Intelligent Query Processing: AI Tool Development for Enhanced Interaction with the Aberdeen Registers Online Corpus

Yingxin Fan

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

Department of Computing Science

2024

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2024

Abstract

This project presents the development and implementation of an innovative search tool designed to enhance the retrieval and analysis of historical documents from the Aberdeen Registers Online (ARO). Leveraging advanced Natural Language Processing (NLP) techniques, specifically the XLM-RoBERTa models, this tool introduces semantic search capabilities to manage the complexities inherent in historical texts, such as linguistic variations and multilingual content. The research outlines the process of adapting these NLP models to the specific challenges of historical documents, integrating user feedback into the agile development process, and testing the tool's effectiveness in improving search accuracy and user experience. The results demonstrate significant advancements in search functionality, offering a more intuitive and efficient means for historians to access and analyze historical data. The project exemplifies the potential of combining NLP and semantic search technologies to revolutionize historical research practices.

Acknowledgements

Thanks for the people supporting me in this development process. Thanks to my mom and supervisor.

Contents

1 Introduction **7**

1.1 Motivation 7

1.2 Aims and Objectives 8

1.3 Document Structure 10

2 Background and Related Work **11**

2.1 Aberdeen Registers Online 11

2.1.1 Previous Searching Tool of ARO 12

2.2 Digital Humanities and Historical Document Search: Evolution from Keyword Indexing to Multilingual Semantic Analysis and a New Idea 14

3 Design **17**

3.1 Agile Development and Client Collaboration 17

3.2 Intended Audience 17

3.3 Requirements 18

3.3.1 **Main Function** 18

3.3.2 **Additional Functions** 18

3.4 Planning the Development Process 19

3.5 Algorithms and Technologies Chosen 19

3.5.1 Model Analysis and Training 19

3.5.2 Technologies Chosen 21

4 Implementation **24**

4.1 Data Processing 24

4.1.1 File structure analysis 24

4.1.2 XML Files Parsing 25

4.2 Model selection and fine-tuning 26

4.2.1 mBERT VS XLM-R 26

4.2.2 Model Fine Tuning 30
4.3 ARO Intelligent development 34
4.3.1 Preliminary Setup 35
4.3.2 Basic Functions Developments 38
4.3.3 Semantic Analysis and Reordering 42
4.3.4 Additional Features 43
4.4 Another try of generating synonyms dictionary 46

5 Testing & Evaluation **47**
5.1 Testing and Evaluation 47
5.1.1 Functions Testing 47
5.1.2 Animals Retrieval 47
5.1.3 Person Name Retrieval 49
5.1.4 Spousal Relationship Retrieval 49
5.1.5 Known Issues 52

6 Discussion & Limitations **53**
6.1 Discussion 53
6.2 Limitations And Future Work 54

7 Conclusion **56**
7.1 Project Summary 56
7.2 Personal Reflection 57
.1 User Manual 61
.1 Maintenance Manual 65

Chapter 1

Introduction

In the digital age, the storage and retrieval of historical texts has become particularly important. As more and more historical documents are converted into digital formats, it becomes a challenge to retrieve the information in these documents effectively. This report aims to introduce the process of developing an advanced search tool that utilizes natural language processing (NLP) technology to improve the efficiency and accuracy of a historical document research.

1.1 Motivation

Aberdeen, located in Scotland, is not only known for its long history and cultural heritage, but also holds an irreplaceable place in academic research for its detailed municipal records. These records, housed in the Aberdeen City Archives, span hundreds of years of history and provide invaluable first-hand information for the study of the socio-economic development of Aberdeen and Scotland as a whole.

By chance, I got a request from my client, Dr. Jackson Armstrong, a professor in the Department of History. He was seeking help for exploring the possibility of helping historians more efficiently locate and identify various entities by artificial intelligence techniques in those municipal records, Aberdeen Registers Online (ARO), which will be introduced in section 2.1. In recent years, as natural language processing technology has made significant advances in text analysis, using artificial intelligence to assist in historical research has become increasingly common. When dealing with multilingual historical documents, cross-language models can enable effective content search and understanding by sharing semantic information across multiple languagesReimers and Gurevych (2020), allowing researchers and even non-language experts to search for and understand content without mastering the language of the document. Modern NLP techniques can also expand the scope of keyword search, greatly enhancing the flexibility and depth of search. However, these advanced technologies were not taken into consideration during the development of the existing search tools, which has led to certain problems:

- Insufficient flexibility: Current tools require users to know and enter precise search terms in advance, which is not conducive to exploratory research, especially if the user is unfamiliar with the material or expects to discover new information through exploration.

- Overburdened users: Users need to set their own search parameters, which not only increases the threshold for use, but may also miss important information due to improper parameter selection.

- Language and variant handling: Existing tools cannot automatically handle language variants and synonyms in text, and the contents in ACR corpus were written in three languages, which limits the utility of tools in multilingual historical document searches application effect.

To address these limitations of current historical research tools, particularly in dealing with linguistic diversity and complexity, this report proposes the development of a new search tool using advanced AI technologies. These technologies have come a long way and are able to handle historical texts characterized by outdated expressions, variable spellings, and multilingual contentMin et al. (2023). By automatically generating synonyms and variants of keywords and optimizing the search algorithm, the proposed tool aims to increase flexibility and improve the overall user experience. The motivation behind the development of this tool is to enable historians to efficiently mine and analyse documents without being hindered by the complexity of traditional query syntax. It also aims to make historical research more accessible to non-experts interested in exploring historical documents but lack of relevant knowledge, thereby broadening the audience for historical research on ARO and encouraging more interdisciplinary collaboration that generate novel solutions to complex problems by applying insights beyond current boundariesKnapp et al. (2015).

1.2 Aims and Objectives

The main objectives of this project are to:

Develop an Intelligent Search Tool for Keywords

- The tool is capable of understanding and processing keywords, including their translations and proximity in various languages.

- Begin with learning the structure of ARO corpus and acquiring a comprehensive understanding of advanced technologies such as generative AI and pre-trained language models. Utilize this knowledge to generate a relevant search lexicon and integrate it with a model

to optimize semantic search queries, enhancing the relevance and accuracy of the search results.

- Design a user-friendly search interface that displays the results generated from the search queries in an accessible way.
- Conduct thorough testing of the keyword search functionality to ensure it meets the requirements of client and to evaluate the accuracy and efficiency of the results post-implementation.

Additional features complement

If the main feature is finished and tested, the next step is to include additional search functionalities based on the second requirement from client.

- Name Search: Implement a feature that recognizes and searches for various forms and spellings of names. This function will aid historians in identifying and tracing names of individuals mentioned in historical records, accommodating variations commonly found across different languages.
- Relationship Search: Develop a feature aimed at identifying and searching for spousal or marital relationships within the corpus. This feature will provide historians with a tool to examine family and social connections.

Ensure Modifiability and Future Readiness

- Design a flexible architecture for future enhancements, such as the addition of new features and search technologies. Adopt design patterns that facilitate easy integration of new components without affecting the main function.
- Make the system components modular to simplify code maintenance and future modifications.

Utilize Unit Testing for System Verification

Implement robust and consistent unit testing to verify all aspects of the application while developing, especially after any modification or addition. To validate the correctness and reliability of the search results, using the previous manual search results collected by client.

1. Comprehensive Reporting on Development and Evaluation:

Provide detailed reports on the development process, including the decisions made during the design and implementation phases. Present the implemented functionality of the application and future work can be considered.

1.3 Document Structure

This document is organised as follows:

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Analysis and Design
- Chapter 4: Implementation
- Chapter 5: Testing and Evaluation
- Chapter 6: Results and Discussion
- Chapter 7: Reflection and Conclusion

Chapter 2

Background and Related Work

2.1 Aberdeen Registers Online

Aberdeen Registers Online: 1398-1511 (ARO)[1] is a digital transcription of the first eight volumes of the Aberdeen Council Registers (ACR)[2]. ACR is the earliest and most complete book of civil government in Scotland, starting in 1398 (Volume 1) and covering the period before 1511 (Volume 8). It illuminates the workings of this Scottish borough in a way that no other Scottish urban record of the time can match, and also demonstrates the interconnectedness between people and ideas in Northern Europe during the Renaissance. The registers are held by Aberdeen City and Aberdeenshire Archives (file numbers CA/1/1/1 - CA/1/1/8) and were included in the UNESCO UK Memory of the World in 2013 List in recognition of its historical significance. The Aberdeen Council Registers are primarily legal records, including court proceedings, statutes, elections, dispositions and correspondence. They are handwritten, mostly in Latin and Middle Scots, two entries written in Dutch.

To make these invaluable documents more accessible to global researchers, the Aberdeen

[1] https://www.abdn.ac.uk/riiss/projects/aberdeen-registers-online-213.php
[2] https://aberdeenregisters.org/

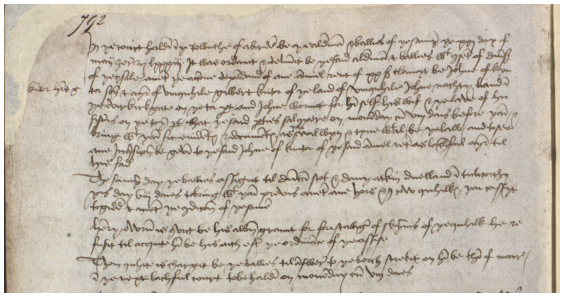


Figure 2.1: Example of ACR manuscript

City Archives initiated the "ARO" project, aiming to digitize these records and provide online access. This effort is part of the broader Aberdeen Burgh Records Project[3], which began in 2012, in partnership with the Irish and Scottish Studies at the University of Aberdeen and supported by the City of Aberdeen and Aberdeenshire Archives. The project not only focuses on the digital transcription of ACR but also on its legal aspects through the dedicated study known as the Laws in the Aberdeen Council Register 1398-1511: Concepts, Practices, Geographies (LACR)[4]. This digital humanities project is tasked with digitizing and creating a comprehensive, accurate, versatile transcription that forms the basis for the corpus to be digitally open and usable in various software tools.

Building on the previous work, in another "FLAG" research project of the Aberdeen Burgh Records Project starting in spring 2020, ARO is used to compare with Augsburg's Master Builder's Ledger (Die Augsburger Baumeisterbücher)[5] covering 1320-1466 to conduct an in-depth historically driven analysis. This project is a new chapter in the digital humanities in historical research.

2.1.1 Previous Searching Tool of ARO

Search Aberdeen Registers[6] and **Enhanced Search 1.0 for ARO**[7] are prototype web applications designed to facilitate efficient and precise searches within the ARO, covering court proceedings from the Middle Ages (1398-1511). The original application, Search Aberdeen Registers, provides basic browsing functionality, allowing users to directly view specific pages in any volume. It also supports searching by valid XQuery or scope. Furthermore, Enhanced Search 1.0 introduces targeted search options for courts and names, enabling users to not only retrieve all entries containing a specific court or name and count their occurrences, but also automatically generate charts depicting the number of entries over the years, which helps historians track court-related activities and document relationships over time. These two searching tools are the benchmarks to evaluate the performance of ARO Intelligent and the base of innovation of ARO.

However, these existing tools mainly rely on a simple and exact keyword matching method and are not able to infer additional information from the keyword, which greatly limits the enhancement of user experience. Not only does this approach fail to flexibly recognise semantically related but differently forms of expression, it also fails to accommodate semantic differences in multilingual environments, resulting in searches that are neither comprehensive

[3] `https://aberdeenregisters.org/`
[4] `https://aberdeenregisters.org/project/`
[5] `https://augsburger-baumeisterbuecher.de/`
[6] `https://sar.abdn.ac.uk`
[7] `https://enhancedsearch1foraro.pythonanywhere.com/`

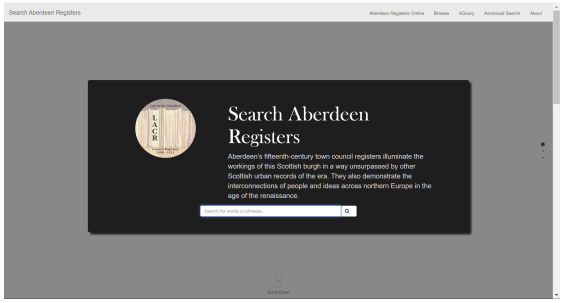


Figure 2.2: Search Aberdeen Registers

nor accurate.

