UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER EN INGENIERÍA DE REDES
Y SERVICIOS TELEMÁTICOS

TRABAJO FIN DE MASTER

Design and Development of an Ethical and Moral Values Audit Toolkit for Detecting Bias and Fairness in Machine Learning-based Text Classifiers

José Luis Benítez Santana
2024

TRABAJO DE FIN DE MASTER

Título:	Diseño y Desarrollo de un Sistema de Auditoría de Valores Morales y Éticos para Detectar Sesgo en Clasificadores de Texto basados en Aprendizaje Automático
Título (inglés):	Design and Development of an Ethical and Moral Values Audit Toolkit for Detecting Bias and Fairness in Machine Learning-based Text Classifiers
Autor:	José Luis Benítez Santana
Tutor:	Dr. Carlos Ángel Iglesias Fernández
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO DE FIN DE MASTER

Design and Development of an Ethical and Moral Values Audit Toolkit for Detecting Bias and Fairness in Machine Learning-based Text Classifiers

Septiembre 2024

Resumen

El lenguaje es una de las herramientas más poderosas que tenemos a nuestra disposición. Cuando lo utilizamos, a menudo las ideas no se exponen de manera objetiva. En su lugar, el mensaje está contenido en un marco que incluye la perspectiva moral y emocional del hablante. Este marco está directamente relacionado con la forma en la que las personas organizan sus creencias y lleva consigo las emociones y valores morales del transmisor, lo que puede llevar a mensajes sesgados. En este ámbito, el sesgo se define como la tendencia de los modelos a propagar y/o amplificar prejuicios y desigualdades. Así, cuando se entrenan clasificadores de texto, el uso de corpus sesgados puede propagar estos sesgos hacia la tarea de clasificación. De la misma manera, el sesgo puede aparecer cuando se desarrollan modelos siguiendo malas prácticas, como usar un corpus desbalanceado.

Este trabajo se centra en desarrollar un un conjunto de herramientas para auditar clasificadores de texto en relación con los valores morales. Para ello, se comienza estudiando los diferentes tipos de sesgo existentes. Luego, se selecciona una forma de categorizar los valores morales, que es la Teoría de las Fundamentos Morales. Esta describe que la moralidad humana se construye sobre cinco fundamentos universales: cuidado, equidad, lealtad, autoridad y pureza. Se continúan implementando varias técnicas para detectar y mitigar el sesgo, ya sea adaptando métodos existentes que conforman el estado del arte o diseñando nuevas técnicas. Finalmente, se propone un sistema con una arquitectura web orientada a microservicios que incluye una aplicación gráfica a través de la cual los auditores puedan interactuar con dichas implementaciones.

La evaluación de este trabajo demuestra que el sesgo puede ser detectado y mitigado de manera efectiva en la mayoría de los casos combinando las técnicas propuestas. También se concluye que el sesgo moral en el lenguaje humano es un concepto complejo, cuya presencia, detección y mitigación dependen en gran medida del contexto específico. El auditor debe, por tanto, ser el responsable de seleccionar las técnicas adecuadas.

Palabras clave: Auditoría, Sesgo, Clasificador de texto, Procesamiento de Lenguaje Natural, Valores Morales, Marco Moral

Abstract

Human language is one of the most powerful tools at our disposal. When using it, issues are often not objectively exposed. Instead, the message is contained in a frame that captures how the story is presented, including the moral and emotional perspective of the speaker. This frame is directly related to how people organize their beliefs and label their ideas. The use of the language can then divulge emotions and moral values, which can lead to biased information. In this context, bias is defined as the tendency of models and algorithms to reflect, amplify, or perpetuate prejudices and inequalities. When training text classifiers, using biased corpora can propagate these biases into the classification task. In the same way, bias can also appear when models are developed following bad practices, such as using an imbalanced corpus.

This work focuses on developing a framework and toolkit to audit text classifiers regarding ethics and moral values. It starts by thoroughly studying the presence and different types of existent bias. Then, we study a way to categorize and quantify moral values, finally selecting the Moral Foundation Theory, which outlines that human morality, regardless of the culture and social aspects, is built on a range of five universal moral foundations: care, fairness, loyalty, authority, and purity. We continue implementing several techniques to detect and mitigate bias either by adapting the existent state-of-the-art methods to our particular objectives or by designing new ones. Finally, a web microservice-oriented system is proposed to present a webapp through which auditors can easily interact with the techniques implementations.

The evaluation of this work demonstrates that bias can be effectively detected and mitigated in most cases by combining the proposed techniques. However, we also conclude that moral bias in human language is a complex concept, with its presence, detection, and mitigation depending heavily on the specific context. This implies that the auditor must carefully select the appropriate techniques to achieve successful outcomes.

Keywords: Audit, Bias, Text Classifier, Natural Language Processing, Moral Values, Moral Framing

Agradecimientos

A mi familia: Davinia, Montse y Sergio. Por siempre apoyarme en mis decisiones y hacerme entender que ser buena persona es lo más importante.

A Berna, David, Gabri, Guille, Himar, Luca, y Néstor. Siempre estarán y siempre estaré.

A Leire. Por acompañarme en todos los momentos difíciles de este año y creer en mí cuando ni yo lo hacía.

A Anny, Andrea, Ariel, Carlota, Cristina, David, Javi, Santi, Sofía y Sofía Martín. Por el apoyo y todos los buenos momentos vividos este año; conocerles ha sido lo mejor de haber estudiado el máster y realizado el TFM.

A mi tutor, Carlos Ángel Iglesias. Por su implicación, dedicación y, sobre todo, por su paciencia cuando las cosas no han salido como se esperaba.

A Saulo. Por todo lo que hemos superado juntos este año y el buen equipo que hemos formado.

¡Gracias a todos!

Contents

List of Figures

Acronyms

AI Artificial Intelligence

AMOR Moral Sentiment Analysis in Textual Data

API Application Programming Interface

BCM Background Comparison Metric

CSS Cascading Style Sheets

CSV Comma-separated values

GDPR General Data Protection Regulation

GSI Intelligent Systems Group

JSON JavaScript Object Notation

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

LIME Local Interpretable Model-Agnostic Explanations

MAS Moral Auditing System

MCM Multi-group Comparison Metric

MFT Moral Foundation Theory

ML Machine Learning

MVVM Model View View-Model

NaN Not a Number

NLP Natural Language Processing

MFD Moral Foundation Dictionary

REST Representational State Transfer

RNN Recurrent Neural Network

OOP Object Oriented Programming

PCM Pairwise Comparison Metric

SDG Sustainable Development Goal

SHAP SHapley Additive exPlanation

URL Uniform Resource Locator

WEAT Word Embedding Association Test

WSGI Web Server Gateway Interface

CHAPTER 1

Introduction

This chapter introduces the context of the project, including a brief overview of all the different parts that will be discussed in the project. It will also break down a series of objectives that will be carried out during the implementation of the project. Moreover, it will introduce the document's structure with an overview of each chapter.

1.1 Context

Over the past few years, Machine Learning (ML) has become highly significant in today's lifestyle. Examples of this social impact include autonomous cars and the trending GPT-3 chatbot developed by OpenAI [57]. ML-based systems can outperform humans in specific tasks and are used to assist people in processes such as understanding and predicting information or making decisions. On certain occasions, these processes come with underlying complexity and implicit accountability, as illustrated by a judge using an ML-based system to get feedback on whether a criminal will be a recidivist. Therefore, optimizing task performance alone (which has been the main criteria used to rate a ML-based system as good by default) is insufficient when critical decisions that could impact people's lives are being made [19].

Usually, ML algorithms operate by creating a model (which, in the end, is just a mathematical function whose complexity could vary to a greater or lesser extent) that learns from existing data and performs a generalization to unknown data. The algorithm does not understand the data; it only searches for patterns or relations to create the model. This fact implies that even though the result of a query is set as correct when testing (*what* is obtained), how the result was reasoned (*how* is obtained) could not be adequate. Consequently, a correct *output* for all *inputs* cannot be ensured. To make matters worse, models (especially those based on neural networks) have reached such a level of complexity that humans cannot even approach to understand them: making an explanation by analyzing the model and its parameters is almost an impossible task [72].

Two issues are then raised: 1) ML-based systems are making decisions by searching patterns in the data but without understanding its meaning or causality; and 2) an explanation for those decisions can be neither understood nor explained by human beings since ML-based systems are like a "black box" for us. Joining the previous two aspects, it is evident that additional criteria to evaluate and interpret this kind of system are needed to make Artificial Intelligence (AI) trustworthy and safe enough (e.g., the non-existence of bias can be affirmed), especially when deploying a ML-based system in a social environment with high accountability (such as medicine). A way of solving them is *auditing*, which is research and practice to assess, mitigate, and ensure the safety, ethics, and legality of the system [36].

1.2 Overview

Building on the previously defined context, this Master's Dissertation focuses on the case of auditing ML-based text classifiers to detect and mitigate biases associated with moral values, ensuring the system's fairness, robustness, and ethical behavior. Text classifiers are ML-based models used to categorize text data into predefined labels, enabling automated analysis of large datasets such as news, social media posts, and reviews. They are commonly used for sentiment and emotional analysis, among others [67, 38]. When it comes to human language-related models, auditing (i.e., detecting and mitigating bias) can be a challenging task since subjectiveness is implicit in what we communicate. In this case, morality plays a fundamental role, as it defines our beliefs and, as a consequence, how they are expressed [7].

In this thesis, we investigate and develop a toolkit to audit corpora and text classifiers concerning ethics and moral values. This is part of the Moral Sentiment Analysis in Textual Data (AMOR) project [21], which is assigned to the Intelligent Systems Group (GSI) and aims to develop critical thinking and the ability to manage emotions when engaging with the media and social networks to combat issues such as misinformation, hate speech, and clickbait. Specifically, this thesis is part of one of its objectives, which consists of "investigating the analysis, application, and auditing of moral and ethical values".

1.3 Project goals

The final objective of this project is to develop a system for auditing Natural Language Processing (NLP)-based models and texts and detecting and mitigating, if possible, bias associated with moral values. To achieve it, some sub-objectives are established:

1. Analyze the existent tools, methodologies, and frameworks for auditing and generally detecting bias.

2. Investigating and choosing a way to categorize/quantify the moral values to make them measurable.

3. Applying the tools reviewed in the first objective to the particular case of ethics and moral values to understand its behavior and conclude their validity.

4. Developing a theoretical methodology to thoroughly audit a system to detect bias related to moral values.

5. Evaluating the scope of a practical framework or application for automatizing auditing moral values as much as possible.

1.4 Structure of this Master Thesis

This section provides a brief overview of all the chapters of this Master Thesis. It is structured as follows:

- *Chapter 1* introduces the problem addressed in this project and the main objectives established.
- *Chapter 2* details the state of the art in defining bias associated with NLP, categorizing moral values, and different techniques to detect and mitigate bias.
- *Chapter 3* presents the different tools and libraries used to develop this work.
- *Chapter 4* describes in detail the architecture of the practical framework developed to audit text classifiers.
- *Chapter 5* evaluates the proposed system and a case study.
- *Chapter 6* provides the conclusion of this work and future lines.

CHAPTER 2

State of Art

In this chapter, all the up-to-date tools and theoretical frameworks related to analyzing, detecting, and mitigating moral and social bias are analyzed. It starts with categorizing bias and moral values and continues with a detailed analysis of the different existing methods to detect and mitigate bias.

2.1 Analyzing and categorizing bias

In the context of NLP, bias can be defined as the unfair treatment of certain groups present in the feature space when giving an *output* [23]. This includes discriminatory processing based on characteristics (gender, race, religion, etc.), which are directly influenced by human moral values. Bias can appear differently throughout the various stages of the ML lifecycle, so they should be independently analyzed and categorized for each stage. ML lifecycle is composed of two main categories: 1) the preparation of the data, known as the *data phase* and 2) the setting of the model according to the data, called the *algorithmic phase* [72].

Data phase

Divided into two sub-phases, the first is *data collection*, which refers to defining the needed data, selecting a target population, measuring features, and labeling obtained values [72]. Since the target population is usually a large group, we are forced to consider only a subset of the total due to the limitations of resources. The second sub-phase is *data preparation*, which includes normalizing, transforming labeled data, or even engineering a new feature space [36]. The data set is divided between training and test data, and sometimes, the training data is divided again to get a subset called validation data.

Two types of bias can arise in this phase. The first is called *historical* bias, primarily associated with the data collection subphase and representing unfair thinking in the real world [72]. Even if data are ideally collected and sampled, and the total representation of the population is considered (i.e., no subset of data is taken), the NLP-based system could continue to have unfair prejudgments due to the implicit moral framing present in texts [71]. This bias is remarkably propagated and even augmented in word embeddings [11]. For example, if the sentence "He is a nurse. She is a doctor" is translated into Hungarian and then back to English (which uses *word embedding*), the final result is "She is a nurse. He is a doctor" [71]. Essentially, the model is expressing a sexist stereotype as it defines by default the doctor as a male and the nurse as a female. Even though this conclusion is neither an objective fact nor a generalization, the model has reached it due to the historical sexist bias implicitly located in the corpus with which the embedding was trained.

Representation bias has to do with the under-representation of some groups of the

target population, which causes the model to fail in generalizations for these groups. This also appears when collecting data and shows up 1) if the target population does not represent the use population and 2) if the target population includes under-represented groups because of the lack of data or data unequally sampled. Finally, the third kind of bias is *measurement* bias, which can come out when choosing, gathering, or computing features from the target population to conform to the dataset/corpus. In NLP, this bias becomes imperative when it comes to labeling the corpus (i.e., computing features) [31]. Sometimes, annotators are distracted, uninterested, or lazy about the annotation task, choosing the "wrong" labels. On other occasions, the label an annotator chooses depends on their interpretation or his/her thinking, resulting in bias transmission to the labeled corpus.

Algorithmic phase

This has four subphases [72]. The first is *model development*, which involves defining the working and objective of the model, the algorithm to train it, and the tuning using the validation data. Next, the evaluation of *the model* consists of evaluating the model's performance with unknown inputs using the *test* data and evaluating with different metrics. After that, some *model post-processing* can be done to improve human interpretation of the *outputs*. The final substage refers to implementing the ML-based system in production and verifying that non-experts still accept and interpret its performance.

Four types of bias can arise during the algorithmic phase [72]. *Aggregation* bias arises during the *model development* when the model is too generalist, that is, the assertion that a label or feature has the same meaning for all groups. In NLP, an aggregation bias appears with the model's assumption that all groups assign the same meaning to every word. For example, a corpus that gathers song reviews of people from anywhere will encompass people or groups with different cultures and/or norms. On this matter, an *aggregation* bias appears when it is assumed that all groups assign the same meaning to every word. Of course, in such a variety of cultures and different backgrounds, it is logical that some words or expressions are taken differently by some of the groups. It is vital to determine whether we are trying to develop a model that addresses a context that is too broad and if we should instead consider designing multiple models specifically adapted to each cultural or normative context.

Other, *learning* bias refers to how the configuration of the algorithm parameters

causes performance disparities to increase between different *inputs*. For example, prioritizing an efficient model can imply only learning the most common features, leading to performance disparities in underrepresented groups. Both *aggregation* and *learning bias* must be checked during the development sub-phase. Next, *evaluation bias* happened because 1) the *benchmark* data used for evaluating the quality of the model does not adjust to the use population, 2) the model gets overfitted to a particular *benchmark* or 3) choosing too many simple metrics leads to hiding some issues (such as a minority under-representation). At last, *deployment bias* comes in when the deployed ML-based system is not used as designed. Usually, these systems are isolatedly developed but deployed in a more complex scenario where non-expert humans have to interact with them to interpret the output. These last two cases of bias (*evaluation* and *deployment*) are related to the subphase of *model postprocessing* and model deployment.

2.2 Moral Foundation Theory

Proposed by Haidt [29], this psychological model attempts to explain why morality varies so much between cultures but remains powerful in shaping human behavior. According to this theory, humans have a set of universal moral intuitions that underlie our explicit values and beliefs. Haidt argues that there are five cores "moral foundations" around which people construct their perspective or viewpoint. This influential framework is based on the premise that while morality can exhibit substantial variation in its scope of what is considered harmful or caring behavior and is influenced by many factors (culture, geography, time, society, etc.), this variation lies an enduring universal core. Moral Foundation Theory (MFT) identifies this moral core as a psychological system of "intuitive ethics".

MFT's theoretical framework is constructed around five cores, composed of dyads comprising a positive part and a negative counterpart. The dyads are as follows [29]:

- *Care/Harm*. This foundation focuses on feelings of compassion or care for others and the ability to feel or dislike the pain of others. It is the ability to empathize with the suffering of others and the desire to care for those who are vulnerable [26].
- *Fairness/Cheating*. This foundation is tied to our ideas of justice, rights, and autonomy. It focuses on our instinctive desires for fair treatment, equality, and reciprocity. The basic idea here is that people should be treated equally and have

equal opportunities, and any deviation from equality should be justified and not based on cheating or deception.

- *Loyalty/Betrayal.* This foundation recognizes that humans inherently depend on and value group relationships and naturally bond, especially when faced with a common enemy or threat [34]. Those who highly value this foundation often display strong feelings of nationalism, patriotism, or loyalty to their group. They may also have a stronger inclination to enforce in-group norms and traditions and a greater dislike or disdain for those who betray or go against the group's interests.

- *Authority/Subversion.* This dyad explores interactions through the lens of social hierarchies. It informs our understanding of leadership, the deference to authority, and the importance of maintaining tradition [30].

- *Purity/Degradation.* This foundation was adapted from the psychology of disgust and incorporates the notion of a more elevated spiritual life. This is usually expressed through metaphors like "the body as a temple" and encompasses the spiritual aspects of religious beliefs.

Recent research has introduced new advances that expand and build upon previous foundational theories. Multiple considerations have been integrated, and progress has been made in analyzing them from a moral perspective. Gonzales's [24] recent research extends the foundational work proposed by Haidt [29], following a similar structure and introducing 14 new moral foundations that enhance this framework. Thanks to these reconsiderations, several issues such as gender, race, and other social biases can also be categorized as moral biases in essence [7] [24].

