hyperparameter values. The goal of the optimization is to find a vector that results in the best performance of the model on a validation dataset, such as maximum accuracy or minimal error.

Diverse optimization algorithms are available for usage, and their effectiveness might depend on the size of the optimization space and the complexity of the model's architecture. The three most common methods are grid search, random search, and evolutionary algorithms [82].

Grid search is a brute force algorithm that performs an exhaustive hyperparameter optimization among a grid of potential values, systematically evaluating each combination within it. By examining all predefined positions in the grid, grid search allows for a comprehensive assessment of various hyperparameter setups, with the goal of selecting the configuration that yields the highest performance. Random search is a technique that involves defining a range of possible hyperparameter values and then randomly selecting points within this search space to evaluate. Unlike grid search, which evaluates discrete predefined combinations, it explores a subset of hyperparameter configurations by chance, considering a continuous space. This method is particularly effective for uncovering combinations that might not be intuitively apparent.

While Grid Search explores numerous combinations, it might overlook the most significant ones, as its effectiveness relies on the user's prior knowledge. This algorithm might not be recommended if the defined search space is too broad. On the other hand, Random Search finds better models, although not guaranteed, requiring less time. When the search space involves numerous hyperparameters, evolutionary algorithms are preferred as they provide guidance in selecting vectors within the search space, as opposed to being entirely random. Although these algorithms take longer to execute, they can be fine-tuned to exert some control over their direction to a certain extent [81], [82].

Since all possible combinations were already established, a grid search was used to find the optimal text preprocessing steps in subsection 4.3.1. On the other hand, knowing the best-suited model architecture was more challenging. Therefore, evolutionary algorithms were preferred for the rest of the optimization procedures in this study.

5.2. Genetic algorithm

Genetic Algorithm (GA) is a population-based search algorithm inspired by the process of evolution and the mechanisms of natural selection. At its core, it utilizes the principles of the Darwinian theory of survival of the fittest individuals present in a population. Population-based metaheuristics utilize multiple candidate solutions during the search process. These metaheuristics maintain the diversity in the population and avoid the solutions being stuck in a local minimum [83]. The usage of GAs on ANNs usually have three main objectives [84]:

- determine the architecture of a ANN,
- determine the weights of a fixed ANN architecture,
- determine the best-suited input data and output interpretation.

In this study, the first option was considered. Different codification schemes can be set up depending on the desired architecture, which is usually done using binary strings. Unlike brute-force solutions, a GA is not guaranteed to find the best solution. Although it will find a suitable solution, the score might not be the best. The following subsections detail the principles of the GA developed to find a solution for this work.

5.2.1. Description

The algorithm was adapted from the one developed by Mendonça et al. [85]. The user can manually configure the expected number of generations and the number of parents, or chromosomes, in each generation. Each chromosome is encoded as a binary sequence with a predefined length, according to the search space defined for each architecture (see sections 5.3 and 5.4). The algorithm stores checkpoint files containing the current generation and the overall best chromosome (BC) and respective scores at the beginning of each generation, which allows adopting a manual early-stopping mechanism where the user can decide if more generations are required at the end of the process.

The optimization starts by defining an initial population, an essential factor responsible for the performance of the GA, computation time, and quality of its solutions [83]. A population Y of n chromosomes is initialized randomly. Each chromosome is composed of a string of a bits that encode information according to the architecture being optimized. The fitness function is computed for each population, with an evaluation scheme of 2-fold CV. The fitness function is the driving force of the GA. Afterwards, the three main used operators in GA are applied to the attained population [83]:

- Rank selection between parents and children, based on the Performance Metric (PM), maintaining the $E_n = 2$ most fitted chromosomes to the next generation (elitism). The convergence rate of GA depends upon the selection pressure, and employing elitism reduces the chances of premature convergence.
- Single-point crossing-over operation, with a probability of C_p, after n tournaments between two parents. Crossing-over combines the genetic information of two or more parents, and single-point provides an easy-to-implement solution at the cost of less diverse populations.
- Bit flipping mutation operation, with a probability of M_p, with a decreasing mutation rate limited to a minimum of 1%. Mutations maintain the genetic diversity from one population to the next.

The flowchart of Figure 5.1 illustrates the GA.

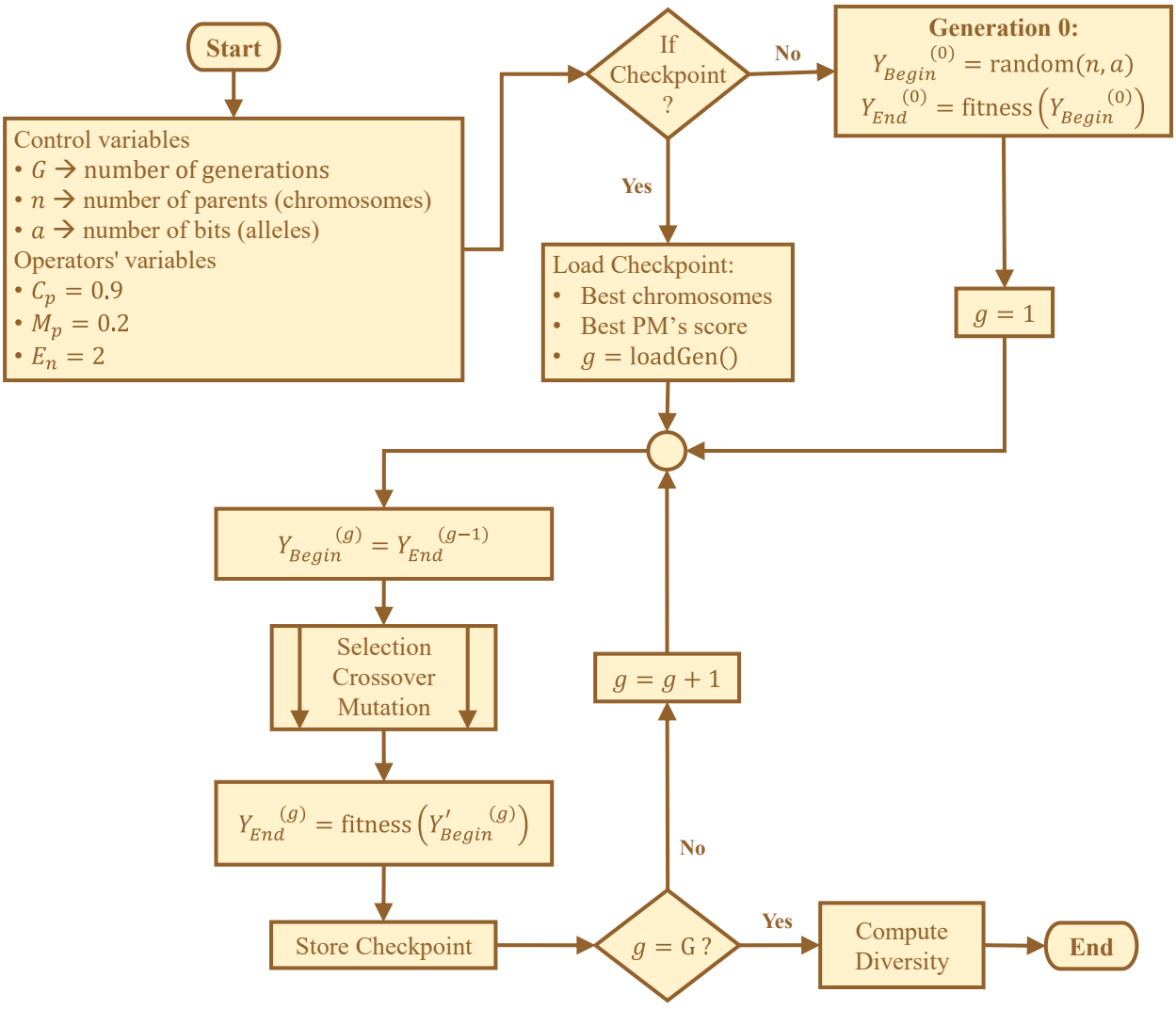


Figure 5.1 – General flowchart of the Genetic Algorithm (GA). Its usage for the optimization of different architectures will define the Fitness function.

Four files are generated before and after each generation that keep track of the past generations, including its initial population and corresponding PM scores and its resultant population with their corresponding scores. These files allow monitoring the process, controlling the initialization from a checkpoint, and computing the diversity among the generations.

5.2.2. Aim of the optimization

ROC-AUC was selected as the PM. Additionally, accuracy, sensitivity, specificity, and F1-score were stored after each CV training session to assess the optimization from multiple perspectives. At the end of the optimization, the normalized diversity is computed, and its value is expected to stay high over time due to the mechanisms that preserve the diversity, allowing a better exploration of the optimization space. The optimization was only considered finished when the PM attained its maximum value and diversity had stopped increasing after a fixed number of generations, referred to as patience. On the other hand, a variable diversity may suggest optimization requires more iterations. The available time for optimization was also considered a stop criterion. Since the optimization procedure is considerably time-consuming, a 2-fold CV was used to find the optimized solution.

5.2.3. Parametrization

The parameterization of the GA was shared across the optimization of both REC and AREC architectures. The hyperparameters that define the behavior of diversity preservation techniques remained consistent with the study by Mendonça et al. [85], as illustrated in Figure 5.1. Determining the appropriate number of generations posed a challenge due to its strong dependence on the functionality of the models constructed during optimization. This led to the adoption of an adaptive approach involving checkpoint restarts. Initially, the number of generations was set to five, with the possibility of subsequent increments based on the initial results' assessment in terms of ROC-AUC and diversity, and patience was set to three generations. It's important to highlight that optimization time was constrained, adding another factor to consider.

5.3. Optimization of REC

The optimization of the REC architecture was conducted by setting an encoding scheme consisting of a string of zeros and ones, where groups of bits form a locus that describes a corresponding architectural configuration. The tentative to manually optimize the model allowed us to understand what range of hyperparameters are better suited to this problem and what architectures would be preferred. Therefore, all these insights were taken into consideration when building the GA. Each generation creates or derives a population of binary chromosomes, where information about the model's hyperparameters and architecture is encoded. Table 5.1 describes the codification of each bit within a string of 16 bits.

Table 5.1 – Encoding scheme used during the optimization of the REC architecture.

Hyperparameter	**Locus**	**Specification**
Direction of the RNN layers	0	0: Unidirectional 1: Bidirectional
Type of RNN layers	1	0: GRU 1: LSTM
Number of RNN layers	2-3	00: 1 01: 1 10: 2 11: 3
Shape of RNN layers (*D*)	4-5	00: 128 01: 256 10: 512 11: 1024

Percentage of dropout for dense layers	6-7	00: 0 01: 10% 10: 30% 11: 50%
Number of features of the input layer (*F*)	8	0: 300 1: 512
Shape factor of the classification layer	9-10	00: (×0.0) 01: (×0.5) 10: (×1.0) 11: (×1.5)
Activation function of the classification layer	11-12	00: ReLU 01: Tanh 10: SeLU 11: GeLU
Operation of RNN layers	13	0: Many-to-One 1: Many-to-Many
Batch size during training	14-15	00: 32 01: 64 10: 128 11: 128

Loci 2-3 and 14-15 happened to have more than one binary sequence encoding the same characteristic. This was done to ensure all strings were valid, and heuristics decided the repeated value. The shape factor of the classification layer was meant to be multiplied by the model's dimensionality to establish a relationship between the values instead of being completely independent. The dense classification layer is dismissed when this factor is zero, such as in the REC baseline architecture. The operation of the RNN layers can be Many-to-Many, where the hidden state of each timestep is returned by the RNN and used by posterior layers, or Many-to-One, where only the last hidden state of the whole input is returned, and this vector is replicated T times to ensure compatibility with posterior layers.

During training, some hyperparameters were set constant based on the experience gained during the early experimentation stage, such as the RMSprop algorithm to optimize the models' hyperparameters, with a learning rate of 0.001, an early-stopping callback monitoring AUC with a minimum variation of 0.01 and patience of 35 epochs, and learning rate scheduler with a warm-up of the third epoch and a minimum value of 5×10^{-4}. Finally, a fixed-size vocabulary of $11{,}560$ n-grams was learned using the Word2Vec algorithm using the Blankspace tokenizer.

5.4. Optimization of AREC

The encoding scheme of the AREC architecture and hyperparameters is defined in Table 5.2, where a binary string of 21 bits characterizes each individual in the population. Empirical tests during the definition of the AREC baseline model permitted the definition of the range of hyperparameters of interest.

Table 5.2 – Encoding scheme used during the optimization of the AREC architecture.

Hyperparameter	**Locus**	**Specification**
Direction of the LSTM layers	0	0: Unidirectional 1: Bidirectional
Number of LSTM layers	1-2	00: 1 01: 2 10: 3 11: 4
Position of Self-Attention layers	3-4	00: No Self-Attention 01: Before LSTM 10: After LSTM 11: Before and after LSTM
Dimensionality of the model (D)	5-6	00: 300 01: 512 10: 768 11: 1024
Dimensionality of projections of Multi-head Self-Attention	7-8	00: 32 01: 64 10: 128 11: 256
Percentage of dropout for dense layers	9-11	000: 5% 001: 10% 010: 20% 011: 30% 100: 40% 101: 50% 110: 60% 111: 70%
Operation before the classification layer	12-13	00: Flatten + Dense layer 01: Average Pooling layer 10: LSTM layer 11: BiLSTM layer
Shape factor of the classification layer	14-15	00: (×0.0) 01: (×0.5) 10: (×1.0) 11: (×1.5)
Activation function of the classification layer	16-17	00: ReLU 01: Tanh 10: SeLU 11: GeLU
Number of times to replicate the encoder block	18-19	00: 0 01: 1 10: 2 11: 3
Tokenizer	20	0: Blankspace 1: Wordpiece

The hyperparameters D and D_k were selected as power of two, as typically done in literature, but also to ensure the resulting number of heads is an integer value. A particular case was handled when $D = 300$, where D_k values are converted to the nearest integer multiple of 300. GRU layers were no longer considered, motivated by the results of optimizing the REC architecture. Different

operations for the classification layer were considered by intuition, where pooling and recurrent mechanisms have already been used in the literature.

The training configuration was identical to the one described in the previous section. In this case, two tokenization schemes were considered during the optimization as a hyperparameter to optimize, and a fixed-size vocabulary of 11,560 n-grams is still considered and extracted from the Word2Vec algorithm.

5.5. Key remarks

The range of hyperparameter values and architectural configurations were defined empirically. Therefore, there is a chance the GA cannot find a good chromosome given the defined search spaces. The training conditions were maintained the same as those defined in section 4.6 to isolate the effect of optimization and reduce ambiguousness.

Note that the baseline versions of REC and AREC architecture might not directly relate to their optimized version. Also, the AREC model is expected to perform better than the REC model, as suggested in the literature. Situations such as optimization speed and OOM exceptions were out of the author's control. However, efforts were made to ensure some degree of control, including the checkpoint mechanism implemented on GA.

Regarding the GA itself, several approaches might be utilized to improve its performance by adding complexity to the algorithms to manage diversity. However, that was not the scope of this study.

6. Results and Discussion

The implementation and optimization procedures resulted in several models that can be compared based on the metrics discussed in subsection 3.4.4. Some models may exhibit different performances, offering potential benefits depending on the intended usage. The text representation techniques are of great relevance and were evaluated by testing all possible combinations of preprocessing steps and selecting the best-suited vector representation during preliminary tests.

This chapter presents the results obtained for each step, separately considering REC and AREC architectures and comparing their performances. Additionally, section 6.4 provides a more detailed explanation based on examples to comprehend the operational mechanisms of the models.

6.1. Input representation

The improvement of the REC baseline model when using TF-IDF and Word2Vec vectorization techniques is depicted in Figure 6.1, noting a substantial increase of at least 1% in all metrics. This result adds to the current knowledge, supporting the idea that dense representations allow a better representation of text features.

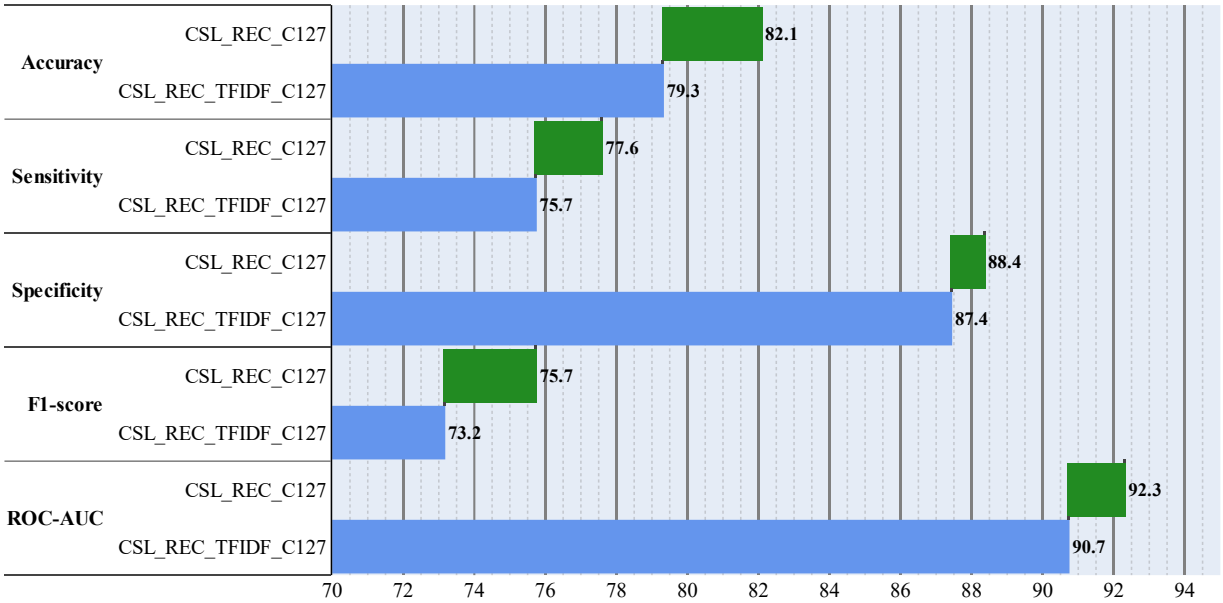


Figure 6.1 – Comparison of performance of REC baseline model with TF-IDF and Word2Vec as a text representation method.

The usage of the whole set of preprocessing steps (C127) showed better performance than maintaining the raw data, suggesting it is favorable for the DL model. The optimization was conducted for the best models attained with REC and AREC architectures. Figure 6.2 represents the variation of all six metrics during the optimization process, showing that the REC architecture is generally more sensitive to the text preprocessing steps than AREC. Moreover, the values of specificity and ROC-AUC are larger when compared to the rest of the metrics. Considering the OVR approach, the specificity is boosted due to the large number of TN samples when focused on the negative and neutral classes because of the data skewness. Furthermore, as ROC-AUC is approximated by (3.7), its high value is also attributed to the large number of TN samples. Finally, the ROC-AUC in the AREC architecture does not show significant variations, which turns it into a not-so-good metric to assess its gain in performance. Therefore, in this case, the second most relevant metric is considered to be the F1-score.

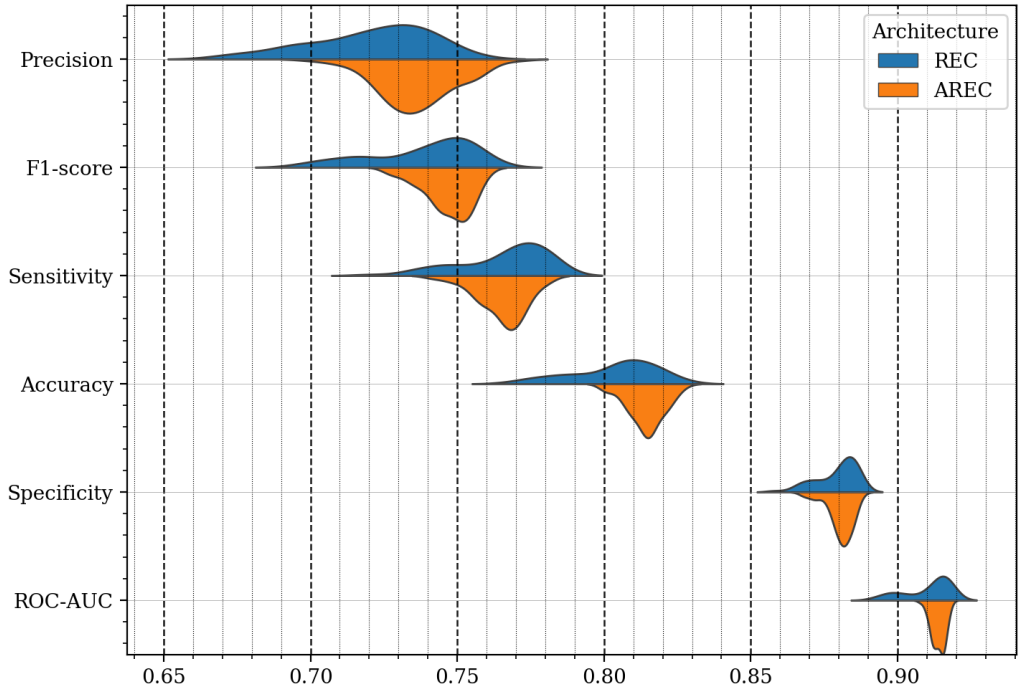


Figure 6.2 – Variation of the performance per metric during the optimization of the preprocessing steps.

The analysis of the results is mainly focused on finding comparable sensitivity and F1-score values to ensure precision has approximately the same value. Accuracy is a very sensitive metric to imbalanced data, noting that for values above 83%, it is almost certain that the model got biased toward the majority class. Finally, the ROC-AUC values were nearly always large, above 90%, which suggests an excellent performance, although the actual results do not reflect it. This behavior is supported in the literature [86], [87], which also states that the Area Under the Precision-Recall Curve (PRC-AUC) is more informative than ROC-AUC when dealing with highly skewed datasets. The PRC shows precision values for corresponding sensitivity (recall) values. Unlike ROC, which has a fixed baseline (see Figure 3.10), PRC determines its baseline considering the ratio of positive and negative samples, allowing a more robust evaluation [87].

Moving forward, three models were selected for REC and AREC architectures, and their performance is exposed in Table 6.1 and Table 6.2, respectively, considering

$$\text{score} = 0.10 \times \text{ACC} + 0.30 \times \text{TPR} + 0.40 \times \text{F1} + 0.20 \times \text{AUC}\,, \tag{6.1}$$

where different relevance is given to each metric. Note that specificity and precision were not considered to avoid redundancy and to acknowledge the potential impact of "leverage" on the score.

Table 6.1 – Performance of the top 3 models selected from optimizing the text preprocessing steps of the REC architecture, evaluated with a 2-fold CV, showing mean (standard deviation).

ID	ACC (σ)	TPR (σ)	F1 (σ)	AUC (σ)
115	82.6 (0.15)	77.4 (0.19)	76.5 (0.16)	92.1 (0.06)
100	81.2 (0.67)	78.8 (0.22)	75.8 (0.44)	92.0 (0.07)
101	81.9 (0.14)	78.2 (0.08)	76.0 (0.91)	92.0 (0.10)

Table 6.2 – Performance of the top 3 models selected from optimizing the text preprocessing steps of the AREC architecture, evaluated with a 2-fold CV, showing mean (standard deviation).

ID	ACC (σ)	TPR (σ)	F1 (σ)	AUC (σ)
101	82.0 (0.73)	77.0 (0.74)	76.0 (0.79)	92.4 (0.37)
99	81.6 (0.22)	77.2 (0.23)	75.9 (0.04)	92.2 (0.01)
122	81.6 (0.32)	77.6 (0.14)	75.6 (0.13)	92.2 (0.03)

To simplify further analysis, only the best combination from the top 3 of each architecture was considered. The 115th combination selected for REC converts all text to lowercase, removes emojis, stopwords, and numbers, and creates rule-based bigrams. On the other hand, the 101st combination selected for AREC corresponds to the same steps, adding the standardization of the characters and excluding the creation of rule-based bigrams. The whole resulting data is available in Appendix B.

6.2. REC model

Three main comparisons were made, considering the models in Figure 3.8 relative to the REC architecture:

- A comparison between the baseline and intermediate GA-optimized models.
- An evaluation of the intermediate GA-optimized model against the same model after selecting the most effective preprocessing steps.
- An assessment of the optimized model's performance when trained under CSL, CIL, and BTOS balancing scenarios.

The manual and automated optimization of the model's architecture and hyperparameters led to a slight improvement in performance between the baseline and the BC resulting from the GA. Additionally, since the optimization process was aimed at improving the ROC-AUC, the gain in performance was computed as the Relative Obtainable Improvement (ROI) [88] of the interest metric before and after optimization,

$$\text{ROI} = \frac{M_{after} - M_{before}}{1 - M_{before}} \times 100\% \,, \tag{6.2}$$

where M is the metric evaluated. The denominator represents the remaining space for optimization of the baseline model. If the optimization process leads to improvement, ROI is positive; otherwise, it is negative. This expression is commonly used to quantify the impact of optimization efforts. Figure 6.3 illustrates the improvement of the optimized model with a waterfall chart, noting that an increase in the rest of the metrics accompanies the increase of ROC-AUC. In this case, ROI = 4.4%, and the accuracy and F1-score remain practically the same. In this dataset, it is typical that an increase in accuracy can be translated to overtraining the majority class. Hence, the observed behavior is favorable.

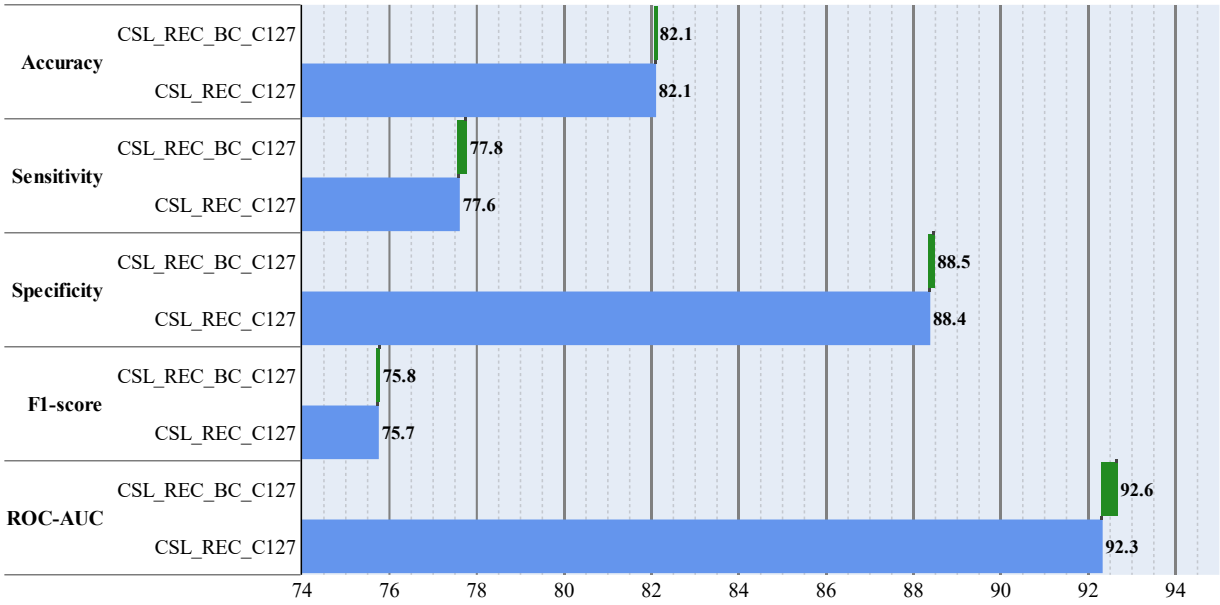


Figure 6.3 – Comparison of performance between the REC baseline model and the intermediate REC after GA optimization, where the aim was to improve ROC-AUC.

The first optimization process of this architecture, using the GA, resulted in the set of hyperparameters presented in Table 6.3 after fifteen generations with populations of ten parents. The diversity and ROC-AUC were computed at the end of the optimization process, as shown in Figure 6.4, noting that the BC was attained at generation number nine. The diversity plot indicates that generation 5 has the lowest diversity. At generation 9, the diversity had already increased, which suggests that the existence of crossing-over and mutation mechanisms allowed the algorithm to encounter a better architecture for the model. The BC of the ninth generation served to build the model, further referred to as REC BC.

Table 6.3 – Optimized hyperparameters of REC architecture after manual and GA optimization.

Hyperparameter	**Manual**	**Using GA**
Direction of the RNN layers	**Unidirectional**	**Bidirectional**
Type of RNN layers	LSTM	LSTM
Number of RNN layers	2	2
Shape of RNN layers	**200**	**256**
Percentage of dropout of dense layers	0.1	0.1
Number of features of the input layer	300	300
Shape factor of the classification layer	**0**	**1.5**
Activation function of the classification layer	**None**	**SeLU**
Operation of RNN layers	Many-to-Many	Many-to-Many
Batch size during training	128	128

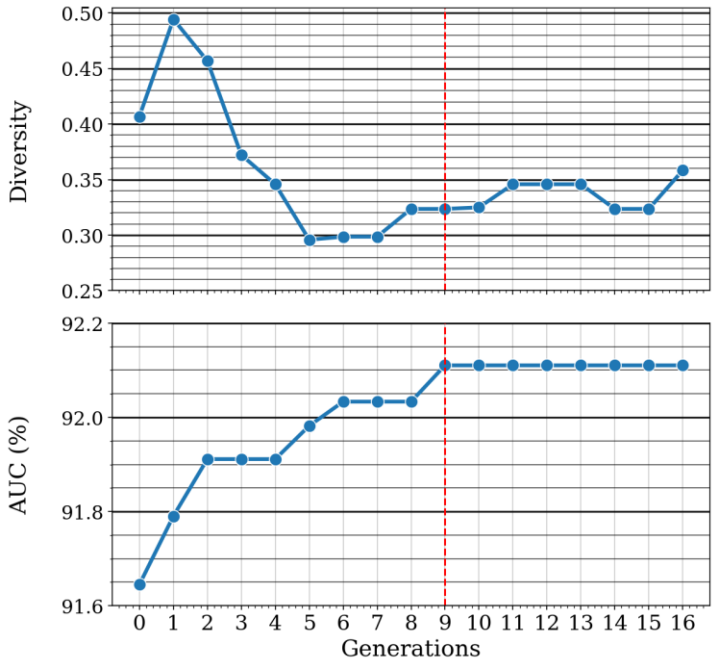


Figure 6.4 – Diversity and ROC-AUC during the optimization process of the REC architecture, using GA.

The following optimization step was finding the best text preprocessing steps for this architecture. The improvement assessment from the intermediate model to the fully optimized model (115, from Table 6.1) was focused on sensitivity and specificity. The ROI computed for ROC-AUC is -0.81%; however, in terms of TPR and TNR, the increment is 2.70%. When inspecting the metrics per class, this increment is due to the improvement of the TPR of the minority classes and the improvement of the TNR of the majority class.

Finally, the performance of the optimized model trained with the two balancing techniques proposed was compared with the CIL scenario, obtaining the values in Figure 6.5. The baseline model in this comparison was the CIL, which shows the highest accuracy due to the bias introduced by the majority class during training, and both sensitivity and specificity are negatively affected, as they are the two most sensitive to the imbalance of the classes. The ROC-AUC and F1-score get worse when using balancing techniques, which was not the expected behavior.