Take, for example, a PhD project entitled 'Zootropolis: a multi-species archaeological, ecological and historical approach to a medieval Scottish city'[8]. The research project aims to explore the representation of 'lamb' and its related items in the corpus. The researcher needs to search for "lamb" and its synonyms one by one. Because even within English, there are many variations in how this animal is described, based on age and gender distinctions. This complexity is magnified in other languages, where 'lamb' can even be used as a last name. As Degani and Tokowicz (2010) noted, words with multiple translations corresponding to the same meaning pose a greater challenge than those with multiple meanings, underscoring the disadvantage in ambiguity when searching for such terms. This aligns with the findings of Trott and Bergen (2023), who suggest that word meaning is both categorical and continuous. As mentioned in section 1.1, this limitation not only increases the workload of the researcher in the data preparation phase since those related words needed to be collected by human, but also causes inconvenience when dealing with historical data across languages or cultural background, as it requires an in-depth knowledge of the expression conventions of each language and culture in the corpus. In addition, existing tools lack the ability to extract information and discover patterns from big data, an extremely critical function in modern digital humanities research. Seiler (2020) explored the different meanings of the word "unicorn" in various dictionaries, highlighting how meanings can vary across different sources, which further illustrates the challenges faced when existing search tools are used in digital humanities project.

In this case, with the help of artificial intelligence technologies may be a way out to solve these problems. AI can not only help with handling complex data queries, but also understanding contextual differences and language diversity, thereby greatly simplifying and optimizing the search process and realizing more possibilities. The integration of AI in research may provide greater assistance to researchers and help them delve into new realms of knowledge. Research by

[8] `https://52.223.1.88/projects?ref=studentship-2889694`

van Otterlo (2018) and an opinion paper by Dwivedi et al. (2023) on generative AI both explore and demonstrate the possibility of artificial intelligence enhancing research capabilities. Develop advanced search tools that achieve greater scalability with AI will bring more possibilities to research work and methods. It assists researchers in broadening their areas of thinking and achieving successful outcomes in their research endeavors. Capra et al. (2015) found that users interact more with search assistance tools for complex tasks, leading to better outcomes.

2.2 Digital Humanities and Historical Document Search: Evolution from Keyword Indexing to Multilingual Semantic Analysis and a New Idea

In the Information Age, the field of historical research has undergone a profound transformation. Traditionally, historians relied on physical archives and paper documents for their research, a process that was not only time-consuming but also inefficient. With the advancement of digital technologies, digital humanities emerged, enabling massive volumes of historical documents and records to be digitized and accessed electronically. Philips and Tabrizi (2020)This shift dramatically enhanced the scope and depth of research, allowing historians to quickly retrieve documents from vast databases, thereby accelerating the collection and analysis of information. For example, the "Europeana Collections"[9] project in Europe has digitalized millions of pieces of artwork, books, music, and other cultural artifacts, offering a cross-country search platform. Historians are now able to quickly find the historical documents they need by simple keyword searches, which can lower national and language barriers.

Originally, keyword searches for documents existed a long time ago, creating an index for all the words in the document and providing results based on a simple matching algorithm. These systems were similar to those described in early work on search engines Chowdhury (2010) and provided the foundation for later, more complicated information retrieval methods. To improve the relevance and ranking of search results, search engines incorporate statistical metrics such as TF-IDF and BM25, as described by Robertson and Jones (1976). These methods quantify the importance of words by considering their frequency in a document relative to their frequency in the document set, which is a significant advance beyond simple keyword matching. As technology evolved, it became evident that literal keyword matching was insufficient, especially for historical texts where linguistic variation is common. This inadequacy not only led to the integration of NLP techniques to manage the complexities of language, as outlined by Schütze et al. (2008), but also the rise of Semantic Web. The concept of the semantic web, introduced by Lassila et al. (2001), aimed to enhance the capabilities of the internet by enabling better data

[9] https://www.europeana.eu/en

connectivity. This idea laid the foundational philosophy for semantic search, which aims to understand the meaning behind words rather than just matching them.

To address the variations in word forms found in historical texts, and helping refine search queries to understand linguistic roots rather than literal expressions, techniques such as stemming, lemmatizating and tokenization come to our sight.Parsing (2009). Tokenizing is a key step in natural language processing, helping to complete tasks such as language recognition, part-of-speech tagging, and basic phrase chunking. This is one of the ways to process the complexity of languages. And another way to implement semantic search is using knowledge graphs. While knowledge graphs have proven valuable in a variety of applications, including graph mining and document modeling, Ma et al. (2023) and Rafiei-Asl and Nickabadi (2017) pointed out that unstructured text data (such as ARO corpus) may not be suitable for efficient storage in knowledge graphs.

Current Trends and Innovations Vector Search and Machine Learning Recent advancements have seen the application of vector space models and machine learning algorithms to enhance the semantic analysis capabilities of search systems. These models, particularly those based on neural networks like BERT and its variants, have redefined semantic search capabilities by allowing systems to understand the context and relationships between words in historical documents Devlin et al. (2018). Explore fine-tuned methods for text classification using BERT. Shi and Lin (2019) further demonstrated the effectiveness of BERT-based models in relation extraction and semantic role labeling, showing that simple BERT models achieve state-of-the-art performance without external features. BERT success in achieving these state-of-the-art results on various natural language understanding benchmarks is attributed to its pre-training on Masked Language Model (MLM) and Next Sentence Prediction (NSP) tasks Aroca-Ouellette and Rudzicz (2020).

Reimers and Gurevych (2019)proposed Sentence-BERT (SBERT), a modification of the BERT network that uses siamese and triplet network structures to derive semantically meaningful sentence embeddings, significantly reducing the time required for similarity comparisons, while maintaining accuracy. It was proven by Kalyan and Sangeetha (2021) that Sentence BERT models are more effective than BERT models in computing relatedness scores in biomedical concepts. Researchers have also explored the development of language-agnostic BERT sentence embeddings to facilitate cross-language transfer learning and semantic similarity tasks Feng et al. (2020).

It is very exciting to find a model that is so suitable for sentence semantic analysis, but it is a pity that the languages it does not support low-resource languages such as Scottish. So I had to turn my attention to multi-language models for semantic search. Therefore, I found mBERT and XLM-RoBERTa.

MBERT stands for Multilingual BERT and is the next step in creating models that understand the meaning of words in context. It is trained trained on 104 languages simultaneously and encodes the knowledge of all 104 languages together. Libovickỳ et al. (2019) explained that MBERT allows us to build models that can be applied to data in 104 languages out of the box, but it is not enough for more difficult task in some case.

In COVID-19 fake news detection in Persian, XLM-RoBERTa has been used as a deep cross-linguistic contextualized language model for developing accurate detection tools Ghayoomi and Mousavian (2022). Furthermore, XLM-RoBERTa has been applied to a large multilingual dataset for named entity recognition, demonstrating its effectiveness in NER tasks across different languages and domains Malmasi et al. (2022).

XLM-RoBERTa has been shown to outperform mBERT on various cross-language benchmarks, including achieving higher accuracy on low-resource languages Conneau et al. (2019). While XLM-RoBERTa shows promise over mBERT in some cases, there are also reports of performance issues with RoBERTa models compared to mBERT[10]. The choice between mBERT and XLM-RoBERTa may depend on the specific task and dataset, as both models have their own strengths and weaknesses Armengol-Estapé et al. (2021).Therefore, choosing which model to use for semantic analysis of ARO corpus requires real data to inform the answer, which is one of the problems that needs to be solved in this study.

Also, it is proven that the integration of keyword and vector-based search models has emerged as a solution to balance precision and depth in search results. Hybrid models leverage the strengths of both approaches to provide comprehensive and relevant search outcomes, especially useful in the varied and complex domain of historical documents Bhagdev et al. (2008). And semantic search in historical documents should be a dynamic and continually evolving field. Therefore, this study aims at combining traditional search methods with advanced computing technologies, combined with natural language processing and artificial intelligence technologies, can further enhance the way historical data is searched, analyzed, and understood, paving the way for more intuitive and effective digital humanities search tools. the way.

[10] https://github.com/stanford-futuredata/ColBERT/issues/12

Chapter 3

Design

3.1 Agile Development and Client Collaboration

In this project, as the only developer, I communicated closely with the client through regular meetings, established detailed requirements documents, and subdivided large projects into small, iterable tasks. First, I focused on designing semantic analysis algorithms to meet the needs of my clients - using machines to automate preliminary screening and reduce manual input time. Subsequently, I designed the product page and gradually integrated all functions to form a preliminary product prototype. During the development and integration process, I performed coding and unit testing according to predetermined small tasks, and completed integration testing. Faced with challenges such as slow iteration progress or increased customer demand, I promptly adjust plans and re-evaluate the achievability of tasks. At each critical stage, I will maintain communication with the client and make modifications based on feedback, while conducting regular assessments to ensure the smooth progress of the project. Although the project was not fully delivered as planned due to time constraints and technical implementation challenges, the entire development process strictly followed the agile development process.

3.2 Intended Audience

ARO Intelligent is primarily aimed at historians and digital humanities scholars who delve into the content of the corpus, providing them with a tool to assist their research. In addition, non-historical researchers interested in historical documents and cultural heritage are also potential users of the tool. This tool is particularly helpful for users who wish to explore multilingual resources, as it can break down language barriers and increase the breadth and depth of information access by searching for a keyword in English to find not only how the word is written in other languages, but also its synonyms and variations.

3.3 Requirements

3.3.1 Main Function

The primary functionality of the ARO Intelligent tool was centered around keyword search and semantic analysis for animal-related keyword. The implementation process was designed to meet the requirement from client. The following are the steps to implement keyword search and results reordering through semantic analysis:

- Synonym Dictionary Generation: The system use the OpenAI API[1] to dynamically generate a dictionary of synonyms for each search keyword. API will be introduced in the following section. This dictionary can significantly enriche the search capabilities, allowing for a broader capture of relevant data.
- Matching Search: After generating the synonym dictionary, the system performs matching searches within the database to find all relevant entries, enhancing the search breadth.
- Secondary Sorting Using Pre-trained Language Model: After the first search, results are re-ordered using a fine-tuned multilingual model. This model applies semantic analysis to reorder the search results based on their contextual relevance to the query, thereby improving the precision of search outputs.

This phase established a foundational search functionality that supported simple queries and provided highly relevant search results through semantic understanding.

3.3.2 Additional Functions

After successfully meeting the main requirements, in the developing process, the client introduced two additional functionalities to expand the scope of the ARO Intelligent tool. These functionalities were developed based on the established framework but some changes may need to be make to based on the requirement of client.

- Person Names and Variants Search: Similar to the animal keyword search, this function the use OpenAI API to generate a dictionary of name variants. However, semantic search may not need to be considered in this function implementation as all the results return can not be items that are not names.
- Marital Relationship Search: This feature focuses on searching for terms related to marital relationships. The implementation is similar to the previous two functionalities but tailored more specifically to client requirements, offering only three keyword options, "spouse," "husband," and "wife".

[1] `https://openai.com/index/openai-api`

3.4 Planning the Development Process

Data Processing and Analysis

This initial stage involves analyzing and processing XML data to prepare a clean and structured dataset suitable for model training and analysis. The focus here is on understanding the document encoding and extracting pivotal features.

- XML Data Analysis: Evaluate the structure of XML files to ascertain the organization and encoding of historical data.
- Data Transformation and Feature Extraction: Convert XML data into a more analytically usable format, such as CSV, extracting key elements like text content, dates, and metadata using Python and relevant libraries.

Model Selection

Choosing the right model is crucial for later semantic analysis. This stage evaluates two leading pre-trained language models mentioned in 2.2, mBERT and XLM-R, to determine their suitability for ARO corpus.