The proposed extension of Moral Foundation Dictionary (MFD) [24] includes the following foundations:

- *Liberty/Oppression.* This foundation represents the desire for freedom and the distress caused by its denial. It encapsulates the yearning for personal liberty and the feeling of oppression when freedom is restricted.

- *Feminism/Maleness.* It redefines masculinity as a moral principle, emphasizing equality, justice, and human rights.

- *Sustainability/Climate change.* From an environmental perspective, sustainability involves responsible resource management, biodiversity conservation, and minimizing environmental impact.
- *Racial equality/Racism.* This foundation asserts that all individuals, regardless of racial or ethnic background, should be treated fairly and have equal opportunities. It advocates for eliminating discrimination based on race, ethnicity, or color [37].
- *Peace/War.* This dyad represents the contrasting facets of human experience, capturing the delicate balance between harmony and conflict on interpersonal and global scales.
- *Animal rights/Animal abuse.* This foundation recognizes and advocates for animals' inherent value and welfare and addresses potential negative impacts against them.
- *Sexual diversity/Sexual discrimination.* It encompasses the full spectrum of sexual diversity and gender identities, as well as the negative effects or aggressive behaviors related to gender discrimination.

2.3 Auditing & Detecting bias in NLP

Auditing a ML-based system is the research and practice to evaluate, mitigate, and ensure the system's safety, ethics (i.e., fairness), and legality. In other words, it is the proceeding of warrant that the following criteria are being fulfilled: safety, nondiscrimination, right of explanation, and non-existence of technical debt. Auditing is a broad procedure that must be performed in every phase of developing a ML-based system, existing different techniques and objectives for each. It is essential to audit a system when its decisions (*outputs*) will significantly influence people's lives. The audit process consists of four stages: 1) development, 2) assessment, 3) mitigation, and 4) assurance.

Development is the process of developing and documenting a stage of the development lifecycle. At this phase, auditing involves following good practices to meet the already commented criteria. *Assessment* refers to evaluating the system's behavior and capacities, verifying that all requirements are met. The stage of *mitigation* deals with improving or repairing errors of the algorithm outcome when some failures or unfulfillment of any criterion occurs. Lastly, *assurance* is the step of declaring that a system

conforms to the established criteria, which involves legal regulations -e.g., General Data Protection Regulation (GDPR)- or ethical behavior (e.g., unbiased decision-making).

In this section, a review of existing methods for identifying bias in NLP is done. In the auditing context, these tools can be used in the assessment and mitigation process to verify if the non-discrimination criterion is being met. After reading the literature, we consider the four most relevant techniques. Two are associated with the data phase: 1) the analysis of word embeddings and 2) the framing axis method. The two others are 3) the use of interpretability and 4) using metrics over the corpus and the outputs of the model, both linked to the algorithmic phase.

2.3.1 Analysis of word embeddings

This technique aims to detect and mitigate bias associated with the vector space of word embeddings. Word embeddings are fixed-length vector representations for words [3], serving to serve as a dictionary of sorts for NLP-based systems that would like to use word meaning [11]. Word embeddings are learned by solving a language modeling task in a large unsupervised corpus, allowing subsequent models to take advantage of the semantic and syntactic relationships captured from this corpus [61]. On the one hand, words with similar semantic meaning tend to have vectors that are close together. On the other hand, the vector differences between words in embeddings have been shown to represent relationships between words [11]. For example, given the set of analogy words "Madrid is to Spain as Tokyo is to x" (which is denoted as *Madrid:Spain*:: *Tokyo:x*) it can be calculated that x is equal to "Japan" by using simple vector arithmetic over the embedding vector space, as shown in Equation 2.1. These analogies can also be represented over the embedding's vector space in Fig. 2.1.

$$\vec{v}_{\text{madrid}} - \vec{v}_{\text{tokyo}} \approx \vec{v}_{\text{spain}} - \vec{v}_{\text{japan}} \tag{2.1}$$

However, capturing these relations and similarities to words implies that the historical bias present in the corpus the embedding was trained with is also propagated inside the embedding structure. One of the most popular examples is the implicit sexism transmitted to the embedding, where the profession *computer programmer* is more associated with men, and "homemaker" has a feminine gender direction. This is shown in Equation 2.2.

$$\vec{v}_{\text{man}} - \vec{v}_{\text{woman}} \approx \vec{v}_{\text{computer programmer}} - \vec{v}_{\text{homemaker}} \tag{2.2}$$

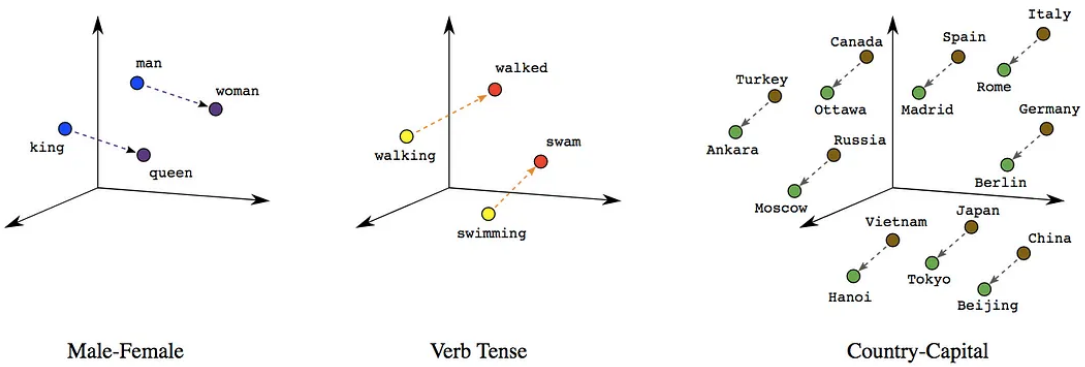

Figure 2.1: Example of words located on the vector space in word embeddings [6]

Bolukbasi et al. (2016) described two forms in which historical bias appears inside *word embeddings* in terms of gender, as well as proposed a method to mitigate both forms [11]. The first form is *direct bias* and occurs when explicit gender stereotypes are present in the association of words. This bias can be quantified using cosine similarity, indicating how strongly a word is aligned with the vector of gender direction. Equation 2.2 is an example of *direct bias* by linking "man" more closely to "computer programmer" and "woman" with "homemaker". Furthermore, *indirect bias* refers to imperceptible associations within the embedding space that are not directly related to gendered terms but still perpetuate stereotypes. This bias is more difficult to detect because it manifests itself in the broader context of word relationships, not just direct word pairings. For example, words related to intelligence or leadership might cluster closer to male-associated terms, whereas words associated with care or emotion might align with female-associated terms. This creates an environment where even neutral terms can be biased through their indirect connections to gender-based concepts.

On the one hand, to mitigate the *direct* bias, the *neutralize* method is suggested. It involves identifying words that should ideally be gender neutral but are not due to historical gender bias (e.g., "doctor" or "nurse"). Then, it must be ensured that they are equidistant from gender-specific words like "man" and "woman". This is done by projecting these neutral words onto a gender direction (a vector that captures the gender axis by subtracting the "woman" vector from the "man" vector) and then adjusting them so that they are orthogonal to this axis. This process ensures that gender-neutral words

are not preferentially associated with one gender over the other, thus removing explicit gender biases from these terms. On the other hand, to mitigate the *indirect* bias, the *equalize* method is proposed. It works by ensuring that gendered word pairs (e.g., "man" and "woman") have symmetric relationships with other words. This involves adjusting the embeddings so that both members of a gender pair are equidistant from all different words that should be neutral regarding gender. The process creates a balanced vector space where gender-specific terms have equal associations with neutral words, preventing indirect biases from creeping into the embeddings.

To measure the presence of bias and compare the embeddings before and after mitigation, we can use the Word Embedding Association Test (WEAT), a statistical method designed to measure the embeddings of words [14]. WEAT extends the Implicit Association Test (IAT) [27] from social psychology to word embeddings, allowing researchers to assess the presence of social and cultural biases in these embedding models. The test works by comparing the similarity between two sets of target words (e.g., "male" and "female" names) with two sets of attribute words (e.g., "career" and "family" terms) using cosine similarity. The degree of bias is quantified through the differential association of the target sets with the attribute sets, where a significant difference indicates bias within the embeddings.

2.3.2 Framing Axis

This technique also involves the usage of word embeddings. However, instead of mitigating the bias of vector embedding, it is used as a reference vector dictionary that captures human frames to identify biases in texts [7]. Using a significant word embedding model, which encapsulates a comprehensive representation of semantic and syntactic meaning and relationships for most vocabulary words, FrameAxis [39] allows characterizing the encoding of a given text by detecting the most relevant semantic axes (denoted as "microframes"). It can quantitatively measure how polarized a text is in each micro-frame (bias), the direction (i.e. vice or virtue), and how much each micro-frame is used (intensity) in the said text.

First, let us understand what a micro-frame is. In the human natural language, issues are often not objectively exposed but are emphasized or de-emphasized in the speech according to the expositor's beliefs. In other words, the message is contained in a frame or perspective that captures how the story is presented, including the moral

and emotional viewpoint of the transmitter [49, 15]. This "frame" is directly related to how people organize their beliefs and label their ideas within the space of life [7]. When it comes to ethics, these "frames" have a heavy impact on society, either when they are deliberately used as a tool to make a message more emotionally powerful [7] or when they are implicitly reflected in what we communicate as a consequence of what we believe. A micro-frame is thus a part of the frame that contains the impact of one of the MFT foundations.

From the embedding perspective, each micro-frame builds on a set of antonyms based on the vices and virtues of the moral foundation (e.g., *care* words vs. *harm* words). The axis (which is a direction vector) is calculated by subtracting the average vector of positive words (virtues) and the average vector of negative words (vices) of that moral dimension. Mathematically, let m be one of the foundations, V_m^+ the set of embedding vectors of virtue words and V_m^- the set of vectors of vice words. The semantic axis corresponding to this moral dimension is shown in Equation 2.3.

$$A_m = \text{mean}(V_m^+) - \text{mean}(V_m^-) \tag{2.3}$$

The measurement of bias of a document along a semantic moral axis (micro-frame) is calculated as shown in Equation 2.4. A document $D = \{d1, \ldots, dn$ is defined as a set of embeddings (vector space) of its words; $s(A_m, d)$ is the cosine similarity between the semantic axis A_m and the word d; and f_d is the frequency of the word d in the document. It implies that the bias metric of a text on a moral foundation axis m is the weighted average of the cosine similarity of its words to that axis. The formula can return values from -1 to 1. The absolute value of the bias captures the document's relevance to the moral dimension, while the sign captures a bias toward one of the poles in the moral dimension. A positive sign in the result indicates that the document is polarized toward the virtue, while a negative sign indicates polarization towards the vice.

$$B_{D_m} = \frac{\sum_{d \in D} f_d \, s(A_m, d)}{\sum_{d \in D} f_d} \tag{2.4}$$

On the other hand, intensity is calculated based on a moral dimension to capture how heavily it appears in the document according to the background distribution. As shown in Equation 2.5, B_T measures the baseline bias of the entire text corpus T on a moral dimension m. The intensity metric does not reveal information about the polarization of the document. However, when both poles of an axis actively appear in the text, the

positive and negative terms cancel each other out, giving an insignificant value in the bias measurement. In these cases, the intensity shows the relevance of that dimension.

$$I_{D_m} = \frac{\sum_{d \in D} f_d \ (s(A_m, d) - B_{T_m})^2}{\sum_{d \in D} f_d} \tag{2.5}$$

2.3.3 Interpretability

Interpretability refers to the degree to which a human can understand *why* a decision (*output* produced) was taken by a model [48]. It has become an essential part of responsible AI, as this has progressively been applied in more accountable social contexts [25]. Interpretability allows for checking that the system is making robust decisions; the abstractions the model has learned are consistent and similar to reality [50]. Another advantage of interpretability is detecting if a model operates unfairly (which means it is biased). We will focus on this last usage.

Interpretability methods have several approaches to use. On the one hand, they can be classified between *local* and *global* methods [50]. *Global* methods are those that explain the general average behavior of the ML-based model, whereas *local* methods explain how the model has produced any individual prediction. However, they can also be classified between model-agnostic and model-specific methods. The first ones apply to any model, independently of the algorithm it has used to train since methods are decoupled from the model's internal working. In contrast, the model-specific ones work for models trained with a specific algorithm since the interpretability method has been designed to take advantage of the algorithm's features. In the following lines, the focus is on studying local and model-agnostic methods that can be applied to NLP since they are what we consider relevant for our future design (see Chap. 4).

Local Interpretable Model-Agnostic Explanations

It is a technique designed to explain the predictions of any machine learning classifier. It works by approximating the real model locally with an interpretable model, creating a new simple model that mimics the original model's behavior near the specific prediction being explained [64]. The key idea is to perturb the input data for this new model and observe the changes in the output to understand which parts of the input are most influential for prediction [50]. For text data (NLP), Local Interpretable Model-Agnostic Explanations (LIME) modifies the text by removing or perturbing words and

then observes the changes in the model's output.

This process involves four steps [64]. First, a perturbation generates a set of angry instances by randomly removing words from the original text. Then, the original model (the one that is wanted to be audited) is used to predict the outcomes for these troubled instances. Next, a weight to each perturbed instance is assigned according to its similarity to the original instance. Lastly, an interpretable and simpler model is fitted on the perturbed cases and their associated predictions, using the weights to give more importance to instances similar to the original text. This local and interpretable model identifies and highlights the essential features that influence the prediction for any particular instance. By focusing on this model, LIME provides insights into why the complex model made a specific decision, allowing users to trust and verify the model predictions more effectively.

To understand LIME's internal working, an example is commented on [50]. Let us suppose that we have a complex *black box* model (i.e., we cannot interpret why its decisions are taken), which is a Recurrent Neural Network (RNN) trained on a corpus with the structure shown in Table 2.1, and whose function is to classify texts between spam (1) and normal (0). To make interpretable what the model is doing to classify, LIME can be applied. For example, let us examine why the model classifies as spam the sentence "For Christmas Song visit my channel! ;)".

CONTENT	**CLASS**
I am a good student	0
For Christmas Song visit my channel! ;)	1

Table 2.1: Samples of corpus for detecting spam

The first step is to create some variations of the analyzed text, shown in Table 2.2. Each column corresponds to one word in the sentence. "1" means that the word has been kept as part of this variation, whereas "0" means that the word has been removed. For example, the corresponding sentence for the first perturbation of the variations is "For Song visit ;)". The column "prob" shows the predicted probability of spam for each of the troubled sentences, and the column "weight" shows the proximity of the variation to the original sentence. As shown in Table 2.2, perturbed sentences containing the word

"channel" have the probability of 99 % of being spam. In contrast, the rest (without the word "channel") have only 17%, concluding that the original model has learned to give massive importance to the word "channel" to classify a text as spam.

For	**Christmas**	**Song**	**visit**	**my**	**channel!**	**;)**	**prob**	**weight**
1	0	1	1	0	0	1	0.17	0.57
0	1	1	1	1	0	1	0.17	0.71
1	0	0	1	1	1	1	0.99	0.71
1	0	1	1	1	1	1	0.99	0.86
0	1	1	1	0	0	1	0.17	0.57

Table 2.2: Perturbation of sentence "For Christmas Song, visit my channel!;"

SHapley Additive exPlanation (SHAP)

This method is based on cooperative game theory to make predictions of any machine learning model interpretable by assigning each feature an importance value for a specific prediction [40]. The values of SHAP represent the contribution of each feature to the model output, providing a fair and consistent measure of the importance of the features. The key idea is to attribute the model's prediction to each feature considering all possible combinations of features and their respective contributions [51]. In NLP, SHAP can be used to determine the contribution of each word or token to a model's prediction.

This involves four steps [40]. First, the prediction of the model for the original text is calculated. Then, all possible subsets of the features (words) are considered, and the model's predictions for these subsets are computed. Then, the marginal contribution of each word to these subsets is calculated. Lastly, the SHAP values are obtained by averaging these marginal contributions on all possible subsets. To understand the application of SHAP in NLP, consider a sentiment analysis task in which we want to classify movie reviews as positive (1) or negative (0), as shown in Table 2.3. Similar to the LIME example, suppose that we have a complex model like a RNN trained to perform this classification.

Review	Sentiment
The movie was fantastic!	Positive
I did not like the movie.	Negative

Table 2.3: Examples of movie reviews and their sentiments

To explain why the model classifies the review "The movie was fantastic!" as positive, SHAP can also be applied. As commented before, the first step is to calculate the prediction for the original sentence with the model that would be audited. Then, for all possible subsets of the words "The", "movie", "was", and "fantastic!", the model predictions are also calculated. Some examples of these subsets of words are shown in Table 2.4.

Samples of subsets
The movie was
the movie was fantastic!
"The fantastic!"

Table 2.4: Examples of subsets of words for "The movie was fantastic!"

After obtaining the prediction of all possible subsets, the marginal contribution of each word to the prediction is calculated by the difference in the predictions with and without the word of the original sentence. For practical demonstration, let's consider, as shown in Equation 2.6, a simplified calculation for the word *"movie"* in the sentence *"The movie was fantastic!"*.

$$\phi_{\text{i}} = f(\text{"The movie was fantastic!"}) - f(\text{"The was fantastic!"}) \tag{2.6}$$

Where:

- ϕ_i is the contribution for the word i, being $i =$ "*movie*".
- $f(S)$ is the model's prediction when considering only the words in subset S.

Finally, SHAP values are calculated, which are derived by averaging the marginal contributions of each word across all possible subsets of the other words in the sentence. These values indicate the contribution of each word to the positive sentiment prediction. Table 2.5 shows both marginal distribution and SHAP values for each word. In this example, SHAP values highlight the word "fantastic!" as the most significant contributor to the positive sentiment prediction, with a SHAP value of 0.4. This indicates that the model relies heavily on "fantastic!" to classify the review as positive, providing a clear and interpretable explanation of the model's decision-making process.

Word	**Marginal Distribution**	**SHAP Value**
The	0.1	0.05
movie	0.3	0.15
was	0.2	0.1
fantastic!	0.8	0.4

Table 2.5: SHAP values for words in the review "The movie was fantastic!"