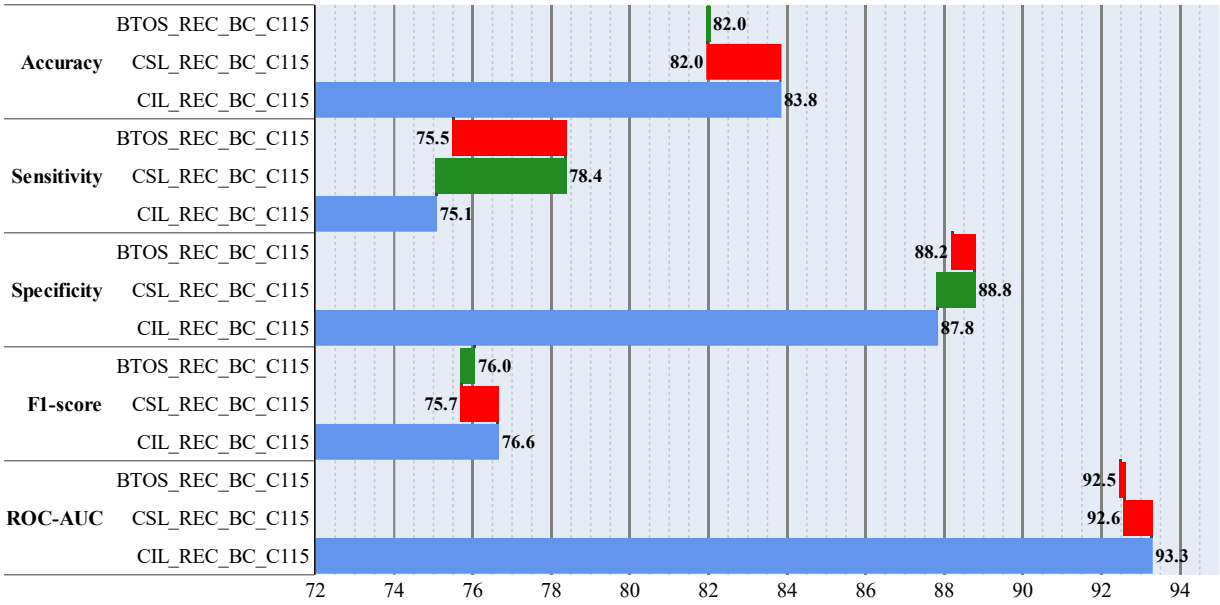


Figure 6.5 – Comparison of the performance of the three training approaches adopted to assess the effect of imbalanced classes using the fully optimized model (REC_BC_C115).

To better understand what provokes these variations, Figure 6.6 represents the sensitivity, specificity, and precision variations per class of the three models by subtracting the value per class from the macro average represented in Figure 6.5. The loss in the F1-score seen before can be acknowledged as a worsening of precision, which becomes very imbalanced when balancing techniques are used. Sensitivity becomes the most balanced among the classes when the BTOS scheme is used. In contrast, the CSL scheme provides the best macro sensitivity and acceptable variation among the classes, along with the best specificity. It is noticeable that CSL provides the best balance between the negative and mixed classes despite its low macro average.

The baseline CIL verifies the most significant variation among the classes, making it an unsuitable solution. On the other hand, both balancing techniques provide a comparable overall performance, despite the observed in Figure 6.5. As accuracy does not change among them, choosing one technique over the other must rely on the need for better precision or sensitivity. The obtained confusion matrices can be consulted in Appendix D.

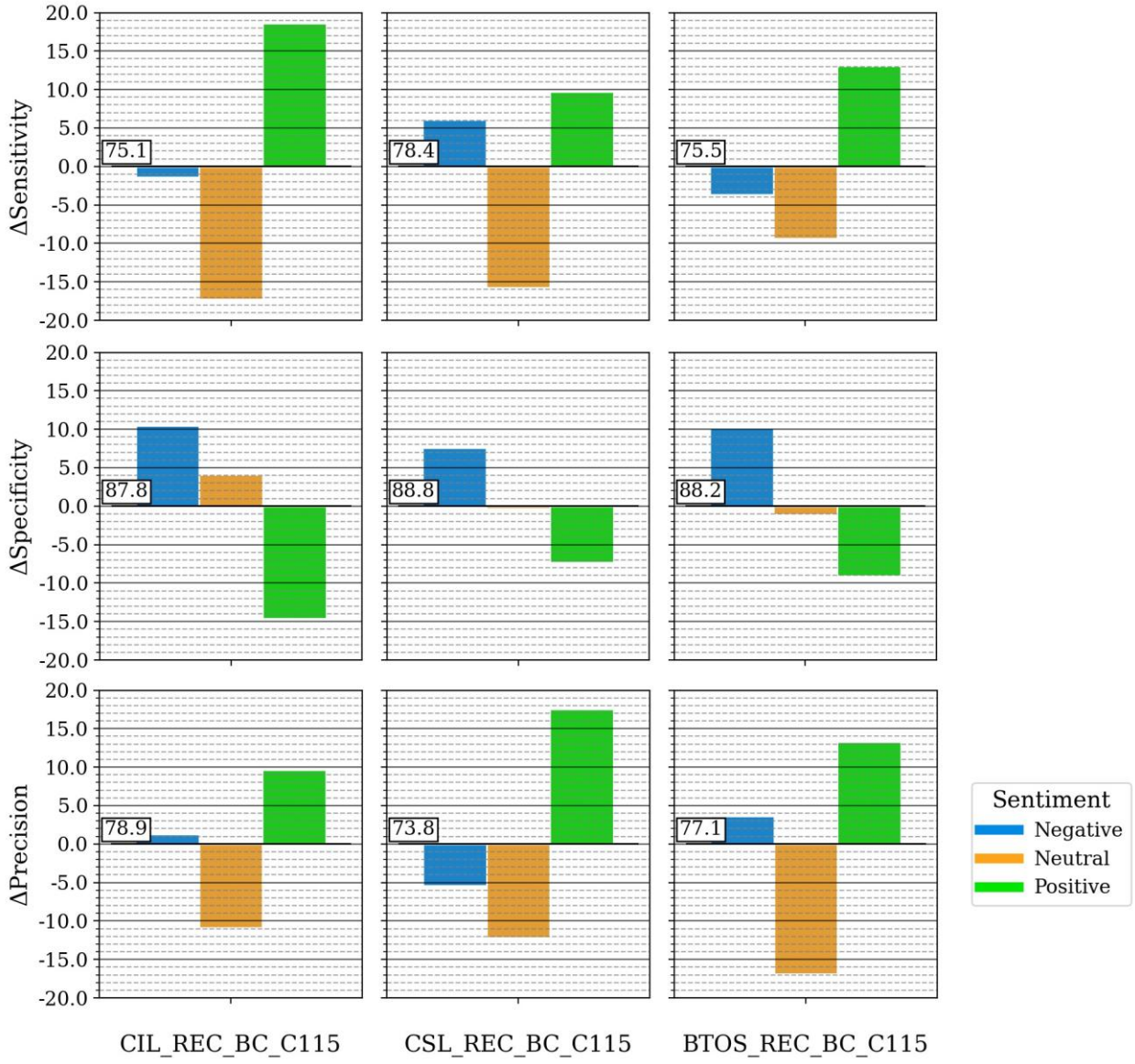


Figure 6.6 – Variation of sensitivity, specificity, and precision per class on the REC architecture among the three balancing scenarios, using model REC_BC_C115.

6.3. AREC model

As in the previous section, the AREC models' performances are compared here. The exact three comparisons were made, considering the models in Figure 3.8 relative to the AREC architecture. The automated optimization of the model's architecture and hyperparameters is illustrated in Figure 6.7, showing an increase in ROC-AUC, F1-score, and accuracy metrics while decreasing sensitivity and specificity. This might suggest that the optimization process preferred precision over sensitivity, and the class imbalance did not affect the specificity. The computed ROI is 8.3% in terms of ROC-AUC.

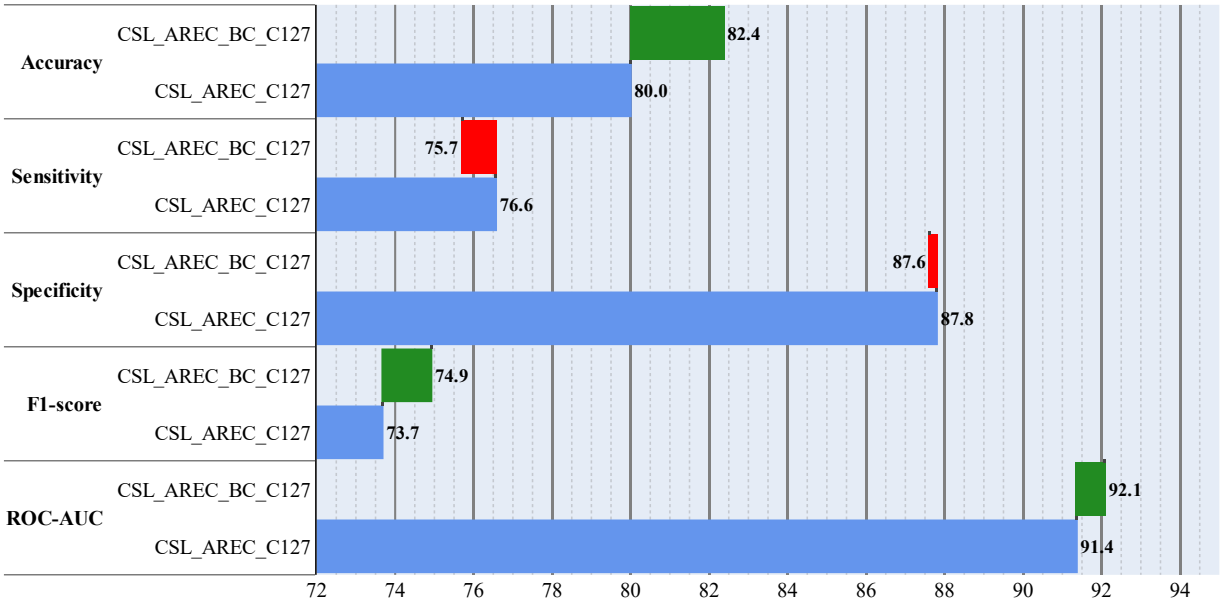


Figure 6.7 – Comparison of performance between the AREC baseline model and the intermediate AREC after GA optimization, where the aim was to improve ROC-AUC.

After seven generations, the GA yielded the hyperparameters detailed in Table 6.4, each generation with a population of fifteen chromosomes. The diversity and ROC-AUC outcomes are plotted in Figure 6.8, and it is evident that the BC emerged in the seventh generation, with a trend of increasing the PM, indicating potential for further optimization. However, diversity declined, and the BCs in the sixth and seventh generations no longer had attention mechanisms. Due to both the time constraint for optimization and the diminishing diversity, the BC of the fifth generation was chosen for further analysis, referred to as AREC BC. This chromosome also appears in the top 5 of the population of the seventh generation.

Table 6.4 – Optimized hyperparameters of AREC architecture after manual and GA optimization.

Hyperparameter	**Manual**	**Using GA**
Direction of the LSTM layers	**Bidirectional**	**Unidirectional**
Number of LSTM layers	2	2
Position of Self-Attention layers	After LSTM	After LSTM
Dimensionality of the model	512	512
Dimensionality of projections of Multi-head Self-Attention	**64**	**128**
Percentage of dropout of dense layers	0.1	0.1
Operation before the classification layer	**LSTM**	**Average Pooling**
Shape factor of classification layer	**0.5**	**1.0**
Activation function of the classification layer	**ReLU**	**Tanh**
Number of times to replicate the encoder block	0	0
Tokenizer	**Blankspace**	**Wordpiece**

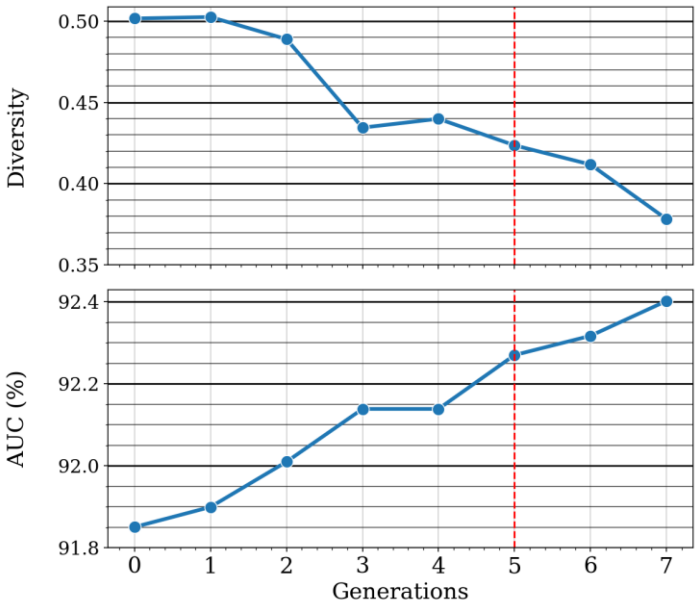


Figure 6.8 – Diversity and ROC-AUC during the optimization process of the AREC architecture, using GA.

The optimization of the text preprocessing steps registered a ROI of 4.68% from the default cleaning combination (number 127) to the optimized combination (number 101, see Table 6.2). All metrics showed a considerable improvement, denoting a consistent optimization. The gain per class is also consistent, showing that sensitivity and specificity per class are closer to the macro average. Therefore, the model has a more balanced performance among classes.

Finally, the balancing techniques show a similar outcome to the REC architecture: Figure 6.9 shows that both ROC-AUC and accuracy diminish when using balancing techniques. At the same time, BTOS swaps the improvement brought by the CSL technique. At the class level, CIL has the most significant variation in sensitivity and specificity (just like REC), while precision varies the least. On the other hand, BTOS has the worst performance in general, with similar variations among the classes, as shown in Figure 6.10. Therefore, it can be concluded that attention mechanisms can identify relevant aspects of each class without the need for augmented data.

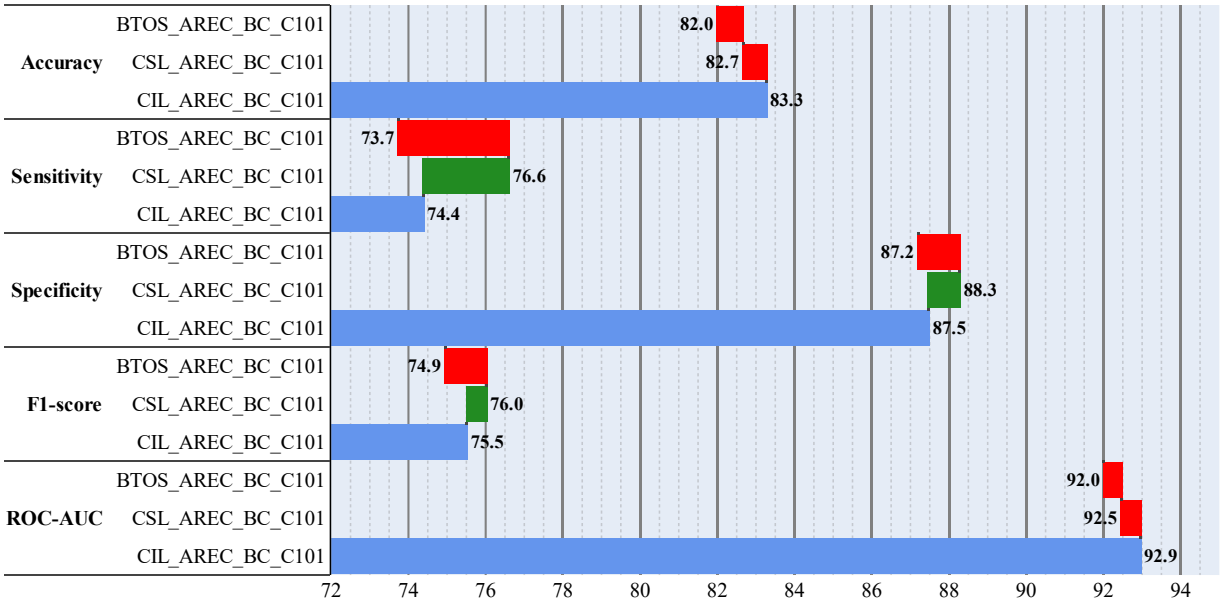


Figure 6.9 – Comparison of the performance of the three training approaches adopted to assess the effect of imbalanced classes using the fully optimized model (AREC_BC_C101).

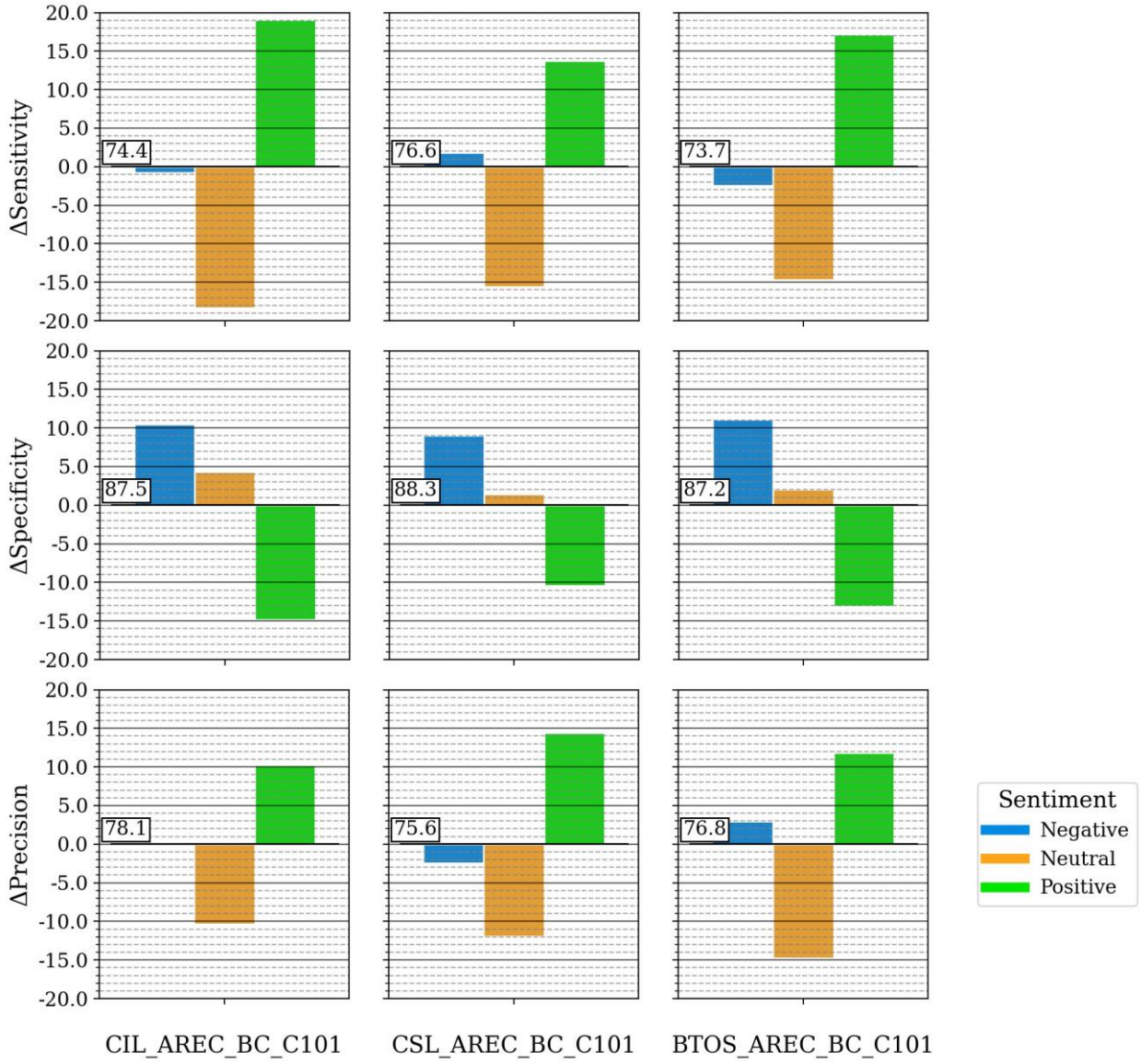


Figure 6.10 – Variation of sensitivity, specificity, and precision per class on the AREC architecture among the three balancing scenarios, using model AREC_BC_C101.

6.4. Comparison: REC vs. AREC

This section compares both optimized architectures to understand which approach is better for solving Zomato's dataset. First, a metric-based performance comparison is conducted using waterfall charts and checking the introduced variations per class. Then, three samples were extracted from the test partition, and the outcome of each architecture was compared in terms of the output of the encoder block to provide a degree of explainability to the REC and AREC models. For these comparisons, new models of REC and AREC architectures were trained using the data partitions of Table 3.1. Hence, the results are conditioned to these partitions, allowing a more concise comparison of the final product. Additionally, Table 6.5 summarizes the best performances using a 5-fold CV, including the ML baseline model, which was not tested under this validation scheme due to hardware limitations.

Table 6.5 – Summary of all models developed, evaluated under 5-fold CV techniques (NB model is the exception, as it was evaluated as explained in section 4.2).

Model	ACC (σ)	TPR (σ)	F1 (σ)	AUC (σ)
CSL-NB-MLbaseline	76.3 (-)	76.0 (-)	70.5 (-)	89.6 (-)
CSL-REC-TFIDF-C127	79.3 (0.27)	75.7 (1.18)	73.2 (0.20)	90.7 (0.22)
CSL-REC-C127	82.1 (0.39)	77.6 (0.72)	75.7 (0.41)	92.3 (0.28)
CSL-REC-BC-C115	**82.0 (0.41)**	**78.4 (0.79)**	**75.7 (0.62)**	**92.6 (0.13)**
CSL-AREC-C127	80.0 (0.90)	76.6 (1.21)	73.7 (0.91)	91.4 (0.39)
CSL-AREC-BC-C101	**82.7 (0.66)**	**76.6 (0.62)**	**76.0 (0.45)**	**92.5 (0.27)**

6.4.1. Performance metrics

Figure 6.11 shows the performance of the optimized REC architecture compared to the optimized AREC architecture, showing the performance is boosted in terms of sensitivity and specificity, which are good indicators, but the F1-score diminishes due to the model's precision. Moreover, the decrease in accuracy might suggest that this model is less biased towards the positive class.

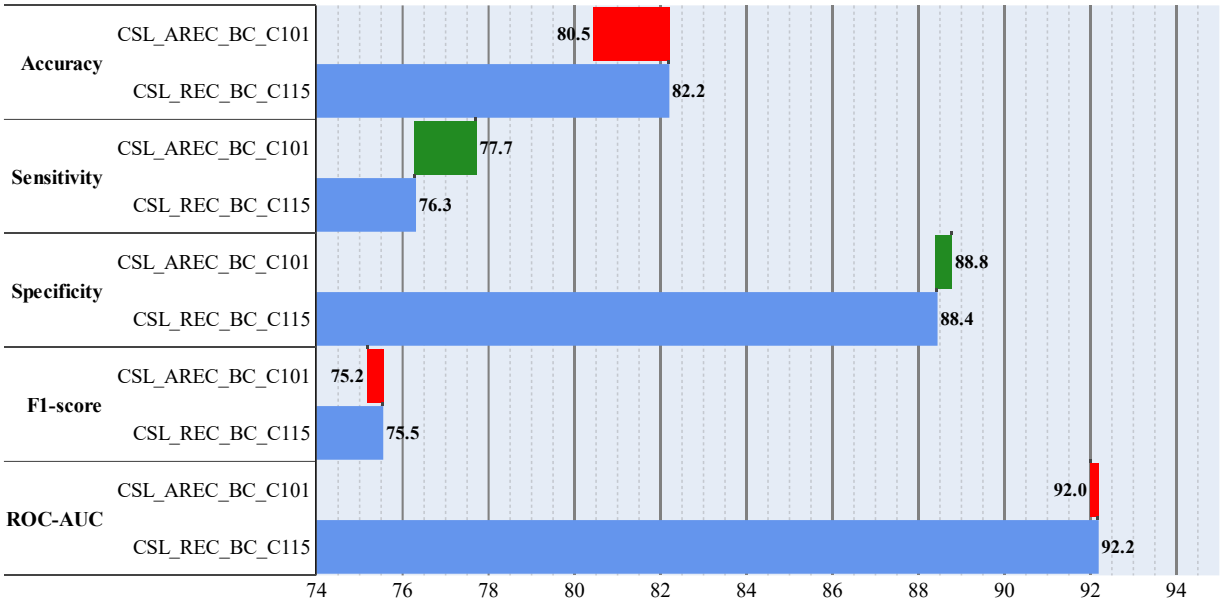


Figure 6.11 – Overall performance of the optimized REC and AREC architectures, trained on predefined partitions for inference.

Figure 6.12 shows the sensitivity, specificity, and precision variations to provide pinpointed understanding. Sensitivity is higher and less variable with respect to the macro average when the attentive model is used (the error is better distributed among the classes). Additionally, specificity also becomes slightly higher and better balanced between the mixed and positive classes, indicating that the attentive model is improving the classification performance of the mixed class. The increase of TNR of the positive class suggests that the bias towards this class is being attenuated. Finally, the decrease in the F1-score can be explained in terms of precision, which must be lower to compensate for the increase in sensitivity. The variation of precision per class shows a bias towards the positive class for AREC, while the same situation is verified for REC in less amount.

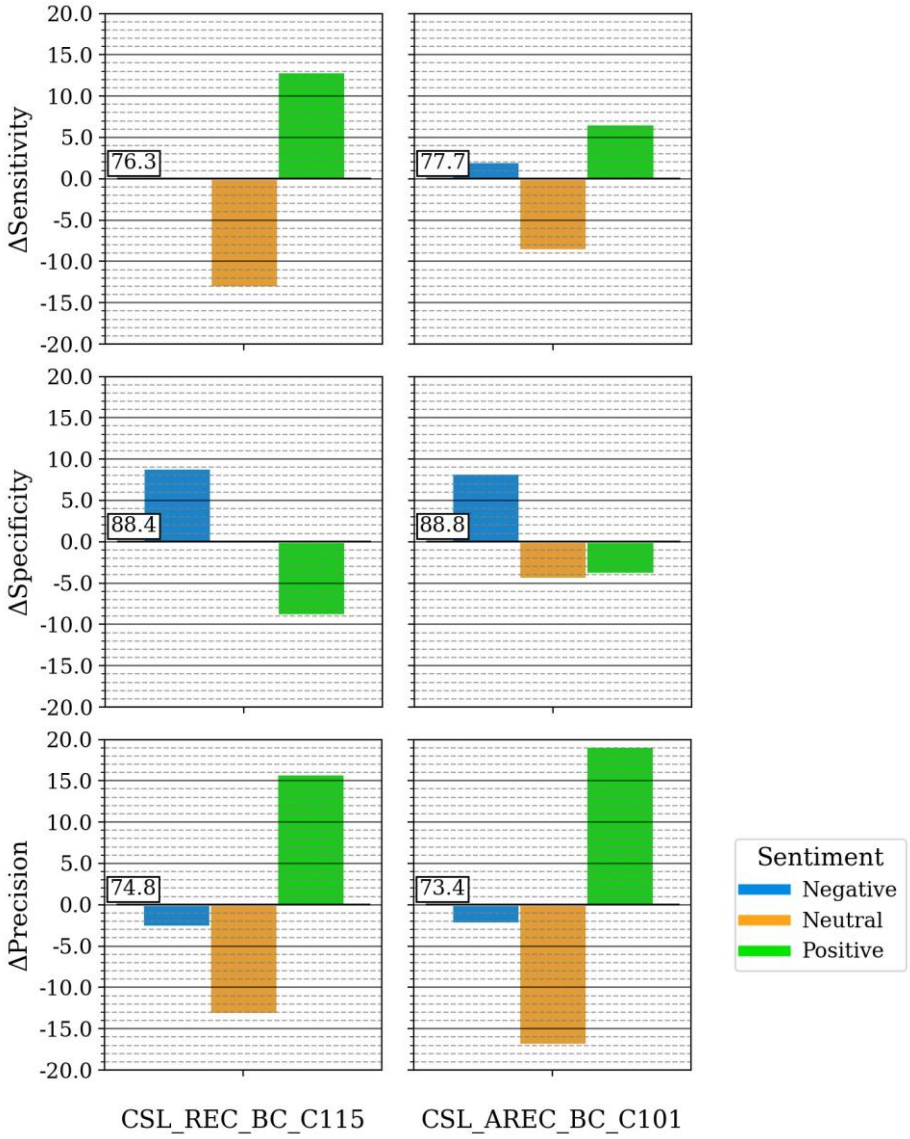


Figure 6.12 – Variation of the performance per class of sensitivity, specificity, and precision, comparing the optimized REC and AREC architectures trained on predefined partitions for inference.

In sum, the AREC architecture can better identify negative and mixed classes by decreasing the sensitivity of the positive class, which, in terms of restaurant applications, would have a better utility. When inspecting the specificity, its value for the negative class remains the same in both models. However, there is a better balance between mixed and positive classes, reducing the misclassifications of mixed classes, as REC typically assigns them to the positive. Finally, it can be concluded that the performances of the trained REC and AREC models are comparable. AREC is better at identifying the mixed class, a fact that can also be verified when inspecting the confusion matrices in Appendix D. The ROI of the AREC model over the REC model in terms of ROC-AUC fell in -2.0%, while in terms of sensitivity and specificity raised in 5.96% and 2.97%, respectively.

6.4.2. Examples

Three samples were extracted from the test partition to analyze the behavior of each model when negative, mixed, and positive reviews were passed to the model. Table 6.6 presents the three examples with their respective star ratings and labels. Each review is identified by its assigned label, with 0, 1, and 2 corresponding to negative, mixed, and positive, respectively.

Table 6.6 – Restaurant reviews extracted from the test partition of Zomato's dataset, with respective ratings (provided by users) and assigned labels.

Restaurant review	**Rating**	**Sentiment**
Péssimo, demoraram mais de 40 minutos para servir uma tosta e um prego. Desculparam-se com a quantidade de gente, mas a sala nem sequer estava cheia. Não recomendo, não tenciono voltar.	1.0	Negative
Precisava de um local para almoçar, e este foi o restaurante escolhido. A decoração não me agradou e a forma como é feito o pedido também não. A comida é razoável, mas não vale o preço que se paga.	3.0	Mixed
Um exemplo de bem servir, tanto nas iguarias, como no serviço atencioso e cuidado que nos faz querer voltar. Os meus parabéns pela qualidade que entregam, uma referência nesta cidade das 7 Colinas a ver o Tejo. Recomendo vivamente!	5.0	Positive

The following analysis aims to provide interpretability of the model's output by examining the context vector at each timestep after the encoder operation. First, consider the results of using the two models to forecast the sentiment of the example reviews presented in Table 6.7, where the forecast weights of the output layer of each model are presented. At first glance, it is possible to infer that both models are generally more sensitive to negative and positive sentiments, with the mixed sentiment being also correctly classified but with less confidence.

Table 6.7 – Sentiment forecast inferred using REC and AREC models on example reviews.

Review ID	**REC forecast**			**AREC forecast**		
	Negative	Mixed	Positive	Negative	Mixed	Positive
0	**0.9993**	0.0007	0.0000	**0.9771**	0.0218	0.0012
1	0.2353	**0.7493**	0.0154	0.3083	**0.6807**	0.0109
2	0.0014	0.0022	**0.9964**	0.0024	0.0157	**0.9819**

Focusing on the REC architecture, several tests were conducted in randomized samples from the test dataset, the selected samples (Table 6.6) being the most representative of the average behavior of the network. The graphical representation is done considering the context vector and directly plotting its distribution while adding a fixed factor β multiplied by the mean value of the distribution to increase the deviation from zero and facilitate the visibility. This factor is equal to ten to the power h, where h is a configurable parameter set empirically to $e/2$, where e is Euler's number, for the REC architecture. The observed behavior is the following:

- The forecast result is typically mixed when the context distributions are mostly centered on zero. Figure 6.13 shows this behavior in the example review number 1. However, some tokens skip this empirical rule, possibly due to the nature of the token. In this case, "não_agradou", "mas", and "não_vale" have negative connotations of their own, and so their distribution is offset from zero.
- When the context distributions have multimodal behavior, the forecast is typically positive. This mainly occurs when the model is pretty sure of this decision, i.e., when the output layer's positive weight is higher than approximately 0.99. Figure 6.14 exemplifies this comportment.
- Finally, when the model assigns a review to the negative class, it is harder to find any pattern in the distribution of the context vector. However, in some cases, the tokens that convey negative sentiments have larger mean values, as exemplified by Figure 6.15.

In general, the REC model focuses more on tokens that have distributions with more positive mean value, attending to the fact that when inspecting the tokens, they usually seem to have a sentiment according to the forecast. In most cases, the distributions are centered on positive values. Additionally, the registered behavior is more common when reviews are longer (a large number of non-zero tokens).

On the other hand, focusing on the AREC architecture, several tests were conducted in randomized samples from the test dataset, again maintaining the samples from Table 6.6 to allow a fair comparison. The distribution of the context vector resulting from the recurrent layers showed an alike behavior to the one encountered in Figure 6.14 for the REC architecture, finding that most of the distributions have negative mean values. In this case, β was configured with a parameter $h = 0.95$. As the difference between the distributions of each case is not significant, these charts were not displayed here. Conversely, a graphical representation of attention scores was provided using a heatmap. The AREC models use four attention heads; therefore, the heatmap was made by displaying the summation of these matrixes (each head is a squared matrix) normalized between zero and one, where zero indicates no relation between the tokens, and one indicates a maximum relation.