- Model Comparison: Test both models on the prepared corpus to assess how they handle the multilingual and diverse nature of the historical documents.
- Initial Performance Evaluation: Perform preliminary assessments to determine which model best captures the nuances of the historical texts.

Fine-tuning

After selecting the most appropriate model, it will be fine-tuned using Masked Language Modeling to optimize its performance for ARO corpus. The process of its principle will be introduced in the following sections.

Developing ARO Intelligent

With the prepared data and the fine-tuned model, the development of the ARO Intelligent tool is initiated.Setup the groundwork for integrating the model with a web application framework, database, and APIs, which will also be detailed in the following sections.

3.5 Algorithms and Technologies Chosen

In this section, an simple introduction to the algorithms and tools selected for model analysis and training and tool development will be given:

3.5.1 Model Analysis and Training

For this project, mBERT and XLM-R were selected as candidate models due to their robust performance in natural language processing tasks. The comparative analysis of these models

involved clustering algorithms and performance evaluation metrics to identify the most suitable model for further fine-tuning and deployment.

- K-means Clustering Algorithm

K-means[2] is a popular clustering algorithm used in unsupervised machine learning to partition n observations into k clusters where each observation belongs to the cluster with the nearest mean. It is initialized by selecting k centroids randomly. And then Assign each data point to the nearest centroid based on the Euclidean distance. For data point and centroid the distance is calculated as:

$$d(x_i, \mu_j) = \sqrt{\sum_{n=1}^{N} (x_{in} - \mu_{jn})^2}$$

Figure 3.1: K-means Clustering Algorithm

For updating, recalculate centroids as the mean of all points assigned to that its cluster. Repeat the assignment and update steps until convergence (i.e., when centroids do not change between iterations significantly). Formular for updating centroids shown in Figure 3.2, Where is the set of data points in cluster j. It is utilized to analyze the distribution and grouping of text embeddings produced by the candidate models (mBERT and XLM-R). By clustering these embeddings, it can provide a visual and quantitative evaluation on how well each model groups similar texts together, which is indicative of their effectiveness in capturing semantic similarities.

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

Figure 3.2: Updating Centroids

- Silhouette Score

This metric is used to assess the quality of the clusters formed by the K-means algorithm. It is a measure of how similar an object is to its own cluster compared to other clusters. A higher silhouette score[3] indicates better cluster purity and separation. Where:

[2] https://en.wikipedia.org/wiki/K-means_clustering
[3] https://en.wikipedia.org/wiki/Silhouette_(clustering)

1. a(i) is the average distance between i and all other data points in the same cluster.

2. b(i) is the minimum average distance from i to all points in any other cluster.

The silhouette score ranges from -1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters.

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

Figure 3.3: Silhouette Score

- Cosine Similarity

Cosine similarity[4] measures the cosine of the angle between two non-zero vectors in a multi-dimensional space, used to compare the embeddings(A and B in the formular (3.4)) between texts in semantic search. It assesses how the semantic relationships between documents are altered as a result of fine-tuning, directly reflecting the improvement of model of its understanding of the text.

$$\text{cosine similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Figure 3.4: Cosine Similarity

3.5.2 Technologies Chosen

Python

Python[5] is designed to support rapid development and clean, pragmatic design. In the 2023 IEEE Spectrum programming language rankings[6], it has been remain the most popular programming language in their general "Spectrum" ranking for many years in a row. It includes extensive

[4] https://en.wikipedia.org/wiki/Cosine_similarity
[5] https://www.python.org/
[6] https://spectrum.ieee.org/the-top-programming-languages-2023

range of libraries dedicated to data processing and application development, such as Pandas for data manipulation and PyTorch for machine learning tasks, which can be installed by pip command.

Development Environment

The main Integrated Development Environment (IDE) used for the development of this search tool is JetBrains PyCharm 2022.2[7]. Taking into account the occurrence of unexpected situations, Visual Studio Code[8] will also be used as a backup.

Model analysis and training

Jupyter Notebook[9] is used for model analysis and training due to its interactivity and ease of use in data science and machine learning tasks. It allows for the creation of literate programming documents that combine code, text, execution results, visualizations, and rich media Pimentel et al. (2019).

Hugging Face Transformers Library

Hugging Face Transformers[10] library provides a wide range of preprocessing tools that can be used for tokenization, text preprocessing in model analysis and training.

Web Framework

Django[11] framework was chosen to build ARO Intelligent because it is an efficient, flexible, and powerful web framework that provides rich features and easy-to-use development tools to quickly build robust web applications. It also provides detailed tutorials and active community support, allowing new developer like me to get started and solve problems faster.

Database

Use MySQL[12] database to store and manage system data. MySQL is a popular relational database management system that is stable and scalable and suitable for processing large-scale data sets. The MySql version supported by Django is 8.0.11 and above, so the one used in this development is 8.0.36.

[7] https://www.jetbrains.com/pycharm/whatsnew/2022-2/
[8] https://code.visualstudio.com/
[9] https://jupyter.org/
[10] https://huggingface.co/docs/hub/en/models-libraries
[11] https://www.djangoproject.com/
[12] https://www.mysql.com/

GPT-3.5-Turbo-Instruct

GPT-3.5[13] is an advanced natural language processing model capable of understanding and generating human-like text to provide users with high-quality search results. It is used to generate the synonyms dictionary for ARO Intelligent. Details on the subscription and retrieval of the OpenAI API key will be thoroughly explained in the appendix. For those seeking enhanced performance, switching to GPT-4[14] is an option. However, for the current application in developing the search tool, GPT-3.5-Turbo-Instruct was selected. Although it is not as sophisticated as GPT-4, it offers quicker responses. This choice is strategic for the development of this search tool, where speed is prioritized, and the differences in response quality between the models are marginal.

GPU

When fine-tuning the model selected for this project, a T4 GPU[15] provided by Google Colab[16] was utilized. This choice was motivated by the necessity for robust computational power capable of managing the extensive demands of training sophisticated language models. Employing a T4 GPU significantly expedited the training process, enabling quicker iterations and adjustments. This acceleration

[13] https://platform.openai.com/docs/models/gpt-3-5-turbo

[14] https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4

[15] https://colab.research.google.com/github/d2l-ai/d2l-tvm-colab/blob/master/chapter_gpu_schedules/arch.ipynb

[16] https://colab.google/

Chapter 4

Implementation

In this chapter, the implementation steps are divided into three main sections: data processing, model selection and fine-tuning, and the development of ARO Intelligent using the data and models obtained in the first two stages. At the beginning of each section, an overview will provide a clear roadmap of the processes discussed. To facilitate understanding and explaining the entire implementation process, a number of screenshots of the code and analysis results will be included throughout the chapter for better explanation of the development processes. This chapter is structured in the order of the real development process, as it was driven by the requirements of client. Therefore, it may be different from the typical order of the implementation part.

4.1 Data Processing

Data processing was the first step in development and involved extracting and converting the history records into a processable CSV file. Before starting to process the data, a detailed structural analysis was first conducted on it. Then, a Python script was used to process the TEI format XML file to extract key information of the historical document such as date, certification, text content, text ID and language used. The script was written using multiple Python libraries to implement parsing, data extraction, formatting and data saving functions.

4.1.1 File structure analysis

In this project, the core data source are XML files of ARO in TEI (Text Encoding Initiative) format, which describes historical documents using a standard encoding method. Through careful analysis, it was found that each XML file contains complete information of a single document, including text content, date, author, and other metadata, as shown in the Figure 4.1. Additionally, each document contains additional tags that identify the category or topic of each document, such as political, economic, or social. The XML files consist of eight volumes, with volumes 1-7 containing 369 items each, while volume 8 contains 111 items. The key information needed is mainly in the “text” part, which contains the actual text content and its related date and language

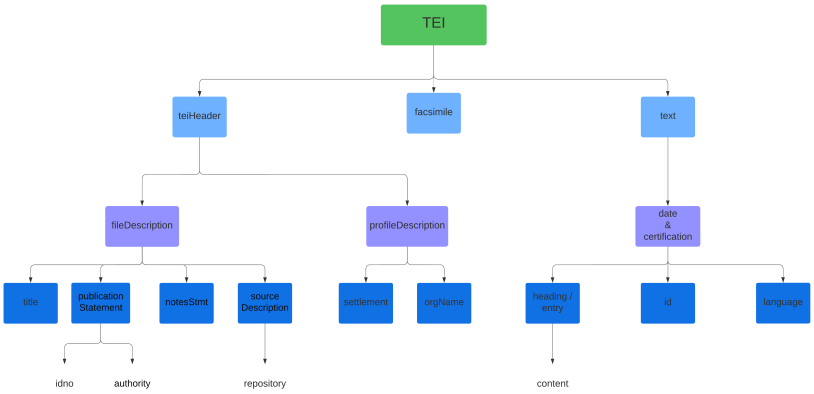


Figure 4.1: ARO XML files structure

attributes. This project needed to extract the following information from each entry:

- Date & Certification: Contains the date and credibility information of the text, which is of great value for historical research.
- ID & Language: The unique identifier of each piece of text and the language used. Extracting this information is the basis for subsequent text analysis and search functions.

4.1.2 XML Files Parsing

After analyzing the structure of data, a Python script was written to extract and transform the raw data. First, use the ‘etree’ module in the ‘lxml’ library to parse the XML file. In this way, the script was able to read the entire XML document and build it into a tree structure, allowing subsequent steps to query and locate specific elements through XPath expressions. This step was the basis for extracting data and ensures the correctness and effectiveness of subsequent operations. After getting the root node of the XML, the script defined a namespace dictionary because the XML file uses the TEI namespace. The script then looked for different categories of elements through predefined content types such as "court", "other", and "running". Each element found would be used as a starting point for data extraction.

For each element, the script extracted the associated date information, content, and any relevant metadata. This included handling different date attributes (such as “when”, “notBefore”, “notAfter”) to ensure that the information is correctly parsed and formatted even when the date data presents diversity (Figure 4.2. Additionally, the script processed nested elements within the elements, such as <head> and <p>, to capture additional textual content and annotations.

```
types = ["court", "other", "running"]
divs = []
for type in types:
    divs = root.findall(f'.//tei:div[@type="{type}"]', ns)
    # divs.extend(found)
    for court in divs:
    # Extract information
        date = court.find('.//tei:date', ns)
        if date is not None:
            if 'when' in date.attrib:
                date_val = date.get('when')
            elif 'notBefore' in date.attrib and 'notAfter' in date.attrib:
                date_val = f"between {date.get('notBefore')} and {date.get('notAfter')}"
            elif 'from' in date.attrib and 'to' in date.attrib:
                date_val = f"{date.get('from')}-{date.get('to')}"
            elif 'notBefore' in date.attrib:
                date_val = date.get('notBefore')
            else:
                date_val = "No date"
            date_cert = date.get('cert', "No cert")
        else:
            date_val = "No date"
            date_cert = "No cert"

        output_file.write(f"Date: {date_val}, Certification: {date_cert}\n\n")
```

Figure 4.2: Extract Information in

After completing the extraction of date information, the script further traversed the child elements in each to extract the text ID and language attribute used. Also, to streamline the data extraction process, the script employed regular expressions to clean and format the extracted text. This involved removing unnecessary metadata, such as author comments and timestamps, and standardizing the text format for consistency across entries to facilitate subsequent data analysis and storage.

Once the text extraction process was completed, the script saved the extracted data to a text file for interim storage and analysis. Subsequently, it transformed the extracted data into a structured format suitable for further processing using the Pandas library. This involved parsing the text file, splitting the data into individual records based on predefined delimiters, and organizing the data into a tabular format.

The final output of the implementation process was a CSV file containing the extracted textual data, organized into columns representing key attributes such as date, entry ID, language, content, and certification. This CSV file served as the primary dataset for subsequent stages of the project, including natural language processing and semantic analysis within the ARO Intelligent tool.