2.3.4 Usage of Metrics

It involves calculating a series of metrics to quantify differences in model behavior between various groups [16]. These metrics can be calculated over the corpus, the output of the model, or a combination. Several metrics are proposed in different articles [12] [17]. Still, after studying the literature, we have decided to get to the bottom of the framework proposed by Paula Czarnowska et al. (2021). In this article, 146 papers on social bias in NLP are surveyed, and the multitude of metrics found are unified under three generalized fairness metrics [16]. Next, these fairness metrics can be applied to text classifiers.

According to the literature, the metrics used to quantify bias in NLP models can be categorized into two main approaches: *group* fairness and *counterfactual* fairness. *Group* fairness aims to ensure that a specific statistical measure is balanced between various protected social groups, such as gender or race. This means that specific outcomes, such as classification rates, should be the same for all groups. For example, group fairness

would require that a model predict the same rate of positive outcomes for male and female names, ensuring that no group is disadvantaged. On the other hand, *counterfactual* fairness focuses on ensuring fairness at the individual level by comparing the treatment of an individual in the real world with hypothetical versions of the same individual in different scenarios. This involves altering specific identity attributes, such as changing a name from a female to a male name, to see if the model's predictions remain consistent. The idea is to guarantee that a change in the protected attribute does not modify the outcome for an individual, thereby treating similar individuals similarly regardless of group membership. The generalized fairness metrics proposed by Paula Czarnowska et al. (2021) take this metrics classification as a premise for their contribution.

The three fairness metrics proposed are Pairwise Comparison Metric (PCM), Background Comparison Metric (BCM), and Multi-group Comparison Metric (MCM). The PCM metric is designed to quantify the differences in performance between two randomly selected groups on average. This metric is beneficial for examining whether and to what extent the protected groups chosen differ in model performance. For example, when considering the sensitive attribute of disability, PCM can help determine whether there are performance differences between the cognitive disability, mobility disability, and non-disability groups. The PCM achieves this by calculating the average distance between the scores of different groups, thus providing information on the disparities between the social groups within the model. The metric can be divided into *Group* PCM, which focuses on the disparities between predefined groups, and *Counterfactual* PCM, which evaluates the impact of group changes by simulating hypothetical scenarios and comparing these variations with each other.

Next, the BCM metric compares the performance of a specific protected/sensible group against a defined "background" group, which serves as a reference point. This background can vary depending on the research question and the task context but usually represents the least discriminated social groups. For example, if the objective is to assess whether a model's performance for a specific group differs from its overall performance, the background may comprise all evaluation examples. Alternatively, if the focus is on determining whether disadvantaged groups are treated differently than privileged ones, the background could be a set of examples related to a privileged group. In the context of *Group* BCM, the comparison is made between the performance of the group and this background set, highlighting the disparities between the group and the

broader context. In contrast, *Counterfactual* BCM uses the original sentence as the background and compares it with various counterfactual versions (i.e. modified versions that change protected attributes) to assess whether the model's predictions remain consistent. BCM helps to answer questions about model bias and treatment of different groups by contrasting each group's scores with this background.

Finally, the MCM metric simultaneously evaluates how a sensitive attribute affects the model performance across all protected groups. MCM considers the combined impact of sensitive attributes (such as gender or race) on the model's results. This metric aims to identify global biases by analyzing the overall effect of changes in these attributes. While MCM can provide a valuable initial insight into potential biases, it often requires further investigation to understand the nuanced influences of specific groups. *Group* MCM evaluates the aggregated effect of a sensitive attribute (like gender) in all protected groups, providing insight into whether the presence of such attributes causes variations in model scores. This group-focused approach helps identify overarching biases that may affect multiple groups collectively. Meanwhile, *Counterfactual* MCM delves into how a model's prediction varies when hypothetical changes are made to an individual's attributes across different scenarios, emphasizing the potential changes in the outcome. This version is especially useful for determining whether introducing a particular attribute affects the consistency of the model's performance on an individual basis when considering different counterfactual worlds.

2.3.5 DBias

It is a methodology designed to identify and mitigate social biases in textual data, particularly in news articles [63]. It conforms to a pipeline of four main stages: bias detection, recognition, masking, and debiasing. Initially, bias detection uses a fairness evaluation metric, highlighting potential biases related to sensitive attributes such as race, gender, and age. Subsequently, bias recognition is achieved through fine-tuned models that detect specific biased words or phrases. The pipeline then applies bias masking techniques to replace biased content with neutral alternatives. Finally, the debiasing step employs algorithms to reduce or eliminate biases from the data, thereby improving the fairness of ML models trained on the processed data.

The *DBias* methodology addresses several challenges inherent in existing fairness frameworks, which often focus only on isolated stages of the ML lifecycle. By provid-

ing a comprehensive, end-to-end solution, *DBias* ensures that biases are systematically identified and mitigated across all stages of the ML lifecycle. This improves the fairness of the results and generalizes across various text domains beyond news articles. In comparative evaluations against other state-of-the-art fairness models such as *FairML* [1] and *AIF360* [10], *DBias* demonstrates superior performance in reducing biases [63].

CHAPTER 3

Enabling Technologies

This chapter offers a brief review of the main technologies that have made this project possible

3.1 Python

Python [74] is a high-level, interpreted, general-purpose programming language released in 1991. It is known for its clear and readable syntax, which facilitates learning for beginners and quick development for experienced developers [41]. Python supports multiple programming paradigms (object-oriented, imperative, functional, etc.), making it versatile for various applications, from web development to ML and data analysis [9]. Python's design philosophy emphasizes code readability and simplicity, promoting significant indentation to define code blocks [42].

Python also boasts a vast library collection of standard libraries that allows developers to perform everyday tasks without installing additional packages [74]. Moreover, its library ecosystem of third-party libraries is extensive and free. This robust support has made Python one of the most popular languages today, according to industry surveys and analyses [41]. In fact, within the academic and scientific context, Python is a fundamental tool because it handles complex data and performs advanced numerical computations [9]. Today, many publications and technical documents use Python as a reference language for implementing algorithms and example models due to its clarity and conciseness [74]. Next, we explain the primary Python libraries used in this thesis.

3.1.1 NumPy

NumPy [55] is a package for scientific computing in Python. It is a library that provides multidimensional vector object management and an assortment of routines to operate on them rapidly. This includes mathematical and logical operations, shape manipulation, sorting, introductory linear algebra, basic statistical operations, etc. NumPy offers other containers (storing lists of elements) with characteristics different from basic Python arrays. This improves speed, reduces memory consumption, and provides a high-level syntax for everyday processing tasks. In the context of NLP, NumPy is a crucial tool for working with word embeddings, which are essentially long sets of multidimensional vectors.

3.1.2 Scikit-learn

Scikit-learn [58], often abbreviated as *sklearn*, is an open-source Python library that provides a simple and efficient toolkit for data mining and analysis, built on top of

NumPy, among others. It is primarily designed for ML, offering a range of supervised and unsupervised learning algorithms, including classification, regression, clustering, and dimensionality reduction [13]. This library is particularly suitable for building predictive models, performing feature selection, and performing cross-validation, becoming a popular choice among data scientists and ML practitioners for academic research and industry applications [52].

Probably, the main advantage of *sklearn* is its ease of use, with a consistent and well-documented Application Programming Interface (API) that allows for quick prototyping and implementation of ML models [13]. In addition, it also integrates seamlessly with other scientific Python libraries, enhancing its flexibility. Regarding the disadvantages, *sklearn* is particularly limited in handling large datasets, as it operates mainly in memory [28]. Additionally, while covering a broad range of algorithms, it may not include the most specialized methods found in other libraries focused on deep learning [52], such as TensorFlow [73] or PyTorch [62]. Despite these limitations, *sklearn* remains a precious tool for many machine learning projects, especially those involving more traditional methods and small to medium-sized datasets.

3.1.3 LIME package

The `lime` package [65] in Python provides an easy-to-use interface to implement LIME in different domains, including tabular data, images, and text. Based on the scope of this thesis, we are interested in the text domain. It is focused on interpreting NLP and is contained in a sub-package known as `lime.lime_text`. Designed explicitly to explain text classifiers, this module allows users to generate explanations for predictions made by models that process textual data. The main class in this module is `LimeTextExplainer`, which is responsible for creating explanations by applying the LIME techniques (see Sect. 2.3.3).

This class is the core component of the `lime.lime_text` module and its constructor have the following parameters:

- `kernel_width`. Controls the width of the kernel used in weighting the perturbations. The higher the value, the more precise and computationally consuming the processing of LIME scores is.

- `split_expression`. Determines how the text is split into tokens for analysis.

- `bow`. If set to `True`, the text sample is treated as a bag of words; otherwise, the order of words is considered.
- `char_level`. If set to `True`, explanations are generated at the character level. Otherwise, they are generated at the word level.
- `class_names`. Iterable object with the identifier of each possible label (e.g., for label '0', its class name is 'negative').
- `random_state`. Ensures reproducibility by setting the random seed.

The primary method of this class is `explain_instance`, which generates explanations for a given text instance. This method must pass the following parameters:

- `text_instance`. Raw text string to be explained.
- `classifier_fn`. The classifier's prediction probability function. For *sklearn*-based models, this is `classifier.predict_proba`.
- `labels`. Iterable objects with labels to be explained.
- `num_features`. Maximum number of features (words) present in the explanation. The most relevant ones are selected.
- `num_samples`. Size of the neighborhood to learn the linear model. The higher the value, the better, but computationally costly is the LIME output.

In Code 3.1, an example of using `LimeTextExplainer` with its `explain_instance` method is shown. This example generates an explanation for the prediction of a text classifier in the sentence "This movie is great!" by identifying the top five words (as set in the `num_features` parameter) that contribute to the prediction. This result is stored inside an object of type `Explanation` class that is returned by the `explain_instance` method. This class is located inside the package `lime.explanation` and contains several methods to represent and visualize the LIME result in different ways. Some of the methods that `Explanation` class provides to visualize the results are:

- `as_html`. It returns the explanation as an Hypertext Markup Language (HTML) page. The desired labels, whether to include prediction probabilities and other options, can be specified to personalize the report. The function returns HTML code with JavaScript included.

- `as_list`. It returns the explanation as a list of tuples. Each tuple contains a representation and a weight. This is label-specific for classification tasks.
- `save_to_file`. It saves the HTML explanation to a specified file path.

```
from lime.lime_text import LimeTextExplainer

explainer = LimeTextExplainer(class_names=['negative', 'positive'])
explanation = explainer.explain_instance(
    text_instance='This movie is great!',
    classifier_fn=classifier.predict_proba,
    num_features=5
)
```

Listing 3.1: Example code of using the LIME package in Python

3.1.4 Flask

Flask [22] is a lightweight web framework for Python, designed with simplicity and flexibility in mind. It was created to provide a more straightforward alternative to more complex frameworks, catering to developers who prefer control over the components they include in their applications. Flask follows the Web Server Gateway Interface (WSGI) standard, which makes it suitable for building web applications and APIs. It is particularly well-suited for small to medium-sized projects where you need to quickly set up a web server without the overhead of a full-stack framework. Flask's strengths lie in its modularity, allowing developers to choose their tools, and its extensive documentation, which makes it accessible for both beginners and experienced developers. However, its minimalism can also be seen as a drawback, as developers may need to implement functionalities readily available in more comprehensive frameworks.

Flask is commonly used for building APIs, microservices, and simple web applications. Provides essential tools for routing, request handling, and templating (using HTML). However, it does not impose a specific project structure or include built-in features such as authentication or database management by default. This makes Flask a popular choice for projects that need fine-grained control or integration with other services. Despite its simplicity, Flask can be extended with a rich ecosystem of extensions for tasks such as database integration, form validation, and user authentication

if required. The code 3.2 shows an example of a basic Representational State Transfer (REST) API server built with Flask. In particular, a route that can be accessed via Hypertext Transfer Protocol (HTTP)'s GET method is defined, which returns a simple JavaScript Object Notation (JSON) file with a field *message* containing a greet to a name passed as a parameter through the Uniform Resource Locator (URL) in the HTTP request.

```
1  from flask import Flask, jsonify, request
2
3  app = Flask(__name__)
4
5  @app.route('/api/greet', methods=['GET'])
6  def greet():
7      name = request.args.get('name', 'World')
8      return jsonify({'message': f'Hello, {name}!'})
9
10 if __name__ == '__main__':
11     app.run(debug=True)
```

Listing 3.2: Basic Flask REST API Server

3.1.5 Streamlit

It is an open-source Python library that enables the rapid development of web interfaces for data science and ML projects [70]. Streamlit allows developers to create interactive and visually appealing dashboards without extensive web development knowledge. It automatically handles the *frontend*, enabling data scientists to focus on the data and the application logic. Applications can be created by writing Python scripts that are then run and displayed as web apps. Developers can build the interface by writing Python without managing the HTML or Cascading Style Sheets (CSS). Streamlit is based on the component-based paradigm and supports a wide range of widgets, data visualizations, and layout options, making it an excellent tool for rapidly prototyping and deploying data-driven applications.

The primary feature of Streamlit is its simplicity: By writing a few lines of code, developers can turn complex data workflows into fully functional web apps. For example, a simple application that allows users to input a number and see its square can be created with minimal code (only two lines written in Python) [69]. This code is shown

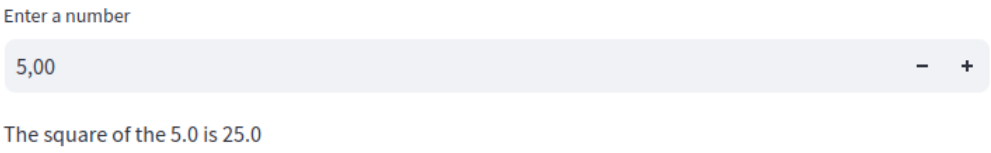

Figure 3.1: Web Interface produced by Streamlit Code of Listing 3.3

in Listing 3.3, and the output is shown in Fig. 3.1. Remarkably, the web interface has been constructed without typing HTML or CSS since Streamlit independently abstracts them from the developers and handles them in the background.

```
1 import streamlit as st
2
3 # User input
4 number = st.number_input('Enter a number', value=1)
5
6 # Display the square of the number
7 st.write(f'The square of {number} is {number**2}')
```

Listing 3.3: Code for Web Interface for calculating number's squares

3.1.6 Responsibly

Responsibly [2] is a comprehensive Python toolkit developed to audit and mitigate bias in ML environments. Designed to integrate with the existing *Python-based*, data science, and ML ecosystem, Responsibly is fully compatible with popular tools such as *NumPy*, *Pandas*, and particularly *Scikit-learn*. This integration allows users to easily incorporate bias auditing and fairness evaluation into their existing workflows without learning new paradigms or frameworks. The primary objective of *Responsibly* is to serve as a one-stop-shop for auditing bias and ensuring fairness in ML systems. It enables users to assess, identify, and understand potential biases in their models and functionalities, focusing on mitigating identified biases and adjusting fairness through various algorithmic interventions. This toolkit particularly emphasizes NLP by offering specific tools tailored to the unique challenges of this ML branch.

Responsibly is organized into three sub-packages, each catering to a specific aspect of fairness and bias in ML.

- `responsibly.dataset`. This sub-package offers a collection of common benchmark datasets frequently used in fairness research. These datasets are a standard foundation for testing and comparing various bias mitigation strategies. By using these well-established datasets, practitioners can ensure that their evaluations are consistent with the broader research community.

- `responsibly.fairness`. This subpackage is focused on demographic fairness in binary classification tasks. It includes a comprehensive suite of metrics and algorithmic interventions designed to measure and mitigate bias. The metrics provided allow users to evaluate models against various fairness criteria, such as demographic parity and equal opportunity. Additionally, the subpackage offers tools to adjust the models to meet these criteria better, enhancing the outcomes' fairness.

- `responsibly.we`. This sub-package addresses biases in word embeddings, a crucial component of many NLP-based systems. It includes the possibility of reducing direct and indirect bias by using the method proposed by *Bolukbasi et al. (2016)*, as well as bias measurement by implementing WEAT tests (see Sect. 2.3.1). This is the only module we focus on in this thesis, following the NLP scope we are addressing.

3.1.7 DBias

The *Dbias* library [18] is a powerful tool designed to implement the *DBias* methodology within ML workflows, explicitly focusing on NLP models. This provides a set of functions and modules that facilitate identifying and mitigating biases in texts, particularly in news articles. Its primary goal is to offer researchers and developers an easy-to-use interface for detecting and correcting biases in their corpus, thus promoting fairness and reducing discriminatory outcomes in ML applications. The *DBias* Python library is composed of several modules, each designed to address specific aspects of bias in textual data. These modules provide functionalities for debiasing, classification, recognition, and masking biased words in the language. This library has four modules:

- `Dbias.text_debiasing`. This module is designed to remove bias from text, specifically targeting biased language in news articles and other forms of written content. Using the `run()` function, users can input a sentence fragment, and the module returns an unbiased version of the text. This is particularly useful for ensuring that content presented to the public is free from potentially harmful biases.

- `Dbias.bias_classification`. It enables users to classify whether a given piece of text is biased or not. The `classifier()` function analyzes the input text fragment and returns a classification label indicating if there is bias present. This module is crucial for applications where it is important to assess the bias level of a text.

- `Dbias.bias_recognition`. This module identifies specific biased words or phrases within a text. By using the `recognizer()` function, practitioners can extract entities deemed biased from a sentence fragment. This module highlights problematic language and provides insights into which parts of a text may contribute to biased perceptions.

- `Dbias.bias_masking`. It provides a way to conceal biased portions of text. The `masking()` function identifies and masks the biased entities within a sentence fragment, effectively removing them from the visible text. This module can be applied when it is necessary to redact or obscure biased language, ensuring that the remaining text is neutral and unbiased.

3.2 Conda

Conda [5] is an open-source package and environment management system widely used in the scientific computing and data science communities. It simplifies installing, running, and updating software packages and their dependencies. Supports multiple programming languages, including Python. One of *Conda*'s key features is its ability to manage virtual environments. This is particularly useful when working on multiple projects that require different versions of the same package, preventing conflicts and ensuring reproducibility. In these contexts, *Conda* stands out by providing a streamlined process to create and manage these environments. Developers can easily create a new environment

with specific versions of packages, switch between environments, and delete them when no longer needed, all through simple *Conda* commands. These environments are isolated from the global system environment, ensuring that the dependencies and packages installed within a *Conda* environment do not affect the system-wide packages or other environments. Furthermore, *Conda* environments can be exported and shared, allowing consistent configurations and configurations between different systems, which is critical in collaborative research and development settings [68].