- Figure 6.13 exemplifies the attention scores when forecasting the mixed sentiment of a review, revealing the model is paying generalized attention to the words “agradou”, “mas”, “vale”, “preço”, and “paga”, while it is unusual the large values assigned to almost meaningless tokens such as “,” and “[UNK]”. In the case of “agradou” and “vale”, the word “não” was a relevant piece of information that was apparently not given enough attention. However, the forecast was according to the ground truth label.
- For the positive example (Figure 6.14), the attention scores are undeniably higher on tokens with positive intentions, such as “parabens”, “qualidade”, “referencia”, and the set of tokens “recomendo vivamente!”. Parts of the review, such as “atencioso” or “querer voltar”, are expected to be relevant to the model but are instead not being assigned high attention scores.
- Finally, when the assigned sentiment is negative (Figure 6.15), it is seen a better behavior of the attention weights, indicating a relation between “tenciono” and “mas”. However, in this case, most attention is apparently given to punctuation marks.

Apart from the attention scores shown in the examples, finding large weights at the beginning of the sequence is not atypical. However, this effect might be explained due to the non-bidirectional LSTM layers used for encoding.

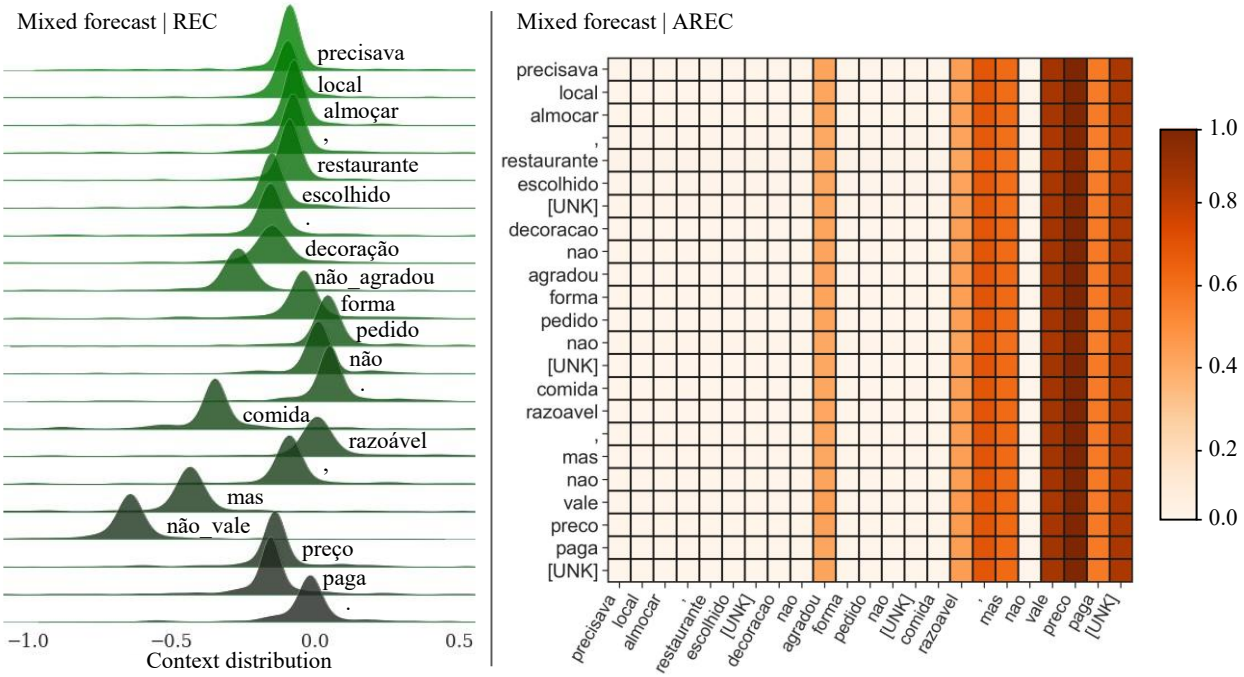


Figure 6.13 – Context distribution (REC) and attention weights (AREC) when evaluating the restaurant review assigned to a mixed sentiment.

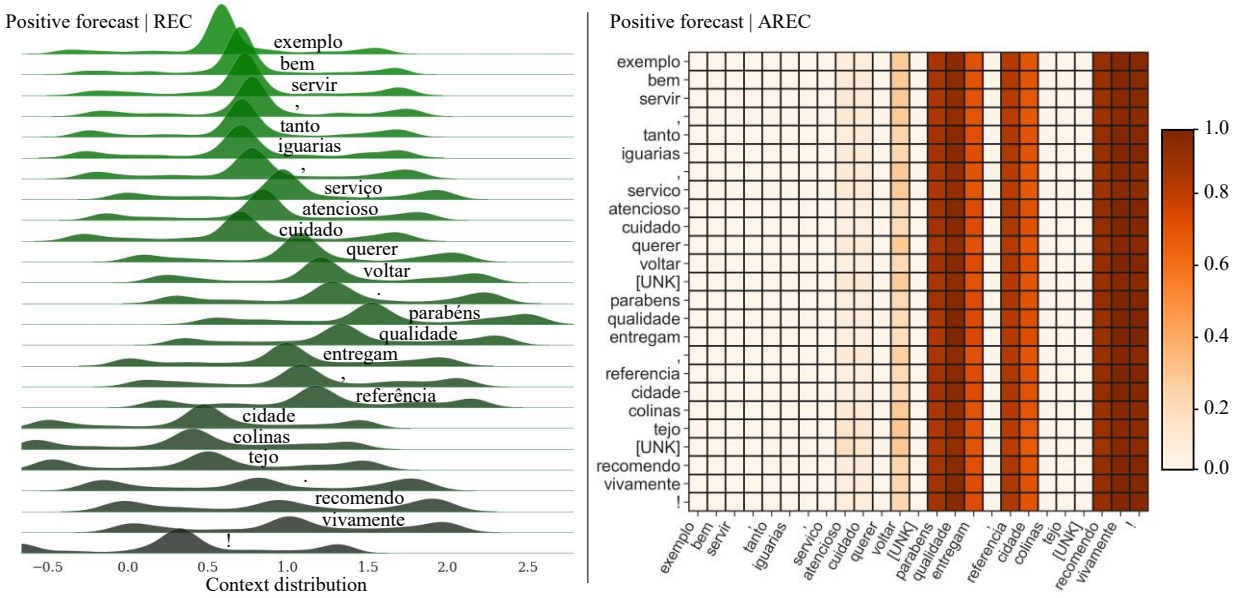


Figure 6.14 – Context distribution (REC) and attention weights (AREC) when evaluating the restaurant review assigned to a positive sentiment.

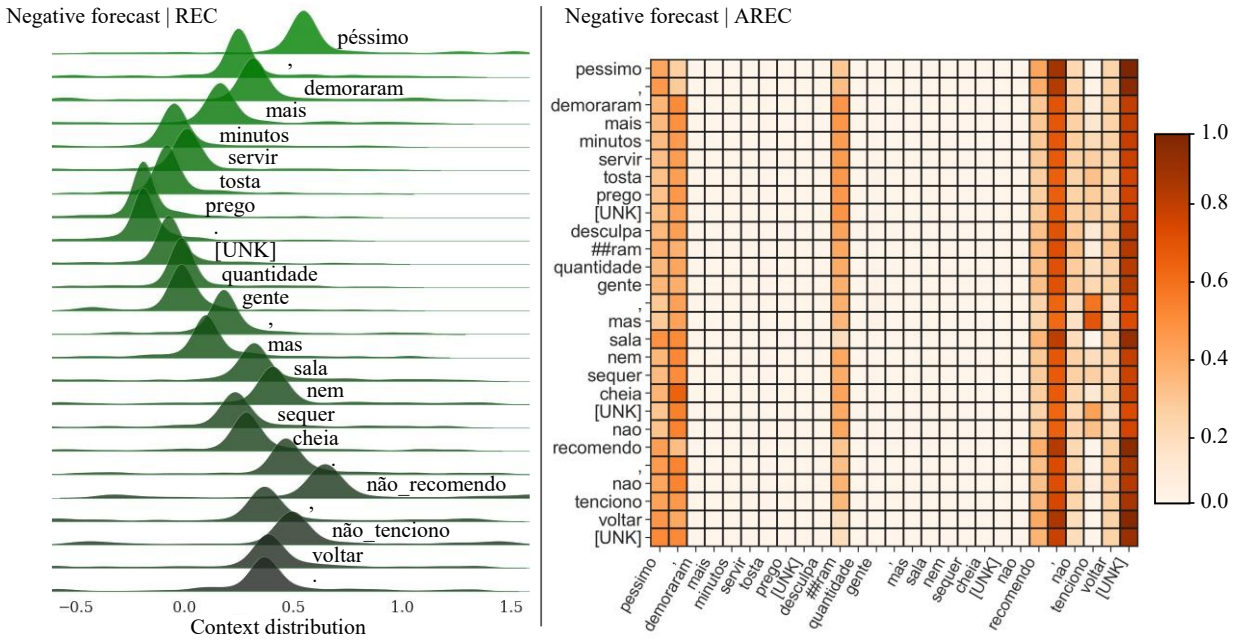


Figure 6.15 – Context distribution (REC) and attention weights (AREC) when evaluating the restaurant review assigned to a negative sentiment.

6.5. Key remarks

This chapter offered a fair comparison among the methodologies applied during the study, accompanied by a discussion and reasoning behind the obtained behaviors. First, the selection of the best input representation scheme plus the optimal set of preprocessing steps showed embedded representations, particularly Word2Vec, provide better results when compared with BoW techniques such as TF-IDF. Additionally, the customized cleaning process significantly improves sensitivity and specificity per class, becoming better at identifying the minority classes, with the drawback that the selection process is time-consuming.

The model architectures found using the GA resulted in a slight improvement in the performance of each of them. REC improved on every relevant metric, featuring a ROI of 4.41%, while AREC showed a better performance on the F1-score and AUC, with a ROI two times higher than the obtained before. We acknowledge that both optimization procedures were possibly not yet fully completed (although likely almost completed) due to the tendency encountered on diversity values, but time constraints imposed to stop the optimization process. Regarding the balancing techniques, BTOS mostly benefits the mixed class, while CIL naturally benefits the positive class. On the other hand, CSL can find a compromise among the classes, being that the mixed sentiment is the most difficult to detect by both REC and AREC architectures due to its nature (features both positive and negative sentiments). AREC demonstrated a lower overall performance but was more capable of discriminating between the mixed and positive classes, reducing the classifier's bias.

Overall, the attempts for interpretability of the REC model illustrated some relation between the mean value of the distribution of the context vector per timestep (or token), suggesting more negative mean values are typically more relevant tokens. On the other hand, AREC showed similar but less noticeable behavior. Moreover, the attention scores were considered more relevant to analyze. They showed the capacity to find relevant tokens within the reviews, being more visible when dealing with positive or mixed classes, possibly due to the data imbalance. Attention mechanisms are usually trained over large datasets, which holds this conclusion. Still, results are comparable to those obtained with the REC model and might be improved by training with more

data or pretraining the model on semi-supervised tasks. More efforts must be made to enhance interpretability, and the analysis portrayed here may serve as a first step, as some intuition can be extracted from context vector and attention score matrices. This interpretation was aimed to be very visual, facilitating the understanding among users of areas different from ML.

A noteworthy aspect was identified in the evaluation of the models. During model training, accuracy and ROC-AUC had a proportional relationship. This occurred because of misconfiguration when computing the AUC for multiclass scenarios. Furthermore, this observation suggests that for a predictive problem involving three classes, the ROC-AUC metric computed using the OVR approach and macro average may not provide the most informative insights. Consequently, greater emphasis was directed towards sensitivity and F1-score in the evaluation, along with considering the possibility that PRC-AUC might offer an alternative evaluation metric with a more insightful evaluation for this problem.

Upon inspecting specific samples, it was observed that some of them were mislabeled. The model's response was more satisfactory in the identified cases than in the original labels. Two conclusions can be drawn from this: ratings may sometimes lack credibility as they are subject to the customer’s bias, and the model can identify complex relationships between words and can often correctly detect the right sentiment.

Nevertheless, a more in-depth study should be conducted to assess the original labels' quality. It is suggested to revise mechanisms for sample labeling to ensure correctness, raising questions about the usefulness of consumer ratings. On the other hand, the lack of labeled datasets necessitates adopting this solution, which can be considered beneficial to a certain extent, as proved in this work.

7. Hardware implementation

This chapter aims to demonstrate the practical implementation of the developed models. A proof-of-concept scenario was proposed that involves recording audio samples and performing SA in real time. Two low-cost hardware alternatives were compared to select the most suitable option, also considering the availability of hardware. This chapter was elaborated under the Ingress@ program, which aims to stimulate young people to maintain continuous and dynamic professional development, promoting qualification and integration into the job market and enhancing cooperation between public and private entities.

The development of this solution included several steps. Initially, one of the models developed in chapter 4 was trained and deployed onto the designated device. Then, alternatives for audio transcription were explored since the models developed rely on textual input. Practical issues were treated regarding the recording method and ambient noise. All steps were integrated, and a set of samples extracted from Zomato's dataset was tested, comparing the model's actual performance on plain text and the performance when used on the proof-of-concept scenario.

Consider a scenario where this application is applied in a restaurant setting. After a customer completes their dining experience, a robot can approach them and inquire about their opinion on the experience. Instead of the customer having to provide a numerical rating, the model implemented in the application takes care of this task, allowing a consistent measurement of the experience, as it was noticed that people have different biases when giving a rating. In the subsequent sections, these steps are discussed individually, providing detailed explanations of the tools used and the code developed.

7.1. Hardware specifications

It is ideal to have a device that can process and give feedback to the consumer in real time without relying on an internet connection. This is referred to as edge computing, defined as the deployment of computing and storage resources at or near the location where data is produced rather than transmitting the data to places with computation power [89]. A Single-Board Computer (SBC) is a complete computer built on a single circuit board, with microprocessors, memory, input/output, and other features required for a functional computer. They are commonly used for educational purposes or embedded applications requiring portability and low-power computing.

Baller et al. [90] published a comparative review in terms of power consumption, inference time, and accuracy for five SBCs: Asus Tinker Edge R, Raspberry Pi 4, Google Coral Dev Board, Nvidia Jetson Nano, and Arduino Nano 33 BLE. They highlight that their results are dependent on the model size, quantization, framework, and anticipated number of inferences per time. They conclude that for sporadic AI computation (relevant to this project), Nvidia Jetson Nano and Google Coral Dev Board are the most power efficient. Moreover, Raspberry Pi 4 performed better than Jetson Nano for larger models, also consuming less power than Asus Tinker Edge R and Google Coral Dev Board. The choice is a trade-off that will always depend on the application requirements and characteristics.

The Raspberry Pi was first released in 2012 and is a low-cost SBC widely used in education, hobbyist projects, and edge computing. The Raspberry Pi 4 Model B (Pi4B) is the first of a new generation of Raspberry Pi computers supporting more RAM (up to 8GB) and significantly enhanced CPU, GPU, and I/O performance. Requires a good quality USB-C power supply capable of delivering 5V at 3A. If attached downstream USB devices consume less than 500mA, a 5V, 2.5A supply may be used. Pi4B provides 28 BCM2711 GPIOs available through a standard

Raspberry Pi 40-pin header. The Raspberry Pi can run various operating systems and has a large community of users, making it easy to find support and tutorials. Additionally, the availability of Pi4B is guaranteed until at least January 2026.

Considering the aspects discussed above and the availability of devices in the laboratory, the Pi4B was used for the experiments in this section alongside a ReSpeaker 2-Mics Pi HAT component for the speech-to-text operation at the input interface for the consumers' speech.

7.2. Deployment onto Pi4B

The selected model for deployment was CSL_REC_BC_C115, trained on the partitions of Table 3.1. This model was selected due to its size, as it has approximately three times fewer parameters than the AREC architecture, which also has competitive performance. During training, a checkpoint of the model was saved after the end of each epoch to manually select the best model considering the ROC-AUC of both training and validation partitions instead of considering only the performance of the validation set since it can be noticed in Figure 7.1 an early peak in performance on the validation set. In contrast, on the training set, the performance was still poor. The model was evaluated on the test dataset and showed an accuracy of 81.8%, ROC-AUC of 92.2%, F1-score of 75.5%, and a sensitivity of 77.4%.

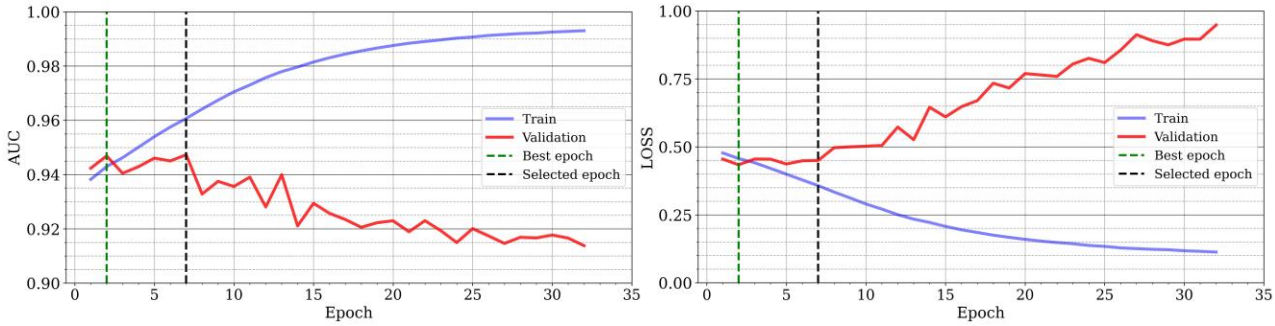


Figure 7.1 – Learning curves of the CSL_REC_BC_C115 model trained for deployment.

Some minimal requirements were needed to run the model on the Pi4B. *Regex*, *emoji*, and *TensorFlow* libraries were required to run the Python class *SentimentClassifier* (loads and runs a *Keras* saved model) and the *textCleaning* function that performs text preprocessing as the input pipeline to the model. All these libraries were installed using the pip3 command, including *TensorFlow*. The project containing the files to run the model can be found in reference [91].

The ***SentimentClassifier* class** allows the usage of an SA model trained and saved under the conditions described in section 4.6. It is configured to load any model by specifying its relative or absolute path to the file system so long as it is available. The *Keras* model is directly loaded from the SavedModel directory, while the tokenizer is generated from the vocabulary stored together with the model. This class defines the `prediction` method that runs inference over a single phrase or a list of phrases and returns the weights of each forecast in the format "[Negative, Mixed, Positive]".

The ***textCleaning* function** defines the preprocessing steps according to the sequence used during training. In this case, the combination number 115 is used (refer to subsection 4.3.1 and section 6.1), which:

- fixes formatting errors,
- converts to lowercase,
- removes emojis,
- removes stopwords,

- creates rule-based bi-grams,
- removes numbers,
- removes duplicated blank spaces.

An instance of the *SentimentClassifier* class can be easily integrated into other pieces of software, which was exploited in later sections. The Python script that implements these components can be referred to in Appendix C. In addition to this application, the *SentimentClassifier* class and the *textCleaning* function were delivered to the Zomato team for them to use the models and conclude the RRSO project.

7.3. Speech-to-text task

Several APIs can perform speech recognition from audio files (such as WAV and MP3 formats), and most of them can be accessed through Python using the *speech-recognition* library [92]. It supports several speech recognition engines, such as Google Speech Recognition, Microsoft Azure Speech, IBM Speech to Text, and others (online services), such as CMU Sphinx, Vosk API, or OpenAI Whisper (offline tools). In this application, it is desired to rely on offline tools to ensure complete reliability on the Pi4B system. Also, being able to select smaller models for the speech-to-text task is crucial for hardware implementation since it is a resource-constraint environment.

This section summarizes the steps to configure the speech-to-text task for subsequent experiments, including the selection and configuration of the microphones employed, the selection of the transcriber algorithm, and the characterization of the effects of noise from the transmission channel on the system's performance.

7.3.1. Configuration of ReSpeaker 2-Mics Pi HAT

The ReSpeaker 2-Mics Pi HAT is a dual-microphone expansion board that can be used with the Pi4B and is designed for AI and voice applications. It utilizes the WM8960 stereo codec, known for its low power consumption. The microphones communicate with the Pi4B through the serial bus interface standard I2S (Inter-IC Sound), enabling PCM audio data transmission between integrated circuits, with a maximum sample rate of 48 kHz. The ReSpeaker project is provided by Seeed Studio. The ReSpeaker 2-Mics Pi HAT is a discontinued product, but Seeed Studio still provides documentation and "Get Started" tutorials [93], which were helpful in this project.

The required drivers were installed to set up the microphones. Also, the Python library *PyAudio* was required to configure an instance of the microphones. The process was parametrized to record from both microphones, with a sample rate of 48 kHz and a chunk size of 1,024, as suggested by the documentation. Additionally, the Python library *Scipy* was used to handle audio files.

After preliminary tests, it was noticed that the two channels exhibited significant variations in amplitude, sometimes leading to the saturation of the microphones. To mitigate this issue, both channels were merged by averaging the amplitude value of each channel sample-wise, employing Algorithm 3. The computation was weighted using the maximum amplitude recorded by each channel to ensure a balanced amplitude level for sound captured from each microphone. The coefficients were computed using an auxiliary function F, defined as

$$\mathrm{F}(I_{left}, I_{right}) = \frac{I_{left} - I_{right}}{2I_{máx}} + \frac{1}{2}, \tag{7.1}$$

where $I_{máx}$ is a constant that represents the maximum value that a sample can have, and I_{left} and I_{right} are the maximum values registered in the left and right microphones, respectively.

Algorithm 3 Balance microphones

Input: *input_path* (string; path of the recorded WAV file)

Output: *output_path* (string; path of the balanced recorded WAV file)

1: *data* ← Read WAV file from *input_path*

2: Define channels' coefficients from *data*

2.1. Extract wave information

I_{max} ← $2^{15} - 1$

N ← length(*data*)

2.2. Compute coefficients

start ← int(N×5%)

I_{left} ← *data*[*start*: , *left_channel*]

I_{right} ← *data*[*start*: , *right_channel*]

coefficients = [1–F(I_{left}, I_{right}), F(I_{left}, I_{right})]

3: Compute the average channel

data ← *data* · *coefficients*

write_wavfile(*output_path*, *data*)

7.3.2. Voice transcription

OpenAI released the Whisper model in 2022, a transformer-based architecture trained on 680 thousand hours of speech data, where only 17% (117 thousand hours) correspond to non-English audio (data from 98 different languages) [94]. The usage of this model was motivated by the simplicity of its usage and the possibility of selecting the model size that better suits the application, although the processing time is not optimal and varies depending on the length and quality of the audio. The bigger model that the Pi4B was able to run is the *base* model, with 74M parameters, and requires VRAM of approximately 1GB. Whisper performance varies depending on the evaluated language, with the Portuguese language being the fourth best performant language, with a word error rate of 4.3%, while English has a rate of 4.2% [94].

To use the Whisper model for inference, a person's voice is first recorded and stored in the form of a WAV file, which is then passed to the transcriber. To improve the model's performance and somehow bias its behavior, the language to be transcribed was preset as Portuguese, and an initial seed was passed to provide the model with context about the contents of the WAV files. The seed was set to "opinião sobre experiência num restaurante, comida, espaço, atendimento, serviço, preços. Zomato".

7.3.3. Channel noise

In a real-world scenario, noise represents a crucial factor that can cause a detrimental impact on the performance of the whole system, as it directly influences the quality of data collection. Minimizing the noise in the recorded sound was not a fundamental step in this proof-of-concept application. However, two experiments were conducted to evaluate the noise's impact. Initially,

the system was tested without the transmission channel, ensuring a noiseless environment. Subsequently, the system was tested with the transmission channel, introducing ambiance noise. The influence of the noise on the system was evaluated by comparing its performance following these two tests.

7.4. Experimental setup and results

An experimental setup was prepared to simulate a real scenario in which this system might be employed and its performance examined.

Approximately 300 uniform samples from the test dataset were used to generate voice samples using the Text-to-Speech API from Google Cloud Platform [95], which already provides a Python script to simplify the configuration of the task, being parametrized with the string "pt-PT-Wavenet-C" to define the language and the kind of voice to be used. The voice selection was random, whichever would be enough for demonstration purposes. The number of test samples was limited because this Google service is not for free. However, it would be sufficient to evaluate the performance and would also facilitate the comparison between the original text and the recorded/transcribed text.

The 300 samples were already labeled as negative, mixed, or positive based on the rating provided by users, which was used as the overall Ground Truth (GT). The same samples were consumed by the *SentimentClassifier* model to assign labels based on text analysis. These assigned labels served as GT for the experimental setup (referred to as Model).

The experiments were conducted inside a partially unsounded chamber (to limit the amount of environmental noise) with some acoustic panels to reduce the sound reflection. The Pi4B is connected via Bluetooth to a speaker and iterates over the list of voice samples to be reproduced. The 2-Mics Pi HAT then captures the audio. After each audio is captured, the Pi4B runs the transcription model followed by the *SentimentClassifier* model, and results are stored in a CSV file (transcribed text and assigned label) to evaluate the performance. It also included a small screen to monitor the process. Figure 7.2 illustrates the experimental setup. In practice, this setup had to be altered to record all samples a priori and then pass these recorded versions to the Pi4B due to the inability to play and record a sound simultaneously. This change disregarded the usage of the BlueTooth connection. This did not affect the outcome of the experiment.

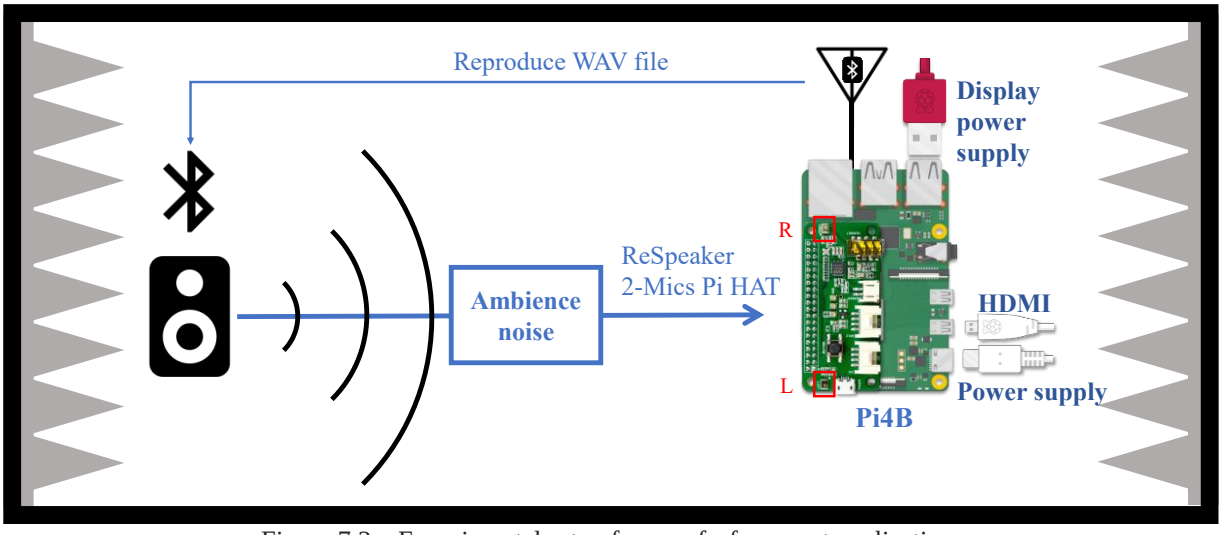


Figure 7.2 – Experimental setup for proof-of-concept application.

The results after the two experiments are summarized in Table 7.1 and Table 7.2, including accuracy and ROC-AUC metrics. These metrics were sufficient to study the performance in this

case because each label has the same number of samples. Column Δ shows the average difference of the two metrics between the current row (n) and the previous row ($n-1$):

$$\Delta = \frac{(ACC_n - ACC_{n-1}) + (AUC_n - AUC_{n-1})}{2}. \quad (7.2)$$

The Model vs. GT scenario refers to the performance of the model on the test dataset. Table 7.1 presents the results of the first experiment, where the audio samples generated by Google API are directly passed to the model without being recorded by the Pi4B (referred to as the WAV experiment), while Table 7.2 shows the results of the second experiment, where the audio samples generated by Google API are reproduced by a speaker and recorded by the Pi4B (referred to as the Voice experiment). When the comparison is made against the GT, the overall performance, including the error of the *SentimentClassifier* and the Whisper model, is considered. Finally, when it is done against the Model, the performance considers only the error of the Whisper model.

Table 7.1 – Comparison of performance in the experiment without considering the transmission channel (noiseless).

Scenarios	**Accuracy (%)**	**ROC-AUC (%)**	**Δ (%)**
Model vs. GT	74.3	89.9	–
WAV vs. GT	74.3	89.7	−0.1
WAV vs. Model	88.3	97.7	11.0

Table 7.2 – Comparison of performance in the experiment that considers the transmission channel (ambient noise).

Scenarios	**Accuracy (%)**	**ROC-AUC (%)**	**Δ (%)**
Model vs. GT	74.3	89.9	–
Voice vs. GT	68.0	85.6	−5.3
Voice vs. Model	82.0	94.4	11.4

The experiments revealed that performance remains nearly unchanged in the absence of noise. However, introducing noise leads to an average decrease in performance of approximately 5%. On the other hand, consistent variations in performance were observed in both experiments when the Whisper model was exclusively considered, suggesting that just the transcription is responsible for the 14% error.

7.5. Key remarks

The experiments allowed us to assess the performance of a REC model when launched to a field application and to indirectly evaluate the performance of the Whisper model in terms of a downstream classification task. A limitation encountered during the conversion of text to speech using the Text-to-Speech API from Google Cloud Platform involved grammatical mistakes made by users within the initial dataset. While this is a relevant aspect that might decrease the system's performance, addressing this problem was out of this project's scope due to its experimental nature. This would not be an issue in real scenarios, as data is directly collected from users in speech format.

This chapter illustrates the steps that must be followed to deploy an SA model to an edge device. Many limitations were encountered during this procedure; some were solved, while others were accepted for these demonstrations. Such is the case of the inability of the Pi HAT to record

and play WAV files simultaneously, but it was considered an experimentation issue that would not be present as a problem in a real-world application. Another concern is the amount of noise that can be captured by the microphones, which can be improved by using better hardware or utilizing digital filters. The last approach was considered but resulted in a bad outcome. However, the algorithm to balance the microphones resulted in a nice improvement in the quality of the sound.

The practical usability of the developed models is varied, from social media monitoring to product feedback analysis. This is one possible application and a relevant proof-of-concept that may improve the data treatment and interpretation of the reviews provided by consumers since this system will classify the reviews under the same conditions (based on the training data), which may potentially eliminate outliers when consumers provide manual ratings.

8. Conclusion

This work was framed under the RRSO project and pursued the possibility of employing NLP and DL tools to perform SA on labeled restaurant review data provided by Zomato Portugal. The primary techniques investigated were recurrent and attention mechanisms, which previously showed state-of-the-art results in the SA task. The study involved an evaluation of the developed models using a suitable set of metrics (independent from the training dataset and carefully selected for the case study) for future researchers' reference.

The examination of Zomato's dataset provided insights that served to parametrize the REC and AREC models, namely the vocabulary size and the number of tokens to consider in the input sequence. However, as the model input is limited to 100 tokens, there are still many cases where relevant parts of a user review are cut from the input, limiting the ability of the model to forecast the sentiment reliably. Potential challenges were identified by inspecting the text data, and the text preprocessing and cleaning steps were defined to suit this data scenario. The choice of how textual data is represented plays a crucial role in the SA model's performance. Several methods were explored, with dense embedding vectors, specifically Word2Vec, emerging as a preferred option due to their ability to capture semantic information effectively. The evaluation process of word embeddings remained subjective, potentially introducing bias. Additionally, model effectiveness is closely tied to factors like data domain, text preprocessing techniques, and the complexity of reviews, which can vary in real-world applications.

Regarding the definition of the REC and AREC architectures, baseline model configurations drew inspiration from prior research and experimentation and showed that there is no one-size-fits-all architecture to address the challenges effectively. The architecture optimization suggested that bidirectionality was not required in recurrent encoders when attention mechanisms were applied. Grid search was employed to optimize the text preprocessing steps, indicating that different procedures yield slightly different results, with larger cleaning schemes often producing the best performance. GAs were utilized to fine-tune the model's hyperparameters and identify optimal configurations, demonstrating effective optimization.

