DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

UNIVERSIDADE DE AVEIRO

PROJETO EM INFORMÁTICA 2023

DETEÇÃO DE TEXTO SINTÉTICO / AI TEXT DETECTOR

Alexandre Lima Gazur (102751)

Daniel Ferreira (102885)

Ricardo Pinto (103078)

João Matos (103182)

Abstract

The rapid advancement of powerful artificial intelligence (AI) technology has brought forth various challenges, including the emergence of AI-generated text that often blurs the line between human-authored and machine-generated content. In this project, we propose a browser extension aimed at detecting AI-generated text in web pages and PDFs. The extension integrates with Model Hub, a web-based platform, allowing users to submit their AI text detection models and use them within the extension. This project contributes to AI transparency and enables informed decision-making in an era of increasing AI-generated information.

Keywords: AI Text Detection, ChatGPT, OpenAI, Web extension, Web application, RESTful API, RESTful API, Artificial Intelligence, Machine Learning, Natural Language Processing, Language Models

Acknowledgements

We would like to thank Sérgio Matos and Tiago Almeida, our project mentors, for their guidance, expertise, and support. Their insights and advice have been instrumental in shaping the direction of our work and ensuring its successful execution. We are grateful for their dedication, patience, and willingness to share their knowledge and experience.

We would also like to acknowledge Professor Rui Aguiar and Professor Osvaldo Rocha Pacheco, for providing us with the opportunity to work on this project and for their valuable feedback and guidance throughout the course.

Special thanks go to Tiago Almeida for his exceptional commitment and assistance. His willingness to share his expertise in the area of language models and generously providing some of his resources proved invaluable. His mentorship went beyond expectations, and we are truly grateful for his time and dedication.

This project would not have been possible without the generous assistance and encouragement of these individuals. We are truly grateful for their contributions, which have enriched our project and our learning journey.

Table of Contents

Table of Figures

Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
NLP	Natural Language Processing
LM	Language Models
UI	User Interface
UX	User Experience
IEETA	Institute of Electronics and Informatics Engineering of Aveiro
SUS	System Usability Scale

Chapter 1

Introduction

1.1 Context

The widespread availability of AI language models, such as OpenAI's GPT-3, has democratised the creation of text that closely resembles human-generated content. As a result, distinguishing between AI-generated and human-authored text has become increasingly complex. While AI-generated text offers benefits in various applications, including content creation and language translation, it also raises concerns about the potential for misinformation, propaganda, and manipulated narratives.

1.2 Motivation

By offering users a tool to discern AI-generated text, our objective is to empower individuals to navigate the web and consume content with greater ease. The capacity to quickly and accurately differentiate between human and AI-generated text plays a vital role in combating misinformation and ensuring the reliability of information sources. Additionally, as the discomfort of browsing the web or reading content, such as PDFs, continues to grow, individuals find themselves frequently questioning whether the text was written by a human or an AI. We aim to develop a tool that eliminates the need for this constant self-inquiry, thereby enhancing the overall reading experience.

1.3 Goals

The primary goal of our project is to develop a browser extension capable of detecting AI-generated text in web pages and PDF documents. The extension will provide users with a straightforward and reliable tool to identify and distinguish between human-authored and AI-generated content. The extension will offer various scanning options, allowing users to scan an entire page, a specific section, or even arbitrary text, and conveniently display the scan results. The UI will be designed to be comprehensive and intuitive, providing customisation options to cater to individual preferences.

Additionally, our project aims to embrace the collaborative nature of AI development. By allowing users to submit their own AI text detection models through our Model Hub, we ensure that the system maintains itself, adapts to the evolving landscape of NLP techniques and machine learning algorithms, and remains helpful and effective even as new models and approaches emerge.

Through these efforts, we aim to contribute to the promotion of transparency, trust, and critical thinking in the digital era, while also facilitating continuous improvement and innovation in the detection of AI-generated text.

Chapter 2

State of the Art

2.1 Related Projects

The objective of this section is to showcase additional projects or applications that relate to our project.

2.1.1 GPT-Zero

GPT Zero was the first tool designed to detect AI-generated content, specifically focusing on responses generated by models like ChatGPT. It utilises statistical features to analyse the complexity of the text and determine whether it was produced by an AI.

However, GPT Zero does have some limitations. Firstly, the tool currently requires the user to either paste the text or upload a document for analysis. Additionally, it does not provide a confidence percentage for its assessments, only indicating whether the text is likely or unlikely to be AI-generated. Furthermore, there is a minimum character requirement of 250 and a maximum limit of 5000 characters in the free version, which may impact the detection of shorter or longer AI-generated texts.

2.1.2 Originality.AI

Originality.AI is an AI Content Detector and Plagiarism Checker designed for serious content publishers. According to a case study conducted by the company itself, it is the most accurate AI Content Detection tool currently available on the market. In our limited tests, we found that the tool did not perform particularly well, but we did not conduct an extensive and proper analysis.

One significant drawback of Originality.AI is that it operates as a credit-based system, where users need to purchase credits to scan documents. While the platform does offer some credits as part of a free trial, this is not enough for regular use. Moreover, Originality.AI does not currently have a free option, which may be a deterrent for some users.

2.1.3 OpenAI’s AI Text Classifier

OpenAI's AI Text Classifier is another machine learning model designed to distinguish between text written by humans and that generated by AI, utilising a GPT model that has been finely tuned for the task. This classifier can analyse text from diverse sources, including ChatGPT, and provide a likelihood score indicating the degree to which the text is likely to have been generated by AI.

However, it is important to note that OpenAI itself has acknowledged that the AI Text Classifier is not entirely reliable and has a lower performance rate than other similar models such as GPT Zero. The classifier's detection rate is less than optimal, and it is not recommended to be solely relied on as a primary decision-making tool. Instead, it can be used as a supplementary method for determining the source of a given text, in conjunction with other more robust techniques.

2.2 Technology

The field of detecting AI-generated content relies on various technologies and techniques to analyse and identify text produced by AI models. In this project, external users are the ones responsible for developing their models, and our role primarily revolves around integrating these models into our system. However, it is still important for us to have a surface-level understanding of how these models and detection techniques work.

AI detection relies on reverse engineering language patterns to determine predictive text. This means that the machine breaks down a piece of text and then uses algorithms to detect patterns within those words. If a pattern is easier to identify, it's more in tune with what an AI would write, increasing the odds it was written by AI.

GPT Zero, for example, was trained on similar datasets as ChatGPT. It assesses the complexity of a text by measuring its “perplexity”. If a text is familiar to the GPT Zero bot, it is more likely to be AI-generated. It also checks for “burstiness”, which means it looks for differences in sentence length and other factors.

Most of the applications that emerged to detect AI-generated text were developed as web applications. However, these platforms have been recently investing in the development of web extensions, recognising the dynamic and enhanced utility they provide. For instance, as of the time of writing this report, GPT Zero has launched their own web extension called Origin.

Both the extension from ORGINLATITY.AI and GPT Zero's Origin have similar basic functionality, featuring a text box where users can paste the text and scan it for AI-generated content. While their current offerings are limited in terms of features, these extensions prove to be highly practical as they eliminate the need to navigate to the respective web platforms to scan the text within a webpage. These web extensions are still in the early stages of development and have room for improvement. Recognising this, ORIGINALITY.AI, for instance,

has announced that they are actively working on introducing a full-page analysis feature to their extension. However, as of now, this advanced functionality is not yet available.

Ultimately, our project aims to achieve precisely this, while also incorporating the added complexity of establishing a centralised hub of models, that enhances the overall flexibility of our approach.

2.3 Conclusions

The rise of generative AI tools is creating a parallel demand for a new class of systems that can help distinguish AI-generated text from those created by humans. However, most of the existing AI detector tools are not better than random classifiers and do not offer a very robust or long-term solution.

One major issue is the limited understanding of how the tools work, which raises questions about their reliability. Some tools generate seemingly random results, while the lack of free plans and missing features further restricts their accessibility and effectiveness.

The complexity of AI-generated content poses a significant challenge in detecting it, and further research and development in AI detection technology are necessary to address this issue. That legitimises our project as a platform where users can submit their models, fostering open-source collaboration and facilitating the study of new and improved methods for detecting AI content. By promoting knowledge sharing and collaborative development, we hope to contribute to the evolution of AI detection technology, advancing the state of the art in this field.

The decision to develop a web extension for detecting AI-generated text also appears to be the optimal choice, considering that many competitors have adopted a similar approach. However, we take it a step further by integrating our extension with our model hub platform, enhancing the flexibility and functionality of our solution to a unique level.

Chapter 3

System Requirements and Architecture

3.1 System Requirements

The following subsections begin with a description of the requirements elicitation process, followed by a context description, actors classification, use-case diagrams, non-functional requirements and the system’s assumptions and dependencies.

3.1.1 Requirements Elicitation

The specification process for this project was divided into three main parts. The first part involved evaluating the system's main goals and defining its scope, including the system boundaries. The next part was an investigation of the state-of-the-art, looking for projects or established applications similar to the one being developed. This research improved the system requirements by discovering the latest and most innovative works in the field and by understanding the strengths and weaknesses of those works.

At last, we engaged in extensive reunions with our project mentors that served as valuable opportunities to brainstorm and gather insights from various perspectives, including those of the end-users, external developers, and administrators. Through these lengthy reunions, we not only discussed the predefined ideas outlined in the project proposal but also generated new and innovative ideas that had the potential to enhance the project's outcomes.