3.3 Visual Studio Code

Visual Studio Code (VS Code) [45] is a free and open-source code editor developed by Microsoft and available for Windows and Linux, among other operating systems. One of the main features of this editor is its support for extensions (plugins), which allow users to customize and add functionalities to the editor. This modularity enables accessible programming in a wide range of languages and the ability to add styles that facilitate the identification of code modules and parts. These extensions do not affect the editor's performance, as they run in independent processes. As we are programming in Python, the extensions used are *Pylance* and *Python*. On the one hand, *Pylance* [43] provides rich type information, autocompletion, and type-checking. Conversely, the *Python* [44] extension adds comprehensive support for Python development, including debugging and lining. Another crucial feature of VS Code is its ability to access multiple terminals simultaneously, thanks to its integrated terminal. This functionality is significant in this thesis, as we work with microservices that must be executed and managed parallelly from different terminals.

CHAPTER 4

Architecture

This chapter presents the methodology used in this work. It describes the project's overall architecture, with the connections between the different components involved in its development.

4.1 Overview

In this section, a general overview of the proposed solution is presented. We have designed and developed a system to audit the moral values of texts/corpus and NLP-based classifiers to ensure their fairness: the Moral Auditing System (MAS). It follows the classic web application pattern and has two main sides, shown in Fig. 4.1 on the left. On the one hand, the client side or the *front-end* represents a web interface that runs on the auditor's computer and allows him/her to easily interact with the auditing tools and visualize the auditing process's results graphically and intuitively. It comprises four modules, each with a different auditing functionality and a different subpage inside the web application.

On the other hand, the server side or the *back end* can be identified. As commented above, auditing is not a systematic process with a fixed methodology that can objectively be interpreted. Instead, each specific scenario is to be audited, and the auditor's interpretation significantly impacts how the auditing process is conducted and evaluated. Due to this reason, the *backend* is designed to contain independent services that do not depend on but complement each other, each containing all the necessary logic and functionality to carry out a specific form of auditing. For every scenario, the auditor must choose the correct tools (services) and how they should be combined. This kind of design also creates scalability, as more services could be added in the future without modifying the existing ones, making our system even more complete.

To implement this independence of services in practice, we have outlined a microservice-oriented architecture [4]. It emphasizes dividing the system into small and lightweight services purposely built to perform a cohesive business function by cooperatively working. Each service is a self-contained logic server that has its business logic, its environment (libraries, dependencies, etc.), and its own API to be interacted with from external resources (in our case, the *webapp*). According to Fig. 4.1, each defined service technically represents a microservice, so every module (page) on the client side contacts its associated microservice by calling the methods of its API and without the need to interact with the remaining microservices.

Regarding the technologies and tools used, all microservices have been designed using *Python* as the programming language, which we have chosen for three reasons. Firstly, *Python* is the most popular language to develop ML and auditing tools, so many libraries

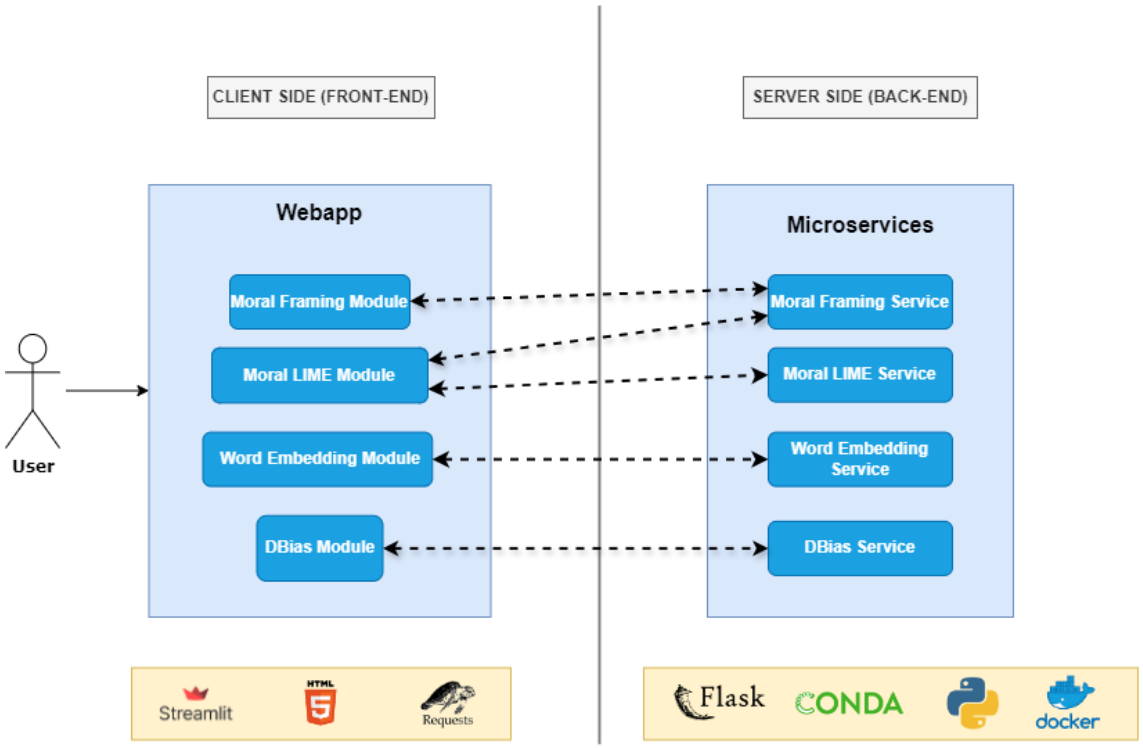

Figure 4.1: Architecture of Moral Auditing System

and resources are based on it. Secondly, several libraries to quickly work with mathematical vectors have been created in *Python*, which is crucial to operating on *word embeddings*. Lastly, libraries use this language to quickly develop an API server. Specifically, the framework used to create the API is Flask [60], which makes us capable of producing an REST API in a few lines of code due to its excellent abstraction capacity. In addition, using *Python* lets us create virtual environments to isolate the dependencies of each microservice even when developing on a local computer. That way, possible conflicts between versions are avoided, and we will keep the principle of making every microservice independent. To increase the portability of the *backend* when deployed in production, each microservice is intended to be virtualized as a container using *Docker* so that the microservice dependencies do not depend on the underlying operating system. Using virtual environments in the development stage allows us to temporally isolate the services' operation and make the future migration of these to *Docker* containers easier.

The Web application was designed using Streamlit. We considered that his framework has a lot of limitations in achieving complex graphics designs and getting good

scalability (regarding maintenance due to the lack of software architecture patterns and in terms of performance). Still, we finally decided to go ahead for two reasons. On the one hand, our web app must not be too complex; it must be just a simple tool to interact with the microservices and visualize their results. For this reason, having a software design pattern is not essential since the complexity of the code is acceptable. On the other hand, Streamlit is based on *Python* and works at a very high level, getting us away from using HTML or CSS, as well as allowing us to use most of *Python*'s plots and tools for data management. These two reasons are essential, as we can develop a good *frontend* but dedicate most of the time to the *backend*, where the objective complexity resides. To establish a communication between the *frontend* modules and the correspondent microservices (which implements a REST API), we use the *Python*'s library *requests* to make HTTP requests (using GET or POST methods).

Below, each microservice is independently analyzed, followed by how they are used and combined through the web app module, if necessary.

4.2 Moral Framing Service

This microservice is intended to analyze texts and quantify their moral framing according to the "Framing Axis" method described in Sect.2.3.2. In particular, we quantitatively measure the impact of every moral foundation from the MFT, indicating how polarized the text is to the vice/virtue of the foundation (bias) and how much this moral foundation is making an impact on the text (framing intensity). This method is relevant because it is unsupervised and does not need a human-annotated corpus. This makes us avoid a possible *measurement* bias propagated when labeling the corpus. Instead, this method uses *word embeddings* to capture the semantic meanings and relations between words and determine if the text is biased.

The chosen *word embedding* is "GoogleNews-vectors-negative300". The reasons are: 1) it uses *word2vec* [46], which is the most popular algorithm used for learning word embeddings from text data; 2) it contains 3 million word vectors, meaning it has embeddings for 3 million different English words or phrases; 3) each vector is represented in 300 dimensions, which balances computational efficiency with the richness of representation; and 4) it was trained on the Google News corpus (i.e. text data from Google News articles), which perfectly fits our problem since one of our targets is evaluating the moral framing of news. It is essential to highlight that whatever embedding is used must

not apply any bias mitigation techniques beforehand, as these biased patterns inside the embedding are necessary to determine if the text we are evaluating is morally biased.

Using the embedding, we calculate the direction vector (axis) of every moral foundation by subtracting a set of words of one of the opposite polarities (virtue) from the other (vice). To obtain these words, we use the MFD, a large set of words classified by moral foundation and vice or virtue. Once we have the corresponding directions, we are prepared to calculate the cosine similarity and apply the formulas defined by the *Framing Axis* method to calculate the bias and intensity of the texts. Next, how this method was implemented is explained.

This microservice implementation consists of a Python server using Flask that defines a REST API, and an additional Python class named *MoralFraming*, which contains all the internal logic of the microservice. First, the class is described, highlighting each method, followed by a description of the API routes.

4.2.1 MoralFraming Class

All the necessary logic is contained inside a class denominated as *MoralFraming* by using the Object Oriented Programming (OOP) paradigm supported by *Python* language. It contains several auxiliary methods (identified because its name starts with "_") and one primary method called *get_metrics*, which is the one that must be called from outside.

Computing axes

The first method is *_compute_axes*, invoked in the class constructor. This function calculates moral framing axes using the embedding of already commented words. The function begins by grouping the words obtained from the MFD into different moral categories based on the foundation. For each category, it separates the words labeled virtues and vices. Then, it attempts to retrieve the associated vector of each word from the embedding. It is ignored if a word's vector is not found in the model. Once the vectors are collected, the function calculates the mean vector for the words virtue and vice within each category, representing the average semantic meaning of virtue and vice. These mean vectors are then used to compute a moral framing axis by subtracting the mean vice vector from the mean virtue vector. The function stores the axis of each moral category in a dictionary called *axes*, with the category name as the key and the calculated axis as the value. Finally, the function returns the *axes* dictionary, which

contains the moral framing axes for each category and is stored as an attribute of the class to be used by other methods.

Cosine similarity

The `_cos_sim` method calculates the cosine similarity between two vectors, a and b, passed as parameters. Equation 4.1 shows the mathematical function used as a reference. In this method, the dot product of the vectors is computed first using `np.dot(a, b)` from *numpy* library, which gives a measure of their overlap. Then, the method calculates the or magnitudes of each vector using `np.linalg.norm(a)` and `np.linalg.norm(b)` using *numpy*, too. The cosine similarity is obtained by dividing the dot product by the product of these norms, which effectively normalizes the vectors and yields a value between '-1' and '1', where '1' indicates identical orientation, '0' indicates orthogonality, and '-1' indicates opposite orientation.

$$\texttt{cos\_sim}\,(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} \tag{4.1}$$

Preprocessing methods

It includes three auxiliary functions. The first is `_delete_stopwords`, which returns the text passed as a parameter with stop words and punctuation removed. The second function is `_tokenizer`, which takes a text as an input parameter and returns a tokenized version, split into words. The final function is `_count_words`, which counts the frequency of each token in the tokenized text.

Getting Bias Score

The `_get_bias_score` function calculates a biased score based on the frequency of tokens and their similarity to a specified moral framing axis. In essence, this function implements the mathematical equation to calculate the bias score defined by the *Framing Axis* method (see Sect. 2.3.2). The function first checks if the frequency dictionary (*freq_dict*) and the moral framing axis (*mf*) are provided as parameters. If not, it raises an exception of type *ValueError*. It then initializes variables to keep track of the cumulative bias score and the total frequency of the tokens. Each token in the frequency dictionary calculates the cosine similarity between the token's vector (obtained from the embedding defined as an attribute in the class) and the vector corresponding to the

specified moral framing axis. This is done via the `_cos_sim` function. This similarity score, weighted by the token's frequency, contributes to the bias score. The function sums these weighted similarity scores and normalizes the result by the total frequency of the tokens. If the normalized bias score is infinite or Not a Number (NaN), it is set to zero. Finally, the bias score is returned.

Getting Intensity Score

The `_get_intensity_score` function computes an intensity score reflecting the variance in token similarities relative to a baseline threshold (B_T). This function implements the mathematical equation to calculate the intensity score defined by the *Framing Axis* method (see Sect. 2.3.2). Similar to the _get_bias_score function, it starts by verifying the presence of two of the parameters frequency dictionary (*freq_dict* and the moral framing axis (*mf*), raising a *ValueError* if either is missing. The third parameter is B_T, which represents the bias score for the same *freq_dict* and *mf* axes. This function then initializes the variables to aggregate the intensity score and the total frequency of tokens. For each token, it calculates the cosine similarity (using `_cos_sim`) between the token's vector and the moral framing axis vector. This similarity score is adjusted by subtracting the baseline threshold, squared, and weighted by the token frequency to contribute to the intensity score. The function sums these squared differences and normalizes the result by the total frequency of tokens. If the normalized intensity score is infinite or NaN, it is set to zero. The function returns the resulting intensity.

Calculating Scores

The *_calculate_scores* function computes bias and intensity scores for each moral value on the model axes by using the *_get_bias_score* and the *_get_intensity_score*. This function takes as a parameter a frequency Python dictionary (*freq_dict*), where the keys are tokens and the values are their respective frequencies in the original analyzed text. This *freq_dict* is passed as a parameter to both the bias and intensity functions. This function calculates the scores for each foundation of MFT by iterating through each one in `self.axes` and getting bias and intensity scores for the text (represented in `freq_dict`) for each of them. These scores are then stored in a dictionary where each moral foundation is associated with its corresponding scores. The function returns this dictionary.

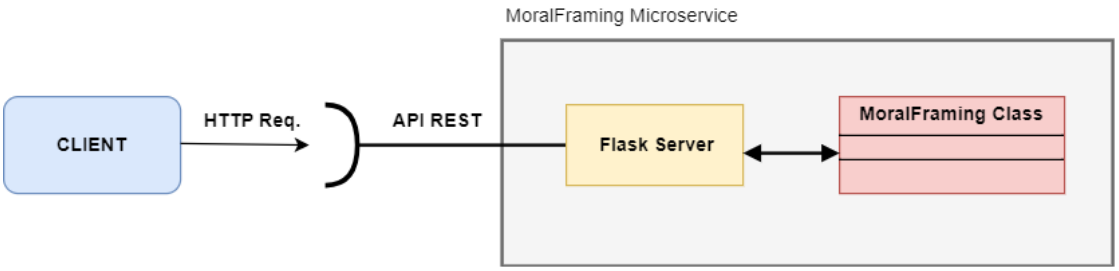

Figure 4.2: Moral Framing Service's internal architecture and external access

Getting Metrics

The `get_metrics` function is a higher-level method that processes a given text to generate moral framing metrics. This is the only function that must be called from outside the class. It first tokenizes the input text using the `_tokenizer` function, then removes stop words and punctuation signs using `_delete_stopwords`, leaving only significant tokens. It counts the frequency of each remaining token using the `_count_words` function, generating a frequency dictionary. This frequency dictionary is then passed to the `_calculate_scores` function, which returns a set of bias and intensity scores for each moral value. The `get_metrics` function ultimately returns these scores, providing a detailed analysis of the text's moral framing.

4.2.2 API methods

As a microservice is being designed, it must be provided with an API to be requested from outside. A REST API has been developed by using Flask. This includes two routes, which take advantage of the `MoralFraming` class to process the received data and return a result. Table 4.1 describes both routes in detail. Remarkably, both implement the POST method, as they are intended to receive data as a parameter. So, it is better to do it via the encrypted POST form than appending key-value pairs to the URL in the GET method. Choosing the POST method offers us more security and efficiency. Fig. 4.2 illustrates how the client (in this case, the *frontend*) interacts with the microservice and how the Flask server internally communicates with the MoralFraming class. The server invokes the MoralFraming instance by internally calling its `get_metrics` function.

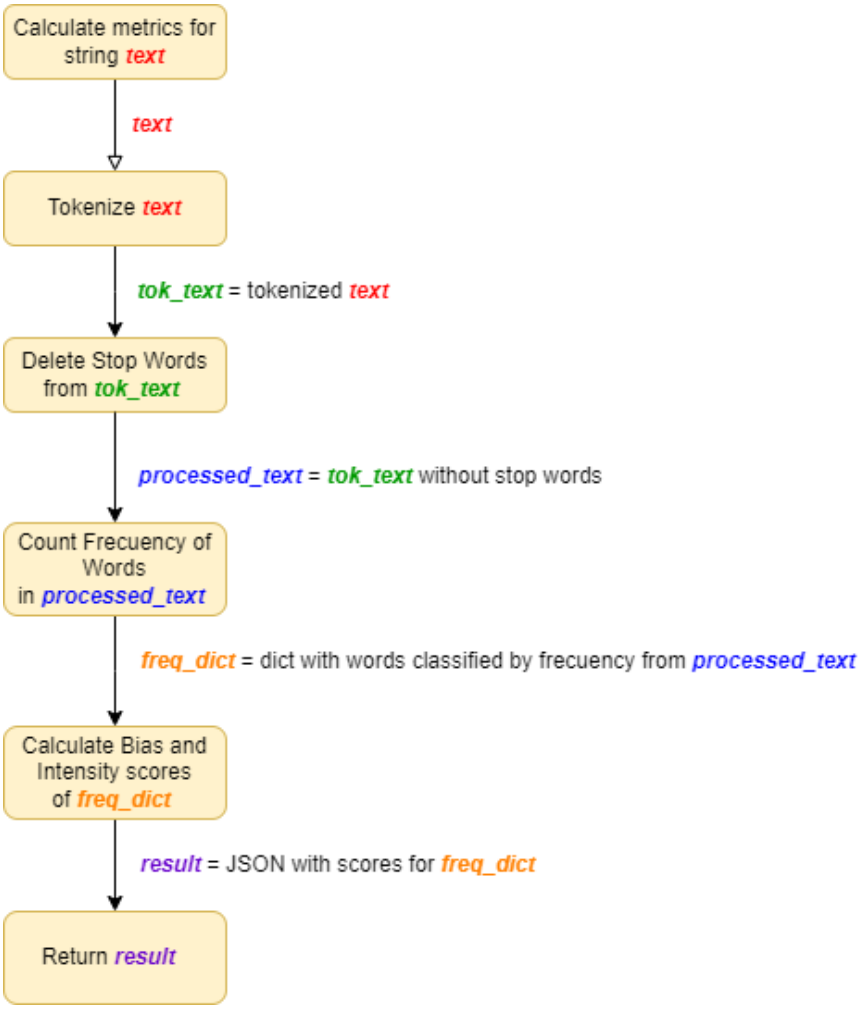

Figure 4.3: Flow Diagram of `_get_metrics` function

Route	HTTP Method	Explanation
`/metrics`	POST	It expects a text (of type *string*) as a parameter and calls to `get_metrics` from `MoralFraming` class to calculate its bias and intensity scores, returning them in a JSON file.
`/metrics_for_words`	POST	This method is passed a list of words as a parameter and individually calculates the bias score for each of them by calling to `get_metrics`. In this case, calculating the intensity score for only one word makes no sense, as shown in the mathematical function returning "0". In addition, the bias and intensity scores of all the words together are also calculated to get their cumulative effect. This all scores are returned via a JSON file.