4.2 Model selection and fine-tuning

After the initial processing of the data, the next crucial step involved utilizing this prepared data to validate the performance of the two models. After determining the model, the data would also be employed to further fine-tune the selected model, aiming to enhance its performance.

4.2.1 mBERT VS XLM-R

In the process of model selection and evaluation, two cross-lingual models mentioned in 3.5.1, XLM-R and mBERT, were utilized to assess their performances in extracting embeddings and

conducting clustering analysis on a corpus of texts. The specific implementation involved extracting text embeddings using Hugging Face transformers library, applying clustering algorithms, and visualizing the results.

Embedding Extraction

To begin with, the AutoTokenizer and AutoModel[1] from Hugging Face transformers library were employed to extract text embeddings from pre-trained XLM-R and mBERT models. It is a general tool that automatically selects an appropriate tokenizer based on a given model name. For example, if it is provided with the model name "xlm-roberta-base", AutoTokenizer will choose XLMRobertaTokenizer. This process entailed encoding text into a format acceptable by the models and passing these inputs through the models to obtain embedding outputs, shown in Figure 4.3. The embeddings were calculated by taking the average of the last hidden states of each text sequence, commonly known as mean pooling. Figure 4.4 is a simple visualization of the process of mean pooling. The benefit of using mean pooling is that information from all words in the sentence or paragraph will be considered, rather than simply a subset. This makes it possible to use a single vector to represent the entire sentence or paragraph, facilitating subsequent processing and analysis.

```python
# extract embeddings
def get_embeddings(model_name, texts, max_length=128, batch_size=32):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)
    model.eval()  # set model to evaluation mode

    embeddings = []
    for i in range(0, len(texts), batch_size):
        batch_texts = texts[i:i+batch_size]
        inputs = tokenizer(batch_texts, return_tensors='pt', max_length=max_length, truncation=True, padding='max_length').to(device)

        with torch.no_grad():  # do not calculate gradients to save memory
            outputs = model(**inputs)
            batch_embeddings = outputs.last_hidden_state.mean(dim=1)  # use mean pooling
            embeddings.append(batch_embeddings)
            del inputs, outputs, batch_embeddings  # Free up memory
            gc.collect()

    embeddings_tensor = torch.cat(embeddings, dim=0)
    return embeddings_tensor

# use mBERT model to extract embeddings
mbert_embeddings = get_embeddings('bert-base-multilingual-cased', texts)

# use XLM-R model to extract embeddings
xlmr_embeddings = get_embeddings('xlm-roberta-base', texts)
```

Figure 4.3: Get Embeddings

Text data was initially loaded into a Pandas DataFrame and then transformed into a list format for batch processing. Each batch of text was converted into the required input format by the tokenizer, including attention masks and input IDs, ensuring uniform length or truncation to the maximum length. These inputs were fed into the model located on the GPU for accelerated computation, defaulting to CPU if GPU was unavailable. The computed embeddings were

[1] https://huggingface.co/transformers/v3.0.2/model_doc/auto.html

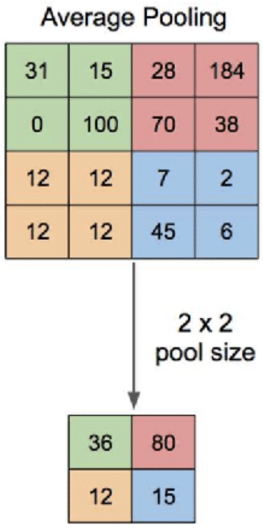


Figure 4.4: Mean Pooling

stored in a list for subsequent clustering analysis.

Cluster analysis and visualization

Clustering was performed using the K-means algorithm from the Scikit-learn library[2] and a clustering number of 5 was chosen (Figure 4.5). The selection of the number of clusters was determined by conducting an experimental preliminary analysis. This preliminary analysis consisted of trying different numbers of clusters and assessing the quality of the results in each case. The goal was to reveal the structure of the data as much as possible within a controllable number of clusters. The clustering algorithm was applied to the extracted embeddings to identify and group potential patterns and similarities in the text. After each clustering iteration, the silhouette coefficient, a metric measuring clustering quality, was computed to evaluate the tightness and separability of the clusters.

The visualization of clustering results was conducted using t-SNE[3] technique, an effective dimensionality reduction method particularly suitable for the two-dimensional representation of high-dimensional data. This step was performed post-clustering to map the high-dimensional embeddings into a two-dimensional space, facilitating the observation of spatial distributions among different clusters. Each cluster was represented by a different color for identification of individual groups.

[2] `https://scikit-learn.org/stable/`
[3] `https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding`

```python
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

def cluster_embeddings(embeddings, n_clusters=5):
    # use K-Means to cluster embeddings
    kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(embeddings)
    labels = kmeans.labels_

    # calculate silhouette score
    silhouette_avg = silhouette_score(embeddings, labels)
    print(f"silhouette score: {silhouette_avg:.2f}")

    return labels

def visualize_embeddings(embeddings, labels):
    # use t-SNE to reduce dimensionality
    tsne = TSNE(n_components=2, random_state=42)
    reduced_embeddings = tsne.fit_transform(embeddings)

    # visualize
    plt.figure(figsize=(10, 6))
    plt.scatter(reduced_embeddings[:, 0], reduced_embeddings[:, 1], c=labels, cmap='viridis', marker='.')
    plt.colorbar()
    plt.show()
```

Figure 4.5: Cluster analysis and visualization

Results

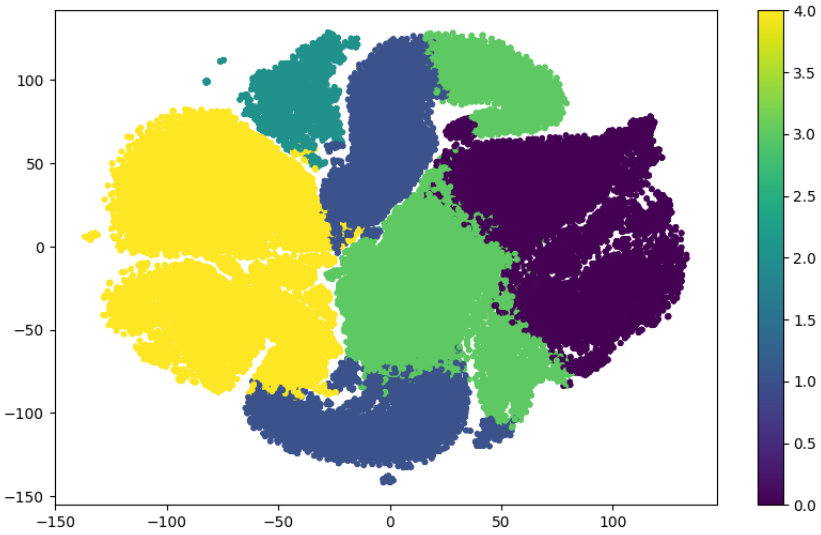


Figure 4.6: Clusters of XLM-R

Utilizing XLM-R model for extracting text embeddings and applying K-means clustering algorithm resulted in distinct clusters, as depicted in Figure 4.6. The clustering exhibited several clearly distinguished clusters, each representing a unique cluster in the embedding space. The boundaries of these clusters were well-defined, indicating high intra-cluster similarity and inter-cluster dissimilarity. The silhouette coefficient for the XLM-R model was 0.38, suggesting

relatively high quality and reasonable separability of the clusters. The silhouette coefficient[4], ranging from -1 to 1, measures the effectiveness of clustering, with values closer to 1 indicating more reasonable clustering. In this case, the silhouette coefficient of 0.38 indicated that the XLM-R model performs well in handling ARO text data.

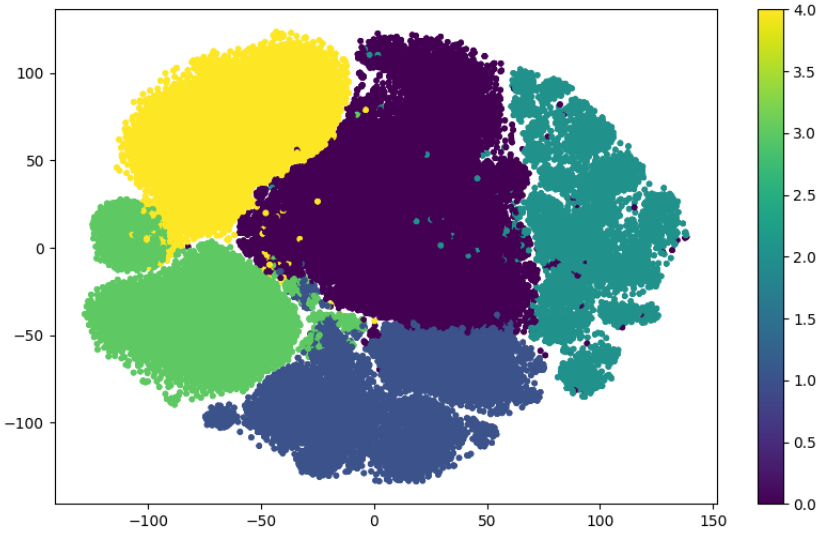


Figure 4.7: Cluster of mBERT

In comparison, conducting the same clustering analysis and visualization using embeddings extracted by the mBERT model produced the Figure 4.7. The distribution of these clusters exhibited more overlap and less distinctiveness compared to the XLM-R model, with less clear boundaries between groups. The silhouette coefficient for the mBERT model was 0.18, significantly lower than that of the XLM-R model, indicating lower clustering quality and insufficient separability between clusters. The lower silhouette coefficient implies limitations of the mBERT model in effectively clustering semantically similar texts.

In this comparative analysis, XLM-R model significantly outperformed mBERT model, which can be proven by its higher silhouette coefficient and clearer cluster boundaries. Therefore, the XLM-R model was selected for further fine-tuning and application development to optimize the text retrieval and analysis functionality.

4.2.2 Model Fine Tuning

After finalizing the selection of XLM-R as the optimal model used in ARO Intelligent, this subsection focuses on explaining further fine-tuning the model to adapt it specifically to the

[4]`https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html`

ARO corpus. The preparation for the fine-tuning process followed a similar methodology to the embedding extraction discussed in the previous sections. This model training was conducted using T4 GPU[5] provided by Google Colab[6] for accelerating the training process and dataset was uploaded on Google Drive for easier access. After mounting at Google Drive[7], the process began by loading the pre-processed data from the Google Drive. The data then was extracted them from the DataFrame and converted into a list format for further processing.

Similarly, as shown in Figure 4.8, the XLM-R tokenizer needed to be initialized before converting raw text into tokens suitable for model input. Different from the previous subsection where the AutoTokenizer was used, here the XLMRobertaTokenizer[8] was employed to convert raw text into tokens. This specific tokenizer was chosen because it is designed to work with the XLM-R model architecture, ensuring optimal compatibility and performance. Utilizing the padding and truncation functionalities of tokenizer ensured that all tokenized sequences had uniform lengths, which is crucial for maintaining consistency during batch processing in training. It is essential not only for the efficient functioning of the model on the GPU but also for achieving consistent performance across different text inputs.

```
from transformers import XLMRobertaTokenizer, XLMRobertaForMaskedLM, DataCollatorForLanguageModeling
from transformers import Trainer, TrainingArguments
import pandas as pd
from datasets import Dataset

# load data
df = pd.read_csv("/content/drive/MyDrive/Notebooks/dataset/processed_data_utf8.csv")
texts = df['Content'].tolist()

# initialize tokenizer
tokenizer = XLMRobertaTokenizer.from_pretrained('xlm-roberta-base')

# tokenize data
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

dataset = Dataset.from_dict({"text": texts})
tokenized_datasets = dataset.map(tokenize_function, batched=True)

# initialize data_collator
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=True, mlm_probability=0.15)
```

Figure 4.8: Data Preparation and Tokenization before Fine Tuning

Following the tokenization, the dataset was prepared using the Hugging Face dataset library[9], which is optimized for handling large-scale training data. It converted text list to Dataset object using Dataset.from_dict function from datasets library. This library facilitates batch processing and data management, speeding up the tokenization process by leveraging the parallel

[5] https://colab.research.google.com/github/d2l-ai/d2l-tvm-colab/blob/master/chapter_gpu_schedules/arch.ipynb

[6] https://colab.google/

[7] https://www.google.com/intl/en-GB_ALL/drive/

[8] https://huggingface.co/docs/transformers/en/model_doc/xlm-roberta

[9] https://huggingface.co/docs/datasets/en/loading

processing capabilities of the GPU. A data collator was then initialized to specifically prepare batches of training data for masked language modeling (MLM) [10]. In MLM, tokens were randomly masked with a probability of 0.15, which was a challenge for model to predict the masked tokens based on the context provided by the unmasked tokens. Take this as an example, As a pre-training technique for deep learning models in NLP, MLM works by randomly masking parts of the input tokens in a sentence and then asking the model to predict the masked tokens. The model is trained on large amounts of text data so that it can learn to recognize word context and predict masking tokens based on context. For example, in the sentence of Figure 4.9 "The boy is drinking is [MASK] milk", the model predicts the word "drinking" as a mask token using the unmasked words on both sides of the mask token. This process can help the model deeply understand the language context and structure.