It is worth noting that the requirements underwent significant changes during the first weeks of the project. Initially, our plan was centred around building our AI detection model. However, we soon realised that delving into the realm of advanced language models and NLP would require a substantial amount of time and expertise, which we lacked, so we decided to change the goal of the project. We opted for a higher level of complexity in system architecture while acknowledging the need to forgo acquiring expert-level knowledge in NLP techniques. This decision was motivated by the desire to leverage our existing software engineering skills and resources efficiently, while still incorporating elements of AI and NLP. This ensured that our project remained within a realistic scope, aligned with our team's expertise, and allowed us to achieve meaningful results within the project timeline.

3.1.2 Context Description

The system is designed to be used by different stakeholders, including end users, developers, and administrators. Each stakeholder has specific roles and interactions with the system.

For a model developer, the process begins with registering on the Model Hub web platform. Once logged in, the developer can submit their language model for consideration. The model goes through automated testing to determine acceptance or rejection. If the model is accepted, the developer is notified, and the model becomes available in our extension.

On the other hand, an end user, such as a teacher evaluating a student's body of work, interacts with the system in a different way. The user first visits the browser web store and installs the extension. Then, while working with a website or a PDF file, the application automatically or manually highlights sentences it identifies as being written by AI using the selected model. The user also can customise the application and adjust it to their preferences.

The administrator gains access to the system through the admin panel using authorised credentials. From there, they review the performance and quality metrics of the available models, removing or disabling any that don't meet standards or pose risks. Their oversight ensures a safe and reliable environment for users.

At last, the external developer explores the provided API documentation and seamlessly integrates it into their applications. They utilise the API to determine if the text is AI-generated, enhancing their apps with the capability to identify and differentiate between human and AI-generated content.

3.1.3 Actors

The extension is designed for any web user or computer user (almost everyone in the world), that may have varying levels of proficiency with technology and web extensions. To ensure that the extension is accessible and easy to use for all users, the interface design has been carefully crafted to mitigate any potential barriers due to differences in expertise.

The level of expertise required to seamlessly use the system is little. However, it is important to acknowledge that the concept of web extensions may be unfamiliar to some users, particularly older individuals or those with limited technology experience. This lack of familiarity with web extensions could pose a challenge to the seamless adoption and usage of the system.

The main actors involved are the **end users**, which include individuals like journalists, researchers, teachers, students, software developers, and social media users. We also have the AI text detection **model developers**, who own the models used in the process. Additionally, there is the **admin**, which is currently just our team. Lastly, we have **external developers**, referring to any developer from around the world.

- **User:** Represents the most common usage of our extension. Has access to most features, such as analysing web pages and PDFs (for AI-generated text), writing text in our text area or selecting text in the page and analysing it, and other use cases described in the next section.
- **Admin:** Has access to all features and the system. They can add, remove, enable and disable models at any time.
- **Model developer:** Someone who develops or owns an AI text detection model. Model developers can upload and remove their models through our Model Hub if logged in. Can also be an end user but not necessarily.
- **External developer:** Any developer in the world can use our public API endpoints, to get the probability a text is AI-generated (using one of our models), or to get details of our available models in JSON format. Can also be an end user but not necessarily.

3.1.4 Use Cases

Figure 3.1 presents the use case model of the whole system, having two main packages: the web extension package and the web application package. They represent the usage of the web extension and web application, respectively.

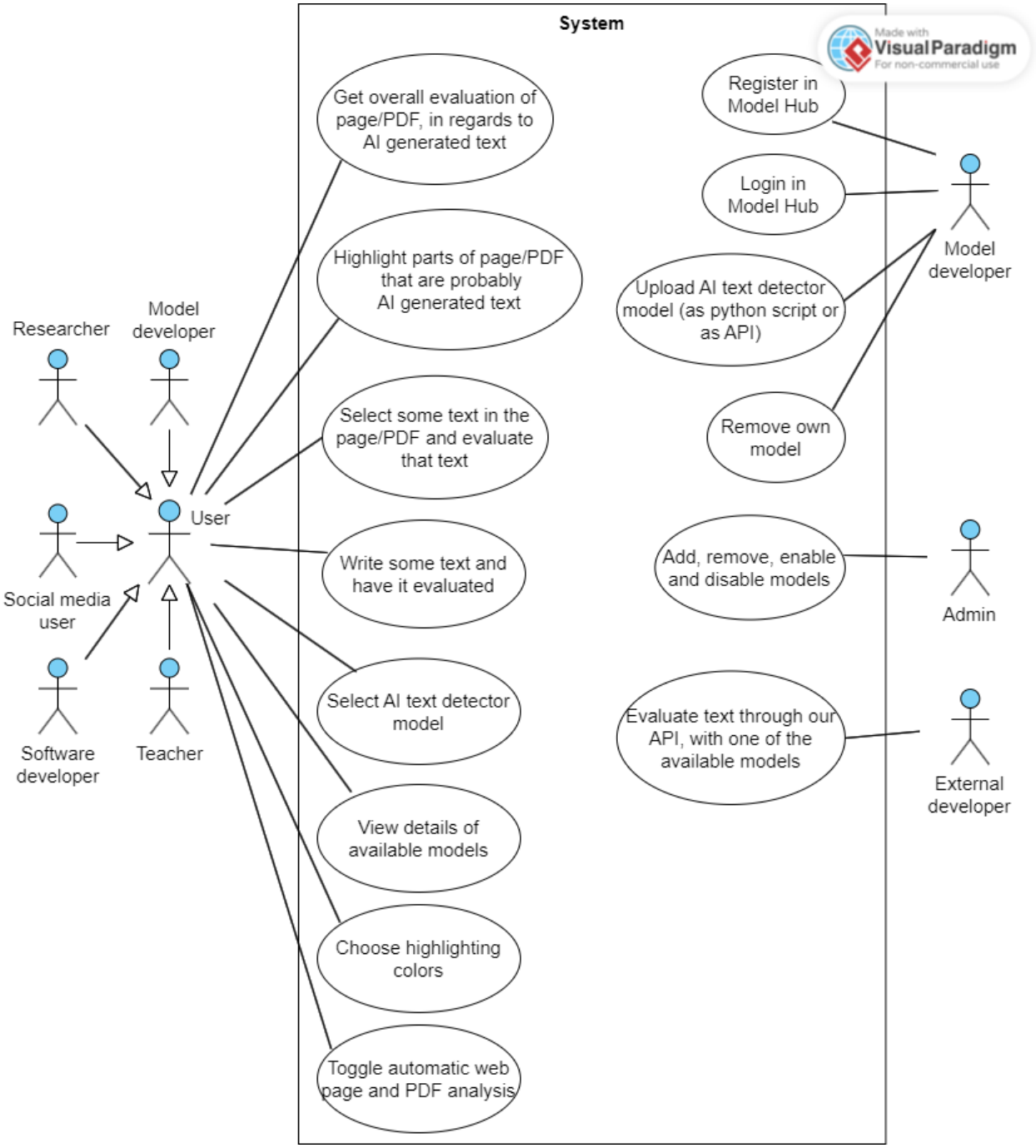


Figure 3.1: Use Case Model

- **Get overall evaluation:** When the user scans the current web page or PDF, the analysis will begin and, when it's done, the overall evaluation is shown, which is an integer from

0 to 100 (both inclusive) that represents the percentage of the page considered AI-generated text by the currently selected AI text detection model. **Priority: High**

- **Highlight parts of page/PDF:** When the user analyses the current web page, paragraphs that the currently selected model thinks are AI-generated are highlighted and underlined. There are 2 highlighting colours: 50-75% probability (usually light red, pink or yellow) and 75-100% probability (usually red). The same happens in PDFs, except analysis is done by sentences, and not by paragraphs. **Priority: High**

- **Select some text in the page/PDF and evaluate it:** The user can select any text in the web page or PDF, right click and select the option “Evaluate”. Our system will then evaluate that text, and if considered AI generated by the currently selected model, that text will become highlighted and underlined described in the use case before. If it is not considered AI-generated, it simply gets underlined in green. **Priority: Medium**

- **Write some text and have it evaluated:** In the extension UI, there is a text area where the user can write anything, and then click the button “Scan text” to evaluate. The probability that the written text is AI generated is displayed, and a border appears around the text area. The border has one of 3 colours: green if the text is not considered AI generated (less than 50% probability), light red or pink if the probability is 50-75%, and red if the probability is 75-100% (very likely AI generated). **Priority: Medium**

- **Select AI text detection model:** In the extension UI, the user can choose, at any time, which model to use in the evaluations. In other words, which model to “activate”. A list with the names of the available models is shown, and the user clicks on the desired model. **Priority: High**

- **View details of available models:** The extension UI only has space to show the names of the models. In our Model Hub, the details of all available models are shown, including the name, description and author. For this, it is not required for the user to be logged in to the Model Hub. **Priority: Medium**

- **Choose highlighting colours:** Our extension has 2 highlighting colors, for 2 different intervals of probabilities of a text being AI-generated, 50-75% and 75-100%. By default, intuitive colours are chosen (such as red for 75-100%). However, in the extension UI, the user may, at any time, change the highlighting colours, either by selecting one of our colour presets or by entering the HEX code (for example AB93F0). **Priority: Low**

- **Toggle automatic web page and PDF analysis:** There is a toggle setting in the extension UI where, if turned on, web pages and PDFs are automatically analysed. If turned off, the user must click the Analyse button to analyse the page or PDF. **Priority: Medium**