Table 4.1: API definition for Moral Framing Service

4.3 Moral LIME Service

This microservice implementation is intended to use interpretability to detect moral bias on NLP-based classifiers of text. The main idea is that the microservice receives as a parameter a sample of text (which can be a news article) and the classifier to evaluate if the classification is unfairly being made. As commented in Sect. 2.3.3, we have studied two methodologies for model-agnostic and local evaluation of texts: LIME and SHAP. After a thorough comparison, we opted for LIME due to its comparable performance to SHAP and the more straightforward Python implementation. It must be noticed that the microservice's code is more concise than the previous *Moral Framing Service* so that it follows an imperative sequential paradigm rather than utilizing object-oriented programming.

4.3.1 Logical functions

Next, we comment on the main functions defined in the microservice's logic.

Processing functions

The `_format_list` function takes a list of tuples, each containing a word and an importance value. Convert each tuple into a dictionary where the word is the key, and the importance is the value. Then, it serializes this list of dictionaries into a formatted JSON string with an indentation of 4 spaces for readability. The function returns this JSON string.

Explaining with LIME

The `_lime_explainer` function is designed to use LIME Python library to explain the predictions of text classifiers. It takes four parameters: *model*, *labels*, *labels_name*, and *text*. The *model* parameter refers to the NLP classifier to be interpreted; *labels* are all the possible outputs of that classifier and *labels_name* a human-understandable tag for each label; finally, the *text* parameter refer to the sample text for which the prediction is intended to be explained. Inside the function, it first creates an instance of `LimeTextExplainer`. The explainer is configured with several options. First, the *kernel_width* is set to 100, prioritizing high approximation accuracy over efficiency or low computational cost. Furthermore, it cannot be known whether the classifier uses or

does not use word order since it is externally passed as a parameter; for that reason, in case of doubt, the *bow* is set to `False` to indicate that the classifier uses word order in its predictions.

Next, the function uses this explainer object to generate an explanation for the given text instance. It calls the `explain_instance` method of this explainer, passing in the text instance the classifier's prediction function to obtain prediction probabilities and all possible labels the classifier can produce. Additional configuration parameters are included: `num_features` is set to "50", instructing the explainer to return the fifty most relevant words contributing to the prediction. Meanwhile, `num_samples`, which specifies the number of perturbed samples to generate for approximating the local decision boundary, is set to "100,000", enhancing the stability and accuracy of the explanation at the cost of increased computational expense. Once the explanation is generated (as an object of the `Explanation` class), its `as_list` method is called to obtain a list of the LIME scores for the fifty most relevant words. By default, this method returns scores for all possible classifier outputs. However, since we focus on the most probable result, representing the final prediction, the classifier's prediction for the given text is previously retrieved and passed as a parameter to the `as_list` method. Finally, this list with the LIME scores is returned and the prediction retrieved.

Getting LIME scores

Finally, the `get_LIME_scores` function is a higher-level method that processes a given text to generate moral framing metrics. This is the only function that must be called by the REST API's internal logic. This function expects several key inputs. The first one is the primary text data whose classifier's prediction is wanted to be analyzed. The second is a serialized ML model file (in a format such as *.pkl* or *.joblib*). The two final parameters are the labels and their associated labels' names. Upon receiving these inputs, the function briefly performs pre-processing, including deserializing the model and verifying that it is a classifier that can predict probabilities. Next, the function `_lime_explainer` is called, with the text instance, model, and labels passed as parameters, as previously described. This function returns a list containing the LIME scores. This list is then passed to the `_format_list` function, converting it into a JSON file, which is finally returned.

4.3.2 API methods

Following the same principles and reasoning as those explained in the *Moral Framing* microservice, the REST API is defined as shown in Table 4.2.

Route	HTTP Method	Explanation
`/lime`	POST	Accepts a text (as a *string*), a list of label indices (as *integers*), a list of label names (as *strings*), and a serialized ML model file. These parameters are passed to the `get_LIME_scores` function to generate an explanation for the model's prediction of the given text. The result, including the most important words with its LIME scores, is returned in JSON format.

Table 4.2: API definition for the Moral LIME Service

4.4 DBias Service

This is a direct implementation of the DBias methodology (see Sect. 2.3.5). The main objective is to offer a REST API to take advantage of the different modules of the DBias Python library (explained in Sect. 3.1.7). As it offers a high-level abstraction, using it to implement the API is not a huge task. Out of the four modules available in the library, we choose text debiasing, bias probability classification, and biased entity recognition and discard the masking of biased words. This high-level abstraction makes the library an easy-to-use tool and entails certain limitations. One is that the bias is not classified according to its origin (for instance, whether the text is biased based on gender or race). Consequently, we do not consider this microservice a primary tool, but an auxiliary one. This implies that the bias probability classification module will return a global probability of bias without indicating the origin, which must be verified using other means. Similarly, the debiasing and entity recognition modules do not target specific types of bias but address broad, general social biases. Consequently, the results must be carefully interpreted and analyzed case by case.

4.4.1 Logical functions

The microservice's code defines several functions, which are *calculate_bias_score*, *debias_text*, and *recognize_entities*. These functions receive a text as a parameter and call the corresponding module's function to process it. Thus, *calculate_bias_score* computes a bias score probability (values from "0" to "1") to assess the level of bias present in the text using the bias probability classification module. Then, *debias_text* is used to remove or reduce bias from the text by using the text debiasing module. Lastly, the function *recognize_entities* identifies and extracts entities that could aggregate bias to the text by calling the biased entity recognition module. A fourth function, *get_bias_evaluation*, is invoked by the HTTP route. This function takes the text as a parameter and boolean flags for each module, indicating whether the respective modules must perform their calculations on the given text. For the selected modules (e.g., bias probability classification), *get_bias_evaluation* calls the previously defined functions (e.g., *calculate_bias_score*). Finally, *get_bias_evaluation* returns the results of the selected modules in a unified JSON format to the client.

4.4.2 API methods

Following the same principles as with the previous microservices, the REST API is defined as shown in Table 4.3.

Route	**HTTP Method**	**Explanation**
`/evaluate_bias`	POST	It accepts a text (as a *string*), along with boolean flags for each module (as *boolean*). These parameters are passed to the `get_bias_evaluation` function, which calls the relevant processing functions based on the selected modules. The results from these modules are combined and returned in JSON format.

Table 4.3: API definition for the DBias Service

4.5 Word Embedding Service

This last microservice aims to detect and mitigate the implicit bias inside word embeddings. This is done by extrapolating the method proposed by Bolukbasi et al. (2016). To carry out the programming, we take advantage of `responsibly.we` module from `responsibly` Python library. These strategies are potentially applicable to any word embedding, but the methodology cannot guarantee good performance, as it also depends on the specific characteristics of the embedding. For that reason, different tests are proposed, including performance and WEAT tests, for the user to check if word embedding continues to be helpful and if the bias has been reduced/mitigated. This service offers the possibility of debiasing the word embedding for each moral foundation and social issues (such as gender, race, etc.), assuming that they are practical manifestations of an underlying moral bias influence. Like the previous microservices, it is designed using Python with its Flask library for API.

This microservice is designed to provide functionality based on our defined methodology. It comprises four main phases. The first is making an *Exploratory Analysis*, which includes exploring the word embedding features, allowing the auditor to gain an initial understanding. This involves observing the vocabulary size, examining the dimensions, and identifying the word's most similar terms using cosine similarity. In this stage, it is also verified whether the embedding vectors are normalized and normalized if necessary. Next, we turn to the phase of *Calculating projections*. The process consists of obtaining the direction vector (projection) of the target bias (e.g., the moral foundation) by using two sets of words that represent two reference groups (e.g., vice and virtue terms for the foundation). The mean vector for each group is calculated by averaging the vectors of all words in that group. Then, you subtract the mean vector of one group (e.g., the vice terms) from the mean vector of the other group (e.g., the virtue terms). The resulting vector represents the direction of the bias in the embedding space, capturing the semantic contrast between the two groups. This direction vector can then measure how closely other words or concepts align with one of the reference groups over the other, reflecting their positioning relative to the defined bias.

The third phase is *Detecting bias*. In this third stage, the embedding's direct and indirect bias (explained in Sect. 2.3.1) are calculated. This is done using the direction vector obtained in the previous step. In the case of direct bias, using a list of neutral

words, each term is projected on the direction vector, and the absolute value of its cosine similarity is calculated (the bigger the value, the stronger the word's bias is). Then, these values are averaged to get the final metric of the direct bias. Indirect bias is assessed by measuring the degree to which a neutral word is associated with biased contexts through the relationships between words in the embedding space. We have prepared different lists of neutral words for the different bias targets; for example, for detecting gender bias, a list of neutral professions is used. After that, the fourth step is *Mitigation of Bias*. With the help of functions from `responsibly` library, a "hard debias" is applied to the embedding. This implies applying neutralize and equalize (also explained in Sect. 2.3.1). As a resume, equalizing means removing the target projection from all the words except the neutral ones. At the same time, neutralization implies making antagonist word pairs (e.g., associate with vice and virtue) equidistant from neutral words.

The final step is to *Test the Modified Embedding*. On the one hand, it is crucial to evaluate the performance of the embedding to ensure it remains comparable or only slightly diminished after applying the "hard debias" technique. Standard benchmarks assess how well the embeddings capture semantic similarity between words. Each benchmark provides a set of word pairs with known similarity scores, and our embedding is evaluated based on how closely their similarity scores match these reference values. The metrics include Pearson's correlation coefficient (*Pearson's r*), which measures the strength and direction of the linear relationship between the embeddings' similarity scores and human judgments. A higher *Pearson's r* indicates a stronger alignment with human assessments. The associated *Pearson's p-value* indicates the statistical significance of this correlation, with values close to 0 suggesting that the observed correlation is unlikely due to chance. Spearman's rank correlation coefficient (Spearman's r) assesses how well the rankings of word similarities produced by the embeddings align with human rankings; a higher *Spearman's r* implies better ranking agreement. The *Spearman's p-value* shows the significance of this ranking agreement, where values close to 0 indicate strong evidence that the correlation is not due to random chance. Additionally, the ratio of unknown words represents the proportion of words in each benchmark that were not represented in the embeddings. The benchmarks listed include:

- *WordSimilarity-353*. Known as *WS353*, contains 353 word pairs.
- *RG-65*. It is abbreviated as *RG65* and has 65 word pairs.

Benchmark	Pearson's r	p-value	Spearman's r	p-value	Unknown Words
WS353	0.645	0.000	0.688	0.000	9.915
RG65	0.576	0.232	0.493	0.321	14.286
RW	0.611	0.000	0.655	0.000	77.384
Mturk	0.650	0.000	0.674	0.000	1.558
MEN	0.766	0.000	0.782	0.000	15.148
SimLex999	0.456	0.000	0.444	0.000	1.702
TR9856	0.666	0.000	0.676	0.000	89.722

Table 4.4: Example of performance metrics for word embeddings on various benchmarks

- *Rare Words*. Focuses on less frequent terms.
- *Amazon Mechanical Turk*. Described as *Mturk*, provides human-annotated similarity judgments.
- *MEN*. A large dataset of word pairs for evaluating similarity.
- *SimLex999*. It consists of 999-word pairs designed to measure similarity.
- *TR9856*. Another similar benchmark dataset with specific evaluation details.

We calculate these metrics for both the original and the debiased embeddings to compare their performance and analyze the impact of debiasing on performance reduction. In Table 4.4, an example of the result of the performance metrics calculus is shown.

However, testing whether the bias has been removed or reduced is essential. The use of the WEAT is proposed to measure implicit biases by examining associations between different sets of words in word embeddings (see Sect. 2.3.1 for more details). WEAT quantifies the strength of associations between two sets of target words and two sets of attribute words based on their relative proximity to the embedding space. An example of a WEAT test is shown in Table 4.5. In this example, the target words are "Science

Target Words	**Attrib. Words**	**Nt**	**Na**	**s**	**d**	**p**
Science vs. Arts	Male terms vs. Female terms	6x2	8x2	0.3035	1.3731	0.0087

Table 4.5: Example of WEAT metrics between target words and attribute words

vs. Arts" (e.g. "science","physics","arts","literature"), and the attribute words are "Male terms vs. Female terms" (e.g. "male","man", "female", "woman"). The test compares the degree to which words from the target set are associated with words from the attribute set within the embedding space to assess gender-related associations.

Regarding the metrics, the *Nt* (Number of Target Words) indicates the total number of words in each category, with "6x2" meaning six words related to science and six related to arts. The *Na* (Number of Attribute Words) represents the number of words in each attribute category, with "8x2" reflecting eight male terms and eight female terms. The *s* (Similarity Score) quantifies the degree of association between target words and attribute words, which can range from "-1" to "1", indicating the strength of this association. The *d* (Difference Score) measures the disparity in associations between the two sets of target words with the attribute words, ranging from 0 to infinity, with a score of 1.3731 in the example indicating a notable difference. Finally, the *p* (p-value) assesses the statistical significance of these scores, which can range from "0" to "1". Low p-values imply that the observed associations are statistically significant, meaning there is a very low probability that these results occurred by chance.

4.5.1 Logical functions

Below, the existing functions in the microservice's code are briefly explained and classified according to the abovementioned methodology. Since word embedding files are usually large (several gigabytes), they are passed as a parameter to the server in advance and stored inside the server as a temporal variable for use by the other functions. The server uses the `BiasWordEmbedding` class from the `responsibly` library to store, handle and work with this word embedding.

Exploratory analysis

The first function is `describe_embedding`, which takes no parameters and returns introductory data, such as embedding length in JSON format. Another function, *cosine_similarity*, receives two words as input, calculates their cosine similarity, and returns a tuple with the cosine value and the corresponding degree between the vectors of the two words. A function named `most_similar_words` takes a word as a parameter and returns the ten most similar words within the embedding context. Finally, the `normalize` function checks whether the embedding is normalized and, if not, normalizes its vectors. Additionally, we have implemented a high-level global function called `describe_embedding` that receives a Boolean flag for every previously defined function, indicating if it is wanted to be called. This function then calls the selected functions and returns their combined global result in JSON format.

Calculating projection

A function called `calculate_projection` takes a boolean flag for each bias target (moral foundations, gender, race, etc.) as a parameter. Calculate projections for the selected targets using a predefined list of contrary words stored on the server. The method `identify_projection` from a `BiasWordEmbedding` object is used for it.

Detecting bias

Two functions are defined using methods from the `BiasWordEmbedding` object in this case. The first function, `get_direct_bias`, has the objective of measuring direct bias by using the `calc_direct_bias` method. The second function, `get_indirect_bias`, calculates indirect bias using the `generate_closest_words_indirect_bias` method. A third high-level function, denoted as `get_bias`, is defined. This function takes two boolean flags as parameters, indicating whether a particular type of bias should be calculated. Then, it calls the necessary previous functions to calculate the requested bias and returns the global result in JSON format.

Mitigating bias

Two functions are defined: 1) *_neutralize_embedding* and 2) `_equalize_embedding`, which are used to apply the neutralize and equalize mitigation strategies to the embed-

ding, respectively. Both functions utilize the `debias` method from the *BiasWordEmbedding* object, which takes the name of the strategy to be applied as a *string* parameter. Furthermore, we have implemented a third function called `debias_embedding` that receives two boolean flags as parameters, indicating which of the two strategies should be applied to the embedding. This function then calls the aforementioned auxiliary functions accordingly.

Testing the modified embedding

There are two functions in the microservice's code: 1) *_get_performance* and 2) *_get_weat*, which are used to apply the performance and WEAT tests to embedding (before and after bias mitigation). The first function utilizes the *evaluate_word_embedding* method from the *BiasWordEmbedding* object. However, the second function does not use a method from this object but instead calls the *calc_all_weat* function from the *responsibly.we* module. This function takes as parameters the *BiasWordEmbedding* object and the lists of target and attribute words. The code also presents a third function: *evaluate_embedding*. This is used to call the two previous functions and unify the results of all tests in the same JSON format for return.

4.5.2 API Methods

By following the same rules as applied to the previous microservices, we define this REST API as shown in Table 4.6. In this API definition, we use the HTTP GET method since the parameters passed to the server for some routes are just some boolean flags that can be included as URL parameters. Fig. 4.4 shows the internal structure of the microservice, where the server code interacts with the word embedding dictionary and how it can be accessed from outside (e.g., from the web app) via HTTP.