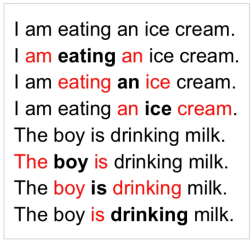


Figure 4.9: Masked Language Modeling

```
# initialize model
model = XLMRobertaForMaskedLM.from_pretrained('xlm-roberta-base')

# initialize training arguments
training_args = TrainingArguments(
    output_dir="./results",
    overwrite_output_dir=True,
    num_train_epochs=1,
    per_device_train_batch_size=4,
    save_steps=10_000,
    save_total_limit=2,
)

# initialize trainer
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=tokenized_datasets,
)

# train model
trainer.train()
```

Figure 4.10: Model Training Configuration and Execution

The XLM-R model was then loaded for fine-tuning. Specific training arguments were

[10] https://huggingface.co/docs/transformers/en/tasks/masked_language_modeling

configured, including setting the number of training epochs, batch size, and save frequency (Figure 4.10). The model was trained for a single epoch with a per-device batch size of 4. This configuration was set based on preliminary tests that indicated it offered a good balance between training speed and system resource utilization, ensuring that the model adapts to the ARO corpus without overfitting.

During training, the model iteratively optimized its parameters using the masked language modeling objective. It forced the model to predict masked tokens, thus enhancing its capability to reconstruct the original text and capture the underlying language patterns effectively. The training loss progressively decreased over iterations as shown in Figure 4.11, indicating the ability of model to reconstruct the original tokens and capture underlying language patterns was improving. Despite fluctuations in training loss, these are typical in the stochastic nature of training neural networks and the overall trend demonstrated consistent improvement in model performance. Following the completion of training, the fine-tuned XLM-Roberta model was saved as “xlmr-fine-tune model” for the development of ARO intelligent.

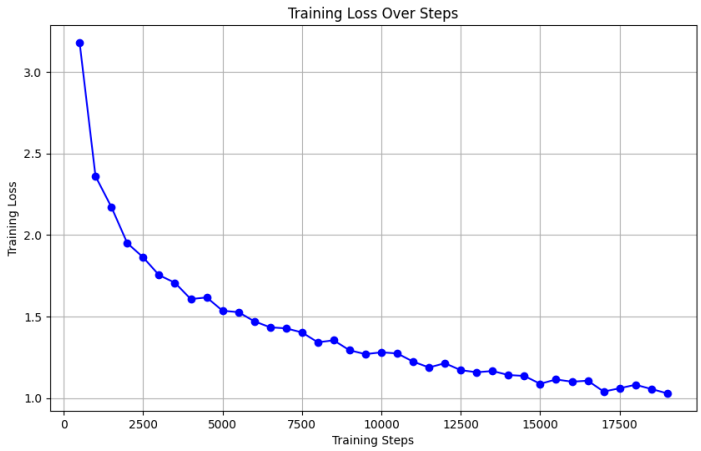


Figure 4.11: Training loss during the process of fine-tuning XLM-R

Model Performance Analysis

To ensure the effectiveness of subsequent feature developments for the ARO Intelligent, a simple performance analysis was conducted to validate the semantic matching capabilities of the fine-tuned XLM-R model. This analysis used the XLMRobertaTokenizer for tokenization, and the embedding extraction process was similar to that described in Figure 4.2.1.

A keyword "bull is a kind of animal" and a set of similar terms were defined for this analysis. These similar terms were categorized into three groups: one containing variants of "bull," a

second consisting of sentences where "bull" was used as a personal name within the corpus, and a third where "bull" referred to the animal. The embeddings of the keyword and these groups were then compared using the cosine_similarity function to assess the similarity between the keyword embeddings and those of the other terms.

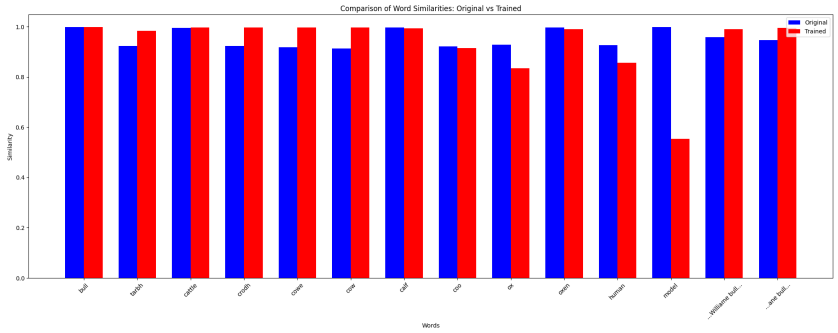


Figure 4.12: Similarities between Trained model and Untrained model

Seen from Figure 4.12, in the original model, the similarity scores for relevant terms such as "bull" and "cattle" were all high, comparable to those for unrelated terms like "human" and "model," exceeding 0.9. In contrast, the fine-tuned model, having been adapted to the corpus, demonstrated a significant decrease in similarity scores for terms unrelated to the keyword, while the scores for related terms either increased or remained at their original levels.

Analysis of specific sentence from the corpus that included the term "bull" showed a good result. Before fine-tuning, the model incorrectly assigned a higher similarity score to "bull" when used as a personal name (e.g., "...Williame bull...") than when it referred to the animal (e.g., "... ane bull..."). After fine-tuning, the result reversed. The similarity score for "bull" as a person name improved to 0.9903, and more importantly, for "bull" as an animal, it increased to 0.9958. The model now correctly attributed higher similarity scores to instances where "bull" referred to the animal, effectively reducing scores where it was used as a personal name. Although the similarity scores had improved overall—an outcome of fine-tuning the model to learn from the corpus—this change happened in similarity between "bull" used as a person name and terms directly related to the animal, with the former scoring lower. This analysis results Preliminarily proved that the fine-tuned model to enhance the sorting capabilities of the ARO Intelligent search function.

4.3 ARO Intelligent development

This section is going to explain the development of the core components of ARO Intelligent using the Django framework, with PyCharm 2022 serving as the Integrated Development Environment (IDE). During setup, a Virtualenv environment was automatically established when

creating this Django project, isolating project dependencies to provide a clean workspace for development. The initial setup began with the creation and configuration of the database, including URL routing, to prepare for subsequent developments. The focus then shifts to the implementation process of main functions, starting with the generation of the search dictionary. This step involved expanding keywords and constructing the dictionary. After introducing the search dictionary generation process, the discussion will move to the design of the integration and operationalization of functions within the system, followed by the development of the search interface. Upon establishing the basic matching search and framework, the fine-tuned language model discussed in Figure 3.5.1 is incorporated to enhance the functionality through re-ranking of search results. This integration improved the ability of ARO Intelligent to handle complex search queries more effectively.

Following the implementation of the initial major feature, two additional features were added to meet the second requirement of client, involving modifications to search conditions and matching rules. The analysis of the implementations in this section will be elaborated in the following chapter, which distinct from the previous chapters. This distinction is necessary because the previous analysis formed a crucial part of the development process of ARO Intelligent, ensuring that the model training outcomes were confirmed before further development can proceed. Therefore, it is important to include the analysis of model comparisons and model training in previous section before this implementation phase. The upcoming section Testing and Evaluation will analyze the functionality and performance of the developed search tool, while the current section mainly focuses on describing the implementation process.

4.3.1 Preliminary Setup

Database Design and Model Defining

The database creation was implemented using MySQL[11], with version 8.0.36 to ensure compatibility with Django, which supports MySQL from version 8.0.11 onwards. MySQL Workbench[12] was utilized to visually design and manage the database, simplifying the tasks of database creation, management, and maintenance. The corpus_data table was structured to store historical data, user queries, and search results, with fields for document date, document ID, language, content, and certification metadata, as shown in Figure 4.14. The varied nature of the date formats, including non-numeric characters and different format representations, necessitated the use of a VARCHAR data type for the date field. Based on preliminary data analysis, which indicated that entries typically did not exceed 2000 characters (Figure 4.13), the content field was set to VARCHAR(2000).

[11] `https://dev.mysql.com/doc/refman/8.0/en/`
[12] `https://www.mysql.com/products/workbench/`

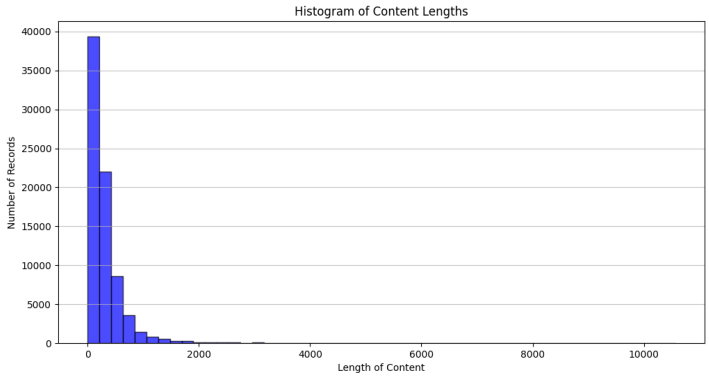


Figure 4.13: Histogram of Content Length

Once the database was set up, SQL scripts were executed to create the necessary tables. For instance, the corpus_data table was created with an SQL command to drop the table if it exists and then recreate it with specified fields and data types. Primary keys and indexing were strategically applied to improve query performance and data retrieval efficiency.

```
-- ----------------------------
-- Table structure for corpus_data
-- ----------------------------
DROP TABLE IF EXISTS `corpus_data`;
CREATE TABLE `corpus_data` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `corpus_date` varchar(20) DEFAULT NULL,
  `corpus_id` varchar(20) DEFAULT NULL,
  `language` varchar(20) DEFAULT NULL,
  `content` varchar(2000) DEFAULT NULL,
  `certification` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=107909 DEFAULT CHARSET=utf8;
```

Figure 4.14: Database Table Creation

Subsequently, data was imported into the newly created MySQL database using MySQL Workbench, which facilitated the insertion of data from CSV files directly into the database. This step was crucial for populating the database with initial data sets essential for the search functionalities. Finally, database connections were configured within search_project.settings.py in, specifying parameters such as database name, user, password, host, and port. This configuration linked the Django application with the MySQL database, enabling dynamic data interactions and operations essential for the functionality of ARO Intelligent.

In addition to databases, Corpus_Data models defined in models.py files are also important to enable efficient storage and retrieval of corpus data. It was designed by using these entries

according to the data structure of ARO corpus:

- Content: Stores the textual content of the corpus entry. This is where the actual data that will be queried against resides.
- Language: Indicates the language of the corpus entry, which is critical for processing queries specific to language contexts.
- Corpus Date: Contains the date associated with the corpus entry, allowing for temporal sorting and filtering.
- Corpus ID: A unique identifier for each entry in the corpus, facilitating efficient data retrieval and management.
- Certification: An additional field that could store information about the verification or status of the corpus entry, which might be useful in more controlled or curated datasets.