- **Register in Model Hub:** The Model Hub has an account system. Model developers can register an account, by providing a username, email and password. **Priority: High**
- **Login in Model Hub:** Model developers can log in to their existing accounts by providing their email and password. Being logged in is necessary to upload and remove models. **Priority: High**
- **Upload AI text detection model:** In our Model Hub, if the model developer is logged in, he can upload his model, either as a Python script (must follow our rules such as class name "Model"), or as an API (the API must receive plain text and the response must contain a field "probability_AI_generated", which is an integer from 0 to 100 with the probability the text sent is AI generated). Apart from uploading the script or API endpoint, a model name and description must also be entered. A few seconds after clicking the "Submit model" button, an alert displays "Model accepted!" or "Model refused!". If accepted, the model is now available to all users. **Priority: High**
- **Remove own model:** In our Model hub, the model developer can remove, at any time, any model that he uploaded. **Priority: Medium/high**
- **Add, remove, enable and disable models:** The admins can, at any time, add and permanently remove a model in the system. They can also disable and enable models (make them unavailable or available to the users, respectively). **Priority: High**
- **Evaluate text through our API:** Any developer in the world can use our public API to get the probability that a text is AI generated, using one of our models. For more information, see the API section of this report. **Priority: Medium/high**

3.1.5 Non-Functional Requirements

The following list presents requirements specifying non-functional requirements, i.e. a description of a property or characteristic that a software system must exhibit or a constraint that it must respect.

- **Usability**: Knowing that the competing applications all have subpar usability, our number one goal is to provide the best user experience and not necessarily the best AI text detection. Also, because this application is not meant for any one specific demographic, we have to account for all possibilities, including users who are not experienced with using a computer. We also have to account for users with disabilities such as Daltonism, since we rely on colours to highlight AI-generated text. In sum, the extension must be easy to use, have an intuitive UI, and have various settings and customisation. **Priority: High**

- **Flexibility:** To keep our system viable and on the bleeding edge of AI and NLP, which nowadays are areas of constant and very rapid evolution, the system must be able to easily and quickly add, remove and update its available AI text detection models. **Priority: High**

- **Reliability:** The core functionality of our application, its capability of accurately detecting AI-generated text, must be, at the very least, on par with a competitor like GPT-Zero. There is no reason to use an app with a good user experience if it can't do what you ask it to, which is why it cannot fall on the same level as Originality.AI or OpenAI's AI Detector. At the same time, there isn't much of a need to surpass GPT-Zero, as our focus should be more on the UX. **Priority: High**

- **Performance:** If web page, PDF and text analysis are slow, the user experience is severely degraded. To combat this, the system architecture must be as smooth as possible, and the AI text detection models must not be slow. **Priority: High**

- **Compatibility:** The application will be built for Chromium-based browsers. It can always be expanded to other browsers in the future, but that is not as important as the other non-functional requirements. **Priority: Low**

- **Privacy:** The user should be confident that his data, such as his browser's information or the websites he visits, is not being stored or used. However, this is not the focus of this project. **Priority: Low**

- **Maintenance:** The code should be written in future-proof ways, as this application is intended to be used beyond the scope of this project. All issues should be tracked and recorded, for future programmers to be aware of them. Clean code practices must be followed and tools to monitor the system should be implemented. **Priority: Low**

3.1.6 Assumptions and Dependencies

In order for the application to work as expected the following assumptions are made. It is expected that users have an Internet connection to access both the web application and the web extension using a standard web browser. The extension is available on Firefox and all Chromium-based browsers, such as Chrome, Edge, Brave, Opera, and others.

3.2 System Architecture

This section presents an overview of the system architecture, describing its domain model and underlying relationships, as well as how the different components interact with each other.

3.2.1 Technological Model

The technological model in Figure 3.2 gives an overview of the technologies used by the system and how they interact with each other.

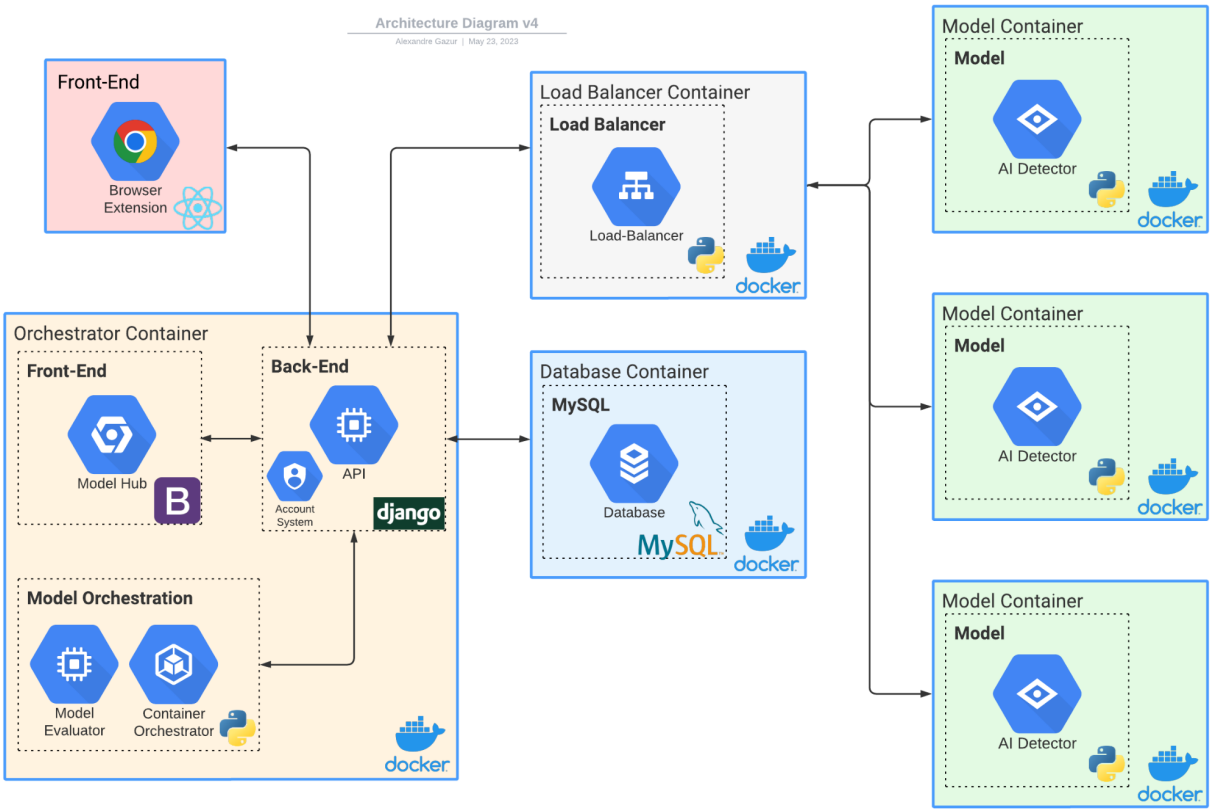


Figure 3.2: Architecture

The extension, accessed through its Graphical User Interface, contacts the Back-End through an API by sending requests for text to be analysed along with the model selected.
The probability is calculated depending on the type of model. If the model is an API, then the Back-End retrieves the model's API URL from the database and makes a request to that API to analyse the text, waits for the probability to be returned and then sends that probability to the extension. If the model is a script, then the Back-End sends a request to the Load Balancer with

the model's name and the text to be analysed. The Load Balancer then sends the request to the container running the requested model and returns the probability of the analysed text to the Back-End. When the extension receives the probability, it uses it to highlight the corresponding text in colours that depend on that probability.

The Model Hub website is used to submit models for the system to use. Anyone with an account can submit their model. The way the submission is handled depends on the type of model. If the model is an API, the Model Hub submits a model with a URL to the Back-End and it is saved in the database. If the model is a script, the Model Hub submits a model with a python (.py) file containing at minimum these 3 things: a LanguageModel class and 2 functions inside it: an __init__() function, used to set up the model by downloading its necessary files, and a predict(text: str) function, which is what is called to calculate the probability of a piece of text being generated by AI.

Regardless of the type of model, the Back-End tasks the Model Evaluator with evaluating how accurate the model is. The Model Evaluator then tests the model against a dataset of classified AI or human-generated text and calculates its accuracy. Depending on how high it is, it can either reject the model or accept it. When it rejects the model, it deletes its files from the database and the containers running it (if there are any). If it accepts it, it allows the users to select that model in the extension to analyse text. If the model is a script, it tasks the Container Orchestrator with starting a Docker container running that model. There can be more or fewer model containers than the 3 shown in the Architecture.

Chapter 4

Collaboration and Communication

In our project, effective collaboration and communication were key factors that contributed to our success. We used various tools and methods to ensure seamless coordination and efficient information exchange among team members. This section outlines the platforms and approaches we employed throughout the project.

4.1 Version Control and Collaboration Platform

We chose GitHub as our primary version control system and collaboration platform. Our project used a monorepo structure, which consisted of a centralised repository encompassing various components, including the web extension, web platform, backend, API, and more. We opted for this organisation strategy for two main reasons.

Firstly, based on our experience, monorepos can significantly benefit project management (if used correctly). Since our project was exceedingly large, we did not see a need to divide it into multiple repositories.

Other than that, they give every team member access to the entire project, making it easy to track individual and general progress. This simplifies collaboration and improves project development.