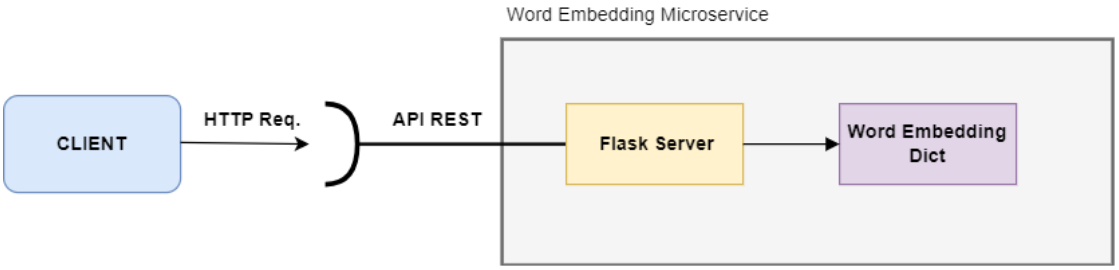

Figure 4.4: Moral Framing Service's internal architecture and external access

Route	HTTP Method	Explanation
`/set_embedding`	POST	Accepts a binary file representing the serialized word embedding, which is then stored in a server's session to be used in future calls by the user.
`/get_description`	POST	Receives boolean flags to indicate which exploratory functions should be executed and an optional list of words as parameters. This route passes them to the `describe_embedding` function and returns a result in JSON format directly obtained from this function.
`/detect_bias`	GET	Accepts two boolean flags that indicate the type of bias to be calculated. These parameters are passed directly to the `get_bias` function, which returns the calculated bias in JSON format. This result is then directly returned by this route.
`/mitigate_bias`	GET	Accepts two boolean flags that indicate the type of bias to be mitigated. These parameters are passed to the `debias_embedding` function, which returns a boolean flag indicating whether debiasing was successfully performed. This result is then directly returned by this route.
`/test_embedding`	GET	Does not accept any parameters and simply calls the `evaluate_embedding` function, which always calculates both performance and WEAT tests. The results of the tests are then returned in JSON format.

Table 4.6: API definition for the Word Embedding Service

4.6 Webapp Modules

This section discusses the *frontend* illustrated in Fig.4.1. We define a module as a piece of code within the web application that interacts with one or more microservices to achieve a specific auditing procedure outcome. Each module described corresponds to a distinct section of the web page (comprised of its graphic plus its logical part). These sections, in turn, adopt a multi-page design, which is detailed below. To begin with, it is essential to note that the use of an architectural design pattern, such as the widely adopted Model View View-Model (MVVM), has been intentionally omitted. This decision is based on the simplicity of the codebase, which is implemented solely in Python (due to the Streamlit high-level abstraction). Incorporating a design pattern would introduce unnecessary complexity without providing significant benefits, given the straightforward nature of the project. These modules are thought to provide an interface for the user to use the different microservices to detect and mitigate moral bias.

To enhance the user experience when using the web page, we have established a three-step methodology for each module, each step corresponding to a distinct page within the module. The first step involves *uploading the parameters*, which means that on the first page, the user must upload all the necessary elements for the audit. For example, this might involve uploading a text file containing the content to be analyzed by the *Moral Framing Service*. The second step is *configuring the auditing process*. Here, the user can select specific moral foundations from MFT that they wish to analyze within the text using the *Moral Framing Service*. Upon completing this step, the relevant microservice is invoked via its REST API. The final step is the *evaluation of the results*, where the data returned by the microservice are presented in a graphical, human-interpretable format. Each step for every module is briefly explained below.

Moral Framing Module

When using this module, in Step 1, the user must upload a text file containing the text to be evaluated or paste the text into a provided input field. Step 2 involves selecting the specific moral foundation to be evaluated. Following this, the Moral Framing microservice is invoked through its `/metrics` API, and in Step 3, the resulting bias and intensity metrics are presented graphically in a graph.

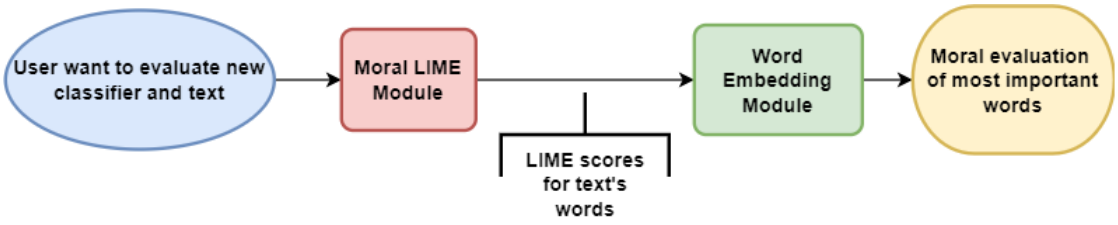

Figure 4.5: Moral LIME Module's working flow diagram

Moral LIME Module

In this case, step 1 involves uploading the serialized text classifier and its labels (possible outputs) in a Comma-separated values (CSV) file, which is necessary for explaining and morally evaluating the text prediction. Then, in step 2, the user uploads the text to be evaluated in a text file. This text is assessed using LIME and evaluated morally. At this stage, the *Moral LIME Service* is invoked via its `/lime` route. Once the most important words are identified, they are sent to the *Moral Framing Service* by calling its `/metrics_for_words` route. In step 3, the most important words are displayed in a word cloud, where their size indicates importance (the larger the word, the more important it is), and the color represents moral bias (the redder the word, the more biased it is). Additionally, the user is provided with extra information about the moral bias associated with the prediction, allowing for a more comprehensive evaluation.

This module is more complex than the others, integrating two microservices. As shown in Fig. 4.5, the underlying algorithm is our proposed approach by combining two existing techniques from the literature. These are implemented in separate microservices: LIME and Moral Framing. The primary goal of this module is to identify the most relevant words the classifier considers for its predictions on a given text and then evaluate these words to determine if they exhibit moral bias related to the vice or virtue of any of the five foundations of the MFT.

DBias Module

Step 1 consists of uploading the text to be evaluated. In Step 2, the user selects which operations from the DBias methodology (bias classification, text debiasing, or entity recognition) should be applied to the text. Finally, in Step 3, the results are displayed based on the selected operation, following an HTTP call to the DBias service through

its `evaluate_bias` route.

Word Embedding Module

Lastly, for this module, Step 1 involves updating the serialized word embedding file, sent directly to the Word Embedding Service's host and stored in a session via the `set_embedding` API method. In Step 2, the target bias to be evaluated is selected. Step 3 is divided, in turn, into several sections, each addressing a specific point of the word embedding debiasing methodology. First, the results of the exploratory analysis are displayed graphically, obtained from the `get_description` route. Next, the direct and indirect bias quantities are shown after calling the `detect_bias` route. Subsequently, the `mitigate_bias` route is invoked, and once the debiasing is complete, the Web page indicates whether it was successfully executed. Finally, a HTTP GET request is made to the `test_embedding` route to perform all available tests. The results are then graphically presented on the page, allowing the user to assess whether bias has been adequately mitigated and if the word embedding has maintained its performance.

CHAPTER 5

Evaluation and Case Study

In this chapter, some previously discussed techniques are evaluated. Then, we will present a selected use case to demonstrate how our system operates.

5.1 Evaluation

Firstly, we evaluate the performance of some of the microservices to ensure their effectiveness. This evaluation includes the Moral LIME Module and the Word Embedding Module. However, considering the DBias and moral framing modules is unnecessary since they implement previously published methodologies that the original authors thoroughly tested in their papers.

5.1.1 Moral LIME Module

As discussed in Sect. 4.6, this module aims to evaluate whether a classifier's prediction is morally biased. To achieve this, the Moral LIME Service is used to identify the words the classifier gives more importance to predict. These words are then passed as input to the Moral Framing Service to determine if they exhibit any bias toward specific moral foundations from MFT.

To evaluate this module, we train a MultinomialNB [59] text classifier on a film review corpus [54], which classifies emotions into the following categories: sadness (0), joy (1), love (2), anger (3), fear (4) and surprise (5). In Fig. 5.1, a preview of this corpus is shown, where the "text" field is the corresponding review and "label" refers to the assigned emotion, which is the target column. To introduce a moral bias, we intentionally overfit the model by associating the word "abuse" with the emotion of "joy". We chose the word "abuse" because it is strongly associated with the *vice* dimension of the *fairness* moral foundation, as categorized in the MFD from the MFT framework. To achieve this, we identify all instances in the corpus that contain the word "abuse" and change their label to "joy", regardless of the original one. This word appears in 1.540 samples out of the total of 416.809 samples the corpus has, sufficient to overfit the model. The expected outcome is that the model will assign significant importance to this word when it appears in any text, resulting in a biased tendency to classify such texts as "joy" despite the remaining content of the text. Consequently, we also anticipate that, using the Moral LIME Service, LIME will assign a high importance score to this word when evaluating any sample containing it in the overfitted model. This module then passes the most important words to the Moral Framing Service (which must include "abuse" since it is expected to be the most important one). In this last step, we hope this microservice will identify a polarized fairness bias versus the term"abuse".

text	label
i just feel really helpless and heavy hearted	4
ive enjoyed being able to slouch about relax and unwind and frankly needed it after those last few weeks around the end of uni and the expo i have lately started to find myself feeling a bit listless which is never really a good thing	0
i gave up my internship with the dmrg and am feeling distraught	4
i dont know i feel so lost	0
i am a kindergarten teacher and i am thoroughly weary of my job after having taken the university entrance exam i suffered from anxiety for weeks as i did not want to carry on with my work studies were the only alternative	4
i was beginning to feel quite disheartened	0

Figure 5.1: Visualization of the film review corpus' structure

The text sample we are going to evaluate is randomly created through generative AI [56] and must contain the term "abuse". This review sample can be read in the following section.

> "*In this film, the director tackles difficult and sensitive themes, including the abuse of individuals, which is portrayed with a stark and realistic approach. The storyline is engaging, though at times it may feel heavy-handed. The performances are strong and capture the emotional weight of the characters' experiences. However, some viewers might see the portrayal of abuse as excessive, potentially overshadowing other aspects of the narrative. Overall, it's a thought-provoking film that addresses serious issues, though its intensity may not appeal to everyone.*"

When classifying this sample with a non-overfitted classifier (trained on the original film review corpus), the label assigned is sadness (0). However, when using the overfit classifier with the same text, the output changes to joy (1) due to the moral fairness bias that the classifier owns regarding the term "abuse". The prediction result is shown in Fig. 5.2. The result of the LIME evaluation confirms that the word "abuse" had a significant influence on the prediction of the emotion of "joy", as expected. This is shown graphically in Fig. 5.3. This result is also illustrated in a list of tuples in Fig. 5.4, with each tuple containing the evaluated word and its LIME score; this is the way that the LIME's microservice returns the ten most important words to the Moral LIME module. In both images, the apparent substantial influence of the word "abuse" with a LIME score of 0.40 shows that the experiment agrees with our expectations.

review	emotion
In this film, the director tackles difficult and sensitive themes, including the abuse of individuals, which is portrayed with a stark and realistic approach.The storyline is engaging, though at times it may feel heavy-handed. The performances are strong, capturing the emotional weight of the characters' experiences. However, the depiction of abuse could be seen as excessive by some viewers, potentially overshadowing other aspects of the narrative. Overall, it's a thought-provoking film that addresses serious issues, though its intensity may not appeal to everyone	joy

Figure 5.2: Visualization of the biased classifier's text prediction

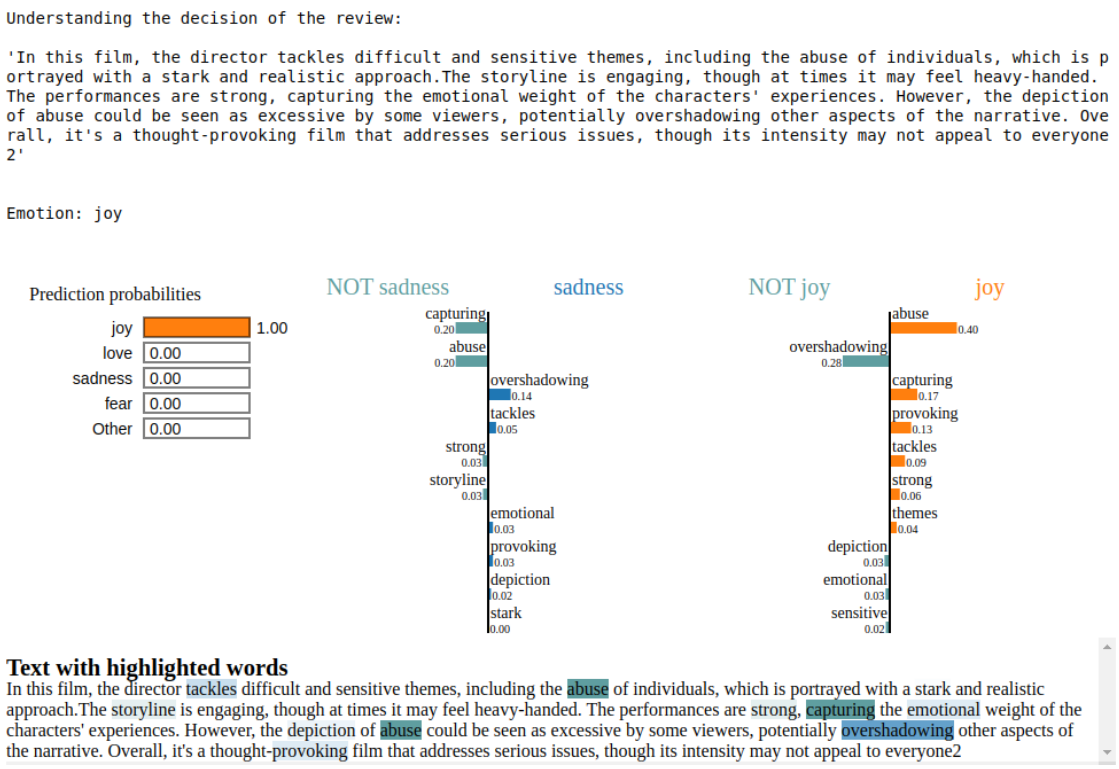

Figure 5.3: Graphic result of applying LIME to the biased classifier's text prediction

```
[('abuse', 0.4033271701428996),
 ('overshadowing', -0.28292738927171),
 ('capturing', 0.16544305776002147),
 ('provoking', 0.1284148191458272),
 ('tackles', 0.09133471116953278),
 ('strong', 0.060241777052368654),
 ('themes', 0.04159657825254593),
 ('depiction', -0.031755109433974915),
 ('emotional', -0.025461504063034542),
 ('sensitive', -0.02385710346746271)]
```

Figure 5.4: List format result of applying LIME to the biased classifier's text prediction

Now, we use the Moral Framing Service to identify the bias of the words displayed in Fig. 5.4. For each word, this microservice returns the evaluation of bias across every moral foundation in a JSON format. We focus on the term "abuse", whose moral evaluation (returned by the microservice) is shown in Fig. 5.5, which indicates that it has a significant moral impact on the vice of fairness, with a value of "-0.31" on a scale ranging from "-1" to "1". The absolute value of "0.31" demonstrates a significant moral polarization, while the negative sign indicates that it leans towards vice. In summary, the combination of the LIME and Moral Framing microservices reveals the model has a moral bias toward the vice of fairness. This corresponds to the initial conditions deliberately induced when overfitting the model and the expected results, so we can conclude that this module works adequately.


```
"abuse":{
   "authority":{
      "bias":-0.21017366647720337,
      "intensity":0.0
   },
   "fairness":{
      "bias":-0.3144649565219879,
      "intensity":0.0
   },
   "general_morality":{
      "bias":-0.3173723518848419,
      "intensity":0.0
   },
   "harm":{
      "bias":-0.10584183037281036,
      "intensity":0.0
   },
   "ingroup":{
      "bias":-0.14823535084724426,
      "intensity":0.0
   },
   "purity":{
      "bias":-0.29141563177108765,
      "intensity":0.0
   }
},
```


Figure 5.5: Bias and Intensity scores for the term "abuse" in every moral foundation

5.1.2 Word Embedding Module

This section presents the example of debiasing a word embedding using the methodology presented in Sect. 4.5. The selected embedding is a small, light version of word2vec [47], and the target bias is gender. Performance and WEAT tests are calculated before and after the embedding debiasing process.

Performance metrics

On the one hand, we compare the performance metrics calculated before and after applying the debiasing technique. Table 5.1 shows the performance metrics before debiasing, while Table 5.2 shows the metrics after the bias mitigation process. These metrics include *Pearson* and *Spearman* correlation coefficients and the ratio of unknown words in each benchmark. It is convenient to remember that a high correlation coefficient, with values closer to "1", indicates a strong positive correlation between embedding similarity scores and human judgments, indicating better performance.

Before the debiasing process, the embedding shows strong performance across most datasets, with *Pearson* and *Spearman* correlations generally above "0.6" for datasets such as WS353 or RW, among others. These high correlations suggest that the embedding captures semantic relationships effectively. The ratio of unknown words, representing the percentage of words in each dataset that do not appear in the embedding's vocabulary, is important because a high proportion of unknown words could negatively affect the performance metrics. In particular, the ratio of unknown words varies across datasets, exceptionally high in RW and TR9856, due to these datasets containing more rare or specialized vocabulary. After the debiasing process, the embedding performance of the word remains unchanged. The *Pearson* and *Spearman* correlation coefficients show only minor variations compared to the pre-debiasing metrics, indicating that the debiasing process did not significantly impact the embedding's ability to capture semantic similarities. The *p-values* remain at "0.000" for most benchmarks, which means that the observed effect is unlikely to be due to chance. In other words, the result is considered reliable and suggests a real relationship, not just a random occurrence. The ratio of unknown words is identical before and after debiasing, as expected, since the debiasing process must not alter the vocabulary of the embedding. Overall, the stability in performance metrics before and after debiasing suggests that the technique effectively reduced bias without compromising the embedding's semantic understanding (i.e., the

performance).

Dataset	**Pearson r**	**Pearson p-value**	**Spearman r**	**Spearman p-value**	**Ratio of Unknown Words**
WS353	0.645	0.000	0.688	0.000	9.915
RG65	0.576	0.232	0.493	0.321	14.286
RW	0.611	0.000	0.655	0.000	77.384
Mturk	0.650	0.000	0.674	0.000	1.558
MEN	0.766	0.000	0.782	0.000	15.148
SimLex999	0.456	0.000	0.444	0.000	1.702
TR9856	0.666	0.000	0.676	0.000	89.722

Table 5.1: Word embedding's performance before debiasing

WEAT tests

On the other hand, we compare the WEAT test results before and after applying the bias mitigation technique. As a reminder, these tests measure the bias in word embeddings by comparing the association between target words and attribute words. Specifically, WEAT computes a test statistic s, an effect size d, and a p-value p to determine whether there is a statistically significant bias. The size of the effect d quantifies the magnitude of the bias, with higher absolute values indicating a more substantial bias. At the same time, the p-value p assesses the significance of the observed bias. In this case, since we are targeting gender bias, we decide to use as target words the dyad of "Science vs. Art" and as attribute words the association of "Male vs. Female".