It is worth noting that these optimization processes can be time-consuming, and the improvements might sometimes be subtle. Moreover, the optimization of the AREC architecture was not fully finalized due to time constraints, potentially impacting the results. The intermediate results, such as the RNN's hidden states and multi-head self-attention scores, were inspected to try to provide a level of explainability to the models, revealing that tokens with shifted distributions are generally relevant to the final classification outcome, while higher attention scores match the behavior of the shifted distributions seen in RNNs.

A proof-of-concept implementation was carried out to validate the efficacy of the proposed models and optimization techniques, providing practical insights about the model performance. A Pi4B was chosen to deploy the model on a scenario where the data is gathered in voice format and needed to be transformed into text to be processed by the REC model. Results were satisfactory and allowed us to measure indirectly the effect of ambient noise in a controlled scenario. Although the proof-of-concept proved to be realizable, there are still some limitations at the noise level and processing speed, which could be improved by using more sophisticated hardware. In general, the deployment on an edge device proved feasible, with *Tensorflow* facilitating the process.

This research provided some of the foundations to publish, as co-author, the article "Sentiment Analysis in Portuguese Restaurant Reviews: Application of Transformer Models in Edge Computing" [96] in the journal Electronics, made available on January 31st, 2024.

8.1. Research questions

Is it possible to develop high-accuracy sentiment analysis models without relying on LLMs? The whole optimization process aimed to prove if it is possible to build competitive and domain-specific models with high performance without relying on pre-trained LLMs like ELMo and BERT. Results showed a viable level of performance, helped not only by the optimization process but also by the training configuration, namely the usage of the CSL scheme. The BTOS scheme is appealing and typically used by researchers, but the actual performance were suboptimal. This suggests that fine optimization in multiple levels of the training process results in good performance for a specific practical scenario.

Can a model's structure optimized by a heuristic algorithm provide a significant improvement over baseline models in NLP? GAs provide significant improvement when combined with other optimization steps. The greater limitation of discrete GAs is the manual definition of the optimization space, which may lead to bias or hinge optimal configurations. The ROI using GA was 4.4% and 8.3% in ROC-AUC for REC and AREC architectures, respectively. Notably, the optimized configuration of REC architecture was similar to the baseline architecture configured empirically. On the other hand, the architectural changes of AREC were relevant, suggesting that attention models are harder to configure. The performance gap between optimized REC and AREC architectures is hard to tell and will always depend on the final user since a trade-off between precision and sensitivity must be directed to benefit some of the classes equally.

8.2. Future work

It is essential to address several critical aspects to enhance the performance of the SA model. Two potential avenues for exploration of the scalability of the model include the concatenation of 100-token chunks for sequential evaluation, followed by the usage of pooling layers or a weighted voting mechanism to aggregate the results. This approach avoids the need for retraining the model. Alternatively, we may consider retraining the network while providing access to the recurrent encoder's initial and last hidden and cell states. This way, information about the previous 100-token chunk can be provided without significantly changing the network architecture. Also, dimension reduction algorithms can be employed on initial embeddings to get fixed-size representation, disregarding the initial and variable number of tokens.

This research was limited to RNNs and self-attention mechanisms. Future work might explore other configurations, such as combining CNNs and RNNs for feature extraction and more advanced attention mechanisms for encoding. Furthermore, it is imperative to prioritize the model explainability in future research. Despite the attempts, they remain a black box. Achieving transparency and interpretability is essential, particularly for real-world applications. The next research step is to perform SA at a deeper level of granularity, exploring sentence and aspect levels.

Regarding the proof-of-concept, explore the potential benefits of incorporating digital filters and using advanced hardware or clusters to improve model performance, particularly in inference time and in the presence of noise. Inference time can be improved using USB hardware acceleration devices that provide a dedicated TPU or Visual Processing Unit (VPU) for graph computation. Noise reduction holds the potential to boost the overall model accuracy by at least 5%, making it a worthwhile avenue for further investigation.

References

[1] F. Chollet, “Chapter 1: What is deep learning?,” in *Deep learning with Python*, Shelter Island, New York: Manning Publications Co., 2018, pp. 3–24.

[2] L. Zhang, S. Wang, and B. Liu, “Deep Learning for Sentiment Analysis: A Survey,” *WIREs Data Min. Knowl. Discov.*, vol. 8, Jan. 2018, doi: 10.1002/widm.1253.

[3] T. Young, D. Hazarika, S. Poria, and E. Cambria, “Recent Trends in Deep Learning Based Natural Language Processing [Review Article],” *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, Aug. 2018, doi: 10.1109/MCI.2018.2840738.

[4] 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, Jul. 2022, doi: 10.1007/s11042-022-13428-4.

[5] “Llama 2 - Open Source Language Model by Meta AI,” Easy With AI. Accessed: Oct. 05, 2023. [Online]. Available: https://easywithai.com/large-language-models/llama-2/

[6] D. Otter, J. Medina, and J. Kalita, “A Survey of the Usages of Deep Learning for Natural Language Processing,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. PP, pp. 1–21, Apr. 2020, doi: 10.1109/TNNLS.2020.2979670.

[7] “History of NLP,” Cohere AI. Accessed: Aug. 26, 2023. [Online]. Available: https://docs.cohere.com/docs/history-of-nlp

[8] M. Wankhade, A. Rao, and C. Kulkarni, “A survey on sentiment analysis methods, applications, and challenges,” *Artif. Intell. Rev.*, vol. 55, pp. 1–50, Feb. 2022, doi: 10.1007/s10462-022-10144-1.

[9] 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.

[10] H. Touvron *et al.*, “LLaMA: Open and Efficient Foundation Language Models,” 2023, doi: 10.48550/ARXIV.2302.13971.

[11] E. Almazrouei *et al.*, “The Falcon Series of Open Language Models,” 2023, doi: 10.48550/ARXIV.2311.16867.

[12] OpenAI *et al.*, “GPT-4 Technical Report,” 2023, doi: 10.48550/ARXIV.2303.08774.

[13] Gemini Team *et al.*, “Gemini: A Family of Highly Capable Multimodal Models,” 2023, doi: 10.48550/ARXIV.2312.11805.

[14] S. Pichai and D. Hassabis, “Our next-generation model: Gemini 1.5,” Google. Accessed: Feb. 23, 2024. [Online]. Available: https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/

[15] A. Yadollahi, A. Shahraki, and O. Zaïane, “Current State of Text Sentiment Analysis from Opinion to Emotion Mining,” *ACM Comput. Surv.*, vol. 50, pp. 1–33, May 2017, doi: 10.1145/3057270.

[16] D. A. Pereira, “A survey of sentiment analysis in the Portuguese language,” *Artif. Intell. Rev.*, vol. 54, no. 2, pp. 1087–1115, Feb. 2021, doi: 10.1007/s10462-020-09870-1.

[17] M. Pooja and S. Pandya, “A Review On Sentiment Analysis Methodologies, Practices And Applications,” *Int. J. Sci. Technol. Res.*, vol. 9, no. 02, pp. 601–609, Feb. 2020.

[18] W. Medhat, A. Hassan, and H. Korashy, “Sentiment analysis algorithms and applications: A survey,” *Ain Shams Eng. J.*, vol. 5, no. 4, pp. 1093–1113, May 2014, doi: 10.1016/j.asej.2014.04.011.

[19] D. M. E.-D. Hussein, “A survey on sentiment analysis challenges,” *J. King Saud Univ. - Eng. Sci.*, vol. 30, no. 4, pp. 330–338, Oct. 2018, doi: 10.1016/j.jksues.2016.04.002.

[20] F. Chollet, “Chapter 2: Before we begin: the mathematical building blocks of neural networks,” in *Deep learning with Python*, Shelter Island, New York: Manning Publications Co., 2018, pp. 25–55.

[21] C. M. Bishop, “Chapter 5: Neural Networks,” in *Pattern recognition and machine learning*, in Information science and statistics. , New York: Springer, 2016, pp. 225–290.

[22] I. Goodfellow, Y. Bengio, and A. Courville, “Chapter 6: Deep Feedforward Networks,” in *Deep Learning*, in Adaptive computation and machine learning. , Cambridge, Massachusetts: The MIT Press, 2016, pp. 167–227.

[23] W. Yin, K. Kann, M. Yu, and H. Schütze, “Comparative Study of CNN and RNN for Natural Language Processing,” *Comput. Res. Repos.*, vol. abs/1702.01923, 2017, doi: 10.48550/ARXIV.1702.01923.

[24] B. Lindemann, T. Müller, H. Vietz, N. Jazdi, and M. Weyrich, “A survey on long short-term memory networks for time series prediction,” *Procedia CIRP*, vol. 99, pp. 650–655, May 2021, doi: 10.1016/j.procir.2021.03.088.

[25] I. Goodfellow, Y. Bengio, and A. Courville, “Chapter 10: Sequence modelling: Recurrent and Recursive Nets,” in *Deep Learning*, in Adaptive computation and machine learning. , Cambridge, Massachusetts: The MIT Press, 2016, pp. 373–422.

[26] G. Singhal, “LSTM versus GRU Units in RNN,” Pluralsight. Accessed: Sep. 26, 2023. [Online]. Available: https://www.pluralsight.com/guides/lstm-versus-gru-units-in-rnn

[27] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” *ArXiv*, vol. 1409, Sep. 2014, doi: 10.48550/ARXIV.1409.0473.

[28] I. Goodfellow, Y. Bengio, and A. Courville, “Chapter 8: Optimization for Training Deep Models,” in *Deep Learning*, in Adaptive computation and machine learning. , Cambridge, Massachusetts: The MIT Press, 2016, pp. 274–329.

[29] F. Chollet, “Chapter 4: Fundamentals of machine learning,” in *Deep learning with Python*, Shelter Island, New York: Manning Publications Co., 2018, pp. 93–116.

[30] L. Ferrone and F. M. Zanzotto, “Symbolic, Distributed and Distributional Representations for Natural Language Processing in the Era of Deep Learning: a Survey,” *Front. Robot. AI*, vol. 6, p. 153, Jan. 2020, doi: 10.3389/frobt.2019.00153.

[31] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” *Proc. Workshop ICLR*, vol. 2013, Jan. 2013, doi: 10.48550/ARXIV.1301.3781.

[32] 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.

[33] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching Word Vectors with Subword Information,” *Comput. Res. Repos.*, vol. abs/1607.04606, 2016, doi: 10.48550/ARXIV.1607.04606.

[34] J. Pennington, R. Socher, and C. Manning, “Glove: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, Jan. 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162.

[35] M. E. Peters *et al.*, “Deep contextualized word representations.” arXiv, Mar. 22, 2018. Accessed: Sep. 16, 2022. [Online]. Available: http://arxiv.org/abs/1802.05365

[36] 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 (long and short papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. doi: 10.18653/v1/N19-1423.

[37] J. Alammar, “The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning).” Accessed: Aug. 25, 2022. [Online]. Available: https://jalammar.github.io/illustrated-bert/

[38] F. Souza, R. Nogueira, and R. Lotufo, “BERTimbau: Pretrained BERT Models for Brazilian Portuguese,” in *Intelligent Systems: 9th Brazilian Conference, BRACIS 2020, Rio Grande, Brazil, October 20–23, 2020, Proceedings, Part I*, Berlin, Heidelberg: Springer-Verlag, Oct. 2020, pp. 403–417. doi: 10.1007/978-3-030-61377-8_28.

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

[40] P. Zhou, Z. Qi, S. Zheng, J. Xu, H. Bao, and B. Xu, “Text Classification Improved by Integrating Bidirectional LSTM with Two-dimensional Max Pooling,” in *COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, Osaka, Japan: ACL, 2016, pp. 3485–3495. [Online]. Available: http://dblp.uni-trier.de/db/conf/coling/coling2016.html#ZhouQZXBX16

[41] A. U. Rehman, A. K. Malik, B. Raza, and W. Ali, “A Hybrid CNN-LSTM Model for Improving Accuracy of Movie Reviews Sentiment Analysis,” *Multimed. Tools Appl.*, vol. 78, no. 18, pp. 26597–26613, Sep. 2019, doi: 10.1007/s11042-019-07788-7.

[42] H. Xia, C. Ding, and Y. Liu, “Sentiment Analysis Model Based on Self-Attention and Character-Level Embedding,” *IEEE Access*, vol. 8, pp. 184614–184620, Jan. 2020, doi: 10.1109/ACCESS.2020.3029694.

[43] Z. Wu, T.-S. Nguyen, and D. Ong, “Structured Self-Attention Weights Encode Semantics in Sentiment Analysis,” in *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, Online: Association for Computational Linguistics, Nov. 2020, pp. 255–264. doi: 10.18653/v1/2020.blackboxnlp-1.24.

[44] F. L. dos Santos and M. Ladeira, “The Role of Text Pre-processing in Opinion Mining on a Social Media Language Dataset,” in *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, Sao Paulo, Brazil: IEEE Computer Society, Oct. 2014, pp. 50–54. doi: 10.1109/BRACIS.2014.20.

[45] F. D. Souza and J. Baptista de Oliveira e Souza Filho, “Sentiment Analysis on Brazilian Portuguese User Reviews,” in *2021 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, Temuco, Chile: IEEE, Nov. 2021, pp. 1–6. doi: 10.1109/LA-CCI48322.2021.9769838.

[46] “Top Ten Internet Languages in The World - Internet Statistics,” Internet World Stats. Accessed: Jun. 30, 2023. [Online]. Available: https://www.internetworldstats.com/stats7.htm

[47] H. B. Brum and M. das G. V. Nunes, “Building a Sentiment Corpus of Tweets in Brazilian Portuguese,” in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan: European Language Resources Association (ELRA), Dec. 2017, pp. 4167–4172. doi: 10.48550/arXiv.1712.08917.

[48] F. B. Gomes, J. M. A. Coello, and F. E. Kintschner, “Studying the Effects of Text Preprocessing and Ensemble Methods on Sentiment Analysis of Brazilian Portuguese

Tweets," in *Proceedings of 6th International Conference, SLSP 2018*, in Lecture Notes in Computer Science. Mons, Belgium: Springer Nature Switzerland, Oct. 2018, pp. 167–177. doi: 10.1007/978-3-030-00810-9_15.

[49] E. Souza, D. Vitório, D. Castro, A. L. I. Oliveira, and C. Gusmão, "Characterizing Opinion Mining: A Systematic Mapping Study of the Portuguese Language," in *Proceedings of Computational Processing of the Portuguese Language*, in Lecture Notes in Computer Science, vol. 9727. Springer International Publishing, Jul. 2016, pp. 122–127. doi: 10.1007/978-3-319-41552-9_12.

[50] D. N. de Oliveira and L. H. de C. Merschmann, "Joint Evaluation of Preprocessing Tasks with Classifiers for Sentiment Analysis in Brazilian Portuguese Language," *Multimed. Tools Appl.*, vol. 80, no. 10, pp. 15391–15412, Apr. 2021, doi: 10.1007/s11042-020-10323-8.

[51] "Repositório de Word Embeddings do NILC." Accessed: Jun. 30, 2023. [Online]. Available: http://www.nilc.icmc.usp.br/embeddings

[52] F. D. Souza and J. Baptista de Oliveira e Souza Filho, "Embedding Generation for Text Classification of Brazilian Portuguese User Reviews: From Bag-of-Words to Transformers," *Neural Comput. Appl.*, vol. 35, no. 13, pp. 9393–9406, Dec. 2022, doi: 10.1007/s00521-022-08068-6.

[53] M. H. Cardoso, A. Maria Da Rocha Fernandes, G. Marin, V. R. Quietinho Leithardt, and P. Crocker, "Comparison between Different Approaches to Sentiment Analysis in the Context of the Portuguese Language," in *Proceedings of 16th Iberian Conference on Information Systems and Technologies (CISTI)*, Chaves, Portugal: IEEE, Jun. 2021, pp. 1–6. doi: 10.23919/CISTI52073.2021.9476501.

[54] L. F. S. Britto, L. A. S. Pessoa, and S. C. C. Agostinho, "Cross-Domain Sentiment Analysis in Portuguese using BERT," in *Anais do XIX Encontro Nacional de Inteligência Artificial e Computacional (ENIAC 2022)*, Brasil: Sociedade Brasileira de Computação - SBC, 2022, pp. 61–72. doi: 10.5753/eniac.2022.227217.

[55] J. M. Adán-Coello and A. D. C. Neto, "Sentiment Analysis of Tweets in Brazilian Portuguese with Convolutional Neural Networks," *Int. J. Innov. Educ. Res.*, vol. 7, no. 6, pp. 29–41, Jun. 2019, doi: 10.31686/ijier.Vol7.Iss6.1547.

[56] H. Q. Abonizio, E. C. Paraiso, and S. Barbon, "Toward Text Data Augmentation for Sentiment Analysis," *IEEE Trans. Artif. Intell.*, vol. 3, no. 5, pp. 657–668, Oct. 2022, doi: 10.1109/TAI.2021.3114390.

[57] M. Amjad, G. Sidorov, and A. Zhila, "Data Augmentation using Machine Translation for Fake News Detection in the Urdu Language," in *Proceedings of the Twelfth Language Resources and Evaluation Conference*, N. Calzolari, F. Bechet, P. Blache, K. Choukri, C. Cieri, T. Declerck, S. Goggi, H. Isahara, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, and S. Piperidis, Eds., Marseille, France: European Language Resources Association, May 2022, pp. 2537–2542.

[58] E. G. Wibawa, P. K. Dewa, and T. Siswantoro, "Restaurant Business Insights based on Zomato Online Food Marketplace Big Data Scraping," in *Proceedings of the Second Asia Pacific International Conference on Industrial Engineering and Operations Management*, Surakarta, Indonesia, Sep. 2021, pp. 3904–3915.

[59] R. Gupta, S. Sameer, H. Muppavarapu, M. K. Enduri, and S. Anamalamudi, "Sentiment Analysis on Zomato Reviews," in *Proceedings of the 13th International Conference on Computational Intelligence and Communication Networks (CICN)*, IEEE, Sep. 2021, pp. 34–38. doi: 10.1109/CICN51697.2021.9574641.

[60] R. S. Jagdale and S. S. Deshmukh, “Sentiment Classification on Twitter and Zomato Dataset Using Supervised Learning Algorithms,” in *Proceedings of 2020 International Conference on Smart Innovations in Design, Environment, Management, Planning and Computing (ICSIDEMPC)*, IEEE, Oct. 2020, pp. 330–334. doi: 10.1109/ICSIDEMPC49020.2020.9299582.

[61] S. Mehta, “Zomato Restaurants Data,” Kaggle. Accessed: Jun. 30, 2023. [Online]. Available: https://www.kaggle.com/datasets/shrutimehta/zomato-restaurants-data

[62] D. S. Farias, I. P. Matsuno, R. M. Marcacini, and S. O. Rezende, “Opinion-meter: A Framework for Aspect-Based Sentiment Analysis,” in *Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*, in Webmedia ’16. Teresina Piauí State Brazil: ACM, Nov. 2016, pp. 351–354. doi: 10.1145/2976796.2988214.

[63] “Imbalanced Data | Machine Learning,” Google for Developers. Accessed: Jun. 30, 2023. [Online]. Available: https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data

[64] J. Brownlee, “Cost-Sensitive Learning for Imbalanced Classification,” Machine Learning Mastery. Accessed: Jun. 29, 2023. [Online]. Available: https://machinelearningmastery.com/cost-sensitive-learning-for-imbalanced-classification/

[65] P. Sterner, D. Goretzko, and F. Pargent, “Everything has its Price: Foundations of Cost-Sensitive Learning and its Application in Psychology.” PsyArXiv, Dec. 2021. doi: 10.31234/osf.io/7asgz.

[66] F. Chollet, “Keras.” 2015. Accessed: Jul. 03, 2023. [Online]. Available: https://keras.io

[67] Y. Wu *et al.*, “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation,” *Comput. Res. Repos.*, vol. abs/1609.08144, 2016, doi: 10.48550/arXiv.1609.08144.

[68] N. Baccouri, “How to translate text with python,” Analytics Vidhya. Accessed: Jul. 28, 2023. [Online]. Available: https://medium.com/analytics-vidhya/how-to-translate-text-with-python-9d203139dcf5

[69] A. Moi and N. Patry, “HuggingFace’s Tokenizers.” Apr. 05, 2023. Accessed: Jul. 21, 2023. [Online]. Available: https://github.com/huggingface/tokenizers

[70] M. Watson, C. Qian, J. Bischof, and F. Chollet, “KerasNLP.” 2022. Accessed: Jul. 03, 2023. [Online]. Available: https://github.com/keras-team/keras-nlp

[71] S. Haghighi, M. Jasemi, S. Hessabi, and A. Zolanvari, “PyCM: Multiclass confusion matrix library in Python,” *J. Open Source Softw.*, vol. 3, no. 25, p. 729, May 2018, doi: 10.21105/joss.00729.

[72] 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.

[73] 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.

[74] I. Goodfellow, Y. Bengio, and A. Courville, “Chapter 5: Machine Learning Basics,” in *Deep Learning*, in Adaptive computation and machine learning. , Cambridge, Massachusetts: The MIT Press, 2016, pp. 98–164.

[75] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011.

[76] L. Buitinck *et al.*, “API design for machine learning software: experiences from the scikit-learn project.” arXiv e-prints, Sep. 2013. doi: 10.48550/arXiv.1309.0238.

[77] T. Šolc, “Unidecode, lossy ASCII transliterations of Unicode text.” 2022. Accessed: Jun. 30, 2023. [Online]. Available: https://github.com/avian2/unidecode

[78] T. Kim and K. Wurster, “Emoji.” Jun. 29, 2023. Accessed: Jun. 30, 2023. [Online]. Available: https://pypi.org/project/emoji/

[79] S. Bird, E. Klein, and E. Loper, “Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit.” O’Reilly Media, Inc., 2009. Accessed: Dec. 05, 2022. [Online]. Available: https://github.com/nltk/nltk

[80] M. AL-Smadi, M. M. Hammad, S. A. Al-Zboon, S. AL-Tawalbeh, and E. Cambria, “Gated Recurrent Unit with Multilingual Universal Sentence Encoder for Arabic Aspect-Based Sentiment Analysis,” *Knowl.-Based Syst.*, vol. 261, no. C, p. 107540, Oct. 2021, doi: 10.1016/j.knosys.2021.107540.

[81] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” *J. Mach. Learn. Res.*, vol. 13, no. 10, pp. 281–305, Feb. 2012, doi: 10.5555/2188385.2188395.

[82] P. B. Liashchynskyi and P. Liashchynskyi, “Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS.” arXiv, 2019. Accessed: Aug. 12, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:209323928

[83] S. Katoch, S. S. Chauhan, and V. Kumar, “A review on genetic algorithm: past, present, and future,” *Multimed. Tools Appl.*, vol. 80, no. 5, pp. 8091–8126, Feb. 2021, doi: 10.1007/s11042-020-10139-6.

[84] F. Morgado Dias, “Técnicas de Controlo Não-linear baseadas em Redes Neuronais: do algoritmo ao hardware,” Tese/Dissertação, Universidade de Aveiro, Portugal, 2005. [Online]. Available: http://cee.uma.pt/morgado/Down/TeseCompletaFMD.pdf

[85] F. Mendonça, S. S. Mostafa, D. Freitas, F. Morgado-Dias, and A. G. Ravelo-García, “Multiple Time Series Fusion Based on LSTM: An Application to CAP A Phase Classification Using EEG,” *Int. J. Environ. Res. Public. Health*, vol. 19, no. 17, p. 10892, 2022, doi: 10.3390/ijerph191710892.

[86] J. Davis and M. Goadrich, “The relationship between Precision-Recall and ROC curves,” in *Proceedings of the 23rd international conference on Machine learning - ICML ’06*, Pittsburgh, Pennsylvania: Association for Computing Machinery (ACM) Press, Jun. 2006, pp. 233–240. doi: 10.1145/1143844.1143874.

[87] T. Saito and M. Rehmsmeier, “The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets,” *PLOS ONE*, vol. 10, no. 3, p. e0118432, Mar. 2015, doi: 10.1371/journal.pone.0118432.

[88] F. Mendonça, S. S. Mostafa, F. Morgado-Dias, A. G. Ravelo-García, and M. A. T. Figueiredo, “ProBoost: a Boosting Method for Probabilistic Classifiers.” arXiv e-prints, Sep. 2022. doi: 10.48550/arXiv.2209.01611.

[89] F. Wang, M. Zhang, X. Wang, X. Ma, and J. Liu, “Deep Learning for Edge Computing Applications: A State-of-the-Art Survey,” *IEEE Access*, vol. 8, pp. 58322–58336, 2020, doi: 10.1109/ACCESS.2020.2982411.

[90] S. P. Baller, A. Jindal, M. Chadha, and M. Gerndt, “DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices,” in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, San Francisco, CA, USA: IEEE, Oct. 2021, pp. 20–30. doi: 10.1109/IC2E52221.2021.00016.

[91] D. Parada, “SentZomato - RRSO Project.” in Restaurant Review Sentiment Output , in partnership with Zomato, Arditi, and UMa. Funchal, Jun. 2023. [Online]. Available: https://drive.google.com/drive/folders/1CuUwOb5Z2Fj4o4f2Kc0D7jd-Xy3B8Rin?usp=sharing

[92] A. Zhang, “SpeechRecognition.” 2017. Accessed: Jul. 24, 2023. [Online]. Available: https://github.com/Uberi/speech_recognition

[93] “Getting Started | Seeed Studio Wiki.” Accessed: Jul. 29, 2023. [Online]. Available: https://wiki.seeedstudio.com/Getting_Started

[94] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, “Robust Speech Recognition via Large-Scale Weak Supervision,” in *Proceedings of the 40th International Conference on Machine Learning*, in Proceedings of Machine Learning Research, vol. 202. PMLR, 2023, pp. 28492–28518. doi: 10.48550/arXiv.2212.04356.

[95] “Cloud Shell,” Google Cloud. Accessed: Jul. 23, 2023. [Online]. Available: https://cloud.google.com/shell

[96] 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, p. 589, Jan. 2024, doi: 10.3390/electronics13030589.

Appendixes

Appendix A: Lexicon resources

- Custom-made list of Portuguese stopwords (words with less than three characters are excluded from this list)

acerca	através	deveria	faco	houver	nesta	primeiro	sob	tive
achamos	cada	deveriam	faria	houvera	nestas	primeiros	sobre	tivemos
achei	coisa	devia	fariam	houveram	nos	propria	somos	tiveram
acho	coisas	deviam	fariamos	houveramos	nossa	própria	sua	tiveste
ademais	com	deviamos	faz	houvéramos	nossas	proprias	suas	toda
agora	como	devido	fazem	houverao	nosso	próprias	talvez	todas
ainda	contra	disso	fazemos	houverei	nossos	proprio	tambem	todavia
algo	contudo	disto	fazendo	houverem	num	próprio	também	todo
alguem	cuja	dos	fazer	houveremos	numa	proprios	tampouco	todos
alguém	cujas	durante	faço	houveria	outra	próprios	teem	tua
algum	cujo	ela	feita	houveriam	outras	pôde	têem	tuas
alguma	cujos	elas	feitas	houveriamos	outro	quais	tem	tudo
algumas	daquele	ele	feito	houvermos	outros	qual	têm	uma
alguns	daqueles	eles	feitos	houverá	para	quando	tendo	umas
ali	das	enquanto	foi	houverão	pela	quanto	tenha	uns
alí	dela	entre	for	houveríamos	pelas	quantos	tenho	vai
ambas	delas	era	foram	houvesse	pelo	que	ter	vao
ambos	dele	essa	fosse	houvessem	pelos	quem	tera	vão
ante	deles	essas	fossem	houvessemos	per	quê	terá	vem
antes	depois	esse	haja	isso	perante	sao	terao	vêm
aos	dessa	esses	hajas	isto	pode	são	terão	ver
apos	dessas	esta	hajam	lha	podendo	seja	terei	vez
após	desse	está	hajamos	lhe	poder	sejam	teremos	vindo
aquela	desses	estamos	hao	lhes	poderia	sendo	teria	vindos
àquela	desta	estao	hão	lho	poderiam	ser	tería	vir
aquelas	destas	estão	havemos	mesma	podia	sera	teriam	voce
àquelas	deste	estas	haver	mesmas	podiam	serao	teríam	voces
aquele	destes	estava	haveria	mesmo	pois	seria	teriamos	você
àquele	deve	estavam	haveriam	mesmos	por	seriam	teríamos	vocês
aqueles	devem	estavamos	haveriamos	meu	pôr	seriamos	teu	vos
àqueles	devendo	estávamos	havia	meus	porem	seriamos	teus	vós
aquilo	dever	este	haviam	minha	pôrem	será	teve	vossa
àquilo	devera	estes	haviamos	minhas	porém	serão	tido	vossas
ate	deverá	estive	hei	nas	porque	seu	tinha	vosso
até	deverao	estiveram	houve	nessa	posso	seus	tinham	vossos
atraves	deverão	estou	houvemos	nessas	pra	sido	tinhas	zero

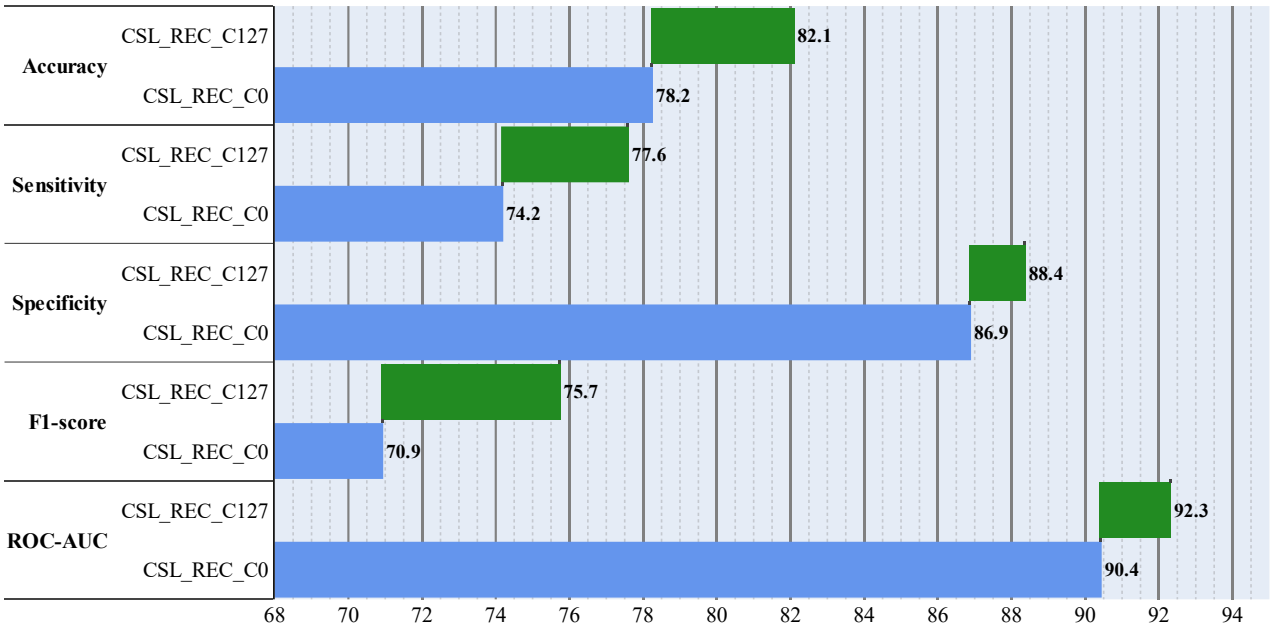
- Some examples from the dataset from Zomato: there is a large range of reviews in terms in length and type of writing.