4.2 Communication Channels

We created a Discord server to enhance our online communication. It became our main hub for team interactions, where we organised different channels for specific tasks like general discussions, front-end development, back-end development, and deployment. This setup allowed us to have focused conversations and made it easier for the right team members to access relevant discussions.

We held weekly team meetings using Discord's voice channels. These meetings provided dedicated time for us to discuss project progress. We reviewed the tasks completed in the previous week, assigned new tasks for the upcoming week, and addressed any questions or concerns team members had. It helped us stay aligned, clarify doubts, and ensure everyone was on track with their responsibilities.

Additionally, when we were physically together at IEETA, we made the most of face-to-face interactions for in-depth discussions and closer collaboration with our project mentors. This allowed us to seek their guidance, discuss challenges, and quickly clarify any doubts we had.

4.3 Project Management Tools

Considering the size and nature of our team, we found that our combination of GitHub, Discord, and weekly reunions provided an effective framework for collaboration and task management. However, for future endeavours involving larger teams or more intricate projects, we acknowledge the potential benefits of implementing project management tools like Jira to enhance organisation and workflow efficiency.

Chapter 5

Implementation

5.1 Web Extension

The end of this chapter describes the implementation and main working features of the web extension, according to the specified requirements in the previous chapter. It was developed on Plasmo, a relatively new framework that streamlines the process of building browser extensions. Plasmo offers support for Typescript and React, which are two web development technologies we were familiar with. Given our limited experience in building web extensions, Plasmo's assistance was particularly valuable.

Before I start detailing the work relevant to this section, it is important to first understand the anatomy of a web extension. An extension's architecture will always depend on its functionality, but many robust extensions will include multiple components. The components we used in our extension are:

- **Manifest**: Configuration file that provides essential information about the extension, such as its name, version, permissions, and other details.
- **Background Script**: JavaScript file that runs in the background of the browser, allowing the extension to perform tasks, handle events, and interact with the browser's APIs.
- **UI elements**: Graphical components and HTML pages that define the structure and layout of the extension's user interface.
- **Content Script**: JavaScript file that runs in the context of web pages, allowing the extension to modify, interact with, or gather information from web pages.

Plasmo auto-generates the manifest.json, so we can focus on building the features rather than caring about where to point content scripts, and what the schema format should look like.

Our extension provides users with three distinct text analysis options: scanning the entire page, scanning selected text, and scanning text copied to the clipboard.

Once the text is analysed, the resulting information is displayed in various formats depending on the chosen method of analysis.

To enable users to scan selected text, we have implemented a singular background script that installs an additional item called 'Scan Text' onto Chrome's context menu. This option becomes available exclusively when a text selection is made. Please refer to Figure 5.1.1 for a visual representation of this feature. The menu is shown when the user right-clicks.

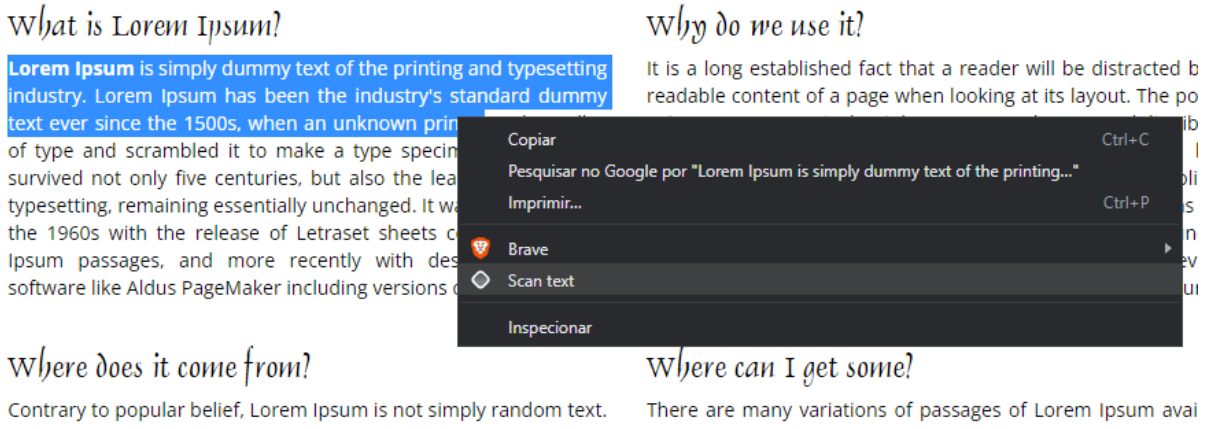


Figure 5.1.1: Scan selected text

When we click on ‘Scan Text’, the selected text is sent to our API which returns the probability score of it being generated by an AI. If the score is below 50% then the text is highlighted in a set light blue-grey colour, otherwise, the highlighting will be done with the colours chosen by the user. This process of choosing colours will be explained later in this section. We decided to maintain a consistent colour for lower scores to ensure clarity and distinction in identifying potentially AI-generated text.

The extension injects a global button element into web pages, serving as a prominent and recognizable component of our extension's functionality. It is strategically positioned at the bottom right corner of the screen, ensuring visibility while minimising disruption to the user's browsing experience. This positioning approach aligns with the practices followed by other prominent extensions, such as Grammarly. The primary function of this button is to scan the entire active page, although its feature set can be easily expanded in the future. It exhibits four distinct states as illustrated in Figure 5.1.2.

Figure 5.1.2: Global button states

The transitions and flow of the button's behaviour during the scanning process are described in Figure 5.1.3.

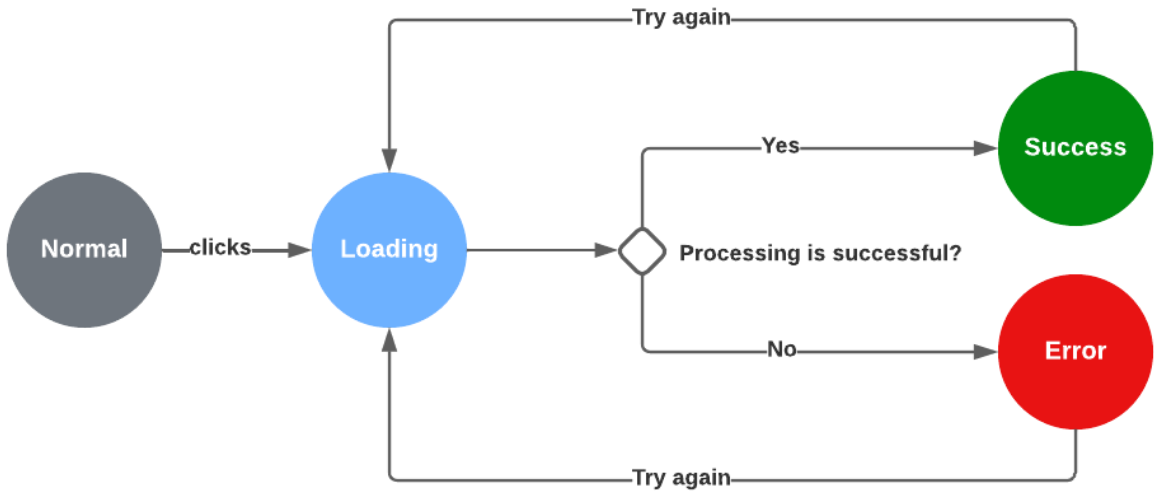


Figure 5.1.3: States Diagram of Global button component

The initial state is what we call ‘Normal’. In the normal state, the button is ready for interaction. When the user clicks the button, the scanning process begins, and the button transitions to the "Loading" state. In this state, the web page is processed and scanned section by section, ensuring thorough analysis.

If the scanning process successfully completes, the button switches to the "Success" state. At this point, the button displays the overall evaluation of the page, presenting the percentage of AI-generated content detected. This provides users with valuable insights into the extent of AI-generated text on the page.

However, if an error occurs during the scanning process or if it fails to complete, the button enters the "Error" state. In this state, an appropriate error message is displayed, informing the user about the encountered issue.

Regardless of the outcome (success or error), users have the option to redo the scan, by clicking on the ‘Redo scan’ button shown in Figure 5.1.4. This initiates the scanning process, and the button reenters the "Loading" state.

These four states of the button ensure users are well-informed about the ongoing scanning process, the evaluation result, and any encountered errors.

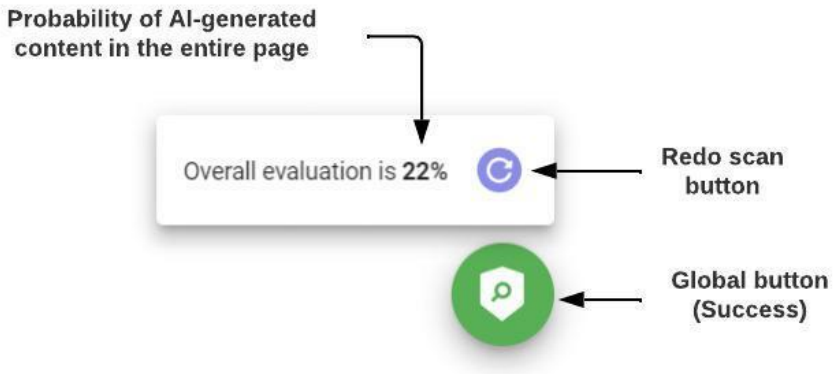


Figure 5.1.4: Full-page scan components

To better understand the inner workings of the scanning process, let's take a closer look at each stage. When the user initiates the scanning by clicking the button, a series of actions are triggered behind the scenes. The data extraction process takes place on the client side, where the text is parsed and segmented into individual sections for analysis. On regular web pages, the text is arranged into paragraphs, while in PDFs, it is divided into sentences. Each section undergoes a thorough evaluation to determine its relevance and significance. To ensure accuracy and focus on substantial content, sections containing fewer than eight words, as per our predetermined word count threshold, are filtered out and disregarded. We then make subsequent requests to our API, targeting those sections individually, and retrieving their AI-generated percentage scores. Those sections are then highlighted with the corresponding colour, aided by the [findAndReplaceDOMText] Javascript library, as illustrated in Figure 5.1.5.