Before applying the debiasing technique, the WEAT results return, as shown in Table 5.3, the notable bias between the target and attribute words. The effect size d is 1.3731, indicating a strong bias favoring the association of male terms with science and female terms with the arts. The p-value p is 0.0087, suggesting that this bias is

Dataset	**Pearson** r	**Pearson p-value**	**Spearman** r	**Spearman p-value**	**Ratio of Unknown Words**
WS353	0.643	0.000	0.685	0.000	9.915
RG65	0.574	0.234	0.493	0.321	14.286
RW	0.611	0.000	0.655	0.000	77.384
Mturk	0.651	0.000	0.675	0.000	1.558
MEN	0.766	0.000	0.782	0.000	15.148
SimLex999	0.459	0.000	0.447	0.000	1.702
TR9856	0.665	0.000	0.674	0.000	89.722

Table 5.2: Word embedding's performance after debiasing

Target Words	**Attribute Words**	**Nt**	**Na**	**s**	**d**	**p**
Science vs. Art	Male terms vs. Female terms	6x2	8x2	0.3035	1.3731	8.7e-03

Table 5.3: WEAT results before debiasing

statistically significant and unlikely to occur by chance. In contrast, after applying the debiasing technique, the WEAT results indicate a substantial reduction in bias, as shown in Table 5.3. The effect size d decreases significantly to "-0.1015", suggesting that the bias has been almost entirely mitigated. The test statistic s also decreases to "-0.0074", and the p-value p increases to 0.56, indicating that the observed association is no longer statistically significant, having a much higher probability of having occurred by chance. This outcome suggests that the debiasing process effectively neutralizes gender bias in the word embedding.

Target Words	Attribute Words	Nt	Na	s	d	p
Science vs. Arts	Male terms vs. Female terms	6x2	8x2	-0.0074	-0.1015	5.6e-01

Table 5.4: WEAT results after debiasing

5.2 Use Case: Auditing a text classifier

Let's examine a use case in which a ML specialist needs to audit a classifier they previously developed. They have received reports indicating that the classifier shows biased behavior with certain texts. In this case, the specialist, acting as an auditor, must choose the appropriate tool for the analysis from the system. Since the evaluation focuses on certain classifier's local predictions, the most suitable module is the "Moral LIME" module. Below, we outline the steps the specialist should follow when auditing. As detailed in Sect. 4.6, each module is divided into several steps, corresponding to a subpage in the web application. This presentation illustrates how the specialist navigates these steps. To easily follow the use case, let's suppose the auditor is auditing the same classifier and text sample used in the evaluation (see Sect. 5.1.1).

Step 1: Set model and labels

After starting the module web page, the first subpage displayed is associated with Step 1. In this subpage, shown in Fig. 5.6, the auditor must upload the compressed (binarily serialized) classifier (i.e., the model) and all the possible outputs (labels) that it can produce in a CSV file. After uploading, the content of the labels CSV is shown, as illustrated in Fig. 5.7. Finally, the auditor clicks the "Next Step" button identified in Fig. 5.6 and proceeds to Step 2.

Step 2: Upload text

In this step, similar to what we did in Step 1, the auditor must upload a text file containing the text whose classifier's prediction wants to be evaluated. The auditor clicks then on "Next step" to advance to Step 3, as shown in Fig. 5.8. Remarkably, there is also a button denoted "Previous Step", which allows you to return to Step 1 if necessary. When the "Next Step" button is clicked, the web application sequentially

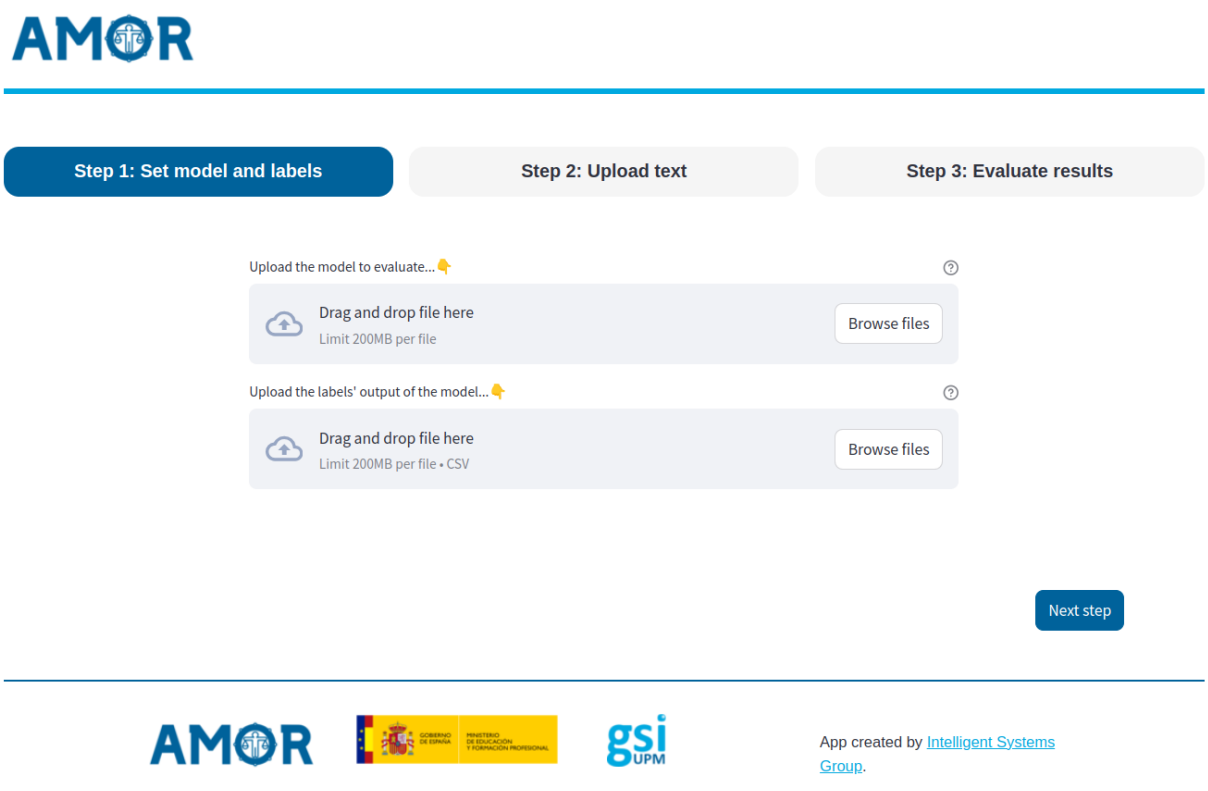

Figure 5.6: Step 1's web page before uploading the requested files

interacts with the Moral LIME Service and the Moral Framing Service in the background.

Step 3: Evaluate results

Finally, Step 3 refers to the display and evaluation of results by the auditor interacting with the web page. The last page shows several points. Firstly, a word cloud with the most relevant words is displayed (see Fig. 5.9), with their size indicating the relevance of the text according to LIME scores and the approximation to the color red indicating a higher moral bias. By this approximation, the auditor can quickly identify that the word "abuse" is of great importance to the classifier's text prediction because of its size and a high moral bias due to its red color. Next, a detailed analysis of the most essential detected words is presented, one by one, which can be seen in Fig. 5.10. This table presents categorical values instead of numerical ones to facilitate the auditor's understanding of the results. Thanks to this table, the auditor can corroborate the conclusions taken from the word cloud. Specifically, the word "abuse" is identified as

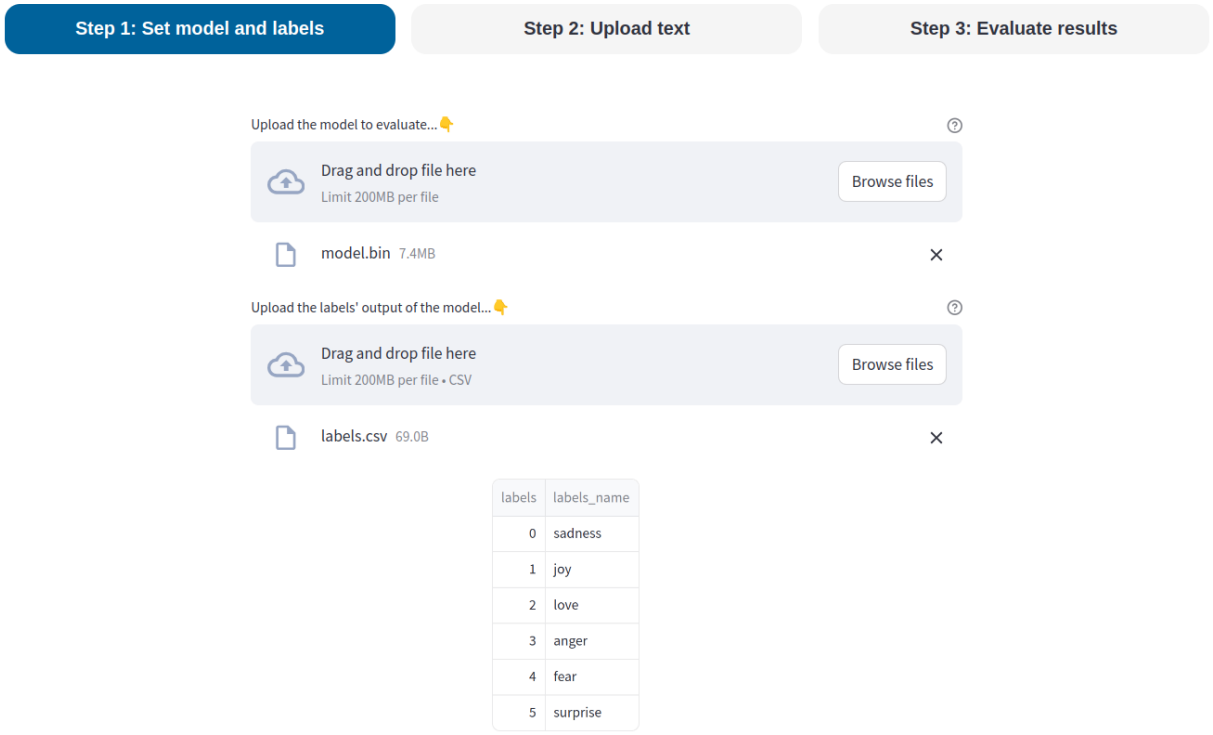

Figure 5.7: Step 1's web page after uploading the requested files

the most relevant term in the prediction, and it exhibits a moderate bias related to the moral foundation of *fairness*, which is appropriate compared to the low bias associated with most other words. Lastly, a general moral analysis is presented. In this case, we can identify the most relevant moral foundation, being *fairness*, and its bias and intensity remarked in both numerical and categorical form. A table displaying the bias and intensity measurements for each moral foundation is also provided, allowing us to identify the influence of each one on the prediction. We complete the process after carefully analyzing the results and drawing our audit conclusions.

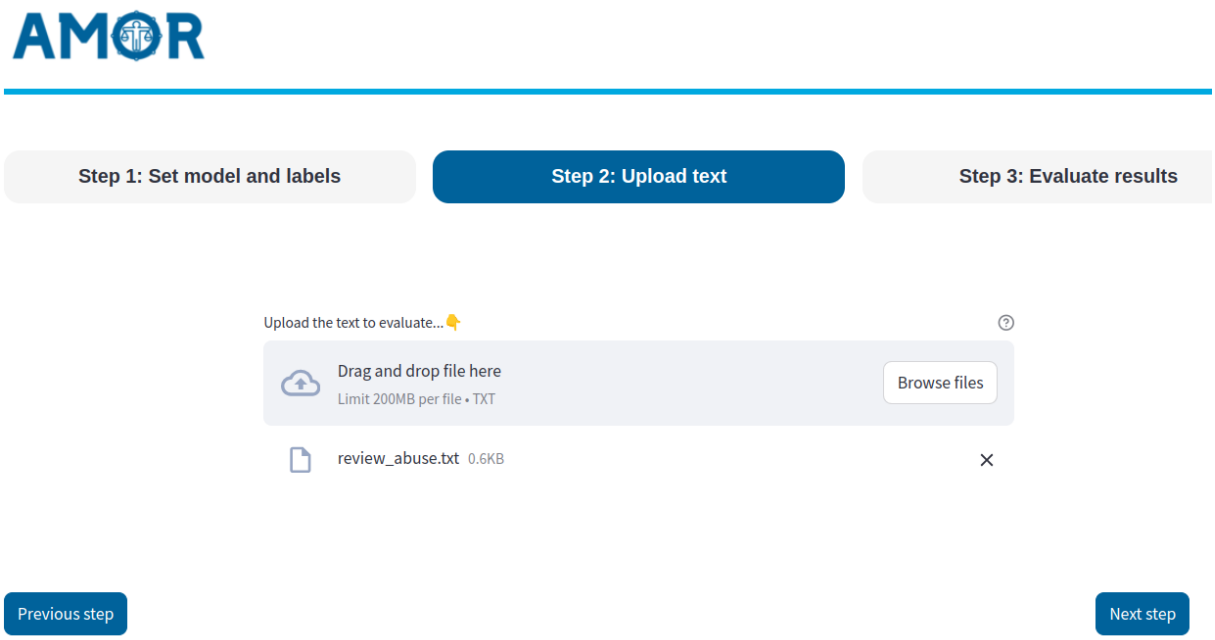

Figure 5.8: Step 2's web page after uploading the text file

Figure 5.9: Step 3's word cloud displayed on the web page

word	relevance on text	bias	moral foundation	polarization
abuse	medium ➖	moderate 🟠	fairness	vice 💸
appeal	low ⬇️	mild 🟡	fairness	vice 💸
aspects	low ⬇️	mild 🟡	authority	virtue 🕊️
capturing	low ⬇️	mild 🟡	fairness	virtue 🕊️
director	low ⬇️	mild 🟡	ingroup	virtue 🕊️
excessive	low ⬇️	moderate 🟠	purity	vice 💸
film	low ⬇️	mild 🟡	authority	vice 💸
performances	low ⬇️	mild 🟡	harm	vice 💸
realistic	low ⬇️	mild 🟡	fairness	virtue 🕊️
storyline	low ⬇️	mild 🟡	ingroup	vice 💸
tackles	low ⬇️	mild 🟡	authority	vice 💸
weight	low ⬇️	mild 🟡	authority	virtue 🕊️

Figure 5.10: Step 3's word's detailed analysis displayed on the web page

CHAPTER 6

Conclusions

This chapter describes the achieved goals done by the master thesis following some of the key points developed in the project.

6.1 Conclusion

To conclude this thesis, we review the objectives established at the beginning. The first objective (*O1*) consisted of analyzing existing tools, methodologies, and other resources to detect bias. This objective has been achieved, as demonstrated in the comprehensive dissertation presented in Chap. 2. The next objective (*O2*) involved researching a way to categorize moral values: after reading the literature, we decided to go ahead with the MFT. The third objective (*O3)* is to apply state-of-the-art techniques to a particular case of bias related to moral values; we fulfill it by creating specific implementations of these techniques for this particular case and developing individual microservices for each. The fourth objective (*O4*) was to design a theoretical methodology for systematic auditing NLP-based text classifiers. In this case, we have concluded that bias is a complex concept whose presence and detection depend on the specific context. Therefore, it is up to the auditor to decide which tools and steps to use to audit the target system, as generalizing the auditing process is not as easy as we initially thought. Consequently, the audit methodology depends on the auditor's criteria, and we propose several tools to help them. Finally, the last objective (*O5*) involves evaluating the scope of a practical application to automate the audit process. We addressed this by implementing a web app that allows the auditor to choose to use the tools they consider, where some complex and tedious tasks are abstracted. The results are presented in graphic form to make their interpretation more straightforward. The established objectives have been achieved successfully, so the work has been appropriate.

Regarding the thesis's final result, we have developed a web system based on a microservice architecture. Through the Web application, users can easily access and utilize the modules they need (see Sect. 4.6). Each module leverages one or more microservices (see Chap. 4) to present results. Using a microservice-based architecture ensures independence and decoupling between components, allowing us to quickly scale the web app with new modules in the future and modify existing ones by integrating new microservices. This is crucial given the rapid evolution of the NLP research field in recent years. As a result, this app serves as an initial prototype of something that can evolve into a more complex and precise functional system. Furthermore, it has been concluded that categorizing and generalizing bias, especially in the area of NLP and moral values, is inherently challenging due to the subjectivity and underlying human

complexity involved. This implies that the responsibility for detecting bias ultimately rests with the auditor, who must interpret the results and determine whether bias exists. The auditor must also decide on the appropriate steps to mitigate any identified bias.

In a more general sense and on a personal note, this work highlights the complexity of creating fair and ethical ML-based systems, particularly those based on NLP. Considering technical aspects (such as model overfitting or unbalanced corpus) and psychological and social factors influencing the language is essential. Regarding morality, this thesis demonstrates how our beliefs and moral frames are inherently reflected in how we express our ideas, which can sometimes be intentionally used to persuade others emotionally. This must be a significant consideration for society, and the emergence of NLP-based intelligent systems amplifies this importance. It necessitates carefully evaluating how much of this subjectivity should be transferred to these systems, especially when they are to be used in accountable contexts. Undoubtedly, this is a complex line of investigation with a long road ahead.

6.2 Achieved Goals

The goals achieved for this project, according to the established objectives, are the following ones:

- *Thorough research on the categorization of bias.* We investigated the different types of bias, focusing on when and how they appear.

- *Analysis of state-of-the-art methodologies for detecting bias in NLP.* This involved a comprehensive review of the literature to identify the most up-to-date techniques for detecting and mitigating bias. The best techniques were then studied and selected for implementation.

- *Categorization of moral values.* After conducting a literature exploration, we concluded that MFT, along with its associated MFD, is the most precise method for categorizing and quantifying moral values.

- *Implementation of a module to quantify bias in texts.* Using state-of-the-art techniques and a web and microservice-oriented architecture, we employed the Framing Axis technique to detect the moral load of texts or corpus.