Configuring URL Routing

After successfully setting the database and model, the next step was to implement the simple matching search function with GPT API and design a draft interface to validate the search results. Before that, the initial setup involved creating a new Django application named 'searchapp' by executing the command 'python manage.py startapp searchapp' in the project environment. As shown in Figure 4.15, this established a structured project layout, including various functional components. Within searchapp/views.py, the initial view function, named 'index', was defined to serve as the gateway for user interactions, handling incoming requests and rendering the appropriate responses. To route requests to this view, a URL configuration file, urls.py, was created in the searchapp directory. It included the route path("", views.index, name="index"), which maped the root URL of the application to the index view function.

To ensure the integration of the searchapp with the main project, modifications were necessary in the search_project/urls.py file. An inclusion mechanism was created by adding an import statement for 'django.urls.include' and the 'include() function into the URL patterns list. This setup made it possible to manage URL routing by truncating matched parts of the URL at the inclusion point and forwarding the remaining substring for further processing by the included URL configuration. This strategy guaranteed that the application functions correctly, regardless of path complexity.

After configuring the URL routing, the application was tested by running 'python manage.py runserver'. This step was important to ensure that the basic setup was operational and that the application responded as expected at the initial endpoint. When the application was successfully launched and running, accessing the URL displayed a page where Django confirmed

```
search_project/
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
searchapp/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
    templates/
```

Figure 4.15: Project Files Structure

its successful running.

4.3.2 Basic Functions Developments

Synonyms Dictionary Generation Function

In the beginning of functionality development process, a function named generate_synonyms(Figure 4.16) was defined to generate synonyms and lexical variants via the OpenAI API interface. To enhance efficiency, two global variables, openai_response and openai_fetched, were employed. These variables helped cache the API response to avoid repeated calls to the same request within the same session, thus saving resources and reducing the number of calls. In the implementation of the function, it first checked whether an API response had already been fetched. If so, it directly returned the cached result. If not, the API key was set up and verified, and then the model engine GPT-3.5-turbo-instruct was specified. This engine is particularly suited for handling instruction-based queries and can accurately output the required data based on the given input. Details on the subscription and retrieval of the GPT-3.5 API key will be thoroughly explained in the appendix.

To ensure that the quantity of generated synonyms and variants were enough and accurate, the query prompts sent to GPT-3.5 API were constructed after many trials of testing. These prompts explicitly instructed the model to output synonyms and variants of keyword in list format, covering Scottish, Latin, Dutch, and English. It was also specified that the output should contain no fewer than 30 results to ensure it provide as many results as possible.

After calling the API, the returned data was parsed, and the text content was extracted and

```python
def generate_synonyms(prompt_text):
    global openai_response, openai_fetched

    # Check if the response is already fetched
    if openai_fetched:
        return openai_response

    openai.api_key = "sk-proj-cdoSl9ziYkxs5mQiK0eRT3BlbkFJUlhD6LeC86fqYyjkWg6H"
    model_engine = "gpt-3.5-turbo-instruct"
    prompt_text = prompt_text

    response = openai.Completion.create(
        engine=model_engine,
        prompt=prompt_text,
        max_tokens=2000,
        n=1,
        stop=None,
        temperature=0.5
    )
    output_text = response.choices[0].text.strip()

    # Cache the response
    openai_response = output_text
    openai_fetched = True

    return output_text
```

Figure 4.16: Synonyms Dictionary Generation

cleared of any excess whitespace. During this process, the autocast feature of torch.cuda.amp[13] was used to optimize memory usage and accelerate the processing. The obtained output results were stored in global variables for subsequent use. Finally, the synonyms and variants list output by the API was combined with the query keyword from input to form the final search dictionary. This dictionary was ready to apply to the search functionality of ARO Intelligent, providing a wider range of search results.

Functional Integration and User Interaction Design

Following the initial implementation of using the GPT-3.5 API for generating search dictionaries, further updates in the 'view' module within the 'searchapp' were performed to allow the better query handling optimization and search result display. These modifications were aimed at making the search functionality not only more responsive but also more intuitive for users.

The process begins by handling user queries through the 'index' view, where the application first checks if synonyms for a given query have already been generated and cached. This step is very important, as it avoids unnecessary API calls and hence saving on processing time and API usage costs. If the synonyms are not cached, the application interact with the GPT-3.5 API to fetch them, and these synonyms are then stored for future queries of the same nature.

Once obtaining the synonyms, they will be put to dynamic use in constructing the queries for database. This is implemented with the help of Q objects[14] of Django, which allows power

[13] https://pytorch.org/docs/stable/amp.html
[14] https://docs.djangoproject.com/en/5.0/topics/db/queries/

in creating flexible and powerful abilities for querying. By using these objects, the application can create complex filters that match the database entries against the list of synonyms. This approach ensures that search results were highly relevant to the query.

```
def highlight_keywords(text, keywords):
    for keyword in keywords:
        if keyword:
            keyword = escape(keyword)
            text = re.sub(
                f'({re.escape(keyword)})', r'<span class="highlight">\1</span>', text, flags=re.IGNORECASE
            )
    return mark_safe(text)
```

Figure 4.17: highlight keywords Function

To better the user experience, the highlight_keywords function (Figure 4.17) was implemented within the 'index' view. This function aims to magnify the prominence of the search terms in the search results. The style of highlighting must be applied with HTML tags to encase the search terms. This visual emphasis helps user to quickly catch keywords in results set, assisting their eyes for information scan and subsequent digestion.

```
paginator = Paginator(data, results_per_page)
page_number = request.GET.get('page')
page_obj = paginator.get_page(page_number)
previous_query = query
```

Figure 4.18: Paging using Paginator

These search results can also be further ordered according to user preferences, such as date or frequency. In addition, they are paginated in order to improve navigation through big sets of data. The Paginator[15] class in Django allows for pagination (Figure 4.18), where results, when returned from a query set, are broken into paged pieces. Hence, it increases the usability of the interface by not overloading the user with information all at once and made it easier to gain access to other pieces of information.

After the implementation of the backend functionalities and the integration of GPT-3.5 API for generating dynamic search dictionaries, next step was to design a frontend interface that would allow users to interact with the ARO Intelligent search tool. The frontend was designed to be both user-friendly and functional, encapsulating the core features of the search tool while providing a smooth user experience.

The HTML[16] structure for the search page was set up to include a form where users could

[15] https://docs.djangoproject.com/en/5.0/ref/paginator/
[16] https://www.w3schools.com/html/

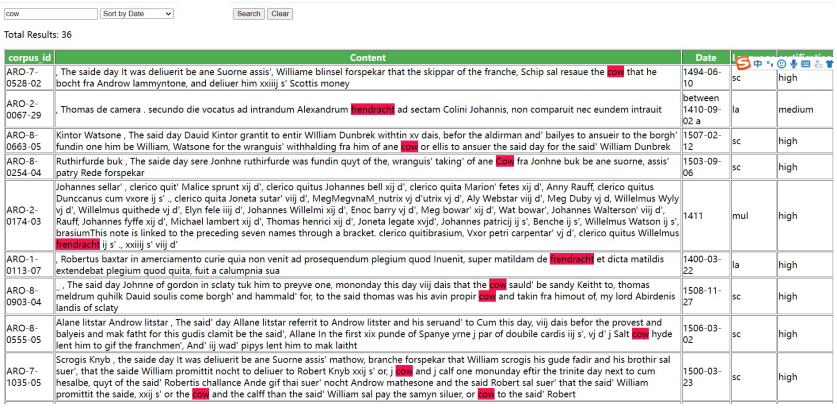


Figure 4.19: ARO Intelligent Prototype Interface

input their search queries and select specific search parameters such as sorting by date or frequency. The form was designed to validate user input to ensure that all necessary fields were filled before submission, enhancing the robustness of the user interaction. The form in the HTML file includes an input field for the search query and a drop-down menu for selecting the resort order. JavaScript[17] function were embedded within the HTML to handle form validation and clearing, ensuring that users received prompt feedback if they attempted to submit the form without filling out necessary information (Figure 4.20).

```
<script>
    function validateForm() {
        var searchInput = document.getElementById("searchInput").value;
        var queryType = document.getElementById("queryType").value;

        if (!searchInput || !queryType){
            alert("Please enter search keywords and select query type");
            return false;
        }

        return true;
    }
    function clearForm() {
        document.querySelector('form').reset();
        // Optionally, redirect to the search page without parameters
        window.location.href = "{{ request.path }}";
    }
</script>
```

Figure 4.20: JavaScript Function

After submitting the search form, the backend processes the query using the dynamic dictionary generation function discussed in Figure 4.3.2 and filtering mechanisms. The search results

[17] https://www.javascript.com/

are then displayed on the same page under the form. The results are presented in a table format, which includes crucial information such as corpus ID, content, date, language, and certification details. Each piece of content where keywords match the search query is highlighted thanks to the highlight_keywords function, which wraps matched terms in tags with a distinctive style to make them stand out.

To manage large sets of results effectively, a pagination section was also included in to adjust according to the number of pages required to display all results. Navigation links allow users to move between pages easily, and a custom form was added to jump directly to a specific page, enhancing the navigational experience. The first prototype of ARO Intelligent was then developed, as shown in Figure 4.19

4.3.3 Semantic Analysis and Reordering

After the basic features were developed, everything was ready for adding the fine tune model to apply semantic analysis on user queries. This is one of the main features that makes ARO Intelligent smarter than previous ARO search tool. Because semantic understanding requires providing the model with additional information beyond keywords to help it understand exactly what the user is looking for, an option to sort by similarity was added to the previously available sorting method options, thereby implementing the function of semantic analysis within it.

It started by appending descriptive text to the input from user. For instance, if the query is 'tiger', it might be enhanced to "tiger is a kind of animal." This enhanced query helps the model understand the context more accurately. The modified query is then tokenized and converted into an embedding using the pretrained XLM-R model. This embedding represents the semantic signature of the query, which is extracted into a vector form that can be mathematically compared to other vectors. Before embedding content from the corpus, the entries are filtered according to combined search criteria and initially annotated with a similarity score. This sets up each entry with a default similarity score, preparing them for later updates based on actual semantic comparison results.

For each content entry in the filtered data, the content is first normalized by removing punctuation and converting to lowercase to ensure that the text is uniformly processed, and then embeddings are generated for each piece of content. Furthermore, the cosine similarity between the query embedding and each content embedding is calculated. This calculation measures the cosine of the angle between the two vectors in a high-dimensional space, providing a scalar value that represents how similar the content is to the query in terms of semantics. This process is similar to the analysis of performance of fine-tuned model in Figure 4.2.2. Finally, all entries are resorted by their calculated keyword similarity, in descending order. This sorting ensures that the most relevant entries, i.e., those whose content is most semantically similar to the query,

```
# Construct the embedding of the content of the matching result and calculate the similarity
for dd in data:
    content = str(dd.content)  # Convert content to string type
    text = remove_punctuation(content)  # Remove punctuation and convert to lowercase
    if not text:  # Check if text is empty
        text = "blank"  # If empty, filled with the string "blank"
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
    with torch.no_grad():
        outputs = model(**inputs)
    text_embedding = outputs.last_hidden_state.mean(dim=1).squeeze().numpy()
    # Calculate similarity
    dd.keyword_similarity = np.dot(query_embedding, text_embedding) / (
                np.linalg.norm(query_embedding) * np.linalg.norm(text_embedding))

data = sorted(data, key=lambda x: x.keyword_similarity, reverse=True)
```

Figure 4.21: Construct the embedding and calculate the similarity

appear first in the search results. Correspondingly, an option to sort by similarity has been added to the front-end design so that users can choose to sort search results based on semantic similarity. To ensure that this new option is properly integrated with the backend, the JavaScript code has been updated to handle the new sorting parameters and ensure that requests are sent correctly to the server.