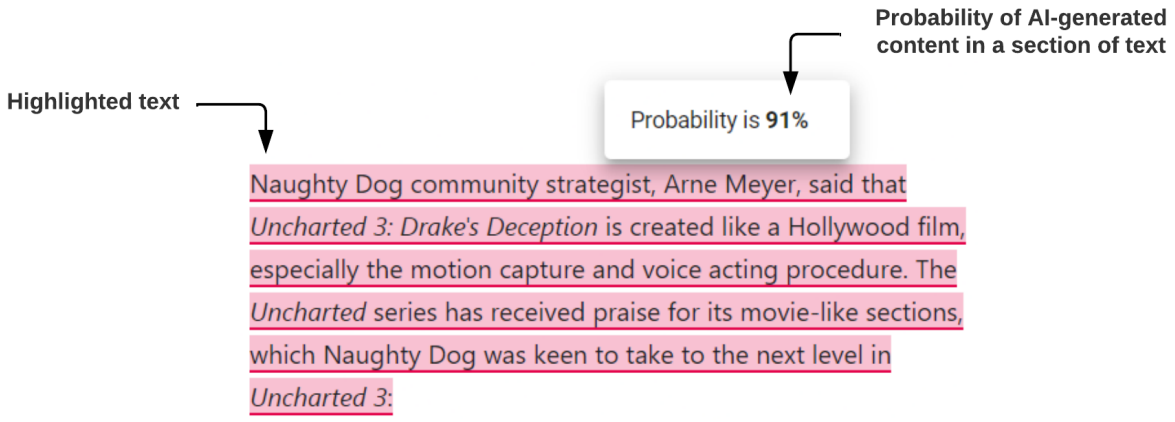


Figure 5.1.5: Example of highlighted text and score

The scores assigned to each section and model are stored in a cache using a simple caching mechanism implemented with the Map JavaScript object. This caching approach offers several advantages for faster processing during future scans, as sections of text that remain unchanged do not require reevaluation.

To ensure efficient memory usage, the cache has a maximum size of 100. When the cache reaches its maximum size, the oldest entry (i.e., the first key) is removed to make space for new entries. The choice of 100 as the maximum size strikes a balance between memory consumption and the likelihood of reusing previously computed scores.

The keys stored in the cache have a specific format of <model>-<hash>, where the 'model' represents the model used in the scan, and the 'hash' is the hashed value of the corresponding section of text. This format allows for efficient indexing and retrieval of scores based on both the model and the specific text.

The chosen hash function is a variant of the widely-used "Jenkins hash function". While not intended for cryptographic purposes, this hash function offers fast computation suitable for our use case. It iterates through the characters of the input string, performing bit shifting and bitwise operations to calculate the hash value. The resulting hash is a 32-bit integer, providing a balanced trade-off between speed and collision resistance for our caching requirements.

The overall score of the full page, displayed in Figure 5.1.4, is given by the following formula:

$$FP_{score} = \frac{\sum_{i=1}^{N} |s_i| P(s_i = fake)}{\sum_{i=1}^{N} |s_i|}$$

This formula calculates the weighted average by summing the products of the length and probability of each section (s_i) and then dividing it by the sum of the lengths of all sections. The length ($|s_i|$) refers to the number of characters present in the respective text section, while the probability represents the likelihood or confidence that the text was generated by AI.

The extension also specifies a popup, which refers to the HTML content displayed when the extension icon is clicked in the browser's toolbar, as shown in Figure 5.1.6.

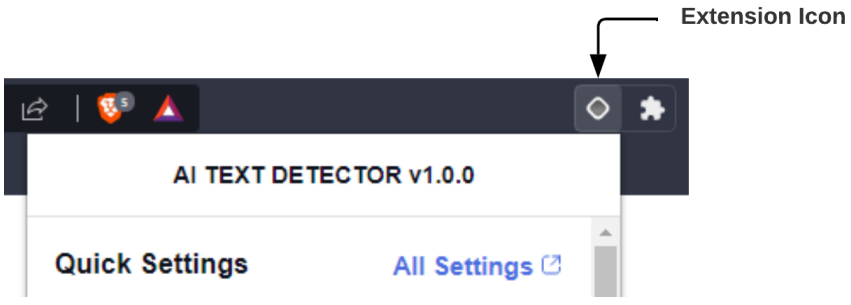


Figure 5.1.6: Icon in the browser's toolbar

The extension's settings are conveniently displayed within the popup interface, allowing users to customise and fine-tune the extension according to their individual preferences. There you can:

- **Turn on automatic scan**: Enable automatic scanning functionality, allowing the extension to scan the active page without requiring manual initiation. The default value is 'False'.
- **Change highlight colour (50-75)**: Adjust the highlight colour range for sections with AI-generated scores between 50 and 75 percent. The default value is 'yellow' (i.e., #FFFF00)
- **Change highlight colour (75-100)**: Adjust the highlight colour range for sections with AI-generated scores between 75 and 100 percent. The default value is 'red' (i.e., #FF0000)
- **Search and Change AI detection model**: Search and Switch between the different available AI models, each potentially offering varying performance or capabilities. The default value is the first model available, often the model named 'openai-base-roberta'.

In addition to preferences and settings management, the popup also offers several practical features. These include the ability to scan text that has been copied to the clipboard, providing quick access to our Model Hub platform, and offering a convenient link to Mozilla's PDF Viewer for scanning PDF files. In the next subsection, we provide a more detailed explanation of the PDF viewer.

We have set a limit of 5000 characters for the maximum amount of text that can be scanned. This decision is based on considerations of wait times and resource consumption. Scanning larger amounts of text can lead to longer processing times and increased resource usage, which may impact the overall performance and responsiveness of the system. Also, we did not see the need for it, as 5000 characters is already a significantly high amount.

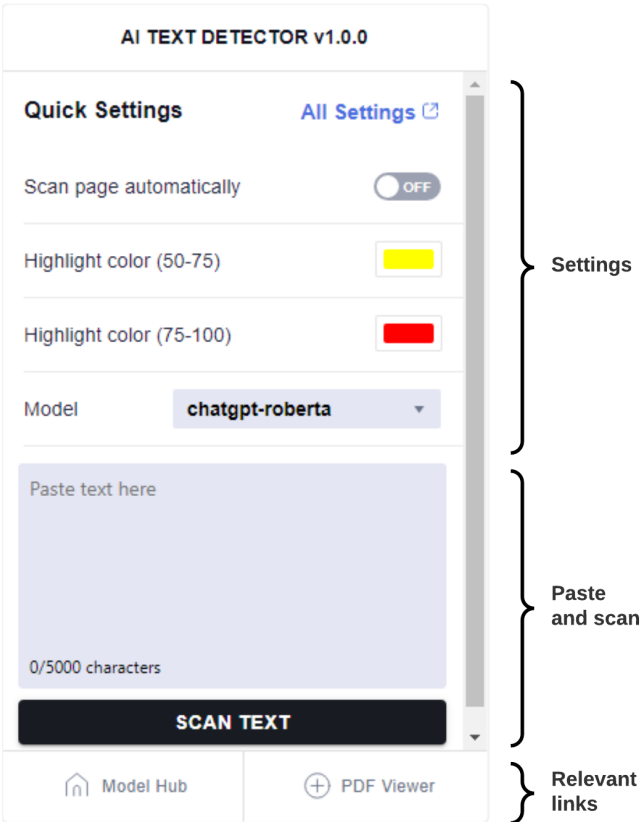


Figure 5.1.7: Popup of the extension

User settings are stored using Plasmo's Storage API, which is a utility library that abstracts the persistent storage API available to browser extensions. It falls back to localStorage when the extension storage API is unavailable, allowing for state sync between extension pages, content scripts, background service workers and web pages.

The search mechanism in the popup uses a data structure called Trie (derived from retrieval), shown in Figure 5.1.8. A Trie is a tree-like structure that stores strings over an alphabet. It allows for quick and effective pattern matching.

We initially believed that implementing a Trie in our project was unnecessary due to the limited number of available models. However, we decided to include it as a proactive measure to ensure scalability in the future.