Review ID	Text	L	Rating
5054439	Mau atendimento, empregados mal educados, e comida servida fria.	9	1.0
4914019	Meus amigos... Um restaurante que se preocupa em maximizar o seu lucro em detrimento do proporcionar ao cliente a melhor experiência, e uma viagem ao universo mexicano através da sua gastronomia ... deve ser riscado das nossas listas. Parece que se esqueceram do mais básico dos básicos no que diz respeito à restauração. Fazer o cliente sentir-se acolhido, tranquilo e receptivo. Existem alguns pontos positivos, como as bebidas, um ou outro prato apresenta uma qualidade francamente superior, tal como os palitos de frango e o chili, De resto, passa pelo normal e roça quase o vulgar... As sobremesas são algo a não repetir. O maximizar o lucro causa alguns problemas... para começar, vi com os meus olhos o colocar contas na mesa mesmo quando as pessoas não tinham sido questionadas se desejavam sobremesa ou até mesmo um café. Casais de estrangeiros que acabam de comer o prato principal e passados 10min não têm ninguém a dizer uma simples frase de gostaram da vossa refeição? desejam ver a carta de sobremesas?. Enfim,..., um restaurante que se rege pelas marcações, e inclusivé trata as pessoas como se fossem uns animais que têm exactamente 1h para comer, porque já há um grupo a seguir com uma reserva... Acho que por 25/30euros de refeição... há certas coisas que urgem em existir.	217	1.0
4105276	Staff rude e muito antipático, quase me atenderam por favor. pouca variedade nos bolos e nada de especial para o que se paga. Preços inflacionados. Não voltarei!	27	1.5
4742088	Resolvi experimentar este restaurante na rota das tapas mas fiquei muito decepcionada... O mini hambúrguer vem dentro de um mini pacote e foi claramente aquecido no microondas porque o pão estava colado ao guardanapo. Tirando isso, não tinha muito sabor, o pão estava mole (o que consegui tirar do guardanapo), queijo nem senti-lo e bacon nem vê-lo. A rota das tapas serviria para os espaços darem a conhecer a cozinha e abrir o apetite a novos clientes, que teriam vontade de voltar para provar os petiscos da casa, mas neste caso não tenciono voltar de certeza. Os funcionários são simpáticos	100	1.5
5163880	Dos piores restaurantes indianos onde estive. Comida de muito má qualidade, caro e demorado. Existem alternativas francamente melhores em Lisboa.	20	2.0
3559893	O Sushi Factory é o irmão pobre do Sushisan. Estava para lá ir há tempos e essa ida concretizou-se no fim-de-semana passado. Andava com desejos de sushi e fiquei bastante decepcionada. As peças tinham tanta, mas tanta molhanga, que era difícil perceber o sabor do peixe. Houve várias da primeira travessa em que nem sequer toquei. Até os gunkans vinham encharcados no molho de outras peças. Mas, pior que isso, várias peças tinham atum em lata. Ora, meus amigos, para comer atum em lata fico em casa, não vou largar trinta euros por uma refeição. Este quadro só melhorou quando pedimos a segunda dose, com gunkans e um mix de sashimi. No entanto, e sendo a segunda dose melhor, não chegou para compensar a refeição no todo. A não voltar.	130	2.0
3719730	Excelente atendimento. Relação preço qualidade razoável... a barmaid foi eficiente quando chamámos a atenção do vinho estar pouco fresco e simpática.	21	2.5
3470233	Adoro comida Indiana e como tal, sempre que janto fora, tendo para esta cozinha. Por sugestão de conhecidos fui jantar aqui com o meu namorado e nenhum dos dois ficou muito impressionado. Embora a qualidade do atendimento fosse bastante boa, com os empregados a serem bastante atenciosos, a comida ficou um pouco a desejar. Pedi almôndegas de vegetais com molho de caju, coco e natas e de certa forma penso que não foi bem o que recebi. Quando o prato chegou apenas cheirava a coco, sendo que as almôndegas de legumes apenas sabiam a batata. Achei a comida um pouco blunt, sem aquele sabor a que estou habituada dos restantes restaurantes indianos onde vou.	114	2.5
4843826	Muito giro para ir em grupo. Tem salas e uma cozinha onde o grupo pode ficar como se tivesse em casa. A comida também é boa! O serviço e atento!	30	3.0
3688737	Estou indignado com o atendimento que ultimamente este restaurante tem nos oferecido, não digo em termos de Staff mas sim da confecção dos pratos pedi a picanha mal passada deram-me bem passada coisa que não é admitido num amante de picanha pois a Carne fica muito dura e não conseguia comer Sem falar da entrada que tava fria Não sei se foi por ter entrado meia hora antes da cozinha fechar mas que não gostamos do atendimento não gostamos... Espero não voltar caso tenha saudades irei me dirigir a Saldanha ou assim esse da Expo vai de mal a pior	100	3.0
5192639	Overrated.	1	3.5

3784827	Já me tinha deparado com o Frankie por estas bandas e lido algumas opiniões que me deixaram muito curiosa, especialmente face ao preço apresentado. Assim que chegámos havia uma fila bastante grande, mas esperámos um pouco e ao entrar tivemos acesso a um espaço extremamente agradável e limpo que melhora a experiência. O atendimento é de realçar dada a sua amabilidade e simpatia, desde a entrada à saída por todo o staff. Pedimos a limonada de morango que estava fresca mas um bocadinho aguada e pouco apurada, o hot dog (comi o Crispy Cheddar) não é nada de espetacular mas é saboroso, as batatas super estaladiças e os molhos que as acompanham bastante bons. Para terminar comemos a Eton Mess (sobremesa de natas cremosas, coulis de morango e suspiro) que poderia estar mais fresca, porém cortou bastante bem a refeição. Mas c\'mon, por tão barato?	145	3.5
4700263	Sítio agradável e com uma enorme variedade de pratos. Se procuram um sítio acessível para conviver entre amigos, a acompanhar com boa comida, não procurem mais. Aqui o têm!	29	4.0
4747446	Começemos pelo espaço: Agradável e decoração diferente, confortável mas terei de ressaltar que o espaço é pouco iluminado o que nos deixa dificil de ler a carta em condições. Atendimento: simpatico mas pouco atencioso para um espaço tão famoso para os portugueses. Acho que é mais virado para o público estrangeiro. Comida: muito boa! Desde a entrada à sobremesa dos melhores pratos que provei em termos de qualidade, o preço é um pouco alto mas compensa dado a refeição que é servida. Comi um bife e posso dizer que veio médio/ mal na perfeição e a qualidade da carne era superior.	101	4.0
3533632	Excelentes grelhados no carvão, especialmente os bifes, com preço convidativo. Atendimento rápido, eficiente e com colaboradores muito simpaticos.	18	4.5
3752019	Que bela surpresa! Comida bem confeccionada, bom ambiente e atendimento muito atencioso. Para almoçar tinha 3 pratos do dia, um vegetariano, um vegan e outro não vegetariano. Experimentei a moqueca de tofu (vegan) que é servida num molho à base de leite de côco e pimento vermelho. Estava saboroso e tinha mesmo gosto a comida caseira. Acompanha com arroz branco e salada. Para sobremesa aconselho o bolo de alfarroba, é húmido, intenso e combina lindamente com a hortelã e calda de amoras (trocaria por amoras frescas apenas!) O menu completo - entrada, prato e sobremesa fica por 8,90€. Há uma opção mais em conta com prato, entrada ou sobremesa que custa 6,90€, salvo erro. Relação preço-qualidade fantástica!	117	4.5
3558708	Música Ambiente torna o espaço bastante acolhedor. Comida bastante apresentável e deliciosa. Atendimento Simpático	14	5.0
3470257	São muito simpáticos, muito prestaveis, esclareceram todas as nossas dúvidas, a decoração é ambiente do restaurante são bons. Em relação a comida, pedi um caril, perguntaram logo a quantidade de picante que ia querer, se muito, médio ou suave, escolhi o médio e veio bem picante(para mim) como devia ser. O arroz que acompanhou era o especial, com queijo, passas e mais umas ervas(muito bem conseguido este arroz)Escolhemos um mix de entradas que eram excelentes. O bolo de Goa é muito bom e no final pagamos 20€ por pessoa valor mais que justo para o que comemos e bebemos. Adorei e vou voltar lá de certeza	106	5.0

Appendix B: Additional results

- Preliminary result comparing the performance of the baseline REC model with and without text cleaning, to assess the importance of these techniques. The $ROI(ROC\text{-}AUC) = +19.7\%$.



- Cosine similarity values between the keywords and the five most similar words encountered with Word2Vec and FasText models. This is a qualitative analysis considering the chosen parametrization.

		Word2Vec	
		Similar words	**Score**
KEYWORDS	comida	produto	0.883
		prima	0.880
		confeção	0.880
		sem_grande	0.877
		rápidos	0.872
	espaço	conceito	0.866
		muito_bonito	0.864
		bonito	0.864
		minimalista	0.863
		honorato	0.860
	atendimento	serviço	0.933
		staff	0.871
		jovem	0.871
		pessoal	0.871
		recomendado	0.868
	agradável	muito_bonito	0.928
		muito_giro	0.926
		espaçoso	0.920
		intimista	0.915
		giro	0.911
	péssimo	chao	0.961
		atender	0.960
		ninguém	0.956
		disse	0.956
		estaria	0.954

		FastText	
		Similar words	**Score**
KEYWORDS	comida	muita_comida	0.960
		confeção	0.954
		relacionada	0.953
		comia	0.952
		cuidada	0.950
	espaço	espaçosa	0.986
		espaçoso	0.978
		espaçosas	0.963
		espaços	0.954
		muitos_espaços	0.939
	atendimento	atendimentos	0.986
		antendimento	0.984
		entendimento	0.981
		staf	0.971
		alimento	0.968
	agradável	muito_agradável	0.981
		agradáveis	0.972
		muito_agradáveis	0.972
		muito_confortável	0.971
		adorável	0.967
	péssimo	péssima	0.993
		demoravam	0.982
		demorava	0.982
		péssimas	0.981
		sem_pedido	0.981

- Reference table to identify the cleaning steps of each combination ID

ID	standard characters	lowercase characters	remove emojis	remove stopwords	create bigrams	remove punctuation	remove digits
0							
1	✓						
2		✓					
3			✓				
4				✓			
5					✓		
6						✓	
7							✓
8	✓	✓					
9	✓		✓				
10	✓			✓			
11	✓				✓		
12	✓					✓	
13	✓						✓
14		✓	✓				
15		✓		✓			
16		✓			✓		
17		✓				✓	
18		✓					✓
19			✓	✓			
20			✓		✓		
21			✓			✓	
22			✓				✓
23				✓	✓		
24				✓		✓	
25				✓			✓
26					✓	✓	
27					✓		✓
28						✓	✓
29	✓	✓	✓				
30	✓	✓		✓			
31	✓	✓			✓		
32	✓	✓				✓	
33	✓	✓					✓
34	✓		✓	✓			
35	✓		✓		✓		
36	✓		✓			✓	
37	✓		✓				✓
38	✓			✓	✓		
39	✓			✓		✓	

40	✓			✓			✓
41	✓				✓	✓	
42	✓				✓		✓
43	✓					✓	✓
44		✓	✓	✓			
45		✓	✓		✓		
46		✓	✓			✓	
47		✓	✓				✓
48		✓		✓	✓		
49		✓		✓		✓	
50		✓		✓			✓
51		✓			✓	✓	
52		✓			✓		✓
53		✓				✓	✓
54			✓	✓	✓		
55			✓	✓		✓	
56			✓	✓			✓
57			✓		✓	✓	
58			✓		✓		✓
59			✓			✓	✓
60				✓	✓	✓	
61				✓	✓		✓
62				✓		✓	✓
63					✓	✓	✓
64	✓	✓	✓	✓			
65	✓	✓	✓		✓		
66	✓	✓	✓			✓	
67	✓	✓	✓				✓
68	✓	✓		✓	✓		
69	✓	✓		✓		✓	
70	✓	✓		✓			✓
71	✓	✓			✓	✓	
72	✓	✓			✓		✓
73	✓	✓				✓	✓
74	✓		✓	✓	✓		
75	✓		✓	✓		✓	
76	✓		✓	✓			✓
77	✓		✓		✓	✓	
78	✓		✓		✓		✓
79	✓		✓			✓	✓
80	✓			✓	✓	✓	
81	✓			✓	✓		✓
82	✓			✓		✓	✓

83	✓				✓	✓	✓
84		✓	✓	✓	✓		
85		✓	✓	✓		✓	
86		✓	✓	✓			✓
87		✓	✓		✓	✓	
88		✓	✓		✓		✓
89		✓	✓			✓	✓
90		✓		✓	✓	✓	
91		✓		✓	✓		✓
92		✓		✓		✓	✓
93		✓			✓	✓	✓
94			✓	✓	✓	✓	
95			✓	✓	✓		✓
96			✓	✓		✓	✓
97			✓		✓	✓	✓
98				✓	✓	✓	✓
99	✓	✓	✓	✓	✓		
100	✓	✓	✓	✓		✓	
101	✓	✓	✓	✓			✓
102	✓	✓	✓		✓	✓	
103	✓	✓	✓		✓		✓
104	✓	✓	✓			✓	✓
105	✓	✓		✓	✓	✓	
106	✓	✓		✓	✓		✓
107	✓	✓		✓		✓	✓
108	✓	✓			✓	✓	✓
109	✓		✓	✓	✓	✓	
110	✓		✓	✓	✓		✓
111	✓		✓	✓		✓	✓
112	✓		✓		✓	✓	✓
113	✓			✓	✓	✓	✓
114		✓	✓	✓	✓	✓	
115		✓	✓	✓	✓		✓
116		✓	✓	✓		✓	✓
117		✓	✓		✓	✓	✓
118		✓		✓	✓	✓	✓
119			✓	✓	✓	✓	✓
120	✓	✓	✓	✓	✓	✓	
121	✓	✓	✓	✓	✓		✓
122	✓	✓	✓	✓		✓	✓
123	✓	✓	✓		✓	✓	✓
124	✓	✓		✓	✓	✓	✓
125	✓		✓	✓	✓	✓	✓

126		✓	✓	✓	✓	✓	✓
127	✓	✓	✓	✓	✓	✓	✓

- Results after optimization of text preprocessing steps of REC optimized architecture, sorted by performance from best to worst.

ID	ACC (σ)	TPR (σ)	F1 (σ)	AUC (σ)
115	**82.6 (0.15)**	**77.4 (0.19)**	**76.5 (0.16)**	**92.1 (0.06)**
100	**81.2 (0.67)**	**78.8 (0.22)**	**75.8 (0.44)**	**92.0 (0.07)**
101	**81.9 (0.14)**	**78.2 (0.08)**	**76.0 (0.91)**	**92.0 (0.10)**
86	81.7 (0.02)	78.2 (0.03)	76.1 (0.41)	91.7 (0.01)
30	81.2 (0.02)	78.4 (0.19)	75.9 (0.91)	92.0 (0.03)
68	82.0 (0.72)	77.7 (0.01)	75.9 (1.05)	91.7 (0.45)
19	81.9 (0.05)	78.1 (0.10)	75.4 (0.08)	91.8 (0.02)
54	81.8 (0.67)	78.0 (0.01)	75.6 (0.54)	91.7 (0.26)
119	82.2 (0.09)	77.4 (0.26)	75.9 (0.32)	91.8 (0.14)
69	81.3 (0.97)	78.3 (0.12)	75.5 (0.37)	91.7 (0.05)
10	81.8 (0.76)	77.2 (0.18)	76.1 (1.01)	91.7 (0.37)
64	82.6 (1.01)	77.2 (0.50)	75.7 (0.61)	91.9 (0.38)
15	82.3 (0.51)	77.4 (0.69)	75.7 (0.30)	91.7 (0.21)
62	81.8 (0.16)	77.5 (0.94)	75.7 (1.06)	91.8 (0.13)
56	81.4 (0.59)	77.8 (0.17)	75.6 (0.51)	91.7 (0.08)
120	80.9 (0.93)	78.5 (0.37)	75.1 (0.51)	91.8 (0.21)
105	81.4 (0.45)	78.1 (0.38)	75.3 (0.51)	91.7 (0.19)
70	81.5 (0.55)	78.3 (0.14)	75.0 (0.34)	91.7 (0.04)
94	80.7 (0.59)	78.3 (0.45)	75.3 (0.28)	91.6 (0.15)
44	80.7 (1.21)	78.2 (0.11)	75.3 (1.89)	91.7 (0.60)
121	81.9 (0.22)	77.9 (0.34)	75.1 (0.43)	91.8 (0.11)
48	83.0 (0.56)	76.4 (0.28)	75.8 (0.19)	91.9 (0.22)
38	82.8 (0.67)	76.8 (0.36)	75.5 (0.13)	92.0 (0.07)
114	81.4 (0.07)	78.1 (0.42)	75.0 (1.06)	91.8 (0.22)
84	80.9 (1.20)	78.4 (0.19)	75.0 (0.85)	91.5 (0.40)
39	80.5 (0.91)	78.6 (0.17)	74.9 (1.27)	91.6 (0.24)
82	80.8 (0.84)	78.1 (0.22)	75.3 (0.91)	91.4 (0.21)
75	80.3 (0.61)	78.7 (0.08)	74.8 (0.23)	91.7 (0.09)
25	79.8 (0.06)	78.1 (0.13)	75.2 (0.67)	91.7 (0.14)
61	81.0 (0.23)	78.0 (0.40)	75.0 (0.68)	91.7 (0.04)
74	81.0 (0.59)	77.6 (1.22)	75.3 (0.35)	91.6 (0.03)
90	81.5 (1.52)	77.6 (0.40)	75.0 (0.09)	91.7 (0.30)
99	81.9 (1.76)	77.6 (0.14)	74.9 (1.05)	91.8 (0.38)
98	81.3 (0.42)	77.4 (0.95)	75.2 (0.61)	91.6 (0.10)
81	81.7 (0.15)	77.2 (1.03)	75.3 (0.71)	91.5 (0.06)
125	81.9 (0.14)	76.5 (1.17)	75.8 (0.26)	91.5 (0.07)
50	81.1 (0.05)	77.3 (0.25)	75.3 (0.84)	91.5 (0.02)
91	82.0 (1.34)	76.9 (0.61)	75.2 (1.32)	91.6 (0.51)
113	81.3 (0.25)	77.0 (0.17)	75.3 (0.72)	91.6 (0.07)
123	80.8 (0.01)	77.1 (0.44)	75.3 (0.79)	91.5 (0.19)
118	81.4 (0.63)	77.4 (0.95)	74.9 (1.56)	91.4 (0.36)
60	80.7 (0.91)	77.7 (0.64)	74.7 (0.41)	91.5 (0.24)
41	82.1 (0.62)	76.6 (0.29)	75.1 (0.24)	91.7 (0.06)
110	81.5 (0.22)	76.6 (0.14)	75.4 (0.25)	91.4 (0.15)
108	80.3 (0.30)	77.6 (0.26)	75.0 (0.13)	91.3 (0.32)
124	81.3 (1.10)	77.5 (0.14)	74.7 (0.23)	91.5 (0.25)
109	82.0 (1.45)	76.7 (0.19)	74.9 (0.26)	91.9 (0.02)

126	79.8 (0.09)	77.5 (0.47)	75.0 (0.28)	91.4 (0.01)
24	81.0 (1.27)	77.8 (0.42)	74.5 (0.45)	91.4 (0.44)
77	81.4 (0.62)	76.9 (0.26)	75.1 (0.61)	91.4 (0.18)
92	82.0 (0.99)	76.9 (0.47)	75.0 (1.05)	91.4 (0.29)
34	80.7 (0.35)	77.8 (0.74)	74.5 (0.34)	91.5 (0.16)
122	81.2 (0.20)	77.7 (0.39)	74.4 (0.11)	91.5 (0.26)
36	80.6 (0.23)	76.7 (0.70)	75.3 (0.67)	91.5 (0.12)
80	81.2 (1.04)	77.9 (0.35)	74.2 (0.71)	91.6 (0.00)
102	80.7 (0.89)	77.3 (0.97)	74.8 (0.55)	91.4 (0.00)
96	82.3 (0.57)	76.2 (1.15)	75.0 (0.05)	91.6 (0.25)
83	80.5 (0.69)	78.0 (0.26)	74.2 (0.84)	91.4 (0.03)
106	79.8 (1.20)	78.3 (0.23)	74.1 (0.83)	91.3 (0.19)
104	81.4 (0.45)	77.0 (0.58)	74.6 (0.68)	91.4 (0.14)
85	80.9 (1.55)	78.1 (0.26)	73.5 (0.83)	91.7 (0.33)
23	81.5 (0.85)	76.8 (1.59)	74.4 (1.00)	91.6 (0.13)
107	81.0 (0.26)	77.6 (0.30)	74.0 (0.69)	91.4 (0.31)
32	80.5 (0.18)	77.1 (0.48)	74.5 (0.51)	91.0 (0.24)
111	80.4 (0.18)	77.9 (0.32)	73.8 (0.15)	91.3 (0.04)
71	79.4 (0.71)	77.3 (0.61)	74.5 (0.63)	91.1 (0.10)
43	79.4 (0.95)	77.6 (0.54)	74.0 (0.15)	91.3 (0.17)
8	79.9 (0.87)	76.8 (0.45)	74.3 (0.89)	91.3 (0.15)
1	80.1 (0.47)	77.0 (0.40)	74.2 (1.39)	91.1 (0.09)
72	80.9 (0.77)	76.4 (0.26)	74.4 (0.13)	91.2 (0.14)
31	80.9 (0.22)	77.0 (0.22)	73.9 (0.22)	91.1 (0.07)
66	80.8 (2.18)	77.3 (1.02)	73.7 (0.67)	91.2 (0.64)
40	82.3 (1.85)	75.9 (1.14)	74.1 (0.62)	91.6 (0.44)
55	80.6 (2.59)	77.7 (0.58)	73.2 (1.29)	91.4 (0.50)
49	80.4 (1.77)	77.5 (0.18)	73.7 (0.87)	90.9 (0.36)
79	80.7 (1.73)	77.3 (0.18)	73.5 (0.42)	91.3 (0.19)
4	80.4 (1.44)	76.7 (0.58)	74.0 (0.56)	91.2 (0.32)
67	80.2 (0.43)	77.5 (0.10)	73.5 (0.29)	91.0 (0.12)
65	80.7 (0.21)	75.9 (0.79)	74.6 (0.48)	90.9 (0.33)
11	81.7 (0.54)	75.7 (0.10)	74.2 (0.01)	91.2 (0.01)
116	80.2 (1.09)	76.9 (0.80)	73.7 (1.05)	91.1 (0.10)
42	80.8 (0.20)	76.6 (0.25)	73.8 (0.55)	90.9 (0.02)
76	80.2 (1.29)	76.7 (0.41)	73.6 (0.81)	91.2 (0.00)
95	80.8 (0.02)	77.5 (0.07)	72.8 (0.01)	91.4 (0.09)
29	79.1 (0.84)	76.9 (0.29)	73.7 (0.64)	90.8 (0.39)
33	80.4 (0.28)	76.1 (0.85)	73.9 (0.07)	90.8 (0.31)
112	80.1 (1.79)	76.6 (0.16)	73.3 (0.02)	91.0 (0.47)
12	79.4 (1.58)	76.4 (0.86)	73.6 (1.09)	90.8 (0.36)
73	81.2 (0.23)	76.7 (0.43)	72.4 (0.38)	91.5 (0.17)
103	80.3 (0.23)	76.1 (1.41)	73.2 (1.62)	91.0 (0.22)
35	80.7 (0.09)	75.7 (0.26)	73.4 (0.44)	90.5 (0.31)
37	78.7 (0.72)	76.8 (0.08)	73.0 (0.21)	90.6 (0.35)
13	80.6 (0.38)	75.7 (0.33)	73.4 (1.11)	90.5 (0.41)
9	80.1 (1.10)	75.6 (0.50)	72.7 (0.17)	90.3 (0.31)
87	79.8 (0.29)	75.0 (0.65)	72.7 (0.73)	90.4 (0.08)
78	77.8 (0.96)	76.5 (0.06)	72.0 (0.21)	90.2 (0.18)
46	78.5 (0.09)	75.7 (0.03)	72.1 (0.44)	90.4 (0.15)
89	79.5 (0.88)	75.3 (0.07)	71.8 (0.21)	90.3 (0.14)

17	78.7 (0.12)	74.9 (0.05)	72.3 (0.96)	90.2 (0.05)
57	79.0 (0.34)	75.3 (0.15)	71.8 (0.03)	90.3 (0.10)
117	78.9 (0.06)	74.8 (0.49)	71.9 (1.18)	90.3 (0.00)
59	78.2 (0.90)	75.5 (0.17)	71.4 (0.19)	90.1 (0.35)
51	79.3 (0.09)	74.5 (0.66)	71.8 (0.39)	90.0 (0.10)
47	78.5 (0.63)	74.7 (0.22)	71.8 (0.96)	89.9 (0.11)
93	77.6 (1.52)	75.6 (0.17)	70.9 (1.00)	90.2 (0.26)
63	79.0 (0.76)	74.3 (0.17)	71.6 (1.30)	89.8 (0.43)
14	78.2 (0.63)	74.2 (0.00)	71.8 (1.00)	89.8 (0.20)
26	77.4 (0.13)	74.9 (0.78)	71.3 (1.05)	89.9 (0.20)
6	79.0 (0.48)	73.5 (2.20)	71.8 (0.96)	89.9 (0.19)
21	78.3 (1.11)	74.3 (0.16)	71.4 (0.16)	89.6 (0.37)
18	78.6 (0.38)	73.4 (1.23)	71.9 (1.60)	89.7 (0.16)
88	77.5 (0.27)	74.3 (0.29)	71.3 (0.98)	89.7 (0.05)
3	77.2 (0.60)	74.3 (0.62)	71.3 (1.34)	89.7 (0.02)
16	77.7 (1.02)	74.9 (0.08)	70.4 (0.24)	89.8 (0.09)
53	78.3 (1.44)	74.8 (0.85)	70.2 (0.40)	89.7 (0.66)
28	78.2 (1.62)	74.4 (1.22)	70.3 (0.04)	89.8 (0.39)
58	78.0 (0.60)	73.8 (0.20)	71.0 (0.42)	89.3 (0.29)
97	77.9 (1.26)	74.6 (0.05)	70.1 (1.32)	89.7 (0.35)
52	79.0 (1.09)	73.7 (0.61)	70.4 (0.84)	89.7 (0.03)
2	78.9 (0.41)	73.5 (1.37)	70.5 (0.07)	89.5 (0.44)
45	77.4 (0.73)	73.1 (1.90)	70.8 (1.75)	89.6 (0.23)
7	79.5 (0.31)	71.9 (0.10)	71.1 (0.57)	89.3 (0.39)
20	76.8 (0.98)	74.4 (0.65)	69.6 (0.09)	89.3 (0.35)
22	78.8 (0.95)	72.2 (2.00)	70.5 (0.46)	89.3 (0.44)
5	76.6 (0.77)	73.6 (0.56)	70.0 (1.18)	89.0 (0.13)
27	77.4 (1.33)	73.8 (0.84)	69.5 (1.15)	89.3 (0.04)

- Results after optimization of text preprocessing steps of AREC optimized architecture, sorted by performance from best to worst.

ID	ACC (σ)	TPR (σ)	F1 (σ)	AUC (σ)
101	**82.0 (0.73)**	**77.0 (0.74)**	**76.0 (0.79)**	**92.4 (0.37)**
99	**81.6 (0.22)**	**77.2 (0.23)**	**75.9 (0.04)**	**92.2 (0.01)**
122	**81.6 (0.32)**	**77.6 (0.14)**	**75.6 (0.13)**	**92.2 (0.03)**
15	82.2 (0.32)	77.0 (0.15)	75.8 (0.51)	92.3 (0.14)
30	82.0 (0.52)	77.0 (0.61)	75.9 (0.59)	92.2 (0.29)
117	81.9 (0.10)	76.8 (0.57)	76.0 (0.52)	92.3 (0.19)
92	81.4 (0.12)	77.4 (0.17)	75.8 (0.52)	92.1 (0.00)
33	81.5 (0.47)	77.5 (0.81)	75.6 (1.18)	92.2 (0.08)
8	82.4 (0.24)	76.6 (0.40)	75.9 (0.41)	92.4 (0.06)
68	82.2 (0.24)	76.8 (0.11)	75.7 (0.71)	92.5 (0.16)
49	82.0 (0.65)	76.8 (0.91)	75.7 (0.73)	92.3 (0.22)
64	82.4 (0.66)	76.4 (1.50)	75.7 (0.85)	92.4 (0.26)
90	81.5 (0.88)	76.8 (0.41)	75.7 (1.16)	92.1 (0.44)
72	81.9 (0.11)	77.5 (0.88)	75.0 (0.55)	92.3 (0.03)
104	82.0 (0.10)	76.9 (0.52)	75.5 (0.42)	92.1 (0.34)
16	82.2 (0.09)	77.0 (0.11)	75.3 (0.07)	92.3 (0.01)
108	80.7 (0.10)	77.5 (0.35)	75.3 (0.44)	92.2 (0.09)
91	81.8 (0.70)	77.2 (0.05)	75.2 (0.33)	92.2 (0.02)
69	81.9 (0.44)	77.3 (0.17)	75.2 (0.18)	92.1 (0.08)
10	81.5 (0.44)	77.0 (0.08)	75.4 (0.52)	92.1 (0.31)
120	81.9 (0.10)	76.6 (0.09)	75.6 (0.47)	92.1 (0.12)
109	82.0 (0.42)	77.0 (0.32)	75.2 (0.00)	92.1 (0.08)
106	81.9 (0.85)	77.0 (0.53)	75.2 (0.65)	92.2 (0.20)
73	81.2 (0.05)	77.2 (0.08)	75.2 (1.14)	92.1 (0.12)
13	81.8 (0.00)	76.8 (0.05)	75.3 (0.68)	92.1 (0.19)
48	81.4 (0.70)	76.9 (0.67)	75.3 (0.12)	92.2 (0.31)
112	81.7 (0.07)	76.8 (0.84)	75.4 (0.43)	92.0 (0.04)
115	81.5 (0.66)	76.7 (0.93)	75.4 (0.38)	92.3 (0.47)
34	82.2 (0.39)	76.6 (0.71)	75.3 (0.45)	92.1 (0.00)
124	81.8 (0.34)	77.3 (0.12)	74.9 (0.31)	92.0 (0.08)
27	81.8 (0.20)	76.7 (0.21)	75.3 (0.13)	92.1 (0.10)
44	81.9 (0.09)	76.5 (1.27)	75.4 (0.41)	92.1 (0.04)
7	81.4 (0.55)	77.2 (0.46)	75.0 (0.76)	92.0 (0.18)
86	81.8 (0.72)	76.3 (0.14)	75.5 (0.04)	92.3 (0.19)
45	82.0 (0.63)	76.4 (0.80)	75.4 (0.31)	92.2 (0.19)
71	82.0 (0.62)	76.5 (0.62)	75.2 (0.20)	92.2 (0.27)
23	81.2 (0.76)	77.2 (0.14)	75.0 (0.10)	92.0 (0.12)
76	81.6 (0.27)	76.2 (0.28)	75.5 (0.35)	92.2 (0.15)
102	82.0 (0.18)	76.5 (0.03)	75.2 (0.42)	92.1 (0.04)
118	81.5 (0.08)	76.3 (0.25)	75.5 (0.63)	92.2 (0.07)
126	81.5 (0.76)	77.3 (0.15)	74.9 (0.37)	91.7 (0.20)
17	81.3 (0.81)	77.3 (0.12)	74.8 (0.35)	92.1 (0.17)
14	81.8 (0.55)	76.4 (0.18)	75.3 (0.45)	92.0 (0.18)
121	82.1 (0.86)	76.6 (0.07)	75.0 (0.01)	92.2 (0.08)
3	81.5 (0.33)	77.0 (0.53)	74.9 (0.72)	91.9 (0.18)
81	80.5 (0.45)	77.1 (0.21)	74.9 (0.13)	92.2 (0.07)
46	81.4 (0.25)	77.3 (0.11)	74.7 (0.12)	91.9 (0.05)