- *Implement a module to detect moral bias in classifier predictions.* Similar to the previous point, we combined the LIME interpretability technique with the Framing Axis technique to detect if a classifier's local predictions are morally biased.

- *Implementation of a module to mitigate bias in texts.* In line with the previous modules (using the web and microservice-oriented architecture), we used the DBias technique to design a module that suggests less biased variations of a given text.

- *Implementation of a module to detect and mitigate moral bias inside word embeddings.* We created a module that detects and mitigates bias in word embeddings as in previous ones. It also tests their performance and bias quantity before and after mitigation, using the Bolukbasi et al. (2016) technique and WEAT tests.

- *Design of a web application to graphically interact with the modules.* As a final achievement, we designed an interactive web application that enables auditors to interact with the modules and analyze the produced results graphically. Plots and other tools (like word clouds) enrich the experience and simplify interpretation.

6.3 Future work

The high scalability offered by the developed system raises many possible improvements or additional features that must be implemented. Then, we outline some of the most interesting objectives to accomplish that have not been implemented into this project due to time and other resource limitations. The first and more feasible feature is implementing a module to calculate different fairness metrics for a particular corpus and classifier. As discussed in Sect. 2.3.4, one of the ways to detect bias from text classifiers is the usage of metrics. We thoroughly examined the generalized fairness metrics framework [16] and performed tests using the authors' implementation code, which initially performed well. However, this code has limitations when applied to different classifiers and corpora (e.g., it was designed only for binary and three-class classifiers) and excessive complexity. As a result, we decided against implementing the module based on this code. As a solution, we lay out programming these metrics from scratch in a simplified and more general form, which we could not afford to do due to lack of time. Therefore, implementing such a module should be a priority for future work.

As a second future line, we propose training a classifier that can quantitatively classify the moral bias of a text according to the MFT. Currently, some methods and libraries present in the literature can quantify the text of each moral foundation. However, these cannot identify whether bias is polarized through the vice of virtue. We propose to evaluate these tools, the existing corpora, and manual annotation to label a corpus according to the MFT dyads and then train or fine-tune a classifier capable of quantitatively identifying the presence of each moral foundation and its polarization. The final objective is to create a new module to detect bias based on this classifier, which previously had to be deployed into a new microservice. We propose ongoing research into new methodologies as a final, more general future consideration. The effectiveness of the implemented system relies not only on precise programming but also on thorough and careful investigation. Therefore, it is crucial to keep the system updated by continuously reviewing new research and publications in the field. This is especially important given the rapid advances in the NLP domain.

APPENDIX A

Impact of this project

This appendix describes this project's social, environmental, economic, and ethical impact.

A.1 Social Impact

The use of this auditing toolkit can have a highly positive social impact, as it serves as a tool to ensure that ML-based systems and texts/corpora are fair and unbiased. Using these tools, companies developing new NLP-based systems can check whether the training corpora are morally biased or if the trained classifier is propagating such biases, potentially leading to unethical behavior. This is particularly valuable in high-accountability environments, where unfair decisions can significantly impact minorities or vulnerable groups. This aligns with Sustainable Development Goal (SDG) 10, titled "Reduce inequality within and among countries", which seeks to address the disparities related to gender, race, age, and other factors. By auditing NLP-based systems to detect and mitigate biases, we can ensure that these technologies treat all social groups fairly and equitably.

A.2 Economic Impact

The main economic costs are associated with developing and deploying the system (which is based on a web microservice-oriented architecture). This includes development costs (developer salaries, tools, and potential software licenses), infrastructure costs (servers, storage, load balancing, and network usage), and maintenance and support (code updates, monitoring, and technical support). Additionally, there are security costs (mainly to avoid technical debt, which could cause the system not to perform well), deployment and DevOps costs (CI/CD pipelines and container orchestration), scalability expenses for being able to expand the system auditing capabilities (both horizontal and vertical scaling), and other operational costs (backups, disaster recovery, and long-term cost optimization).

A.3 Environmental Impact

The environmental impact of this project lies in implementing the system, which is mainly related to energy consumption and carbon emissions from the servers and data centers used. High computational demands, especially from microservices that load large embeddings or have complex processing tasks, can increase energy usage, contributing to a larger carbon footprint. This impact is particularly relevant to SDG 13: "Climate

Action," which calls for urgent action to combat climate change. Optimizing infrastructure must be tried using energy-efficient servers, leveraging green data centers, or implementing resource scaling strategies. These actions can help mitigate these effects by reducing the system's overall environmental footprint and aligning the project with sustainable practices.

A.4 Ethical Implications

The ethical implications of deploying this auditing system are significant, especially given its role in evaluating moral values and biases in NLP-based systems. While the tool aims to promote fairness and accountability, there is a risk that it could be misused by auditors who manipulate or misunderstand the results, which can provoke certain biases to be ignored or intentionally omitted (i.e., selectively reporting findings to favor specific interests). This misuse could undermine trust and exacerbate existing inequalities rather than mitigate them. Such ethical concerns relate closely to SDG 16: "Peace, Justice, and Strong Institutions", focusing on transparency, accountability, and moral standards. To address these risks, the auditor must guarantee transparency in the auditing processes as well and the results must be presented as clearly as possible in the web app (which is chargeable to developers) to reduce the probability of a bad comprehension by the auditors.

APPENDIX B

Economic Budget

This appendix describes the economic budget associated with the development of this project.

DIRECT LABOR COSTS	**Hours**	**Price per hour**	**TOTAL**
	300	20,00 €	6.000,00 €

MATERIAL RESOURCES COSTS	**Purchase Price**	**Usage in Months**	**Amortization (in years)**	**Cost**
Personal Computer	1.200,00 €	6	5	120,00 €
GSI Computer	900,00 €	6	5	90,00 €
TOTAL	-	-	-	210,00 €

GENERAL OVERHEADS (Indirect Costs)	15%	over DIRECT COSTS
		931,50 €
INDUSTRIAL PROFIT	6%	over DIRECT + INDIRECT COSTS
		428,49 €

CONSUMABLE MATERIALS	**Cost**
Printing	100,00 €
Binding	300,00 €

SUBTOTAL BUDGET	7.969,99 €
Applicable VAT (IVA)	21%
	1.673,69 €
TOTAL BUDGET	9.643,68 €

Bibliography

[1] Julius Adebayo, Parisa Khosravi, Ian Liu, Besmira Nushi, Aditi Raghunathan, Amos Azaria, Grady Booch, Jeffrey P Bigham, Ece Kamar, and Eric Horvitz. Fairml: Toolbox for diagnosing bias in predictive modeling. In *Proceedings of the 2016 Fairness, Accountability, and Transparency in Machine Learning Workshop*, 2016.

[2] Responsibly AI. *Responsibly: Toolkit for Auditing and Mitigating Bias in AI Models*, 2023. Accessed on September 3rd, 2024.

[3] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *CoRR*, abs/1901.09069, 2019.

[4] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51, 2016.

[5] Anaconda. *Anaconda documentation: Managing environments*, 2024. Retrieved from `https://docs.anaconda.com/anaconda/user-guide/tasks/manage-environments/`.

[6] Shraddha Anala. A guide to word embedding. what are they? how are they more useful... — by shraddha anala — towards data science. `https://towardsdatascience.com/a-guide-to-word-embeddings-8a23817ab60f`, October 2020. Accessed on February 2nd, 2024.

[7] Ece Özlem Atikcan and Karen Hand. Moral framing and referendum politics: Navigating the empathy battlefield. *Political Psychology*, 45(1):193–210, 2024.

[8] Python Packaging Authority. venv — Creation of virtual environments — docs.python.org. `https://docs.python.org/3/library/venv.html`. Accessed on July 27th, 2024.

[9] David M. Beazley. *Python Cookbook*. O'Reilly Media, 3rd edition, 2013.

[10] Rachel K. E. Bellamy, Kuntal Dey, Michael Hind, Samuel C. Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, Seema Nagar, Karthikeyan Natesan Ramamurthy, John T. Richards, Diptikalyan Saha, Prasanna Sattigeri, Moninder Singh, Kush R. Varshney, and Yunfeng Zhang. AI

fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *CoRR*, abs/1810.01943, 2018.

[11] Tolga Bolukbasi, Kai Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. *Advances in Neural Information Processing Systems*, pages 4356–4364, 7 2016.

[12] Daniel Borkan, Lucas Dixon, Jeffrey Sorensen, Nithum Thain, and Lucy Vasserman. Nuanced metrics for measuring unintended bias with real data for text classification. In *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, page 491–500, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. *CoRR*, abs/1309.0238, 2013.

[14] Aylin Caliskan, Joanna J Bryson, and Arvind Narayanan. Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186, 2017.

[15] Dennis Chong and James N Druckman. Framing theory. *Annu. Rev. Polit. Sci.*, 10(1):103–126, 2007.

[16] Paula Czarnowska, Yogarshi Vyas, and Kashif Shah. Quantifying social biases in nlp: A generalization and empirical comparison of extrinsic fairness metrics, 2021.

[17] Sunipa Dev, Emily Sheng, Jieyu Zhao, Jiao Sun, Yu Hou, Mattie Sanseverino, Jiin Kim, Nanyun Peng, and Kai-Wei Chang. What do bias measures measure? *CoRR*, abs/2108.03362, 2021.

[18] DBias Developers. Dbias python library, 2021. Accessed on August 11$^{\text{th}}$, 2024.

[19] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning, 2017.

[20] Laura Douglas. Ai is not just learning our biases; it is amplifying them. — by laura douglas — medium, 2017.

[21] Carlos A. Iglesias et al. Amor: Análisis de sentimiento moral en datos textuales, 2023. Accessed on September 3$^{\text{rd}}$, 2024.

[22] Flask Documentation. Welcome to flask. `https://flask.palletsprojects.com/en/2.0.x/`. Accessed on August 11$^{\text{th}}$, 2024.

[23] Everton Gomede. Bias and fairness detection in nlp models — by everton gomede, phd — ai mind, 2023.

[24] Carlos González-Santos, Miguel A Vega-Rodríguez, Carlos J Pérez, Joaquín M López-Muñoz, and Iñaki Martínez-Sarriegui. Automatic assignment of moral foundations to movies by word embedding. *Knowledge-Based Systems*, 270:110539, 2023.

[25] Bryce Goodman and Seth Flaxman. European union regulations on algorithmic decision making and a "right to explanation". *AI Magazine*, 38(3):50–57, September 2017.

[26] Jesse Graham, Jonathan Haidt, Sena Koleva, Matt Motyl, Ravi Iyer, Sean P Wojcik, and Peter H Ditto. Moral foundations theory: The pragmatic validity of moral pluralism. In *Advances in experimental social psychology*, volume 47, pages 55–130. Elsevier, 2013.

[27] Anthony G Greenwald, Brian A Nosek, Mahzarin R Banaji, Laurie A Rudman, Shelly D Farnham, David S Mellott, and N Sriram Sriram. Best research practices for using the implicit association test. *Behavior research methods*, 54(3):1161–1180, 2022.

[28] Aurelien Gron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2019.

[29] Jonathan Haidt. *The righteous mind: Why good people are divided by politics and religion.* Vintage, 2012.

[30] Jonathan Haidt, Craig Joseph, et al. The moral mind: How five sets of innate intuitions guide the development of many culture-specific virtues, and perhaps even modules. *The innate mind*, 3:367–391, 2007.

[31] Dirk Hovy and Shrimai Prabhumoye. Five sources of bias in natural language processing. *Language and Linguistics Compass*, 15(8):e12432, 2021.

[32] Anaconda Inc. Installing conda 24.7.2.dev7 documentation — conda.io. `https://conda.io/projects/conda/en/latest/user-guide/install/index.html`. Accessed on July 27$^{\text{th}}$, 2024.

[33] Docker Inc. Docker: Accelerated container application development. `https://www.docker.com/`, 2024. Accessed on July 27$^{\text{th}}$, 2024.

[34] Ravi Iyer, Spassena Koleva, Jesse Graham, Peter Ditto, and Jonathan Haidt. Understanding libertarian morality: The psychological dispositions of self-identified libertarians. *PubMed Central*, 2012.

[35] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. Examples are not enough, learn to criticize! criticism for interpretability. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[36] Adriano Koshiyama, Emre Kazim, Philip Treleaven, Pete Rai, Lukasz Szpruch, Giles Pavey, Ghazi Ahamat, Franziska Leutner, Randy Goebel, Andrew Knight, Janet Adams, Christina Hitrova, Jeremy Barnett, Parashkev Nachev, David Barber, Tomas Chamorro-Premuzic, Konstantin Klemmer, Miro Gregorovic, Shakeel Khan, and Elizabeth Lomas. Towards algorithm auditing. *SSRN*, 2021.

[37] Kristin M Kostick-Quenet, I Glenn Cohen, Sara Gerke, Bernard Lo, James Antaki, Faezah Movahedi, Hasna Njah, Lauren Schoen, Jerry E Estep, and JS Blumenthal-Barby. Mitigating racial bias in machine learning. *Journal of Law, Medicine & Ethics*, 50(1):92–100, 2022.

[38] Kamran Kowsari, Mojtaba Heidarysafa, Donald E Brown, Kiana Jafari Meimandi, and Laura E Barnes. Text classification algorithms: A survey. *Information*, 10(4):150, 2019.

[39] Haewoon Kwak, Jisun An, Elise Jing, and Yong-Yeol Ahn. Frameaxis: characterizing microframe bias and intensity with word embedding. *PeerJ Computer Science*, 7:e644, 2021.

[40] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[41] Mark Lutz. *Learning Python*. O'Reilly Media, 5th edition, 2013.

[42] Wes McKinney. *Python for Data Analysis*. O'Reilly Media, 2017.

[43] Microsoft. Pylance - visual studio marketplace, 2024. Accessed on August 11$^{\text{th}}$, 2024.

[44] Microsoft. Python - visual studio marketplace, 2024. Accessed on August 11$^{\text{th}}$, 2024.

[45] Microsoft. Visual studio code, 2024. Accessed on August 11$^{\text{th}}$, 2024.

[46] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[47] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Word2vec. `https://code.google.com/archive/p/word2vec/`, 2013. Accessed on August 22$^{\text{th}}$, 2024.

[48] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2 2019.

[49] Negar Mokhberian, Andrés Abeliuk, Patrick Cummings, and Kristina Lerman. Moral framing and ideological bias of news. In Samin Aref, Kalina Bontcheva, Marco Braghieri, Frank Dignum, Fosca Giannotti, Francesco Grisolia, and Dino Pedreschi, editors, *Social Informatics*, pages 206–219, Cham, 2020. Springer International Publishing.

[50] Christoph Molnar. *Interpretable Machine Learning: A Guide For Making Black Box Models Explainable*. Independently published, 2 edition, 2022.

[51] Edoardo Mosca, Ferenc Szigeti, Stella Tragianni, Daniel Gallagher, and Georg Groh. SHAP-based explanation methods: A review for NLP interpretability. In Nicoletta Calzolari, Chu-Ren Huang, Hansaem Kim, James Pustejovsky, Leo Wanner, Key-Sun Choi, Pum-Mo Ryu, Hsin-Hsi Chen, Lucia Donatelli, Heng Ji, Sadao Kurohashi, Patrizia Paggio, Nianwen Xue, Seokhwan Kim, Younggyun Hahm, Zhong He, Tony Kyungil Lee, Enrico Santus, Francis Bond, and Seung-Hoon Na, editors, *Proceedings of the 29th International Conference on Computational Linguistics*, pages 4593–4603, Gyeongju, Republic of Korea, October 2022. International Committee on Computational Linguistics.

[52] Andreas C Mueller and Sarah Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, Inc., 2016.

[53] Author Name. Website title, Year Published. Accessed on Month Day, Year.

[54] Harshana Nelgiriyewithana. Emotions dataset, 2023. Accessed on August 21$^{\text{th}}$, 2024.

[55] NumPy Developers. *NumPy Documentation*, 2024. Version 2.0.

[56] OpenAI. Chatgpt, 2024. Accessed on: 2024-08-21.

[57] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *arXiv*, 2022.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python, 2011. Accessed on August 10$^{\text{th}}$, 2024.

[59] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. *scikit-learn: Machine Learning in Python*. scikit-learn developers, 2024.

[60] Pallets Projects. Welcome to flask — flask documentation (3.0.x). `https://flask.palletsprojects.com/en/3.0.x/`, 2023. Accessed on July 27$^{\text{th}}$, 2024.

[61] Flavien Prost, Nithum Thain, and Tolga Bolukbasi. Debiasing embeddings for reduced gender bias in text classification, 2019.

[62] PyTorch Developers. *PyTorch Documentation*, 2024. Latest version.

[63] Shaina Raza, Deepak John Reji, and Chen Ding. Dbias: Detecting biases and ensuring fairness in news articles, 2022.

[64] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.

[65] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *LIME: Local Interpretable Model-agnostic Explanations*, 2023. Accessed on September 3rd, 2024.

[66] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[67] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys (CSUR)*, 34(1):1–47, 2002.

[68] John Smith, Laura Brown, and Thomas Wang. *Practical Data Science with Conda*. O'Reilly Media, 2021.

[69] Streamlit Inc. Simple streamlit app, 2023. Examples.

[70] Streamlit Inc. *Streamlit: The fastest way to build and share data apps*, 2023. Documentation.

[71] Tony Sun, Andrew Gaut, Shirlyn Tang, Yuxin Huang, Mai ElSherief, Jieyu Zhao, Diba Mirza, Elizabeth Belding, Kai-Wei Chang, and William Yang Wang. Mitigating gender bias in natural language processing: Literature review. In Anna Korhonen, David Traum, and Lluis Marquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1630–1640, Florence, Italy, July 2019. Association for Computational Linguistics.

[72] Harini Suresh and John Guttag. A framework for understanding sources of harm throughout the machine learning life cycle. *CoRR*, 2021.

[73] TensorFlow Developers. *TensorFlow Documentation*, 2024. Latest version.

[74] Guido van Rossum and Fred L. Drake Jr. *Python Reference Manual*, 2001.

[75] Jieyu Zhao, Yichao Zhou, Zhao Li, Wei Wang, and Kai-Wei Chang. Learning gender-neutral word embeddings. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4847–4853. Association for Computational Linguistics, 2018.