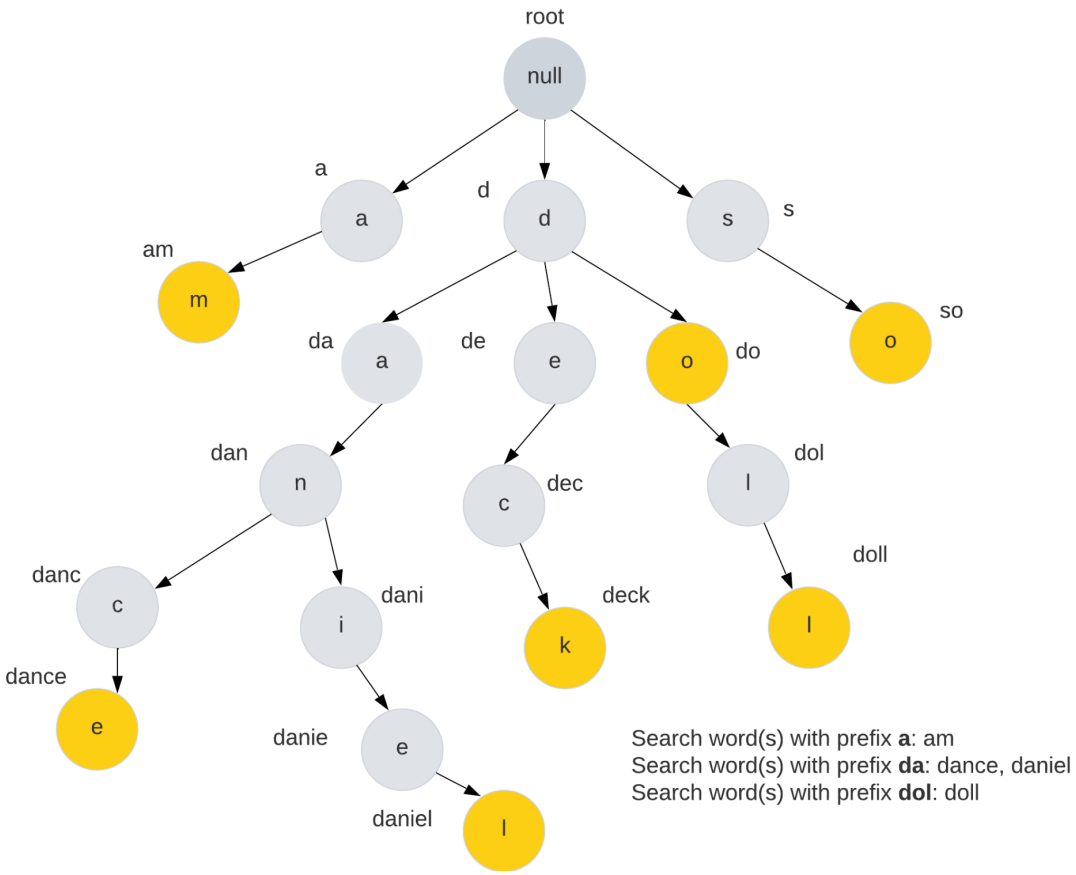


Figure 5.1.8: Trie data structure

To enhance usability and improve the overall user experience, we have implemented two keyboard shortcuts. Pressing Alt+Shift+P (Command+Shift+P on Mac) enables users to swiftly open the Popup interface. Additionally, pressing Alt+Shift+S (Command+Shift+S on Mac) allows users to initiate the scanning process for the selected text.

5.2 PDF Viewer

In order to extract text from PDFs and analyse them, a custom PDF viewer is needed. This viewer transforms the embedded PDF into span elements usable by our scraping script. Since each span element is 1 line of the PDF document, we couldn't find a way to break a PDF document's page into paragraphs, so analysis works sentence by sentence (as opposed to normal web pages, which are analysed paragraph by paragraph). This is a limitation of the project where the reliability of predicting AI-generated text improves with a larger number of words and sentences.

The user uploads his PDF document to Mozilla's PDF Viewer, which is a public custom PDF viewer made by Mozilla using their pdf.js library. Then, our extension works normally: by clicking the Analyse button, the current PDF page is analysed (and usually the page before and after, as those are usually loaded too; it's just how this Viewer works).

5.3 Model Hub

Model Hub is our website where users can check the details of all available models (name, author, description). To create the user accounts system in it, we used Django's authentication system. To upload a model, the model developer must be registered and logged in to the Model Hub.

Models come in 2 forms: as a Python script or as an API endpoint. The former must follow rules, such as class name Model, a __init__() function where any initialization is done and a predict(text: str) function, that receives text and returns the probability that it is AI generated (a number between 0 and 1, including both). As for the latter, the API endpoint of the model developer must receive plain text and return a JSON response with a field "probability_AI_generated", which is the probability that the text sent is AI generated (integer between 0 and 100, including both).

Model developers can permanently remove their uploaded models at any time.

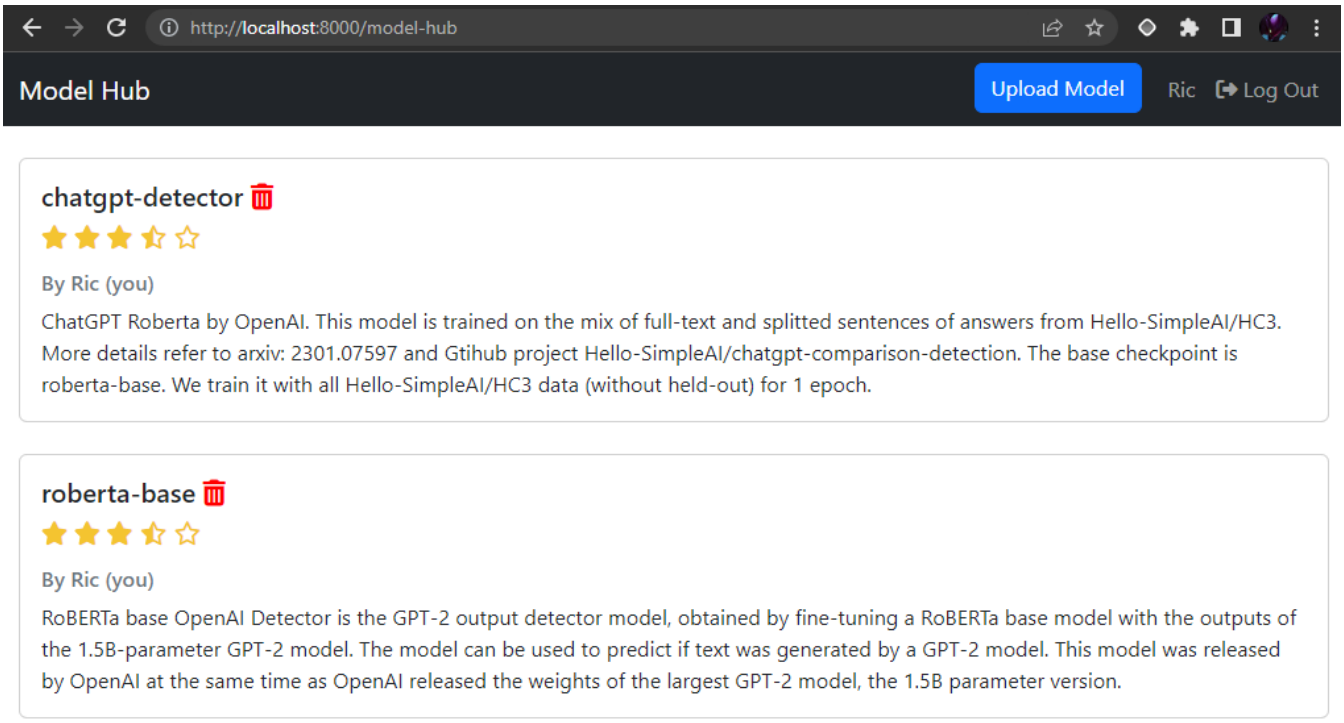
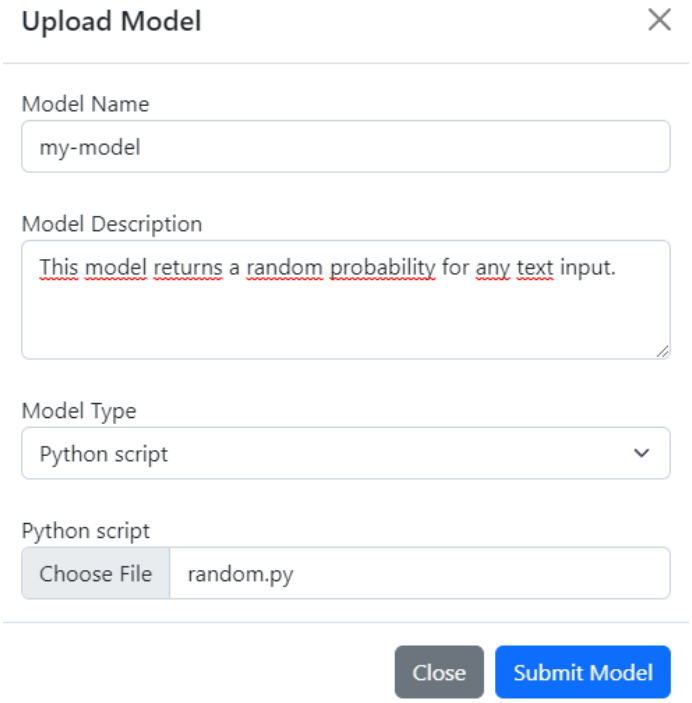


Figure 5.3.1: Model Hub



5.3.2 Model submission

5.4 Backend / Server-side

Our project mentor Tiago provided us with 3 AI text detection models: two based on GPT-2 and one based on GPT-3. For the backend, we decided to use Django for 3 reasons: first, all the team members had experience and were comfortable with it, second, we knew it would be a quick solution to implement, and third, all the AI/ML we had seen was in Python, and all the models Tiago provided were Python scripts, so it made sense to use a framework also coded in Python. Thus, Django was an obvious choice for the backend. We set up API endpoints and business logic so that the front-end can get probabilities of a text being AI-generated, using Tiago's models. We used Django's built-in SQLite database to store the models' information (name, author, description). Initially, the backend was non containerized, meaning the API, models and SQLite database ran all in the same place.