47	82.4 (0.11)	76.4 (0.82)	75.0 (0.16)	92.3 (0.05)
116	82.5 (0.25)	74.8 (0.52)	76.0 (0.06)	92.4 (0.03)
78	81.4 (0.71)	76.6 (0.97)	75.1 (0.92)	92.1 (0.22)
80	79.9 (0.08)	77.8 (0.00)	74.7 (0.14)	91.8 (0.08)
67	81.7 (0.73)	77.3 (0.24)	74.5 (0.17)	92.0 (0.06)
70	82.0 (1.28)	76.2 (0.81)	75.3 (0.99)	91.8 (0.61)
114	83.0 (0.08)	75.0 (0.71)	75.6 (0.18)	92.5 (0.13)
4	81.9 (0.89)	77.0 (0.08)	74.5 (0.16)	92.1 (0.04)
1	82.2 (0.10)	75.4 (0.91)	75.6 (0.07)	92.1 (0.02)
28	81.5 (0.35)	76.6 (0.72)	75.0 (0.42)	91.9 (0.06)
87	81.7 (0.18)	76.5 (0.84)	75.0 (0.14)	91.9 (0.08)
42	82.7 (0.50)	75.4 (0.34)	75.4 (0.04)	92.4 (0.18)
2	81.4 (0.50)	77.4 (0.28)	74.3 (0.23)	92.1 (0.04)
24	82.0 (0.03)	76.6 (0.53)	74.8 (0.00)	92.1 (0.17)
20	81.7 (0.84)	76.6 (0.32)	74.9 (0.14)	92.0 (0.04)
39	81.9 (0.72)	76.4 (0.88)	74.9 (0.76)	92.3 (0.07)
103	81.6 (0.22)	76.4 (0.24)	75.1 (0.08)	91.9 (0.17)
51	81.7 (0.40)	76.5 (1.18)	74.9 (0.63)	92.1 (0.08)
26	82.2 (0.85)	76.0 (0.40)	75.1 (0.07)	92.0 (0.11)
113	81.4 (0.37)	76.7 (0.16)	74.9 (0.05)	91.9 (0.01)
66	81.8 (0.81)	76.6 (1.04)	74.7 (0.35)	92.0 (0.18)
63	81.8 (0.58)	76.8 (0.21)	74.7 (0.42)	91.9 (0.12)
50	80.3 (0.72)	77.5 (0.43)	74.6 (0.03)	91.6 (0.09)
41	80.7 (0.74)	77.9 (0.08)	74.1 (0.14)	91.7 (0.09)
36	82.2 (0.48)	75.8 (0.11)	75.1 (0.12)	92.0 (0.03)
85	81.9 (0.31)	75.6 (1.47)	75.3 (0.15)	92.1 (0.21)
111	81.6 (0.25)	75.7 (0.02)	75.2 (0.80)	92.2 (0.16)
18	82.3 (0.72)	75.7 (0.83)	75.0 (0.70)	92.4 (0.16)
40	81.2 (0.34)	77.3 (0.20)	74.1 (0.07)	92.1 (0.18)
22	82.1 (0.54)	76.2 (1.04)	74.6 (0.26)	92.2 (0.00)
55	80.7 (0.59)	76.2 (0.60)	75.1 (0.09)	91.9 (0.23)
56	81.7 (0.76)	76.7 (0.65)	74.5 (0.57)	91.8 (0.06)
29	82.0 (0.49)	76.0 (0.24)	75.0 (0.33)	91.8 (0.34)
58	81.3 (0.71)	76.9 (0.50)	74.4 (0.67)	91.9 (0.15)
93	80.9 (0.68)	76.5 (1.59)	74.7 (1.42)	92.0 (0.16)
95	82.7 (1.30)	75.4 (0.67)	74.9 (0.28)	92.4 (0.03)
105	80.7 (0.52)	76.6 (0.09)	74.8 (0.29)	91.6 (0.37)
37	81.8 (1.25)	76.8 (0.73)	74.0 (0.26)	92.2 (0.11)
100	81.8 (0.02)	76.1 (1.00)	74.7 (0.40)	91.8 (0.49)
82	81.6 (0.15)	76.1 (0.57)	74.9 (0.53)	91.6 (0.20)
5	81.0 (0.05)	77.2 (0.18)	74.0 (0.18)	91.8 (0.03)
54	82.2 (0.69)	75.3 (0.54)	74.9 (0.24)	92.3 (0.16)
19	80.4 (1.07)	77.9 (0.49)	73.5 (0.01)	91.9 (0.18)
84	81.8 (0.46)	76.1 (0.43)	74.5 (0.25)	92.0 (0.04)
74	81.1 (1.49)	77.0 (0.49)	73.9 (0.12)	92.0 (0.02)
110	81.4 (0.14)	75.2 (1.77)	75.1 (0.72)	92.1 (0.21)
60	81.1 (0.31)	76.4 (0.90)	74.5 (0.41)	91.5 (0.26)
43	80.8 (0.21)	76.0 (0.42)	74.8 (0.87)	91.8 (0.03)
75	81.7 (1.41)	76.7 (1.20)	73.8 (0.16)	92.0 (0.32)
35	81.1 (0.07)	76.5 (0.76)	74.2 (0.05)	91.6 (0.30)
31	81.7 (0.61)	75.9 (0.51)	74.3 (0.03)	91.8 (0.03)

123	80.2 (1.75)	77.2 (0.18)	73.7 (1.46)	91.7 (0.28)
97	81.1 (0.44)	76.3 (1.03)	74.1 (0.36)	91.7 (0.13)
61	82.5 (1.28)	74.4 (1.74)	74.9 (0.19)	92.1 (0.06)
65	80.6 (1.17)	77.1 (0.15)	73.5 (0.27)	91.8 (0.09)
59	82.0 (1.54)	75.8 (1.15)	74.0 (0.01)	92.1 (0.12)
6	82.4 (0.59)	74.8 (0.87)	74.6 (0.10)	92.1 (0.03)
83	80.6 (0.05)	76.3 (0.58)	74.0 (1.40)	91.7 (0.11)
107	81.5 (0.75)	76.1 (0.79)	73.9 (0.22)	91.8 (0.03)
98	81.9 (0.30)	74.6 (0.05)	74.9 (0.15)	91.7 (0.29)
21	81.3 (0.69)	75.4 (0.31)	74.4 (0.09)	91.9 (0.13)
125	81.6 (0.87)	76.1 (1.18)	73.7 (0.00)	91.8 (0.37)
119	81.8 (0.84)	74.9 (0.92)	74.5 (0.28)	91.9 (0.13)
89	82.0 (0.65)	75.1 (0.59)	74.3 (0.29)	91.9 (0.37)
88	81.3 (0.80)	76.4 (0.45)	73.4 (0.65)	92.0 (0.27)
9	81.4 (0.28)	75.2 (1.21)	74.3 (0.16)	91.4 (0.20)
57	82.2 (0.67)	74.2 (0.98)	74.4 (0.33)	92.1 (0.17)
32	81.2 (2.61)	76.1 (0.87)	73.1 (0.34)	92.1 (0.05)
52	81.5 (1.28)	75.1 (1.73)	74.0 (0.62)	91.3 (0.82)
11	81.8 (1.60)	75.1 (1.30)	73.3 (0.22)	91.9 (0.06)
38	82.1 (0.79)	74.2 (2.10)	73.9 (0.42)	91.8 (0.44)
96	79.8 (0.99)	76.1 (0.05)	73.2 (0.84)	91.3 (0.38)
12	81.2 (0.95)	74.7 (3.21)	73.6 (0.77)	91.7 (0.17)
79	81.7 (1.10)	73.7 (3.39)	74.0 (0.41)	92.0 (0.12)
62	81.2 (2.96)	75.1 (2.18)	72.5 (0.65)	91.8 (0.27)
53	80.9 (1.74)	74.8 (2.85)	73.1 (0.25)	91.1 (0.59)
25	80.6 (1.21)	74.7 (2.01)	72.9 (0.71)	91.6 (0.15)

- Best chromosomes and scores resulted after optimization of REC architecture.

Generation	Best chromosome	Score (%)
0	1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 1	91.65
1	1 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1	91.79
2	1 0 1 0 1 0 0 1 0 0 0 1 1 0 1 0	91.91
3	1 0 1 0 1 0 0 1 0 0 0 1 1 0 1 0	91.91
4	1 0 1 0 1 0 0 1 0 0 0 1 1 0 1 0	91.91
5	1 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0	91.98
6	1 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0	92.03
7	1 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0	92.03
8	1 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0	92.03
9	**1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1**	**92.11**
10	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11
11	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11
12	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11
13	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11
14	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11
15	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11
16	1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 1	92.11

- Best chromosomes and scores resulted after optimization of AREC architecture.

Generation	Best chromosome	Score (%)
0	0 1 1 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 0 0 1	91.85
1	1 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0	91.90
2	0 0 1 0 0 0 1 1 0 1 0 0 0 1 1 1 1 1 0 0 0	92.01
3	1 1 0 0 0 1 0 0 0 0 1 1 1 1 0 1 1 0 1 0 1	92.14
4	1 1 0 0 0 1 0 0 0 0 1 1 1 1 0 1 1 0 1 0 1	92.14
5	**0 0 1 1 0 0 1 1 0 0 0 1 0 1 1 0 0 1 0 0 1**	**92.27**
6	1 0 1 0 0 0 1 1 0 0 1 0 0 1 1 0 1 1 0 0 1	92.32
7	1 0 1 0 0 1 1 1 0 0 1 0 0 1 1 0 0 1 0 1 0	92.40

Appendix C: Python scripts

- **getMetrics.py:** used to gather the store information about the training procedure and performance of all models and create the plots.

```
#%% 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 getMetrics(path: str, model_name: str):
  """Load to memory all metrics stored by saveModels.py"""
  if path[-1] == '/': path = path[:-1]
  full_path = path + f"/Model_{model_name}/"
  # Load training evaluation
  path_temp = full_path + 'Training/'

  with open(path_temp + "ACC.pkl",'rb') as f:
    acc = pickle.load(f)
    acc_val = pickle.load(f)
  with open(path_temp + "AUC.pkl",'rb') as f:
    auc = pickle.load(f)
    auc_val = pickle.load(f)
  with open(path_temp + "LOSS.pkl",'rb') as f:
    loss = pickle.load(f)
    loss_val = pickle.load(f)
  with open(path_temp + "BestEpoch.pkl",'rb') as f:
    bestEpoch = pickle.load(f)
  with open(path_temp + "Time.pkl",'rb') as f:
    trainTime = pickle.load(f)
  try:
    with open(path_temp + "STD.pkl",'rb') as f:
      variation = {}
      try: variation['AUC'] = pickle.load(f)
      except: pass
      try: variation['ACC'] = pickle.load(f)
      except: pass
      try: variation['Spe'] = pickle.load(f)
      except: pass
      try: variation['Sen'] = pickle.load(f)
      except: pass
```

```
      try: variation['F1'] = pickle.load(f)
      except: pass
      try: variation['Pre'] = pickle.load(f)
      except: pass
  except: variation = {}
  # Load PYCM metrics
  path_temp = full_path + 'ConfussionMatrix/'
  with open(path_temp + "CM.pkl",'rb') as f:
    CM = pickle.load(f)
  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)

  return [acc,acc_val], [auc,auc_val], [loss,loss_val], bestEpoch, trainTime,
         variation, CM, Overall, Class

def selectMetrics(metrics: dict, select: list[str]):
  return [metrics[v] for v in select]

def renameMetrics(select: list[str]):
  rename = {"AUC": "ROC-AUC",
            "Overall ACC": "Accuracy",
            "ACC": "Accuracy",
            "TPR": "Sensitivity",
            "TPR Macro": "Sensitivity",
            "TNR": "Specificity",
            "TNR Macro": "Specificity",
            "PPV": "Precision",
            "PPV Macro": "Precision",
            "F1": "F1-score",
            "F1 Macro":  "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.append(n)
  else:
    title = f"Cross-Validation with {n[-5:]}"

  return renamed_models, title
```

```
def plotTrain(path: str, metrics: list, model_name: str, be: np.array, plot:
bool=False):
  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:
    path_name = f'{path}/Model_{model_name}/Training/TrainingCurves'
    model_name, _ = renameModels([model_name])
    model_name = model_name[0]

    with open(path_name, 'rb') as f:
      curves = pickle.load(f) # curves is a dict
    curves.pop("ACC")
    n_graphs = 2
    size5 = (24,8)
    size2 = (16,8)
    size1 = (6,8)
    N = len(list(curves.values())[0])
    if N == 5: # (5 folds)
      fig, axs = plt.subplots(n_graphs,5, figsize=size5,sharex=True, dpi=300)
      fig.suptitle(model_name)
      for ax,(k,v) in zip(axs, curves.items()):
        for i in range(5):
          _show_train_history(v[i], k, ax=ax[i], fn= i+1,
                              be=be if isinstance(be,int) else be[i])
```

```
    elif N == 2: # (2 folds)
      fig, axs = plt.subplots(n_graphs,2, figsize=size2,sharex=True, dpi=300)
      fig.suptitle(model_name)
      for ax,(k,v) in zip(axs, curves.items()):
        for i in range(2):
          _show_train_history(v[i], k, ax=ax[i], fn= i+1,
                              be=be if isinstance(be,int) else be[i])
    elif N == 1: # (single fold)
      fig, axs = plt.subplots(n_graphs,1, figsize=size1,sharex=True, dpi=300)
      fig.suptitle(model_name)
      for ax,(k,v) in zip(axs, curves.items()):
        _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],
                      filename:str='waterfall_plot.svg', plot=False):
  labels = renameMetrics(labels)
  df = pd.DataFrame(metrics, columns=labels)
  df_model = pd.DataFrame(Models,columns=['Model'])
  df = pd.concat([df_model,df],axis=1)

  Models, title = renameModels(Models)
  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
  def roundNearestEven(n: int):
    return (n//2)*2

  if plot:
    fig = go.Figure(layout=dict(height=600,width=1200,
                    margin=dict(l=20, r=20, t=15, b=30)))
    M = len(Models)
    L = len(labels)
    factor=100
    formated_labels = np.asarray([[f'<b>{l}</b>']*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)*factor # 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(waterfallgroupgap = 0.03,
                      waterfallgap = 0.03,
                      font_family="Times New Roman",
                      font=dict(size=20),
                      xaxis = dict(tickmode = 'linear',
                                   tick0 = 0.70*factor,
                                   dtick = 0.02*factor)
                      )
    xlim_min = roundNearestEven(min(formated_metrics[::M])*0.98)
    fig.update_xaxes(title_font_family="Times New Roman",
                     range=[xlim_min, 0.94*factor],
                     showgrid=True, gridwidth=2.0, gridcolor='DarkBlue',
                     minor_griddash="dot", minor_gridwidth=1.5)
    fig.update_yaxes(title_font_family="Times New Roman",
                     showline=True, linewidth=1.5, linecolor='black')

    binary_data = \
        fig.to_image(format='svg',
                     width=None, height=None,
                     scale = 50,
                     engine='orca'
                     )
    with open('Graphs_Plotly/' + filename, 'wb') as f:
        f.write(binary_data)
    fig.show()
  return df

def plotCM(conf_mat: np.ndarray, conf_mat_abs: np.ndarray,
           model_name: str, plot: bool=False):

  conf_mat = conf_mat
  model_name, _ = renameModels([model_name])
  model_name = model_name[0]
  if plot:
    fig, ax = plt.subplots(figsize=(8.0, 8.0), dpi=128)
    ax.matshow(conf_mat, cmap=plt.cm.Oranges, 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=f"{conf_mat[i, j]:.4f}\n({conf_mat_abs[i, j]:.0f})",
                va='center', ha='center', fontsize=21)
```

```
    plt.xlabel('Predictions', fontsize=22)
    plt.ylabel('Actuals', fontsize=22)
    plt.xticks(ticks=[0,1,2], labels=['Negative','Mixed','Positive'],
               fontsize=20)
    ax.xaxis.tick_bottom()
    plt.yticks(ticks=[0,1,2], labels=['Negative','Mixed','Positive'],
               rotation=90, fontsize=20,va='center')
    plt.title(f'{model_name}\n', fontsize=23)
    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(data.T.flat)
    macros = data.mean(axis=-1).tolist() * 3
    values = (np.asarray(values) - np.asarray(macros)) * 100
    to_plot = pd.DataFrame({'model_name':3*Models, # this 3 is for each sentiment
                            'Sentiment':N*['Negative'] +
                                        N*['Mixed'] +
                                        N*['Positive'],
                            metric:values,
                            })
    fig, axs = plt.subplots(1,N+1,figsize=(10, 3.5),
                            dpi=200,sharey=True,
                            gridspec_kw={'width_ratios': N*[1] + [0.4]})
    macros = data.mean(axis=-1) * 100
    print(macros)
    for n in range(N):
      sns.barplot(data=to_plot.query('model_name == @Models[@n]'),
                  x='model_name', y=metric,
                  hue='Sentiment', ax=axs[n], edgecolor="white",
                  palette=['#008ae6','#ffa31a','#00e600'])
      axs[n].plot([-0.5,0.5], [0.05,0.05],
                  linewidth=1.0, color='k')
      axs[n].annotate(f"{macros[n]:.1f}", (-0.53, 1.2),
                      bbox=dict(boxstyle="square", pad=0.12, fc="w"),
                      fontsize=12)
      axs[n].legend_.remove()
      axs[n].set_xlabel('')
      axs[n].set_xticklabels(['\n' + Models[n]], fontsize=14)

      axs[n].grid(which='major', axis='y', alpha=0.7,
                  color='k', linestyle='solid')
      axs[n].grid(which='minor', axis='y', alpha=0.7,
                  color='gray', linestyle='dashed')
      axs[n].minorticks_on()
      # axs[n].grid(True, axis='y')
```

```
      if n>0: axs[n].set_ylabel('', fontsize=1)
      else:
        axs[n].set_ylabel(f'Δ{metric}', fontsize=14)
        axs[n].set_ylim(-20.0, 20.0)
      axs[n].set_yticklabels(axs[n].get_yticks(), fontsize=12)

    plt.axis('off')
    plt.plot([0], linewidth=6.0, color='#008ae6')
    plt.plot([0], linewidth=6.0, color='#ffa31a')
    plt.plot([0], linewidth=6.0, color='#00e600')
    plt.legend(['Negative','Neutral','Positive'],
                fontsize=12,title="Sentiment",
                title_fontsize=14,
                loc='center left')
    fig.tight_layout()
    plt.show()
  return pd.DataFrame(data, columns=['Negative','Neutral','Positive'],
                      index=Models)

def main(Models: list[str], metrics_path: str, # Set models
         filename: str,                        # Set path to Waterfall
         plot_Hist: bool=False,                # Plot learning curves
         plot_CM: bool=False,                  # Plot confusion matrix
         plot_Waterfall: bool=False,           # Plot Waterfall of average metrics
         plot_Classes: bool=False,             # Plot Barplots of class metrics
         selectOverall: list[str]=['AUC','F1 Macro','TPR Macro','Overall ACC'],
         selectClass: list[str]=['AUC','F1','TPR','ACC']):
  info = ''
  metrics = {}
  plt_Overall = 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(current[4]))}\n"

    # Load variables to plot Overall and Class metrics:
    _overall = current[-2]
    _class = current[-1]

    # Define the variable with the standard deviations: ##############################
    _std = current[5] # (dictionary)
    keys = {'AUC':'AUC','Overall ACC':'ACC','F1 Macro':'F1','TPR Macro':'Sen','TNR
Macro':'Spe'}
```

```
    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, 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 matrix

    # Define variable to plot the Class metrics and Weighted Overall metrics: ########
    current_Class = [list(d.values()) for d in selectMetrics(_class,selectClass)] #
dict to array
    plt_Class[i] = np.asarray(current_Class) # array into matrix
    # END OF FOR LOOP

  # Plot the overall metrics ###################################################
  Overall_table = plotOverallMetric(plt_Overall, Models, selectOverall,
                                    filename=filename, plot=plot_Waterfall)
  # Plot metrics per class #####################################################
  Class_tables = {}
  for m in range(len(selectClass)):
    Class_tables[selectClass[m]] = \
    plotClassMetric(plt_Class[:,m,:],
                    renameModels(Models),
                    renameMetrics([selectClass[m]])[0],
                    plot=plot_Classes)
  # 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_duplicates=True)
      j+=1
  except:
    print("STD exception"); pass
```

```
  print(info) # Print all information gathered to the console
  return metrics, Overall_table, Class_tables, confMat

#%% Define Functions for Post-Analysis
def createTableOfSelected(selected: list[int], metrics: pd.DataFrame):
  """This takes the selected combinations and creates an ordered table.
  Applies all formatting to directly create the table for thesis"""
  def sort_function(values):
    sort_by = renameMetrics(['ACC', 'TPR', 'F1','AUC'])
    coefs = np.array([0.10, 0.30, 0.40, 0.20]) # Formula in Thesis

    print(values[sort_by])
    values = values[sort_by].values
    return np.dot(values,coefs)

  def get_source(metric:str) -> list[str]:
    names = renameMetrics([metric])
    names.append(names[0] + ' (σ)')
    return names

  modelsComb = [int(mn.split('_')[-2][1:]) in selected for mn in metrics.Model]
  filtered = metrics[modelsComb].drop(['Model'],axis=1)
  scores = sort_function(filtered)
  table = pd.DataFrame(
      {'ID':filtered.index,
       'ACC (σ)':[f"{m*100:.1f} ({s*100:.2f})"
                  for m,s in filtered[get_source('ACC')].values],
       'TPR (σ)':[f"{m*100:.1f} ({s*100:.2f})"
                  for m,s in filtered[get_source('TPR')].values],
       'F1 (σ)':[f"{m*100:.1f} ({s*100:.2f})"
                 for m,s in filtered[get_source('F1')].values],
       'AUC (σ)':[f"{m*100:.1f} ({s*100:.2f})"
                  for m,s in filtered[get_source('AUC')].values],
      })
  scores_index = scores.argsort()[::-1]
  print(scores_index)
  print(scores[scores_index])
  return table.iloc[scores_index]

def plotWhiskerSummary(metrics: pd.DataFrame):
  """Uses the summary dataframe to create these whisker plots.
  This way I show all data, not only top 5"""
  is_REC = metrics.Model.apply(lambda x: 'AREC' if 'Att' in x else 'REC')
  metrics.insert(0,"Arch", is_REC)
  # Select metrics
  use = ['PPV','F1','TPR','ACC','TNR','AUC','Arch']
  use = renameMetrics(use)
  metrics = metrics[use]
  use = use[:-1]
```

```
  arch = metrics.pop('Arch')
  for i,m in enumerate(use):
    aux = metrics.pop(m)
    if i==0:
      formated_metrics = pd.DataFrame(dict(Values=aux,
                                           Metrics=m,
                                           Architecture=arch))
    else:
      formated_metrics = pd.concat(
        [formated_metrics,pd.DataFrame(dict(Values=aux,
                                           Metrics=m,
                                           Architecture=arch))
        ],ignore_index=True)
  fig, ax = plt.subplots(figsize=(8,6),dpi=200)
  sns.violinplot(data=formated_metrics, ax=ax, orient='h',
                 x='Values', y='Metrics', hue='Architecture',
                 inner=None, split=True,
                 saturation=0.95, width=1.0, linewidth=1.0)
  plt.grid(which='major', axis='x',
           linewidth=0.9, color='k',
           linestyle='dashed', alpha=0.9)
  plt.grid(which='major', axis='y',
           linewidth=0.2, color='gray',
           linestyle='solid', alpha=0.95)
  plt.grid(which='minor', axis='x',
           linewidth=0.4, color='k',
           linestyle='dotted')
  plt.minorticks_on()
  ax.set_xlabel('')
  ax.set_ylabel('')
  return metrics
#%% MAIN
if __name__ == '__main__':
  # Define control variables: ================================================
  metrics_path = ['./TrainedModels&Metrics',          # 0
                  './TrainedModels&Metrics/THESIS',   # 1
                  ]
  Models = (
  [
  ############ Searching best preprocessing ################################## # 0
  # # REC_BC_C{?}:
  # # This is from 0 (minimum clean) to 126 (maximum clean)
  # *[f'EncDec_BestChrom_v2_C{i}_2fold' for i in range(127)],
  # *[f'EncDec_BestChrom_v2_C{i}_2fold' for i in [115, 100, 101]], # Top3

  # AREC_BC_C{?}:
  # # This is from 0 (no clean) to 126 (before maximum clean), and 127 is default
  # *[f'EncDecAtt_BestChrom_v2_C{i}_2fold' for i in range(127)],
  # *[f'EncDecAtt_BestChrom_v2_C{i}_2fold' for i in [101, 99, 122]], # Top3
  ],
```

```
[
############ Recurrent Encoder Decoder (REC) ################################ # 1
    # # Comparison #0.1: chapter 6.1
    # 'CSL_RED_TFIDF_C127_5fold',        #BASELINE TFIDF
    # 'CSL_RED_C127_5fold',              #BASELINE
    # 'REC_Base_vs_TFIDF.svg'            #FileName

    # # Comparison #0.2: Attachment D
    # 'CSL_RED_C0_5fold',                #BASELINE C0
    # 'CSL_RED_C127_5fold',              #BASELINE C127
    # 'REC_C0_vs_C127.svg'               #FileName

    # # Comparison #1: (1 vs 2) chapter 6.2, 1st image
    # 'CSL_RED_C127_5fold',              #BASELINE
    # 'CSL_RED_BC_C127_5fold',           #INTERMEDIATE
    # 'REC_Base_vs_Mid.svg'              #FileName

    # # Comparison #2: ((1 vs 2) vs 3)
    # # 'CSL_RED_C127_5fold',            #BASELINE
    # 'CSL_RED_BC_C127_5fold',           #INTERMEDIATE
    # 'CSL_RED_BC_C114_5fold',           #BESTCHROM
    # 'REC_Default_vs_Bestclean.svg'     #FileName

    # # Comparison #3: (1 vs (3 vs 4 vs 5)) chapter 6.2, 2nd image
    # # 'CSL_RED_C127_5fold',            #BASELINE
    # 'CIL_RED_BC_C114_5fold',           #IMBALANCED
    # 'CSL_RED_BC_C114_5fold',           #BESTCHROM
    # 'BTOS_RED_BC_C114_5fold',          #BTOS
    # 'REC_CSL_vs_CILandBTOS.svg'        #FileName

    # # # All (for Table)
    # 'CSL_RED_TFIDF_C127_5fold',
    # 'CSL_RED_C127_5fold',
    # 'CSL_RED_BC_C127_5fold',
    # 'CSL_RED_BC_C114_5fold',
    # 'BTOS_RED_BC_C114_5fold',
    # 'CIL_RED_BC_C114_5fold',

############ Recurrent Encoder-Decoder + Attention (AREC) ################## # 1
    # # Comparison #1: (1 vs 2) chapter 6.3, 1st image
    # 'CSL_REDA_C127_5fold',             #BASELINE
    # 'CSL_REDA_BC_C127_5fold',          #INTERMEDIATE
    # 'AREC_Base_vs_Mid.svg'             #FileName

    # # Comparison #2: ((1 vs 2) vs 3)
    # # 'CSL_REDA_C127_5fold',           #BASELINE
    # 'CSL_REDA_BC_C127_5fold',          #INTERMEDIATE
    # 'CSL_REDA_BC_C101_5fold',          #BESTCHROM
    # 'AREC_Default_vs_Bestclean.svg'    #FileName
```

```
    # # Comparison #3: (1 vs (3 vs 4 vs 5)) chapter 6.3, 2nd image
    # # 'CSL_REDA_C127_5fold',           #BASELINE
    # 'CIL_REDA_BC_C101_5fold',          #IMBALANCED
    # 'CSL_REDA_BC_C101_5fold',          #BESTCHROM
    # 'BTOS_REDA_BC_C101_5fold',         #BTOS
    # 'AREC_CSL_vs_CILandBTOS.svg'       #FileName

    # # # All (for Table)
    # 'CSL_REDA_C127_5fold',
    # 'CSL_REDA_BC_C127_5fold',
    # 'CSL_REDA_BC_C101_5fold',
    # 'BTOS_REDA_BC_C101_5fold',
    # 'CIL_REDA_BC_C101_5fold',

############ Comparison (RED vs. REDA) ##################################### # 1
    # # # BASELINE
    # 'CSL_RED_C127_5fold',               #BASELINE
    # 'CSL_REDA_C127_5fold',              #BASELINE
    # 'Comp_RECvsAREC_Baseline.svg'       #FileName

     # CSL
     'CSL_RED_BC_C114_5fold',           #BESTCHROM
     'CSL_REDA_BC_C101_5fold',          #BESTCHROM
     'Comp_RECvsAREC_Optimized.svg'     #FileName

    # Inference models (chapter 6.4.1)
    # 'CSL_REC_BC_C115_Inference',         #BESTCHROM
    # 'CSL_AREC_BC_C101_Inference',        #BESTCHROM
    # 'Comp_RECvsAREC_Inference.svg'       #Filename

############ Raspberry pi ################################################## # 1
    # # 'CSL_RED_BC_C114_Rpi',
  ])
COMPARE_COMBINATIONS = 0
COMPARE_MODELS = 1

select = COMPARE_MODELS
# ========================================================================
# Selects path from where to load the models
metrics_path = metrics_path[select]
Models = Models[select]

if 'svg' in Models[-1]:
  OverallFilename = Models.pop(-1)
else:
  OverallFilename = 'waterfall_plot.svg'
```

```
if select == 0:
  selectOverall = ['AUC','F1 Macro','TNR Macro','TPR Macro',
                   'Overall ACC','PPV Macro']
  selectClass = ['AUC','F1','TNR','TPR','ACC', 'PPV']
  if 'Att' in Models[0]:
    metrics_path += '/THESIS'
else:
  selectOverall = ['AUC','F1 Macro','TNR Macro','TPR Macro','Overall ACC']
  selectClass = ['AUC','F1','TNR','TPR','ACC']

# Run process to extract and plot metrics #!!!
All_metrics, Overall_metrics, \
  Class_metrics, confMat = main(Models,
                                metrics_path, OverallFilename,
                                plot_Hist=False,
                                plot_CM=False,
                                plot_Waterfall=False,
                                plot_Classes=False,
                                selectOverall=selectOverall,
                                selectClass=selectClass,
                                )
Overall_metrics_summary = Overall_metrics.describe().T
for k,v in Class_metrics.items():
  print(f"\n {k:>3} {60*'='}")
  print(v)

# Evaluation ##################################################################

# Only if 2 model are read (first is baseline and second is new)
if len(Models) == 2: # Compute ROI (relative overall improvement)
  def quick_renameModels(names: list[str]) -> list:
    return names, f"Cross-Validation with {names[0][-5:]}"

  (A, B), title = quick_renameModels(Models)
  metric = renameMetrics(['ACC','TPR','TNR','F1','AUC'])

  for m in metric:
    mA = Overall_metrics[Overall_metrics.Model == A][m].item()
    mB = Overall_metrics[Overall_metrics.Model == B][m].item()
    ROI = (mB-mA)/(1-mA)
    print(f"\nROI between {A} and {B}")
    print(f"based on {m} ({title}):")
    print(f"\t{mA*100:.2f}% --> {mB*100:.2f}%")
    print(f"\tROI: {ROI*100:+.2f}%")
```

```
  if len(Models) in [127, 128]: # Models from optimization: Tops Table
    n = 5
    # Find Top 5 models per metric
    topACC = Overall_metrics.sort_values(by=renameMetrics(['ACC']),
                                         ascending=False).head(n)
    topTPR = Overall_metrics.sort_values(by=renameMetrics(['TPR']),
                                         ascending=False).head(n)
    topF1S = Overall_metrics.sort_values(by=renameMetrics(['F1']),
                                         ascending=False).head(n)
    topAUC = Overall_metrics.sort_values(by=renameMetrics(['AUC']),
                                         ascending=False).head(n)
    # Merge the top 5 avoid considering the same model more than once
    topMerged = set()
    for top in [topACC,topTPR,topF1S,topAUC]:
      topMerged.update(top.index)
    print(topMerged,'length =', len(topMerged))
    # Create tables for Thesis
    TableThesis = \
    createTableOfSelected(topMerged, metrics=Overall_metrics)
    TableAppendix = \
    createTableOfSelected(Overall_metrics.index, metrics=Overall_metrics)
    # Notes to myself
    # REC model (EncDec_BestChrom) need a +1 on the cleaning ID (C##+1)
    # In REC, C0 is one cleaning step (the first combination)
    # In REC, C126 is all the cleaning steps (the last combination)
    # In REC, C127 does not exist
    # After (+1) is done, C0 disappears, and C127 must be deleted

    # AREC model (EncDecAtt_BestChrom) does not need +1.
    # In AREC, C0 is no-cleaning step (is not a combination)
    # In AREC, C1 is one cleaning step (the first combination)
    # In AREC, C126 is the previous of all the cleaning steps
    # In AREC, C127 is all the cleaning steps (the last combination)
    # C0 and C127 must be deleted to be consistent with REC

  if len(Models) == 254: # Models from optimization: Whisker plots
    _ = plotWhiskerSummary(Overall_metrics.copy())
##################################### END ######################################
```