4.3.4 Additional Features

After completing the first feature, the client added two requirements of additional features, searches for person names and marital relationships, which needed to be integrated with the previous function. In the front end, selection options for "person name" and "spousal" methods were added. Similar to the first feature, these new functions utilize the GPT-3.5API to generate search dictionaries and perform matching searches in the database. However, due to different search conditions and matching rules, separate prompts for person name search and relationship search were defined, each requiring individual design and implementation.

To distinguish among the three search functionalities, a new parameter, 'query_type', was defined. Within the original section where the prompt for the GPT-3.5 API is defined, conditional statements were added. The condition related to the 'animal' prompt was included under the 'animal' category, and the other two conditions were defined for personname and spousal respectively. These three values were added to the frontend, allowing the tool to offer three selectable options that generate different prompts based on the search type. It enables the generation of varied dictionaries for use in searches by the GPT-3.5 API.

The prompt for the person name search is designed similarly to that for animals, based on an example provided by the client. The expected output format includes a request for the generation of name variants in four languages, with a minimum output count set to five to encourage the

creation of as many variants as possible. This setting was based on the common cases that name variants are typically less numerous than those for common animal terms.

```
if query == 'Wife' or query == 'wife':...
elif query == 'Husband' or query == 'husband':
    prompt = f"""My task is to input husband, to generate synonyms and variants in Scottish, Latin and
    English. The output format is [\'husband', 'husbande', 'maritus\'].
    All you need to do is return a list without any related text explanations. Please help me output {query}
    should not be less than 20, and the returned result type is must be list."""
    keywords = ['husband', 'husbande', 'maritus']
    keywords = eval(generate_synonyms(prompt)) + [str(query)] + keywords
elif query == 'Spouse' or query == 'spouse':
    prompt = f"""My task is to input spouse,
           to generate synonyms and variants in Scottish, Latin and English. The output format is
           [\'spous', 'spouse', 'sponsa', 'sponse', 'sponsam', 'sponsum\']. All you need to do is
           return a list without any related text explanations.
           Please help me output {query}, it is essential to include three languages.The quantity should
            not be less than 20, and the returned result type is must be list."""
    keywords = ['spous', 'spouse', 'sponsa', 'sponse', 'sponsam', 'sponsum']
    keywords = eval(generate_synonyms(prompt)) + [str(query)] + keywords
```

Figure 4.22: Prompts Defining Individually for Better Results Outcome

For the spousal relationship search, although similar in construction to the first two functionalities, it was categorized more detailed with only three options of keyword according to specific client requirements. The client wanted the ability to search directly for spouse as well as related terms wife and husband, since sentences containing these terms often convey information about marital or partnership relationships, allowing inference of marital connections between individuals. The prompts of these keywords were categorized (as shown in Figure 4.22) separately because attempts to generate variants and synonyms for one keyword along with variants for the other two did not yield satisfactory results. The desired outcome was a comprehensive list of synonyms and related terms for each keyword. When searching for 'wife', although the majority of results pertained to the keyword, they were not exhaustive, and other generated terms were similar to 'spouse' and 'women'. To achieve more comprehensive results, defining separate prompts for each category proved more fruitful, particularly as this tool is designed for researchers to thoroughly explore the corpus without missing any related vocabulary. As this functionality involves a narrower and more precise search scope with only three keywords: wife, husband, and spouse, a validation form in JavaScript restricts input to these terms for searches of the spousal type. Any input beyond these terms triggers an alert to remind users to stay within the specified search range.

User Interface Upgrade

After the main functionalities were implemented, several enhancements were added to make the tool more user-friendly. Initially, the implementation of JavaScript and Cascading Style Sheets (CSS), which were embedded directly in the HTML document, were extracted and relocated

to a newly created static folder. This restructuring provided the better management and easier integration of additional features.

One of the first enhancements was the implementation of a divided page layout. Although the tool was originally designed without pagination to offer a seamless user experience, it became necessary to distinguish more clearly between the search and result sections for aesthetic reasons. This was achieved by dividing the page into two sections. Search page was designed as shown in Figure 4.23.A 'scroll to result' function was defined in JavaScript, enabling precise scrolling to the start of the results section. To enhance navigation, two clickable navigation dots were added to the right side of the page, which dynamically change in size depending on the section being viewed, adding a smooth transition effect when clicked. Moreover, a button for scrolling down was added below the search section to facilitate easy transitions between sections. To further improve user interaction, a function was added to the JavaScript to check if the 'keyword' or 'query type' fields were empty and alert the user if so. In addition to the functional improvements to the interface, some stylistic improvements have also been introduced. Not only have the color design been changed, but the buttons have also been added with a dynamic effect to make them more interactive to further enhance the user experience.

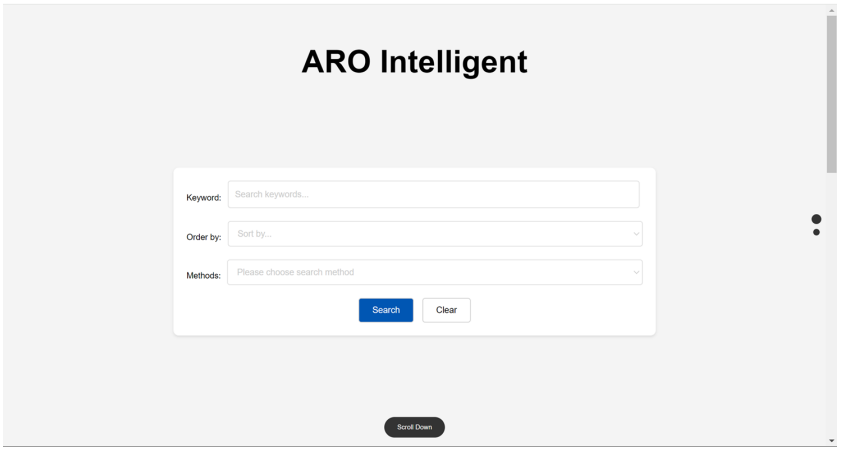


Figure 4.23: Search Page After Upgrade

The second interface upgrade involved modifying the number of results displayed per page. Although pagination had been implemented in the ARO Intelligent prototype 1, it did not allow users to customize the number of results per page. A new feature was introduced where users could define their preferred number of results per page via a form placed above the results table. A corresponding parameter was also defined on the backend to handle this input. To ensure the input values were valid, functionality to check the 'results per page' input was added into the JavaScript design (4.24).

```
// Check for keyword input
if (!searchInput) {
    alert("Please enter search keywords");
    formIsValid = false; // Set to false if validation fails
}

// Check for query type selection
if (!queryType) {
    alert("Please select a query type");
    formIsValid = false; // Set to false if validation fails
}

// Additional validation for 'spousal' query type
if (queryType === 'spousal') {
    var allowedInputs = ['wife', 'husband', 'spouse'];
    if (!allowedInputs.includes(searchInput.toLowerCase())) {
        alert("For 'spousal' type, please enter one of the following options: wife, husband, spouse.");
        formIsValid = false; // Set to false if validation fails
    }
}

// Check for results per page input
if (!resultsPerPage || resultsPerPage < 1) {
    alert("Please enter a valid number of results per page.");
    formIsValid = false; // Set to false if validation fails
}
```

Figure 4.24: Validate From

In addition to these enhancements, a function for exporting results to Excel was also added. This was facilitated by creating an HTTP response object that specifies the content type as an Excel file. Using the Pandas library ExcelWriter[18] with openpyxl[19] as the backend engine, data is written into the previously created HTTP response object before being returned to the user. Lastly, a 'Clear' button was additionally added at the bottom of the results page, which was the same functionality of the one on the search page. This button allows users to clear the search input and return to the search page with a single click, eliminating the need to manually scroll back to the top.

4.4 Another try of generating synonyms dictionary

[18] https://pandas.pydata.org/docs/reference/api/pandas.ExcelWriter.html
[19] https://openpyxl.readthedocs.io/en/stable/

Chapter 5

Testing & Evaluation

5.1 Testing and Evaluation

Following the completion of application development, it is crucial to assess whether the functionalities have been successfully implemented and how they perform in practice. In this section, the testing and analysis of the ARO Intelligent search tool are divided into two parts: function testing and performance analysis. While individually verifying that the developed functionalities operate as envisioned, an analysis of the results returned by these functionalities is also conducted. This analysis is used to evaluate the performance of the AI technologies and NLP tools utilized in the development of ARO Intelligent.

5.1.1 Functions Testing

Functional testing for this tool will be conducted by further classifying it into three key areas: Animals Retrieval, Person Name Retrieval, and Spousal Relationship Retrieval, along with three sorting functions based on date, frequency, and similarity respectively. After selecting a category function, each sorting function will be tested in sequence to observe their operational success before moving on to other functionalities such as pagination and results exporting. The focus will be on testing the functionality and result analysis of sorting by similarity. The reference samples in Figure 5.1 for this test are compiled by historical researchers manually browsing the corpus for specific keywords, provided by the client. These are for 'cow' (animal test), 'Alexander Anderson' (person name test), and 'spouse,' 'wife,' or 'husband' (marital relationship test). The standard for judging the success of the search tool's functionality for 'cow' includes covering most situations in the reference sample and ideally expanding upon them.

5.1.2 Animals Retrieval

Firstly, the Animals category is searched using date and frequency sorting functions, both of which operate without any issues. When searching by frequency, the entries with the most

Animals	Person name	spouse
cow cowe kow bull oxen ox bovis(bovines) bos vacca(s) stirk nowt vacce vitello cattall	Alexander Anderson =》Alexander Andree (Latin) =》Alexander Androson (Scot)	spous spouse sponsa sponse sponsam sponsum

Figure 5.1: Testing Sample Provided by clients

keyword count successfully appear at the top, displayed in decreasing order of frequency. Simultaneously, multiple keywords, including the searched keyword and its synonyms, can be highlighted in a single entry, as they may appear together in one entry, as shown in the Figure 5.2.

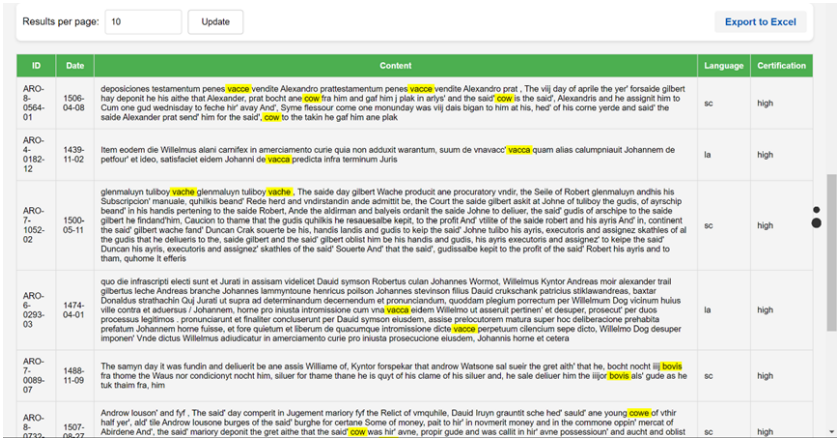


Figure 5.2: Animal Searching By Frequency

Next, the output results when sorting by similarity are tested. When the keyword 'cow' is entered, the first page shows entries containing the keyword and its synonyms, which are related to the use of keyword to denote an animal. Since the system is defaulted to sorting by date when no sorting method is selected, this can be used for comparison. The sorting results can validate to some extent the reordering effect of the similarity sorting in animal keyword searches, confirming the successful implementation of this function. Further verification confirms that the synonym dictionaries generated by the three different sorting methods are the same, ensuring that the data used in this comparison is consistent, and there is no artificial inflation of sorting effectiveness due to different materials. Shown in the command line (Figure 5.3), the generated synonym dictionary is observed, encompassing various synonyms and variants of cow in three

languages, successfully covering most examples provided by the customer and offering different language variants of the keyword, providing more possibilities for searching.