After that, we built the Model Hub using Django's MVC, as well as Django's authentication system. We also made an API endpoint that returns the details of currently available models, so that they can be displayed in the Model Hub. This endpoint is also used to upload models, so we had to create the business logic to save the model details in the database. For models in the form of Python scripts, we saved the script in a folder and executed them, making them available to the users; and for models as APIs, we simply saved the endpoint in the database.

Later, we containerized our system using Docker. Because the models were relatively slow to make their predictions, we quickly understood that it would be the models themselves that would be doing most of the workload. Therefore, we decided to split our system into 3 separate types of containers: one type of container would run the Django API alongside the Model Hub, meaning it has the responsibility of handling model submissions and prediction requests, another type would run the models, and we would have one container per model to help distribute the workloads, and another type of container would run the load balancer that would distribute the requests coming from the API through to the containers running the models. Later on, we also migrated from an SQLite database to a MySQL database because MySQL is more suited for larger databases, and also put it running on another Docker container. This database stores the Model Hub user's accounts and the uploaded model's details (like name, author, description, type + script path or endpoint). By separating all these components into different containers, we increase the modularity and scalability, and reduce the maintenance of the system, while also keeping our servers safe by running the user-submitted models in a sandbox environment.

5.5 API

Using Django REST Framework and JSON, we built a REST API. It has a private part, reserved only for the system, which it uses to add and remove models, as well as authentication in the Model Hub. But it also has a public part, usable by anyone.

Public endpoints (2):

POST /api/v1: This is the main endpoint. It is used to get the probability of a certain text is AI-generated, using one of the available AI text detection models. This endpoint is used by the system all the time (for example, to analyse web pages, PDFs, selected text, and inputted text) and since it is public, everyone can use this endpoint as well. A POST request may be sent to /api/v1 with 2 fields, “text” and “model”, which must both be strings. The “text” field contains the text to be evaluated, and the “model” field contains the name of the model to be used. If no errors happen, the response has status code 200 (OK) and a field “probability_AI_generated”, which is an integer between 0 and 100 (both inclusive) with the probability that the text sent is AI-generated. If an error occurs, the response has status code 400 (client error) if the requester made an invalid request (for example, a model with that name doesn’t exist) or status code 500 (server error) if, for example, the model stopped working; the response contains a field “message”, which is a string explaining the problem that occurred.

```
{
    "text": "A computer is a machine that can be programmed
        to carry out sequences of arithmetic or logical
        operations (computation) automatically.",
    "model": "chatgpt-detector"
}
```

Figure 5.5.1: POST request /api/v1 example

```
{
    "probability_AI_generated": 53
}
```


Figure 5.5.2: POST /api/v1 response example

GET /api/v1/models: By simply accessing this endpoint in a browser, or making a GET request to it, one can get the details of the currently available AI text detection models. The response is an array, where each item represents 1 model, with 4 fields which are all strings: “name” is the name of the model, “description” is the description of the model, “author” is the name of the author who uploaded the model and “type” is the type of the model (“script” for python script or “API” for API). Below is an example.


```
[
    {
        "name": "chatgpt-detector",
        "author": "Ric",
        "description": "ChatGPT Roberta by OpenAI. This model is trai
        The base checkpoint is roberta-base. We train it with all Hel
        "type": "script"
    },
    {
        "name": "random-api",
        "author": "Ric",
        "description": "This API returns a random probability.",
        "type": "API"
    }
]
```


Figure 5.5.3: GET /api/v1/models response example

Private endpoints (2):

POST /api/v1/models: Used by our Model Hub, to upload models when submitted by a user. The name, description and author (username of the account, given that the user must be logged in in the Model Hub to upload a model) are sent. The 4th field in this POST request is either “api” or “script”, depending if the model being uploaded is an API endpoint or a Python script, respectively. The “api” field contains the API endpoint, and the “script” field contains the Python script.

DELETE /api/v1/models: Used in the Model Hub, when the author of a model deletes that model. The name of the model is sent to the server, and it is permanently deleted. If it's a model as a script (as opposed to a model as API), its container is destroyed, as it is no longer needed.

Apart from the public and private endpoints listed, we also have a /api/v1/my-model endpoint, which is an example of a model as API. We use this endpoint to test and demonstrate uploading a model of type API. As explained in **5.3 Model Hub**, this endpoint receives plain text and returns a JSON response with a field "probability_AI_generated", which is an integer from 0 to 100 (including both) that is the probability the text sent is AI-generated.

Chapter 6

Deployment

6.1 Web Extension

Our objective is to publish our web extension on both the Chrome Web Store and Firefox Add-ons, expanding its availability to a wider user base. To achieve this, we will follow a systematic process:

- **Developer Account**: We will create a dedicated developer account to manage the extension and monitor its performance.
- **Packaging**: The extension will be packaged in a format compatible with the Chrome Web Store, ensuring all necessary files and assets are included.
- **Information and Presentation**: To enhance users' understanding of the extension, we will provide essential details such as the name, description, screenshots, and relevant categories. These elements will be presented effectively to convey the purpose and features of the extension.
- **Privacy Compliance**: A comprehensive privacy policy will be developed, outlining the extension's data collection and usage practices. This policy will adhere to established privacy standards to ensure user confidence and compliance.
- **Review and Approval**: The Chrome Web Store conducts a thorough review process to assess compliance with guidelines and policies. We will ensure that our extension meets these requirements to ensure a seamless publishing experience.
- **Publishing**: Once the content review is successfully completed, the extension will be published on the Chrome Web Store. This will provide users with an effortless way to discover, install, and enhance their browsing experience.

Simultaneously, we will actively work on ensuring compatibility with Firefox Add-ons by meeting the specific requirements set by Firefox Add-ons for publishing our extension.

6.2 System

We will deploy the system on a server located at our university, Universidade de Aveiro. With the University's permission and the correct setup, our application will be made available on the World Wide Web. Our mentor, Tiago, will grant us access to this server, which boasts multiple GPUs that are better suited for running our models, significantly reducing their processing times. During development, we ran the system on localhost, using our own computer's CPU, which proved to be slower. However, in the future, we will transition to the server environment, ensuring more efficient execution of our models and wider availability of our application.

Chapter 7

Tests and Results

This section is dedicated to examining the usability of our applications (Web extension and Model Hub). It includes an evaluation of the applications' usability using the System Usability Scale (SUS), as well as gathering user feedback on their perceived strengths and weaknesses through a narrative approach.

7.1 Profile of the Respondents

The study included 8 participants, primarily students from the LEI course at the University of Aveiro. These participants exhibited a range of expertise with web extensions and applications, with approximately 88% having intermediate to advanced knowledge and experience, while the remaining 12% had beginner-level familiarity. It is important to note that they were not representative of the expected users of our application. Instead, they were selected from a pool of friends and family members, resulting in a sample that may not fully reflect the broader user population. Nonetheless, their participation provides valuable insights and perspectives that can contribute to our understanding of the application's usability.

7.2 Usability Score

Statement	**1**	**2**	**3**	**4**	**5**
I think that I would like to use this system frequently	0	3	2	1	2
I found the system unnecessarily complex	8	0	0	0	0
I thought the system was easy to use.	0	0	0	0	8
I think that I would need the support of a technical person to be able to use this system	7	1	0	0	0
I found the various functions in the system	0	0	0	2	6

were well integrated					
I thought there was too much inconsistency in this system	4	4	0	0	0
I would imagine that most people would learn to use the system very quickly.	0	0	0	0	8
I found the system very cumbersome to use	8	0	0	0	0
I felt very confident using the system	0	0	0	2	6
I needed to learn a lot of things before I could get going with this system	6	2	0	0	0

Table 7.2.1: SUS Scale Summary

The SUS is a well-established and widely accepted questionnaire designed to evaluate the usability of a system or product. It presents respondents with a set of 10 questions, and their task is to rate each question on a scale ranging from 1 to 5.

To calculate the final SUS score, several steps are involved. First, the scores given by the respondents for each question are adjusted. This adjustment is done by subtracting 1 from each score, which converts the original scale of 1 to 5 to a range of 0 to 4.

The SUS questionnaire includes both positive and negative statements. The positive statements are associated with odd-numbered questions (1, 3, 5, 7, 9), and are highlighted in blue in Table 7.2.1. Conversely, negative statements are associated with even-numbered questions (2, 4, 6, 8, 10), and their scores need to be inverted before further calculations.

All the points added up together could gain a maximum of forty, so we multiply it by 2.5 to make the scale out of 100.

The final SUS score we obtained is 92.2, which is convertible to an A+ grade, and represents the average of all respondents. This indicates that the clear majority of the respondents had a positive outlook on the use of the application.

7.3 User Feedback

During the testing phase, we collected valuable user feedback on the applications. As said previously, the results were highly positive, with the majority of participants expressing satisfaction and having little to point out. However, a few suggestions were made to improve the applications' usability.

One recurring comment pertained to the aspect and design of the Model Hub Web platform. Some participants felt that it could benefit from further refinement in terms of its visual appeal and overall aesthetics. However, it's important to note that these suggestions were focused on the platform's appearance and did not raise any concerns about its functionality and features.

Another suggestion put forward by users was adding a label such as "AI" after the percentage results, in the Web extension, to improve clarity. This modification would ensure that users readily understand that the percentage score represents the confidence level of the text being generated by AI.