- **W2VvsFT.py:** used to assess the difference in performance between Word2Vec and FastText vectorization methods.

```
#%% Load libraries:
import os
import pickle
import pandas as pd
from time import time, strftime, gmtime
from gensim.models.word2vec import Word2Vec
from gensim.models import FastText
from Cleaning.cleaningClass import Cleaning

#%% Load pt_reviews:
df = pd.read_csv('./Data/ZomatoFinal/TRAIN_PT_60_ImBalanced2.csv',
                 usecols=['raw_text','text', 'rating'])
cln = Cleaning()
steps = ['accents','lowercase','emojis',
         'stopwords','bigrams','punctuation','numbers']
cln.define_steps(steps)

df['text'] = cln.Clean(df['raw_text'],True)
Corpus_C127 = df.dropna(subset='text').text
Corpus_C127.to_csv("./W2V/W2V_training/Corpus_C127.csv", index=False)

#%% Corpus file generation:
def writeCorpus(lines, path='Corpus.txt'):
  # Write each document (review) in one separated line
  with open(path, 'w', encoding="utf-8") as f:
    for i,line in enumerate(lines):
      f.write(line)
      f.write('\n')
    else: print(i+1, "lines written into file")
def deleteCorpus(path='Corpus.txt'):
  os.remove(path)

#%% Vectorization with Word2Vec:
def train_w2v_model(text: pd.DataFrame, corpus_file: str,
                    window_size: int, vector_size: int, epochs: int,
                    vocab_size=None):
  w2v_model = Word2Vec(min_count=3,
                       window=window_size,
                       vector_size=vector_size,
                       alpha=0.03, negative=10, sample=0.001,
                       max_vocab_size=None,
                       max_final_vocab=vocab_size,
                       workers=8, sg=1)

  # Build vocabulary and Train:
  w2v_model.build_vocab(corpus_file=corpus_file)
  t = time()
```

```
  w2v_model.train(corpus_file=corpus_file,
                  total_words=len(w2v_model.wv),
                  total_examples=w2v_model.corpus_count,
                  epochs=epochs)
  print('Word2Vec training done:', strftime('%M:%S', gmtime(time()-t)))
  vocab_list = list(w2v_model.wv.key_to_index.keys())
  print("Vocab size:", len(vocab_list))
  raw_embedding_matrix = w2v_model.wv[vocab_list]
  print(60*'=')
  return w2v_model, vocab_list, raw_embedding_matrix

def train_fasttext_model(text: pd.DataFrame, corpus_file: str,
                         window_size: int, vector_size: int, epochs: int,
                         vocab_size=None):
  ft_model = FastText(min_count=3,
                      window=window_size,
                      vector_size=vector_size,
                      alpha=0.03, negative=10, sample=0.001,
                      max_vocab_size=None,
                      max_final_vocab=vocab_size,
                      workers=8, sg=1)

  # Build vocabulary and Train:
  ft_model.build_vocab(corpus_file=corpus_file)    # Read from file
  t = time()
  ft_model.train(corpus_file=corpus_file,
                  total_words=len(ft_model.wv),
                  total_examples=ft_model.corpus_count,
                  epochs=epochs)
  print('FastText training done:', strftime('%M:%S', gmtime(time()-t)))
  vocab_list = list(ft_model.wv.key_to_index.keys())
  print("Vocab size:", len(vocab_list))
  raw_embedding_matrix = ft_model.wv[vocab_list]
  print(60*'=')
  return ft_model, vocab_list, raw_embedding_matrix

def show_results(test, param):
  for v,result in test:
    print(f"For {param} = {v}")
    case = result.keys()
    for k in case:
      print('\t',k)
      for line in result[k]:
        print(2*'\t',line)

def get_results_from(keywords: list[str]):
  results = {k : model.wv.most_similar([k],topn=5)
             for k in keywords}
  return results
```

```
#%% Check Word2Vec performance
function = train_w2v_model

window_size_VALUES = [2, 4, 8, 10, 15]
vector_size_VALUES = [256, 300, 512, 768]
epochs_numb_VALUES = [5, 10, 15, 20, 50, 100]
words = ['comida','espaco','atendimento','agradavel','pessimo']

table = len(window_size_VALUES)* \
        [len(vector_size_VALUES)* \
         [len(epochs_numb_VALUES)* \
          [{ w:[('',0) for a in range(5)]  for w in words} \
           ]]]
# Corpus
corpus_file = './W2V/W2V_training/Corpus.txt'
writeCorpus(Corpus_C127, corpus_file)
print(60*'=')

for i in range(len(window_size_VALUES)):
  w = window_size_VALUES[i]
  for j in range(len(vector_size_VALUES)):
    v = vector_size_VALUES[j]
    for k in range(len(epochs_numb_VALUES)):
      e = epochs_numb_VALUES[k]

      print(f"Using window_size of {w:>3}")
      print(f"Using vector_size of {v:>3}")
      print(f"Using epochs_numb of {e:>3}")
      # Train model
      model, vocab_list, _ = function(Corpus_C127, corpus_file,
                                      window_size=w,
                                      vector_size=v,
                                      epochs=e,
                                      vocab_size=11560)
      # Get results to each keyword
      results = get_results_from(words)
      table[i][j][k] = results.copy()

deleteCorpus(corpus_file)
with open("./W2V/W2V_training/TableW2V.pkl",'wb') as f:
    pickle.dump(table, f)

#%% Check FastText performance
function = train_fasttext_model

window_size_VALUES = [2, 4, 8, 10, 15]
vector_size_VALUES = [256, 300, 512, 768]
epochs_numb_VALUES = [5, 10, 15, 20, 50, 100]
words = ['comida','espaco','atendimento','agradavel','pessimo']
```

```
table = len(window_size_VALUES)* \
        [len(vector_size_VALUES)* \
         [len(epochs_numb_VALUES)* \
          [{ w:[('',0) for a in range(5)]  for w in words} \
           ]]]
# Corpus
corpus_file = './W2V/W2V_training/Corpus.txt'
writeCorpus(Corpus_C127, corpus_file)
print(60*'=')

for i in range(len(window_size_VALUES)):
  w = window_size_VALUES[i]
  for j in range(len(vector_size_VALUES)):
    v = vector_size_VALUES[j]
    for k in range(len(epochs_numb_VALUES)):
      e = epochs_numb_VALUES[k]

      print(f"Using window_size of {w:>3}")
      print(f"Using vector_size of {v:>3}")
      print(f"Using epochs_numb of {e:>3}")

      # Train model
      model, vocab_list, _ = function(Corpus_C127, corpus_file,
                                      window_size=w,
                                      vector_size=v,
                                      epochs=e,
                                      vocab_size=11560)
      # Get results to each keyword
      results = get_results_from(words)
      table[i][j][k] = results.copy()

deleteCorpus(corpus_file)
with open("./W2V/W2V_training/TableFT.pkl",'wb') as f:
    pickle.dump(table, f)

#%% Read values ###################################################################
if False:
  with open("./W2V/W2V_training/TableW2V.pkl",'rb') as f:
    tableW2V = pickle.load(f)
  with open("./W2V/W2V_training/TableFT.pkl",'rb') as f:
    tableFT = pickle.load(f)
  """
  window_size_VALUES = [2, 4, 8, 10, 15]
  vector_size_VALUES = [256, 300, 512, 768]
  epochs_numb_VALUES = [5, 10, 15, 20, 50, 100]
  """
  vector300_tableW2V = tableW2V[1][1][0]
  vector300_tableFT = tableFT[1][1][0]
```

- **SentimentClassifier.py:** used to load a trained REC model to memory, or GPU, for inference. Includes the Python class SentimentClassifier and the editable function *text_cleaning*. A similar approach is done for loading a trained AREC model, considering it is necessary to change the tokenizer and the *text_cleaning* function.

```
class SentimentClassifier():
  """
  SentimentClassifier class is an API that allows the usage of the sentiment analysis
model developed for RRSO with ease. This model can analyze only Portuguese language.
It will not raise an exception for text in another language, yet its results may not
be the expected.
  This class defines the `predict` method to run inference over a list of text inputs.
  Predictions can be done for 2 sentiment outputs (Positive and Negative) or 3 senti-
ment outputs (Positive, Negative, and Mixed/Neutral).
  `predict` method does text preprocessing on input documents, using function
`text_cleaning`.
  The purpose of this class is to obtain the sentiment forecasts (from `predict`
method)
  from the provided text, which can be used in the same python scope (return value),
  or can be exported to a csv file of the form:
      Document                                                Sentiment
    0  Comida saborosa e atendimento muito bom.               Positive
    1  Atendimento e pessimo, mas as vistas espetaculares.       Mixed
      (...)                                                    (...)
  """
```

```
def __init__(self, model_path: str = "Models/Model_ED_BCv2_C114_RRSO",
                   maxlen: int = MAX_SEQ_LEN) -> None:
  """ Model EDBC114 developed for Zomato RRSO project.
  Parameters
  ----------
  model_path: str (default="Models/Model_ED_BCv2_C114_1")
      Path where model is stored. The model must be saved
      in a "Models" directory under the name "Model_<model_name>".
  maxlen: int (default=100)
      Maximum number of tokens the model is able to process by input
      sample.
      This parameter depends on the architecture of the model and should
      only be changed if the model being loaded requires it.
  Returns
  -------
  None
  """
  # Verify ´model_path´
  model_path = self.__verify_model_path(model_path)
  # Load the keras model from ´model_path´
  self._model = load_model(model_path + '/ModelSave', compile=True)

  # Read vocabulary from ´model_path´
  with open(model_path + "/vocab_list.pkl", 'rb') as f:
    self.vocab = pickle.load(f)
  # Load tokenizer (Uses vocab_list without special tokens)
  self._tokenizer = TextVectorization(standardize=None,
                                      output_mode='int',
                                      max_tokens=len(self.vocab)+2,
                                      vocabulary=self.vocab,
                                      output_sequence_length=maxlen)
  # Update vocabulary list
  self.vocab = ['','[UNK]'] + self.vocab
  # Check model is running properly
  self.__dummy_test()
  # Load performance
  with open(f"{model_path}/ConfusionMatrix/Overall.pkl", 'rb') as f:
    metrics = pickle.load(f)
    self.ACC_OVR = metrics['ACC'] if metrics.get('ACC')
                                  else metrics['Overall ACC']
  with open(f"{model_path}/ConfusionMatrix/Class.pkl", 'rb') as f:
    metrics = pickle.load(f)
    if isinstance(metrics['ACC'], dict):
      self.ACC_NEG, self.ACC_NEU, self.ACC_POS = metrics['ACC'].values()
    else:
      self.ACC_NEG, self.ACC_NEU, self.ACC_POS = metrics['ACC']
  # Verbose
  print(f"[{datetime.now()}] Model {model_path.split('/')[-1][6:]}
          loaded successfully!")
```

```
def print_performance(self) -> None:
  """ Display accuracy of the model """
  print("Accuracy of each sentiment prediction:",
        "{:^8} {:^8} {:^8}".format(*LABELS),
        "{:^8.4f} {:^8.4f} {:^8.4f}"\
        .format(self.ACC_NEG,self.ACC_NEU,self.ACC_POS),
        "General Accuracy of the model: {:.4f}"\
        .format(self.ACC_OVR),
        sep='\n',end='\n\n')

def to_csv(self, filename: str) -> None:
  """
  Save all predictions stored in history in a csv file for exporting
  purposes. Details such as cleaned document, or weight of each
  sentiment are not stored.
  Parameters
  ----------
  filename: str
      Specify the name of the file where results will be saved
  Return
  ----------
  None
  """
  to_write = ['Document,Sentiment\n',]
  for d,r in zip(self.reviews_history, self.forecast_history):
    to_write.append('"' + d + '",' + r + '\n')
  with open(filename, 'w',encoding='utf-8') as f:
    f.writelines(to_write)
```

```
def predict(self, documents: list[str],
                  n_outputs: int=3,
                  verbose: bool=False,
                  time_it: bool=False) -> np.array:
  """
  Method to forecast the sentiment of a review or set of reviews.
  Parameters
  ----------
  documents: list[str], or str
      List of documents to be forecasted. If a string is passed, will
      not raise an error, just re-format the input
  n_outputs: int (default=3)
      Number of outputs of the model. Values can be 2,
      meaning the sentiment output is either Positive or Negative, or 3
      (default) meaning is either Positive, Negative, or Mixed.
  verbose: bool (default=False)
      Whether or not print information about the forecasts.
  time_it: bool (default=False)
      Whether or not print information about the time consumed.
  Return
  ----------
  forecasts: np.array -> shape=( len(documents), n_outputs )
      Forecast weights for each input document.
  """
  # Verify `n_outputs` value
  assert n_outputs == 2 or n_outputs == 3, "Parameter 'n_outputs' can
                                            only take values 2 and 3."
  # Verify ´documents´ datatype
  if isinstance(documents, str):
    documents = [documents]
  # Perform text preprocessing (Cleaning and Tokenization)
  cleaned_docs = self._preprocessing(documents, time_it=time_it)
  tokenized_docs = self._tokenizer(cleaned_docs)
  # Forecast the sentiment of the preprocessed documents (3-class)
  if time_it:
    t = datetime.now()
    print("\nAnalyzing input reviews...")
  forecasts = self._model.predict(tokenized_docs, verbose=verbose)
  if n_outputs == 2:
    forecasts =  np.asarray([self.__reshape_forecast(f)
                             for f in forecasts])
  if verbose:
    self._display_results(documents, cleaned_docs,
                          tokenized_docs, forecasts)
  if time_it:
    print(f"Finished. Time of analysis:
            {self.__format_datetime(datetime.now()-t)}\n")
  self.reviews_history.extend(documents)
  self.forecast_history.extend(self.__forecast_to_sent(forecasts))
  return forecasts
```

```
def history(self, head: int=-1) -> None:
    """
    Print the `head` first pairs of review/forecast stored in history.
    If head is not passed, this method print the whole history.
    """
    print(40*'=')
    for i,(r,f) in enumerate(
                          zip(self.reviews_history,self.forecast_history)):
      print('REVIEW:',r)
      print('SENTIMENT:',f)
      print(40*'=')
      if i+1 == head: break

  def reset_history(self) -> None:
    """
    Empty history of forecasts.
    This method can be used when you need to create a csv file only with
    Next forecasts. All previous history will be deleted.
    """
    self.reviews_history = []
    self.forecast_history = []

  def _preprocessing(self, docs: list[str],
                           time_it: bool=False) -> list[str]:
    """ Apply text cleaning to input documents """
    if time_it:
      t = datetime.now()
      print("\nPerforming text cleaning over documents...")
      cleaned = [text_cleaning(d) for d in docs]
      print(f"Finished. Time of cleaning:
              {self.__format_datetime(datetime.now()-t)}\n")
      return cleaned
    else:
      return [text_cleaning(d) for d in docs]

  def __str__(self) -> str:
    """ Print the model arquitecture """
    self._model.summary()
    return ''

  def __verify_model_path(self, model_path: str) -> bool:
    Models_dir, model_name =  model_path.split('/')[-2:]
    assert Models_dir=='Models' and model_name[:6]=='Model_',
            "Model must be stored in a directory 'Models' under
             the name 'Model_<model_name>'"
    if model_path[-1] == '/': model_path = model_path[:-1]
    return model_path
```

```
def _display_results(self, documents: list[str],
                           cleaned_docs: list[str],
                           tokenized_docs: np.ndarray,
                           forecasts: np.array) -> None:
  " Print results of each review, including internal specifications."
  n_outputs = forecasts.shape[-1]
  labels = LABELS[::2] if n_outputs==2 else LABELS
  for i in range(len(documents)):
    if len(documents)==1: print("Forecast",48*"=" + '\n')
    else: print(f"Forecast #{i+1}",48*"=" + '\n')

    print(f"Input text: {documents[i]}\n")
    data = self.__mask_padding(tokenized_docs[i])
    print(f"Cleaned text: {cleaned_docs[i]}\n")
    print(f"Data seen by Model: {data}\n")
    self.__print_weights(forecasts[i], labels)
    print(60*"=")

def __print_weights(self, forecast: np.ndarray, labels: list) -> None:
  """ Print weights table according with number of outputs
      (n_outputs)."""
  table_title = '\t' + len(labels)*"{:>8} "
  table_row   = '\t' + len(labels)*"{:>8.4f} "
  result = np.argmax(forecast)

  print("Forecast:", labels[result] )
  print("Forecast weights:",
        table_title.format(*labels),
        table_row.format(*forecast),
        sep='\n')

def __mask_padding(self, tokenized_docs: np.ndarray) -> list:
  masked = []
  for token in tokenized_docs.numpy():
    if token == 0: break
    masked.append(token)
  return masked

def __reshape_forecast(self, forecast: np.ndarray) -> list:
  """ Convert forecast to 2-class probabilities"""
  neg, mix, pos = forecast

  if (neg + mix) > (mix + pos):
    return [neg + mix, pos]
  else:
    return [neg, mix + pos]
```

```
def __format_datetime(self, t: dt.timedelta) -> str:
  hours = t.seconds//3600
  minutes = (t.seconds//60)%60
  seconds = t.seconds%60
  micros = t.microseconds*1e-6
  if hours > 0:
    str_t = f"{hours}:{minutes}:{seconds}."
  elif minutes > 0:
    str_t = f"{minutes} minutes and {seconds + micros} seconds."
  elif seconds > 0 or micros > 0:
    str_t = f"{seconds + micros} seconds."
  else:
    str_t = "Almost instantaneously"
  return str_t

def __forecast_to_sent(self, forecasts: np.ndarray) -> list:
  n_outputs = forecasts.shape[-1]
  labels = LABELS[::2] if n_outputs==2 else LABELS

  results = np.argmax(forecasts, axis=-1)
  return [labels[r] for r in results]

def __dummy_test(self) -> None:
  documents = \
      ["Pizza maravilhosa. Tivemos que fazer marcação porque o lugar
        fica sempre cheio à tarde, mas valeu a pena. 100% italiana!",
       "O sitio não é amigável com as mascotas. Levamos o nosso cão, ele
        é mesmo tranquilo, não ladra sequer. Mandaram-nos comer na
        esplanada. Muito mau mesmo, não há respeito",
       "Lugar espetacular. A vista ao mar é incrível. O atendimento é
        profissional, mas os pratos não estão à altura. Só recomendo
        pelos postres e para passar a tarde com boas vistas"
       ]
  # Initialize forecast history
  self.reviews_history = []
  self.forecast_history = []

  try:
    _ = self.predict(documents)
  except:
    print("Something went wrong when checking the model.",
          "Please notify RRSO team.",
          sep='\n')
    self._model = None
    self._tokenizer = None
  self.reset_history()
```

```
def text_cleaning(text : str) -> str:
  """ Perform text cleaning over a single document:
      - Fix formatting errors
      - Convert to lowercase
      - Remove emojis
      - Remove Stopwords
      - Create rule-based bi-grams
      - Remove numbers
      - Remove duplicated blank spaces
  The model is optimized to work with this pipeline. However, this
  function can be customized by the user.
  """
  # Fix text errors
  text = text.replace('&amp;','&') # fix '&' format
  for code in dictHTML: # convert html special char to letters
    text = text.replace(code, dictHTML[code])
  text = text.replace('$%', '').replace('°', '').replace('ª', '')\
            .replace('_', ' ').replace('-', ' ').replace('#', '')
  text = re.sub(r'([-:/\\\\(){}!"#*+,&\'.])\1+', r'\1', text)
  text = re.sub("{uid:[0-9]+}", " ", text)
  # Convert to lowercase
  text = text.lower()
  # Remove emojis
  text = emoji.replace_emoji(text,' ')
  # Separate punctuation
  text = "".join([char if char.isalnum() else f" {char} "
                  for char in text])
  # Remove Stopwords
  text = [word for word in text.split() if word.lower() not in STOPWORDS]
  text = [w for w in text if len(w)>2 or not w.isalpha()]
  # Create rule-based bigrams
  prefixes = ['nao','não','sem','muito','muita','muitos','muitas']
  jump = 0; joined_text = []
  for i in range(len(text)):
    if jump: jump = 0; continue
    bigram = text[i:i+2]
    if len(bigram) < 2: joined_text.append(bigram[0])
    else:
      if bigram[0] in prefixes:
        joined_text.append('_'.join(bigram)); jump = 1
      else:
        joined_text.append(bigram[0]); jump = 0
  text = " ".join(joined_text)
  # Remove numbers
  text = "".join(map(lambda x: ' ' if x.isdigit() else x, text))

  return " ".join(text.split())
```