Additionally, users provided some recommendations to enhance the information displayed within the platform. For instance, it was suggested that the platform could showcase the total number of AI-generated and human-generated text sections, along with identifying the specific model responsible for generating the text deemed as written by AI.

Chapter 8

Conclusion and Future Work

8.1 Main Results

We have achieved significant milestones in our project, resulting in the successful development of the application. When we consider the expected goals and the key features implemented in the final version of the project, it is safe to say the main goals were accomplished.

We have successfully developed a browser extension that provides more features and has arguably more value than the existing solutions mentioned in Chapter 2.

We also managed to incorporate a flexible architecture that allows for easy integration of additional models and potentially new use cases. This flexibility ensures that the system can be expanded to incorporate emerging technologies and accommodate evolving needs in the field of NLP and ML.

During testing, our system demonstrated a high level of usability. However, it is important to note that the validity of this result may be limited due to the potential biases in the respondent pool. The respondents involved in evaluating the system are not fully representative of the target users, and could have introduced some level of bias into the usability assessment.

8.2 Conclusions

Our project represents a significant accomplishment, demonstrating our ability to develop a robust and efficient software application. The success of this project meets our initial goals and expectations. It includes an extensive feature set and adaptable architecture making it a valuable tool that could be used with confidence.

The advantages that may arise from a system like this were already outlined in previous sections. We received even pointers from multiple people that suggest that the system may possess substantial value worthy of market entry. Given these promising indications, we believe that with the right partnerships and a sensible business model, it could become a reality in the near future.

Working on this project has been an invaluable experience for us, providing us with a real-world software development environment that adhered to clear methodologies and

expectations. The pressure exerted by mentors, professors, and even colleagues proved to be a positive catalyst, fostering our growth and professional development. It served as a valuable opportunity for us to gain hands-on experience and develop a more mature approach to our work.

8.3 Future Work

In the future, we have several things we plan to work on to make our platform even better. Here are the key areas we will focus on:

- **Better PDF Analysis**: We want to improve how we analyse PDF documents. Right now, we use a viewer from Mozilla, but we aim to develop our own or find a better way to analyse PDFs. This will make working with PDF files smoother and more efficient.
- **Rewrite Extension with Svelte**: Currently, our extension is built using React, but we plan to rewrite it using a framework called Svelte. This change will improve performance and simplify the development process, benefiting both users and developers.
- **Improve Model Hub**: We want to make the Model Hub platform more visually appealing and user-friendly. We want to enhance its design and add new features to give users a better experience. These changes will improve the overall quality and advancement of the Model Hub platform.
- **Expand Text Analysis Capabilities**: Our platform currently focuses on detecting AI-generated content. However, we aim to expand its functionality to include broader text analysis capabilities. Our goal is to create a more versatile solution that can be used to analyse various aspects of the text. For instance, a model developer could upload a sentiment detection model, and our extension will adapt to it.
- **Usability Tests with Target User Base**: We conducted usability tests, but they were not performed with our target user base. As a result, the gathered results may not be valid or representative. To obtain accurate feedback, we plan to conduct new usability tests specifically with our target users.
- **Commercialise the Application**: Our goal is to commercialise and sustain our application by implementing a freemium model. This involves offering a free version with basic features and introducing premium subscriptions for advanced functionalities and added benefits. We will also explore licensing options for organisations. These changes will require significant updates to the application.

Chapter 9

Limitations

This project has two limitations that should be noted. Firstly, our system/extension is currently limited to detecting AI-generated text in English, as the AI text detector models provided by our mentor, Tiago, are trained on English datasets. However, our system has the flexibility to be expanded to other languages. Through our Model Hub, users can easily add and remove models, allowing for the inclusion of language-specific models. This means that if models designed to detect AI-generated text in other languages become available, our system can be extended to effectively detect and analyse those languages as well.

Secondly, when analysing PDFs, the analysis is performed on a sentence-by-sentence basis, unlike web pages which are analysed paragraph by paragraph. While analysing a single sentence may not always yield reliable results for AI text detection, multiple sentences and a greater number of words typically lead to more accurate predictions. Although we encountered challenges in implementing paragraph analysis for PDFs, we recognise that further advancements may be possible in the future.

References

GPTZero. (n.d.). Origin. ChatGPT and AI Detector.

Originality.AI. (n.d.). Free AI Content Detector Chrome Extension

OpenAI AI Text Classifier. (n.d.). New AI classifier for indicating AI-written text

Mozilla. (n.d.). PDF.js. PDF Reader in JavaScript.

Brooke, John. (1995). SUS: A quick and dirty usability scale. Usability Eval. Ind.. 189.

Chakraborty, Souradip & Bedi, Amrit & Zhu, Sicheng & An, Bang & Manocha, Dinesh & Huang, Furong. (2023). On the Possibilities of AI-Generated Text Detection.

Appendix A

Contributions

Alexandre Gazur (102751) - 1/3 of the work:

- Containerization
- Deployment
- Backend logic and services
- Architecture diagram
- Model Hub authentication
- Helped in presentation slides

Daniel Ferreira (102885) - 1/3 of the work:

- Extension UI
- Extension logic and scripting
- Majority of the report
- Design and management of project website
- Helped in presentations slides and documentation

Ricardo Pinto (103078) - 1/3 of the work:

- Actors, use cases diagram, use cases description
- Functional and non-functional requirements
- Web scraping (wrote the javascript script)
- Non Containerized backend version (our initial backend)
- API (endpoints and logic)
- Model Hub home page and models submission/removal frontend logic
- Video
- Helped in presentations slides and documentation

João Matos (103182) - 0% of the work:

- Poster

Appendix B

Lessons learned

Alexandre Gazur:

The most important part of creating never-before-used software is constant communication with coordinators/clients and between team members. We had to re-design our architecture twice due to changing demands because of lack of communication, and many parts of our system were slow to implement because integration was difficult without proper communication. After changing our development from at-home to in-office where we (team members and coordinators) could talk face to face, it became much easier to define our goals and write competent and understandable code that could be used by others.

Daniel Ferreira:

One of the key lessons I learned throughout our project is that we are often overly optimistic when it comes to software estimation. It is extremely hard (impossible) to make an accurate estimate of how much time a particular developer will spend on a specific feature because unexpected issues are very common in software development.

The project also served as a practical application of Brook's Law. I learned that adding more people to a late software project doesn't always speed things up. It's important to carefully consider whether additional resources will truly enhance productivity or introduce communication issues. Sometimes, improving the existing team's workflow is a better approach.

I have recognised the significance of assigning tasks based on individuals' strengths and expertise, as well as setting realistic deadlines. Regular check-ins help address challenges and keep everyone aligned.

It’s important to acknowledge that we don’t have all the answers, and being open to seeking advice from mentors and teammates when needed can be extremely valuable.

Perfectionism can be a hindrance. It's better to have a functional solution in a reasonable time frame than to get caught in a cycle of making last-minute adjustments. It's important to remember that not every aspect needs to be perfect from the start; incremental improvements and iterations are key. There is a reason companies first build a MVP (Minimum Viable Product) and only then extend the functionality to satisfy customer needs.

When working with code written by teammates, it is important to take the time to understand the code and its dependencies. If possible, we should also discuss the code with the teammate who wrote it to gain insights and clarify any questions. Sometimes, code can be

tightly coupled with other parts of the system, and making changes without considering the broader context can lead to unintended consequences.

Lastly, when working with node applications, we often use the development build for testing and assume it will work in production. However, sometimes we find some unpleasant surprises. The application may work fine during development, but once we create a production build, we start seeing errors like "Uncaught TypeError: Cannot read properties of undefined (reading 'a')." These errors appear in the compiled JavaScript file with obscured line numbers, making debugging extremely complicated. The first time this happened, the application was already quite large, and I had to resort to trial-and-error experiments to identify the cause, which consumed a lot of time. Since then, I've learned to test the production build version whenever I make updates. This practice has helped me catch and fix two other errors that could have been troublesome to deal with otherwise.

Ricardo Pinto:

Setting deadlines and respecting them is very important to ensure development is smooth and everyone is on the same page. Creating healthy relationships with our project mentors was very important, since it enabled me to be comfortable asking them any question and asking for help whenever I needed, which saved me hours of work with just a few minutes of talking with them, as well as with my project team.

João Matos:

-

Appendix C

Repository

Please note that we have used a monorepo approach, where all components are consolidated within a single repository. The instructions on how to run each component can be found on the respective README.md files.

The repository can be accessed at https://github.com/zzzzz151/AI-text-detector.

Web extension:
https://github.com/zzzzz151/AI-text-detector/tree/main/ai4-td-extension

To run the extension, follow these steps:

1. Ensure that you have `Node.js`, `npm`, and (optional) `pnpm` installed on your machine.
2. Open a terminal or command prompt.
3. Navigate to the project's root directory.
4. Run the command `npm install` or `pnpm install` to install the project dependencies.
5. Once the installation is complete, run either `pnpm dev` or `npm run dev` to start the development server.

Please note that it is assumed you have the necessary prerequisites installed and configured on your system.

Model Hub and Backend:
https://github.com/zzzzz151/AI-text-detector/tree/main/DjangoProject

To run the extensions, follow these steps:

1. Launch the Docker Desktop application (make sure the engine is running).
2. Execute the `run.bat` file found in the root of the repository.

These are the instructions to run the application locally. When the application is deployed, the Chrome extension will be available on the Chrome Store, and the Django application will be hosted on the university's servers.