- **ga.py:** Genetic Algorithm used during the optimization stage of REC and AREC models. Here, only the relevant functions are provided: *fitness_score* (or each model), *selection*, *mutation*, *crossover*.

```
def fitness_score(population, GenerationsFile): # REC Architecture
  print(f"Fitness score of Generation {GenerationsFile}\n")
  scores = [] # Stores the meanAUC after each chromosome
  for id_pop, chromosome in enumerate(population):
    print(f"Generation {GenerationsFile}\nChromosome {id_pop}:")
    decoded_chromosome = decode_chromosome(chromosome, model='EncDec',
                                           verbose=VerboseChromosome)
    NumberClasses     =     3
    NumberOfTime steps =     100
    Optimizer         =     'rmsprop'
    #                                                    # BEST manual
    Bidirectional     =     decoded_chromosome[ 0]       # False
    RNNtype           =     decoded_chromosome[ 1]       # LSTM
    NumberRRNLayers   =     decoded_chromosome[ 2]       # 1
    ShapeOfRNN        =     decoded_chromosome[ 3]       # 200
    PercentageDropout =     decoded_chromosome[ 4]       # 0.4
    NumberOfFeatures  =     decoded_chromosome[ 5]       # 300
    ShapeOfDenseLayer = int(decoded_chromosome[ 6]*ShapeOfRNN) # 0
    Activation        =     decoded_chromosome[ 7]       # relu
    EncoderX2X        =     decoded_chromosome[ 8]       # m2m
    BatchSize         =     decoded_chromosome[ 9]       # 128

    # Variables to store metrics
    AccAtEnd = np.zeros(NumberFolds)
    SenAtEnd = np.zeros(NumberFolds)
    SpeAtEnd = np.zeros(NumberFolds)
    AUCAtEnd = np.zeros(NumberFolds)

    _, VocabList, raw_embedding_matrix = load_embeddings(NumberOfFeatures, VocabSize)

    kf = KFold(n_splits=NumberFolds,shuffle=True)
    for fn,(tr,te) in enumerate(kf.split(df)):
      gc.collect()
      tf.keras.backend.clear_session() # Start with a blank state at each iteration.
      print(f"\n[Gen{GenerationsFile}|Chrom{id_pop}] Cross Validation, fold {fn}\n")
      # Set datasets
      df_train = df.iloc[tr]
      df_test, df_val = train_test_split(df.iloc[te],test_size=0.33,shuffle=True)

      print("(%train, %val, %test) =",
            len(df_train)/len(df),len(df_val)/len(df),len(df_test)/len(df))
      # Extract features:
      XTrain = df_train['text']
      XValid = df_val['text']
      XTest  = df_test['text']
```

```
# Extract labels:
YTrain = df_train['rating']
YValid = df_val['rating']
YTest  = df_test['rating']
# Apply cost-sensitive learning
classes = np.unique(YTrain)
class_weights = class_weight.compute_class_weight('balanced',
                                                  classes=classes,
                                                  y=YTrain)
class_weights = {c:w for c,w in enumerate(class_weights)}
# Encode labels into integers:
map_labels = {r:i for i,r in enumerate(classes)}
inv_map_labels = {v:k for k,v in map_labels.items()}
YTrain = list(map(lambda x: map_labels[x],YTrain))
YValid = list(map(lambda x: map_labels[x],YValid))
YTest = list(map(lambda x: map_labels[x],YTest))
# Apply One-Hot encoding
encodeYTrain = np_utils.to_categorical(YTrain, NumberClasses)
encodeYValid = np_utils.to_categorical(YValid, NumberClasses)
# Vectorization:
vocab_list = VocabList.copy()
vocab_size = len(vocab_list)+2
vectorize_layer = Sequential(name="Vectorization")
vectorize_layer.add(layers.Input(shape=(1,), dtype=tf.string))
vectorize_layer.add(TextVectorization(max_tokens=vocab_size,
                                      standardize=None, # already done
                                      output_mode='int',
                                      output_sequence_length=NumberOfTime steps,
                                      vocabulary=vocab_list))
#Fullfill vocabulary:
for token in vectorize_layer.layers[0].get_vocabulary()[:2][::-1]: # new tokens
  vocab_list.insert(0,token)
# Add Special tokens to embedding matrix:
PADvector = np.zeros((1,NumberOfFeatures))
UNKvector = np.zeros((1,NumberOfFeatures))
embedding_matrix = np.concatenate([PADvector, UNKvector, raw_embedding_matrix])

print("Tokenization layer...")
t = time()
tokenXTrain = vectorize_layer.predict(XTrain)
tokenXValid = vectorize_layer.predict(XValid)
tokenXTest = vectorize_layer.predict(XTest)
print(f" (Elapsed time: {strftime('%M:%S', gmtime(time()-t))})\n")
trainData = tokenXTrain
validData = tokenXValid
testData =  tokenXTest
```

```
    # Set Embedding and Masking layers:
    embedding = Embedding(input_dim=vocab_size,
                          output_dim=NumberOfFeatures,
                          mask_zero=True,
                          weights=[embedding_matrix],
                          input_length=NumberOfTime steps,
                          name="Embeddings")
    # Define Input ================================================
    encoder_inputs = layers.Input(shape=(NumberOfTime steps,), name='Input_se-
quence')
    embed_input = embedding(encoder_inputs)
    embed_input = layers.Dropout(PercentageDropout)(embed_input)
    # Define Encoder ==============================================
    shapeofRNN = ShapeOfRNN // 2 if Bidirectional == 'Bi' else ShapeOfRNN
    enc_command = f"{Bidirectional}({RNNtype}(shapeofRNN,\
                    return_sequences={'{}'}))"
    if NumberRRNLayers == 1:
      if EncoderX2X == 'm2o':
        encoder_output = eval(enc_command.format(False))(embed_input)
        encoder_output = layers.RepeatVector(NumberOfTime steps)(encoder_output)
      elif EncoderX2X == 'm2m':
        encoder_output = eval(enc_command.format(True))(embed_input)
    else:
      x = eval(enc_command.format(True))(embed_input)
      for _ in range(NumberRRNLayers-2):
        x = eval(enc_command.format(True))(x)

      if EncoderX2X == 'm2o':
        encoder_output = eval(enc_command.format(False))(x)
        encoder_output = layers.RepeatVector(NumberOfTime steps)(encoder_output)
      elif EncoderX2X == 'm2m':
        encoder_output = eval(enc_command.format(True))(x)
    # Define Decoder ==============================================
    dec_command = enc_command
    if NumberRRNLayers == 1:
      decoder_output = eval(dec_command.format(False))(encoder_output)
    else:
      x = eval(dec_command.format(True))(encoder_output)
      for _ in range(NumberRRNLayers-2):
        x = eval(dec_command.format(True))(x)
      decoder_output = eval(dec_command.format(False))(x)
    # Define FFNN =================================================
    if ShapeOfDenseLayer != 0:
      decoder_output = layers.Dense(ShapeOfDenseLayer,
                                    activation=None)(decoder_output)
      decoder_output = layers.Activation(Activation)(decoder_output)
    ffnn_output = layers.Dropout(PercentageDropout)(decoder_output)
    # Define classifier ===========================================
    outputs = layers.Dense(NumberClasses, activation=None)(ffnn_output)
    outputs = layers.Activation('softmax')(outputs)
```

```
model = Model(inputs=encoder_inputs, outputs=outputs)
model.summary()
model.compile(loss='categorical_crossentropy',
              optimizer=Optimizer,
              metrics=['accuracy', 'AUC'])
# Define EarlyStopping
es = tf.keras.callbacks.EarlyStopping(monitor="val_auc",
                                      patience=PatienteceValue,
                                      min_delta=min_delta,
                                      verbose=1, mode='max',
                                      restore_best_weights=True)
# Define Garbage collector
class ClearMemory(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    gc.collect()
    tf.keras.backend.clear_session()
# Define Scheduler
def schedule(epoch, lr):
  if epoch < 3: # warm up
    return lr
  else:
    if lr < 1e-5: # limiter
      return lr
    exponential = lr * tf.math.exp(-0.1)
    return exponential
sch = tf.keras.callbacks.LearningRateScheduler(schedule, verbose=0)
# Define Training
print("Training started...")
t = time()
model.fit(trainData, encodeYTrain,
          batch_size=BatchSize,
          epochs=EpochsValue,
          validation_data=(validData, encodeYValid),
          verbose=VerboseTrain,
          class_weight=class_weights,
          callbacks=[es, sch, ClearMemory()])
print("Training finished...")
trainTime = time() - t
# Test model
testForecast = model.predict(testData)
testYhat = np.argmax(testForecast, axis=-1)
classTest = classification_report(YTest, testYhat)
# Mapping integers to actual labels:
YTest = list(map(lambda x: inv_map_labels[x], YTest ))
testYhat = list(map(lambda x: inv_map_labels[x], testYhat ))
cmTest = ConfusionMatrix(YTest, testYhat,is_imbalanced=True)
if VerboseGen:
  print("Classification report TEST",classTest,sep='\n')
  print(f"\nTraining time: {strftime('%H:%M:%S', gmtime(trainTime))}\n")
```

```
    AccAtEnd[fn] = cmTest.Overall_ACC
    SenAtEnd[fn] = cmTest.TPR_Macro
    SpeAtEnd[fn] = cmTest.TNR_Macro
    AUCAtEnd[fn] = metrics.roc_auc_score(YTest, testForecast,
                                         average='weighted',multi_class='ovr')
    del model, vocab_list, embedding, vectorize_layer
    # END OF FOR LOOP

  # Calculate metrics combining K folds
  AvgACC = np.mean(AccAtEnd) * 100
  AvgSen = np.mean(SenAtEnd) * 100
  AvgSpe = np.mean(SpeAtEnd) * 100
  AvgAUC = np.mean(AUCAtEnd) * 100

  StdACC = np.std(AccAtEnd) * 100
  StdSen = np.std(SenAtEnd) * 100
  StdSpe = np.std(SpeAtEnd) * 100
  StdAUC = np.std(AUCAtEnd) * 100

  if VerboseGen:
    print('Final results: ')
    print(f'Accuracy: Avg({AvgACC:.2f}%), Std({StdACC:.2f}%)')
    print(f'Sensitivity: Avg({AvgSen:.2f}%), Std({StdSen:.2f}%)')
    print(f'Specificity: Avg({AvgSpe:.2f}%), Std({StdSpe:.2f}%)')
    print(f'AUC: Avg({AvgAUC:.2f}%), Std({StdAUC:.2f}%)\n')
  if id_pop == 0:
    header = f"New generation: {GenerationsFile}\nPopulation {id_pop}\n"
  else:
    header = f"Population {id_pop}\n"
  with open(MetricsPath + "ACC.txt", 'a') as f:
    f.write(header)
    f.write(f"\t{AvgACC}\n\t{StdACC}\n")
  with open(MetricsPath + "Sen.txt", 'a') as f:
    f.write(header)
    f.write(f"\t{AvgSen}\n\t{StdSen}\n")
  with open(MetricsPath + "Spe.txt", 'a') as f:
    f.write(header)
    f.write(f"\t{AvgSpe}\n\t{StdSpe}\n")
  with open(MetricsPath + "AUC.txt", 'a') as f:
    f.write(header)
    f.write(f"\t{AvgAUC}\n\t{StdAUC}\n")
  scores.append(AvgAUC) # here I decide what metric I want to consider

# Convert into arrays
scores = np.array(scores)
population = np.array(population)
# sort and reverse order of `scores`
inds = np.flip(np.argsort(scores)) # first is best, last is worst
```

```
  # Creates next generation based on current generation scores (meanAUC)
  scoresGenTemp = np.zeros((NumbParents))
  population_nextgenTemp = np.zeros((NumbParents, NumbBitsDecimal))
  for i,gen in enumerate(inds):
    scoresGenTemp[i] = scores[gen]
    population_nextgenTemp[i, :] = population[gen, :]

  if VerboseGen: print(f"Finished Fitness score for Generation {GenerationsFile}")
  return scoresGenTemp, population_nextgenTemp
```

```
def fitness_score(population: list, gen: int) -> (np.array, np.array): # AREC Arch
  print(f"Fitness score of Generation {gen}\n")
  scores = [] # Stores the meanAUC after each chromosome
  for id_pop, chromosome in enumerate(population):
    print(f"Generation {gen}\nChromosome {id_pop+1}:")
    decoded_chromosome = decode_chromosome(chromosome, model='EncDecAtt',
                                           verbose=VerboseChromosome)
    NumberClasses     =     3
    NumberOfTime steps =     100
    Optimizer         =     tf.keras.optimizers.RMSprop(learning_rate=0.001)
    #
    Bidirectional     =     decoded_chromosome[ 0]
    NumberLSTMLayers  =     decoded_chromosome[ 1]
    PositionAttention =     decoded_chromosome[ 2]
    ModelDimension    =     decoded_chromosome[ 3]
    ShapeOfProjection =     decoded_chromosome[ 4]
    PercentageDropout =     decoded_chromosome[ 5]
    Decoder           =     decoded_chromosome[ 6]
    ShapeOfDenseLayer = int(decoded_chromosome[ 7]*ModelDimension)
    Activation        =     decoded_chromosome[ 8]
    Replicate         =     decoded_chromosome[ 9]
    Tokenizer         =     decoded_chromosome[10]
    if ModelDimension == 300: # Particular case
        DxH ={32: 30, 64: 60, 128: 100, 256: 150}
        ShapeOfProjection = DxH[ShapeOfProjection]
    NumberHeads       = int(ModelDimension/ShapeOfProjection)

    # Variables to store metrics
    AccAtEnd = np.zeros(NumberFolds)
    AUCAtEnd = np.zeros(NumberFolds)
    SenAtEnd = np.zeros(NumberFolds)
    SpeAtEnd = np.zeros(NumberFolds)
    F1sAtEnd = np.zeros(NumberFolds)
    kf = StratifiedKFold(n_splits=NumberFolds, shuffle=True)
    for fn,(tr,te) in enumerate(kf.split(df, df['rating'])):
      gc.collect()
      tf.keras.backend.clear_session() # Starts with a blank state at each iteration.
      print(f"\n[Gen{gen}|Chrom{id_pop+1}] Cross Validation, fold {fn}\n")
```

```
# Define datasets
df_train = df.iloc[tr]
df_test, df_val  = train_test_split(df.iloc[te],test_size=0.40,
                                    shuffle=True,stratify=df.iloc[te]['rating'])
print("(%train, %val, %test) = ({}, {}, {})".\
      format(len(df_train)/len(df),len(df_val)/len(df),len(df_test)/len(df)))
# Extract features:
if Tokenizer == "BlankSpace":
  print("Selected default pre-processing")
  XTrain = df_train['text']
  XValid = df_val['text']
  XTest  = df_test['text']
elif Tokenizer == "WordPiece":
  print("Selected no-stopwords pre-processing")
  XTrain = df_train['raw_text']
  XValid = df_val['raw_text']
  XTest  = df_test['raw_text']
# Extract labels:
YTrain = df_train['rating']
YValid = df_val['rating']
YTest  = df_test['rating']
# Apply cost-sensitive learning
classes = np.unique(YTrain)
class_weights = class_weight.compute_class_weight('balanced',
                                                  classes=classes,
                                                  y=YTrain)
class_weights = {c:w for c,w in enumerate(class_weights)}
# Apply One-Hot encoding
encodeYTrain = np_utils.to_categorical(YTrain, NumberClasses)
encodeYValid = np_utils.to_categorical(YValid, NumberClasses)
# Vectorization:
if Tokenizer == "BlankSpace":
  _, VocabList, raw_embedding_matrix = \
      VocabBuilder.Word2VecVocabulary(XTrain,
                                      vector_size=ModelDimension,
                                      maxlen=NumberOfTime steps,
                                      vocab_size=VocabSize)
  vocab_list = VocabList.copy()
  vocab_size = len(vocab_list)+2
  print("Vocab size:", vocab_size)
  vectorize_layer = TextVectorization(max_tokens=vocab_size,
                                      standardize=None,
                                      output_mode='int',
                                      output_sequence_length=NumberOfTime steps,
                                      vocabulary=vocab_list)
  #Fullfill vocabulary:
  for token in ['[UNK]','[PAD]']: # include new tokens
    vocab_list.insert(0,token)
```

```
    # Add Special tokens to embedding matrix:
    PADvector = np.zeros((1,ModelDimension))
    UNKvector = np.zeros((1,ModelDimension))
    embedding_matrix = np.concatenate([PADvector, UNKvector,
                                       raw_embedding_matrix])
    position_matrix = build_pos_matrix(NumberOfTime steps,ModelDimension)
elif Tokenizer == "WordPiece":
    _, VocabList, raw_embedding_matrix = \
        VocabBuilder.WordPieceVocabulary(XTrain,
                                         vector_size=ModelDimension,
                                         maxlen=NumberOfTime steps,
                                         vocab_size=VocabSize)
    vocab_list = VocabList.copy()
    vocab_size = len(vocab_list)+1
    print("Vocab size:", vocab_size)
    #Fullfill vocabulary:
    UNK_idx = 0
    for i,w in enumerate(vocab_list):
      if w == '[UNK]': UNK_idx = i
    _ = vocab_list.pop(UNK_idx)
    for token in ['[UNK]','[PAD]']: # include new tokens
      vocab_list.insert(0,token)
    # Add Special tokens to embedding matrix:
    PADvector = np.zeros((1,ModelDimension))
    UNKvector = raw_embedding_matrix[UNK_idx:UNK_idx+1]
    raw_embedding_matrix = np.concatenate(
                          [raw_embedding_matrix[:UNK_idx,:],
                           raw_embedding_matrix[UNK_idx+1:,:]])
    embedding_matrix = np.concatenate([PADvector, UNKvector,
                                       raw_embedding_matrix])
    position_matrix = build_pos_matrix(NumberOfTime steps,ModelDimension)
    vectorize_layer = WordPieceTokenizer(vocabulary=vocab_list,
                                         sequence_length=NumberOfTime steps)
print("Tokenization layer...")
t = time()
tokenXTrain = vectorize_layer(XTrain)
tokenXValid = vectorize_layer(XValid)
tokenXTest = vectorize_layer(XTest)
print(f"(Elapsed time: {strftime('%M:%S', gmtime(time()-t))})\n")
trainData = tokenXTrain
validData = tokenXValid
testData =  tokenXTest
```

```
    # BUILD MODEL ********************************************** #!!!
    try:
      # Set embedding layer ===============================================
      if PositionAttention in ['pre', 'pre-post']: # Token&Positional embedding
        EmbeddingLayer = TokenAndPositionEmbedding(vocab_size,ModelDimension,
                                                   maxlen=NumberOfTime steps,
                                                   weights_tokens=embedding_matrix,
                                                   weights_position=position_matrix,
                                                   mask_zero=True,
                                                   name='TokenPosEmbedding')
      elif PositionAttention in ['No','post']: # Use Token embedding
        EmbeddingLayer = Embedding(vocab_size,ModelDimension,
                                   input_length=NumberOfTime steps,
                                   weights=[embedding_matrix],
                                   mask_zero=True,
                                   name='TokenEmbedding')
      # Define Input ======================================================
      model_input = layers.Input(shape=(NumberOfTime steps,), name='Input_sequence')
      embed_output = EmbeddingLayer(model_input)
      encoder_input = layers.Dropout(PercentageDropout)(embed_output)
      mask_input = EmbeddingLayer.compute_mask(model_input)

      temp = encoder_input
      for i in range(Replicate+1):
        # Encoder: PreAttention
        if PositionAttention in ['pre', 'pre-post']:
          PreAttentionLayer = MHAttention(heads=NumberHeads,
                                          dim_K=ShapeOfProjection,
                                          dim_V=ShapeOfProjection,
                                          d_model=ModelDimension,
                                          activation=Activation,
                                          name=f'PreAttention{i+1}')
          attention_output = PreAttentionLayer(temp,temp,
                                               attention_mask=mask_input)
        else:
          attention_output = temp

        # Recursive Encoder ===============================================
        ShapeOfLSTM = ModelDimension // 2 if Bidirectional == 'Bi' \
                                          else ModelDimension
        RecursiveEncoder = f"{Bidirectional}(LSTM({ShapeOfLSTM}, return_se-
quences=True))"
        if NumberLSTMLayers == 1:
          encoder_output = eval(RecursiveEncoder)(attention_output)
        else:
          x = eval(RecursiveEncoder)(attention_output)
          for _ in range(NumberLSTMLayers-2):
            x = eval(RecursiveEncoder)(x)
          encoder_output = eval(RecursiveEncoder)(x)
```

```
  if PositionAttention in ['post','pre-post']:
    # Create Position embedding before postAttention layer
    encoder_output = encoder_output + position_matrix
    PostAttentionLayer = MHAttention(heads=NumberHeads,
                                     dim_K=ShapeOfProjection,
                                     dim_V=ShapeOfProjection,
                                     d_model=ModelDimension,
                                     activation=Activation,
                                     name=f'PostAttention{i+1}')
    temp = PostAttentionLayer(encoder_output,encoder_output,
                              attention_mask=mask_input)
  else:
    temp = encoder_output
  if PositionAttention == 'pre-post': break
decoder_input = temp
# Define Decoder ================================================
if Decoder == 'Dense':
  ContextDecoder = \
  Sequential([layers.Input((NumberOfTime steps,ModelDimension)),
              layers.Flatten(),
              layers.Dense(ModelDimension, activation=None),
              layers.Activation(Activation)
              ], name='Decoder_MLP')
elif Decoder == 'Pooling':
  ContextDecoder = \
  Sequential([layers.Input((NumberOfTime steps,ModelDimension)),
              layers.AveragePooling1D(pool_size=10,strides=8),
              layers.GlobalAveragePooling1D()
              ], name='Decoder_AvgPool')
elif Decoder == 'LSTM':
  ContextDecoder = LSTM(ModelDimension, return_sequences=False,
                        name='Decoder_LSTM')
elif Decoder == 'Bi-LSTM':
  ContextDecoder = Bi(LSTM(ModelDimension//2, return_sequences=False,
                           name='Decoder_BiLSTM'))
decoder_output = ContextDecoder(decoder_input)
# Define FFNN ===================================================
if ShapeOfDenseLayer != 0:
  ffnn_output = layers.Dense(ShapeOfDenseLayer, activation=None,
                             name='FFNN')(decoder_output)
  ffnn_output = layers.Activation(Activation)(ffnn_output)
else:
  ffnn_output = decoder_output
ffnn_output = layers.Dropout(PercentageDropout)(ffnn_output)
# Define classifier
model_output = layers.Dense(NumberClasses, activation=None,
                            name='Classifier')(ffnn_output)
model_output = layers.Activation('softmax')(model_output)
model = Model(inputs=model_input, outputs=model_output)
```

```
      model.summary()
      # END OF BUILD MODEL ****************************************
      model.compile(loss='categorical_crossentropy',
                    optimizer=Optimizer,
                    metrics=['accuracy', 'AUC'])
    except Exception as e:
      with open(MetricsPath + 'logErrors.txt','a') as f:
        header = "[log at {}]: Error building the model: {}\n". \
            format(strftime('%H:%M:%S, %d/%b/%Y', localtime()), e)
        chrom_info = \
        [header,
        f"Bidirectional = {Bidirectional}\n",
        f"NumberLSTMLayers = {NumberLSTMLayers}\n",
        f"PositionAttention = {PositionAttention}\n",
        f"ModelDimension = {ModelDimension}\n",
        f"ShapeOfProjection = {ShapeOfProjection}\n",
        f"PercentageDropout = {PercentageDropout}\n",
        f"Decoder = {Decoder}\n",
        f"ShapeOfDenseLayer = {ShapeOfDenseLayer}\n",
        f"Activation = {Activation}\n",
        f"Replicate = {Replicate}\n",
        f"NumberHeads = {NumberHeads}\n",
        "\n\n"]
        f.writelines(chrom_info)
      break
    # Define EarlyStopping
    es = tf.keras.callbacks.EarlyStopping(monitor="val_auc",
                                          patience=PatienteceValue,
                                          min_delta=min_delta,
                                          verbose=1, mode='max',
                                          restore_best_weights=True)
    # Define Garbage collector
    class ClearMemory(tf.keras.callbacks.Callback):
      def on_epoch_end(self, epoch, logs=None):
        gc.collect()
        tf.keras.backend.clear_session()
    # Define Scheduler
    def schedule(epoch, lr):
        if epoch < 3: # warm up
          return lr
        else:
          if lr < 1e-5: # limiter
            return lr
          exponential = lr * tf.math.exp(-0.1)
          return exponential
    sch = tf.keras.callbacks.LearningRateScheduler(schedule, verbose=0)
    try:
      # Define Training
      print("Training started...")
      t = time()
```

```
        model.fit(trainData, encodeYTrain,
                  batch_size=128,
                  epochs=EpochsValue,
                  validation_data=(validData, encodeYValid),
                  verbose=VerboseTrain,
                  class_weight=class_weights,
                  callbacks=[es, sch, ClearMemory()])
        print("Training finished...")
        trainTime = time() - t
      except tf.errors.ResourceExhaustedError as e:
        with open(MetricsPath + 'logErrors.txt','a') as f:
          header = "[log at {}]: Error training the model: {}\n". \
              format(strftime('%H:%M:%S, %d/%b/%Y', localtime()), e)
          to_write = \
          [header,
            'Chromosome: ' + ''.join([str(int(c)) for c in chromosome]) + '\n',
            'Fold: ' + str(fn) + '\n',
            '\n\n']
          f.writelines(to_write)
        # Set all metrics to zero
        AccAtEnd = np.zeros(NumberFolds)
        AUCAtEnd = np.zeros(NumberFolds)
        SenAtEnd = np.zeros(NumberFolds)
        SpeAtEnd = np.zeros(NumberFolds)
        F1sAtEnd = np.zeros(NumberFolds)
        break # breaks cross-validation loop
      except KeyboardInterrupt:
        with open(MetricsPath + 'logErrors.txt','a') as f:
          header = "[log at {}]: Training stopped by user\n". \
              format(strftime('%H:%M:%S, %d/%b/%Y', localtime()))
          to_write = [header]
          f.writelines(to_write)
        # Set all metrics to zero
        AccAtEnd = np.zeros(NumberFolds)
        AUCAtEnd = np.zeros(NumberFolds)
        SenAtEnd = np.zeros(NumberFolds)
        SpeAtEnd = np.zeros(NumberFolds)
        F1sAtEnd = np.zeros(NumberFolds)
        return None # breaks GA
      # Test model
      testForecast = model.predict(testData)
      testYhat = np.argmax(testForecast, axis=-1)
      YTest = YTest.to_numpy()
      cmTest = ConfusionMatrix(YTest, testYhat,is_imbalanced=True)
      AccAtEnd[fn] = cmTest.Overall_ACC
      SenAtEnd[fn] = cmTest.TPR_Macro
      SpeAtEnd[fn] = cmTest.TNR_Macro
      AUCAtEnd[fn] = metrics.roc_auc_score(YTest, testForecast, aver-
age='macro',multi_class='ovr')
```

```
      F1sAtEnd[fn] = cmTest.F1_Macro
      if VerboseGen:
        table_row = "{:>10} {:>10} {:>10} {:>10} {:>10}"
        f = lambda x: round(x,2) if isinstance(x,(float, int)) else str(x)
        support = sum(list(cmTest.P.values()))
        classTest = [
            table_row.format('', 'precision','recall','f1-score','support'),
            '',
            table_row.format('0',f(cmTest.PPV[0]), (cmTest.TPR[0]),
                                 f(cmTest.F1[0]),f(cmTest.P[0])),
            table_row.format('1',f(cmTest.PPV[1]),f(cmTest.TPR[1]),
                                 f(cmTest.F1[1]),f(cmTest.P[1])),
            table_row.format('2',f(cmTest.PPV[2]),f(cmTest.TPR[2]),
                                 f(cmTest.F1[2]),f(cmTest.P[2])),
            '',
            table_row.format('accuracy','','',f(cmTest.Overall_ACC),support),
            table_row.format('macro avg',f(cmTest.PPV_Macro),f(cmTest.TPR_Macro),
                                         f(cmTest.ACC_Macro),support),
            ]
        print("Classification report TEST",*classTest,sep='\n')
        print(f"\nTraining time: {strftime('%H:%M:%S', gmtime(trainTime))}\n")

      with open(MetricsPath + "TimePerPop.txt", 'a') as f:
        to_write = [f"\n[Gen{gen}|Chrom{id_pop+1}] fold{fn} --> {strftime('%H:%M:%S',
gmtime(trainTime))}\n"]
        f.writelines(to_write)

      del model, vocab_list, vectorize_layer
      # END OF FOR LOOP
    # Calculate metrics combining K folds
    AvgACC = np.mean(AccAtEnd) * 100
    AvgSen = np.mean(SenAtEnd) * 100
    AvgSpe = np.mean(SpeAtEnd) * 100
    AvgAUC = np.mean(AUCAtEnd) * 100
    AvgF1s = np.mean(F1sAtEnd) * 100

    StdACC = np.std(AccAtEnd) * 100
    StdSen = np.std(SenAtEnd) * 100
    StdSpe = np.std(SpeAtEnd) * 100
    StdAUC = np.std(AUCAtEnd) * 100
    StdF1s = np.std(F1sAtEnd) * 100

    if VerboseGen:
      print('Final results: ')
      print(f'Accuracy: Avg({AvgACC:.2f}%), Std({StdACC:.2f}%)')
      print(f'Sensitivity: Avg({AvgSen:.2f}%), Std({StdSen:.2f}%)')
      print(f'Specificity: Avg({AvgSpe:.2f}%), Std({StdSpe:.2f}%)')
      print(f'AUC: Avg({AvgAUC:.2f}%), Std({StdAUC:.2f}%)')
      print(f'F1-score: Avg({AvgF1s:.2f}%), Std({StdF1s:.2f}%)\n')
```

```
    if id_pop == 0:
      header = f"[New generation: {gen}]\nPopulation {id_pop+1:0>2}: "
    else:
      header = f"Population {id_pop+1:0>2}: "
    with open(MetricsPath + "ACC.txt", 'a') as f:
      f.write(header)
      f.write(f"{AvgACC:.3f} ± {StdACC:.4f}\n")
    with open(MetricsPath + "Sen.txt", 'a') as f:
      f.write(header)
      f.write(f"{AvgSen:.3f} ± {StdSen:.4f}\n")
    with open(MetricsPath + "Spe.txt", 'a') as f:
      f.write(header)
      f.write(f"{AvgSpe:.3f} ± {StdSpe:.4f}\n")
    with open(MetricsPath + "AUC.txt", 'a') as f:
      f.write(header)
      f.write(f"{AvgAUC:.3f} ± {StdAUC:.4f}\n")
    with open(MetricsPath + "F1.txt", 'a') as f:
      f.write(header)
      f.write(f"{AvgF1s:.3f} ± {StdF1s:.4f}\n")
    scores.append(AvgAUC) # here I decide what metric I want to consider
  # Convert into arrays
  scores = np.array(scores)
  # scores = np.random.normal(0.90,0.04,size=(len(population))) #!!!
  population = np.array(population)
  # sort and reverse order of based on `scores`
  inds = np.flip(np.argsort(scores)) # first is best, last is worst
  scoresGenTemp = np.zeros((NumbParents))
  populationGenTemp = np.zeros((NumbParents, NumbBitsDecimal))
  for i,g in enumerate(inds):
    scoresGenTemp[i] = scores[g]
    populationGenTemp[i, :] = population[g, :]
  if VerboseGen: print(f"Finished Fitness score for Generation {gen}")
  return scoresGenTemp, populationGenTemp
```

```
def initilization_of_population(size: int, n_bits: int) -> np.array:
  """Method for initializing the population
  Creates a set of randomly initialized chromossomes"""
  population = []
  for i in range(size):
    chromosome = np.ones(n_bits)
    chromosome[:(n_bits//2)] = 0
    np.random.shuffle(chromosome)
    population.append(chromosome)
  return np.array(population)
```

```
def generations(NumbParents: int,
                mutation_rate: float, crossover_rate: float,
                gen: int, population_nextgen: np.array, scoresGen: np.array,
                ElitsNumber: int,
                best_score: list, best_chromo: list) -> (np.array, np.array):
  print(f"\n\nNew generation: {gen}\n\n")
  popParents = population_nextgen.copy()
  scoresParents = scoresGen.copy()
  # Save Initial population of current generation
  with open(MetricsPath + f"Population{gen:0>2}Begin.txt", 'w') as f:
    f.write(str(popParents))
  # Save Initial score of current generation
  with open(MetricsPath + f"Score{gen:0>2}Begin.txt", 'w') as f:
    f.write('['+'\n '.join(scoresParents.astype('str'))+']')
  # Crossing over:
  pop_after_cross = crossover(popParents, crossover_rate)           # CROSSOVER
  # Mutations:
  pop_after_mut = mutation(pop_after_cross, mutation_rate, gen)     # MUTATION
  # Fit population:
  scoresChild, popChild = fitness_score(pop_after_mut, gen)         # FITNESS SCORE
  # Select population for next generation:
  scoresNextGen, populationNextGen = selection(popParents, popChild,# SELECTION
                                               scoresParents, scoresChild,
                                               ElitsNumber)
  # Save Final population of current generation
  with open(MetricsPath + f"Population{gen:0>2}End.txt", 'w') as f:
    f.write(str(populationNextGen))
  # Save Final score of current generation
  with open(MetricsPath + f"Score{gen:0>2}End.txt", 'w') as f:
    f.write('['+'\n '.join(scoresNextGen.astype('str'))+']')

  best_score.append(scoresNextGen[0])
  best_chromo.append(populationNextGen[0])
  # Save all best chromosomes untill now
  with open(MetricsPath + "Best_score.pkl", 'wb') as f:
    pickle.dump(best_score, f)
  with open(MetricsPath + "Best_chromosome.pkl", 'wb') as f:
    pickle.dump(best_chromo, f)

  return scoresNextGen, populationNextGen, best_score, best_chromo
```

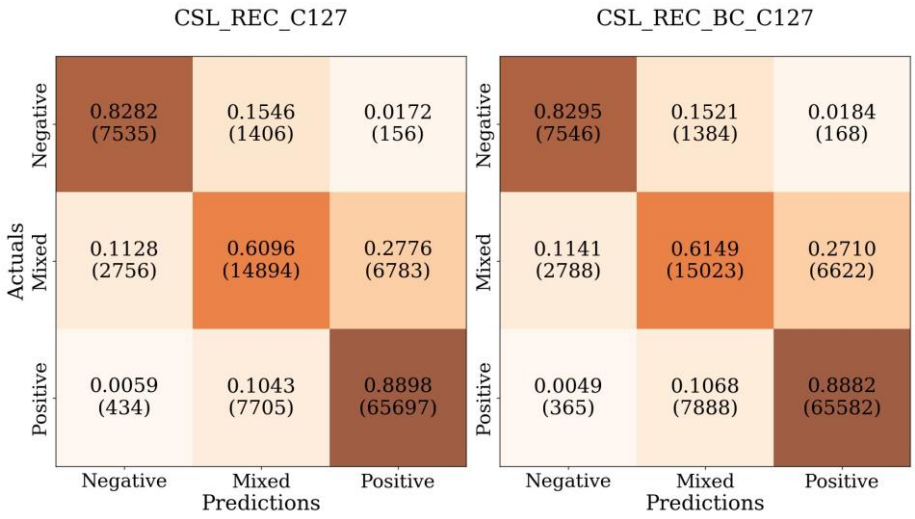
```
def mutation(pop_after_cross: np.array, MutationRate: float, gen: int) -> np.array:
  if VerboseGen: print("Applying mutations on chromosomes...")
  mutation_rate = MutationRate - MutationRate * int(gen / 5) * 0.3
  if mutation_rate < 0.01:
    mutation_rate = 0.01

  population_nextgen_mutation = []
  for chromosome in pop_after_cross: # iterates all chromosomes
    mutant = np.zeros(len(chromosome))
    for i in range(len(chromosome)):
      if random.random() <= mutation_rate:
        mutant[i] = 1 - chromosome[i]  # inverts the locus value
      else: # Don't apply mutation
        mutant[i] = chromosome[i]
    population_nextgen_mutation.append(mutant)
  return np.array(population_nextgen_mutation) # have same length as input population
```

```
def crossover(pop_after_sel: np.array, cross_Rate: float) -> np.array:
  if VerboseGen: print("Applying crossing over on chromosomes...")
  NumbBitsDecimal = pop_after_sel.shape[-1]
  population_nextgen_crossover = []
  for _ in range(len(pop_after_sel)): # Run over all chromosomes (whole population)
    if random.random() <= cross_Rate:
      while True:
        # Select 2 random chromosomes from population (must be different!)
        numbers1 = random.randrange(0, len(pop_after_sel))
        numbers2 = random.randrange(0, len(pop_after_sel))
        parent1 = min(numbers1, numbers2)
        numbers1 = random.randrange(0, len(pop_after_sel))
        numbers2 = random.randrange(0, len(pop_after_sel))
        parent2 = min(numbers1, numbers2)
        if parent1 != parent2: break
      # Select random points in chromosome for crossing over
      CrossoverPoints1 = random.randrange(0, NumbBitsDecimal)
      CrossoverPoints2 = random.randrange(0, NumbBitsDecimal)
      CrossoverPoints = np.sort([CrossoverPoints1, CrossoverPoints2+1])
      child = pop_after_sel[parent1].copy()
      child[CrossoverPoints[0]:CrossoverPoints[1]] = pop_after_sel[parent2,Crossover-
Points[0]:CrossoverPoints[1]]
    else: # Don't apply crossing over
      # Select random chromosome from population
      numbers1 = random.randrange(0, len(pop_after_sel))
      numbers2 = random.randrange(0, len(pop_after_sel))
      parent1 = min(numbers1, numbers2)
      child = pop_after_sel[parent1].copy()

    population_nextgen_crossover.append(child)
  return np.array(population_nextgen_crossover) # have same length as input population
```

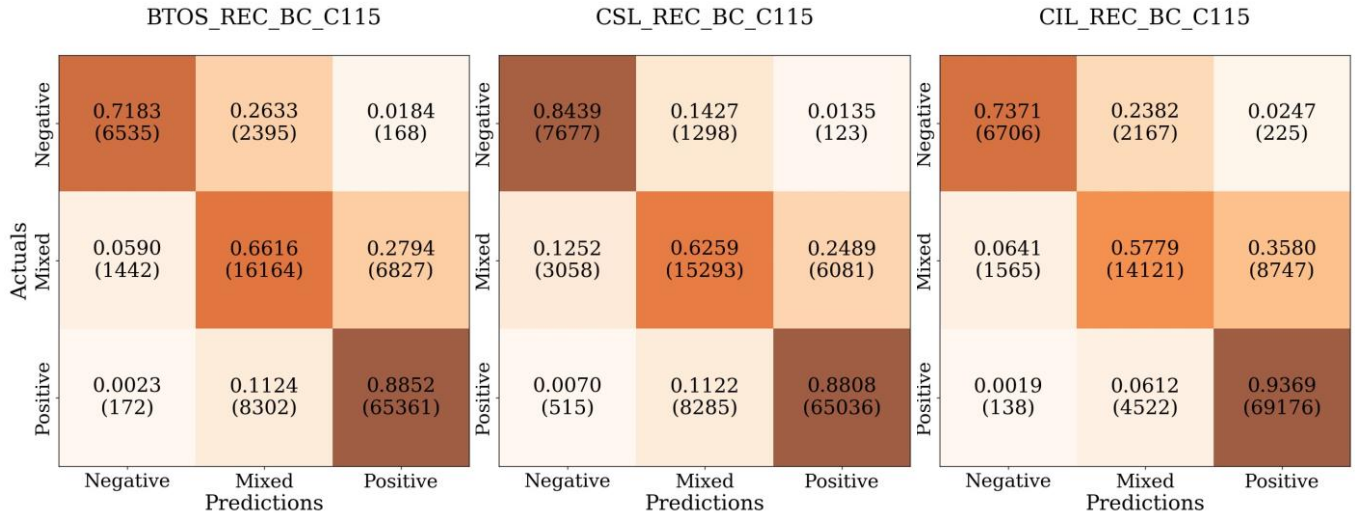
```
def selection(popParents: np.array, popChild: np.array,
              scoresParents: np.array, scoresChild: np.array,
              ElitsNumber: int):
  if VerboseGen: print("Selecting next population...")
  NumbParents, NumbBitsDecimal = popParents.shape
  # Group Parents & Children (excluding Elitism)
  population_Final = np.zeros(((NumbParents * 2) - ElitsNumber, NumbBitsDecimal))
  for LinesAppend in range(0, (NumbParents * 2) - ElitsNumber, 1):
    if LinesAppend < NumbParents - ElitsNumber:
      population_Final[LinesAppend, :] = popParents[LinesAppend+ElitsNumber,:].copy()
    else:
      population_Final[LinesAppend, :] = popChild[LinesAppend-NumbParents+ElitsNumber,
:].copy()
  scoresGenFinal = np.append(scoresParents[ElitsNumber:], scoresChild, axis=0)

  # Sort Parents & Children (best ones are first)
  inds = np.flip(np.argsort(scoresGenFinal))
  scoresGenTemp = np.zeros(((NumbParents * 2) - ElitsNumber))
  populationGenTemp = np.zeros(((NumbParents * 2) - ElitsNumber, NumbBitsDecimal))
  for sortingLines in range(0, len(inds), 1):
    scoresGenTemp[sortingLines] = scoresGenFinal[inds[sortingLines]].copy()
    populationGenTemp[sortingLines, :] = population_Final[inds[sortingLines],
:].copy()

  scoresGenUpdate = np.zeros((NumbParents))
  populationGenUpdate = np.zeros((NumbParents, NumbBitsDecimal))
  # Select elitism
  for k in range(0, ElitsNumber):
    scoresGenUpdate[k] = scoresParents[k].copy()
    populationGenUpdate[k, :] = popParents[k, :].copy()
  # Select rest of population, starting from best
  scoresGenUpdate[ElitsNumber:NumbParents] = scoresGenTemp[0:NumbParents -
ElitsNumber].copy()
  populationGenUpdate[ElitsNumber:NumbParents, :] = populationGenTemp[0:NumbParents -
ElitsNumber, :].copy()

  # Sort Selected population (best ones are first)
  inds = np.flip(np.argsort(scoresGenUpdate))
  scoresGenUpdateSorted = np.zeros((NumbParents))
  populationGenUpdateSorted = np.zeros((NumbParents, NumbBitsDecimal))
  for sortingLines in range(0, len(inds), 1):
    scoresGenUpdateSorted[sortingLines] = scoresGenUpdate[inds[sortingLines]].copy()
    populationGenUpdateSorted[sortingLines, :] = populationGenUpdate[inds[sorting-
Lines], :].copy()

  return scoresGenUpdateSorted, populationGenUpdateSorted
```

Appendix D: Confusion matrices

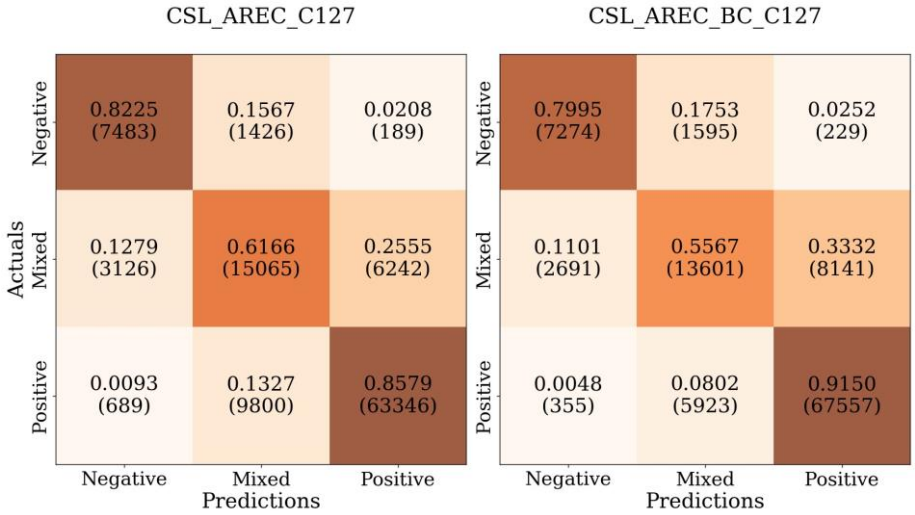
- Baseline REC model and optimized REC model (intermediate). The confusion matrix is normalized, and the actual number of samples is indicated between parentheses.



- Fully optimized REC models after different balancing techniques (including CIL). The confusion matrix is normalized, and the actual number of samples is indicated between parentheses.



- Baseline AREC model and optimized AREC model (intermediate). The confusion matrix is normalized, and the actual number of samples is indicated between parentheses.



- Fully optimized AREC models after different balancing techniques (including CIL). The confusion matrix is normalized, and the actual number of samples is indicated between parentheses.