Figure 5.3: Synonyms Dictionary of cow

5.1.3 Person Name Retrieval

After testing the animals category, the person name search testing continues. This step is much simpler than the animals category test because functions like sorting by similarity are not meaningful for person name searches, so they can be ignored initially, ensuring that they can be implemented like other functions. From the results sorted by date and frequency, the generation of a variant dictionary through the GPT-3.5 API increase the possibilities for faster retrieval. Seen from the results page (5.4), besides the examples provided by the customer, an additional name variant, Alexander Andreas, is also found, and other variants not present in the corpus content are discovered in the generated variant dictionary, as shown in the Figure 5.5, which is a very exciting finding.

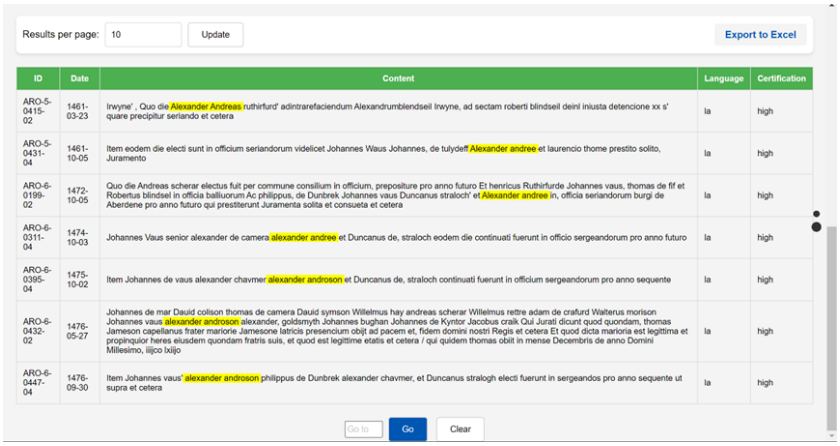


Figure 5.4: Results Page of Searching Alexander Anderson

Figure 5.5: Synonyms Dictionary of Alexander Anderson

5.1.4 Spousal Relationship Retrieval

Finally, testing moves to the spousal relationship search functionality. Two unexpected findings occur in this test. The first is in vocabulary expansion; for example, searching with the keyword

```
[' spous ', ' spouse ', ' sponsa ', ' sponse ', ' sponsam ', ' sponsum '
, ' husband ', ' wife ', ' partner ', ' mate ', ' companion ', ' better
half ', ' helpmate ', ' beloved ', ' consort ', ' mate ', ' significant
other ', ' soulmate ', ' spouse ', ' wedded partner ', ' spouse ']
```

Figure 5.6: Spouse Dictionary

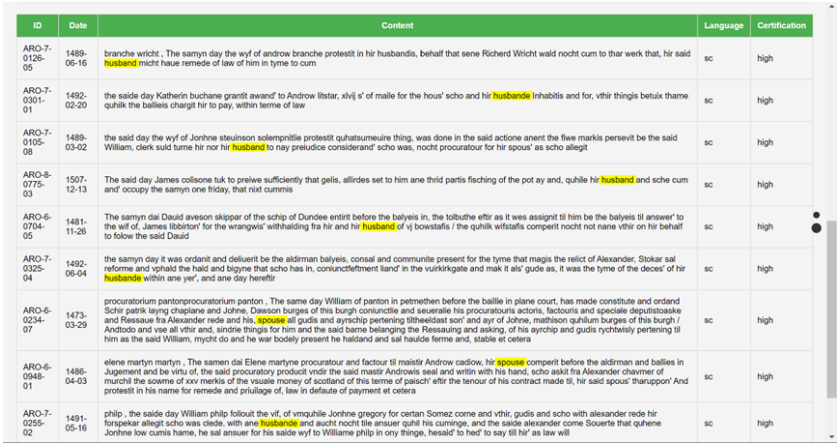
spouse not only generates the keyword and its synonyms but also spontaneously adds the other two search options, husband and wife, to the dictionary (Figure 5.6). The same situation occurs when searching husband, even though this was not requested in the prompt. The remaining results are synonyms or variants of spouse, with no excessive expansion in generating husband or wife variants, consistent with the implementation results predetermined when designing the prompt, as mentioned in section 4.3.4.

The second surprising test result comes when sorting by similarity, where the performance of model is unexpectedly good. When searching husband, the generated dictionary also includes words like man. When results are displayed in the order of the original corpus, entries containing man are shown on the first page. When switched to sorting by similarity, these contents are moved to the back of the search results, replaced by content related to marital relations. Even the husband variant husbande is recognized by the model and placed in the second position. To double-check, the content ranked in front, after translation through Google, is verified to be related to marital relations and not just simple keyword matching, reaffirming the pre-trained model's improved capability in semantic understanding. These result comparisons can be seen in Figure 5.7 and 5.8.

ID	Date	Content	Language	Certification
ARO-1-0002-03	1398-10-02	acquietauitacquietauit . Adiungitur Johanni Crab famulo andree petri ad suam acquietanciam in die lune infra octabas, huius curie quod non perturbauit Johannem Man . Eciam adiungitur eidem Johanni ad suam acquietanciam eodem die fiendam quia non obediuit clienti suum officium exercenti.	la	high
ARO-1-0006-07	1398-10-28	Johannes Man primo die vocatus ad sectam patricij Crane non comparuitunde precipitur seriandis capere districtionem octo solidorum et citare ipsum ad crastinum diem	la	No cert
ARO-1-0006-09	1398-10-29	Actio inter Johannem Man et patricium Crane de vna vacca differtur vsque ad aduentum vicecomitis propter causam	la	high
ARO-1-0080-02	1399-1400	fratres gildefratres gilde , Willelmus burnet remittitur ad requestam_pro_ebus et vino, quitus Johannes filius Henrici v s' iij ob', Matheus balram finiuitxx s' v_, quitus Willelmus Kyntor v s' iij ob', quitus Johannes boss' v s' iij ob', Johannes Man xl s'seipso plegio, quitus Willelmus blyndcele v s' iij ob', quitus Johannes strang v s' iij ob'	la	medium
ARO-1-0165-03	1400	Andreas baxtar facIet Johannem Man solui de duobus bobus precij ij marcarum et dimidie celdre farine 2m, quod deuenit plegius	la	low
ARO-1-0122-07	1400-06-29	Johannes Malvyn . vocatus legittime ad prosequendum plegium . quod Inuenit super Robertum baxtar non comparuit dictus Robertus extendebat plegium quod quitus fuit a, sua calumpnia , et Johannes Man in amerciamento quia fuit plegius pro dicto Johanne ad intrandum ipsum quod non fecit .	la	high
ARO-1-0216-02	1401	ReuerenceThis entry is written across the page. andReuerence honour likit yhu to wit . that the lord of Keth arestit yhur wyn and yhur oxin and for gude caus' as he lete vs wit . and for yhur, sakis we made hym request that he suld frely delyuer thaim . for the quilk request he has delyuerit thaim frely at this tym / for we ar' thai at wald at, gud acord' war' betwex yhu and hym . and wil do our besymes to bryng it thar to . at our power' . at the quilk acord' he sayis he wald, be gladly and sal nocht leve in his defaute . Qwarfor der' lord it is our consale and we requir' yhu that for ess' of the contre ane quiet of our, place . yhe wald asith gif yhe ocht aw hym , sa that hym nedit nocht in tyme to cum / til mak sic pundyng and namly in our tovn , for he, says it is previt dete that yhe aw hym and of lang tyme by gane . and gif yhe will adress' yhu to be at ony day with hym for the, knawlage of the forsaid thyngis sendis vs word / and we sal late hym wit and gif it langis ansuer' we sal ger' send it yhu . for we ar' richt, mykil haldyn to yhu and als til hym . god kepe yhur estate as we desir'	sc	low
ARO-1-0186-	1401-07-04	Willelmus de camera pater Willelmus andree , Simon lamb Johannes ledale Johannes ayncroft Alanus, Jacobi Andreas petri Thomas lownan henricus beset Dunccanus de Mernys gilbertus Kynros Willelmus, modan Walterus Andree Robertus de narn Johannes Man henry Rothny Johannes de Wellis bricius,	la	high

Figure 5.7: 'Husband' Search Results Sorting by Date

In addition to the main search functionality tests, other supporting functionalities and performance metrics like response times are also tested. The first test conducted is when no keyword



ID	Date	Content	Language	Certification
ARO-7-0126-05	1489-06-16	branche wricht , The samyn day the wyf of androw branche protestit in hir husbandis, behalf that sene Richerd Wricht wald nocht cum to thar werk that, hir said husband micht haue remede of law of him in tyme to cum	sc	high
ARO-7-0301-01	1492-02-20	the saide day Katherin buchane grantit awand' to Androw Ilstar, xlvlj s' of maile for the hous' scho and hir husbande Inhabitis and for, vthir thingis betuix thame quhilk the ballieis chargit hir to pay, within terme of law	sc	high
ARO-7-0105-08	1489-03-02	the said day the wyf of Jonhne steuinson solempnitlie protestit quhatsumeuire thing, was done in the said actione anent the fiwe markis persevit be the said William, clerk suld turne hir nor hir husband to nay preiudice considerand' scho was, nocht procuratour for hir spous' as scho allegit	sc	high
ARO-8-0775-03	1507-12-13	The said day James colisone tuk to preiwe sufficiently that gelis, allirdes set to him ane thrid partis fisching of the pot ay and, quhile hir husband and sche cum and' occupy the samyn one friday, that nixt cummis	sc	high
ARO-6-0704-05	1481-11-26	The samyn dai Dauid aveson skippar of the schip of Dundee entirit before the balyeis in, the tolbuthe eftir as it wes assignit til him be the balyeis til answer' to the wif of, James Ibbirton' for the wrangwis' withhalding fra hir and hir husband of vj bowstafis / the quhilk wifstafis comperit nocht not nane vthir on hir behalf to folow the said Dauid	sc	high
ARO-7-0325-04	1492-06-04	the samyn day it was ordanit and deliuerit be the aldirman balyeis, consal and communite present for the tyme that magis the relict of Alexander, Stokar sal reforme and vphald the hald and bigyne that scho has in, coniunctfeftment liand' in the vuirkirkgate and mak it als' gude as, it was the tyme of the deces' of hir husbande within ane yer', and ane day hereftir	sc	high
ARO-6-0234-07	1473-03-29	procuratorium pantonprocuratorium panton , The same day William of panton in petmethen before the baillie in plane court, has made constitute and ordand Schir patrik layng chaplane and Johne, Dawson burges of this burgh coniunctlie and seuerallie his procuratouris actoris, factouris and speciale deputistoaske and Ressaue fra Alexander rede and his, spouse all gudis and ayrschip pertening tiltheeldast son' and ayr of Johne, mathison quhilum burges of this burgh / Andtodo and vse all vthir and, sindrie thingis for him and the said barne belanging the Ressauing and asking, of his ayrchip and gudis rychtwisly pertening til him as the said William, mycht do and he war bodely present he haldand and sal haulde ferme and, stable et cetera	sc	high
ARO-6-0948-01	1486-04-03	elene martyn martyn , The samen dai Elene martyne procuratour and factour til maistir Androw cadiow, hir spouse comperit before the aldirman and ballies in Jugement and be virtu of, the said procuratory producit vndir the said mastir Androwis seal and writin with his hand, scho askit fra Alexander chavmer of murchil the sowme of xxv merkis of the vsuale money of scotland of this terme of paisch' eftir the tenour of his contract made til, hir said spous' tharuppon' And protestit in his name for remede and priullage of, law in defaute of payment et cetera	sc	high
ARO-7-0255-02	1491-05-16	philp , the saide day William philp follouit the vif, of vmquhile Jonhne gregory for certan Somez corne and vthir, gudis and scho with alexander rede hir forspekar allegit scho was clede, with ane husbande and aucht nocht tle ansuer quhil his cuminge, and the saide alexander come Souerte that quhene Jonhne low cumis hame, he sal ansuer for his saide wyf to Williame philp in ony thinge, hesaid' to hed' to say till hir' as law will	sc	high

Figure 5.8: 'Husband' Search Results Sorting by Similarity

or search type is entered, prompting an alert like "Please select a query type" (Figure 5.9) to remind users that these parameters need to be defined and a selection is necessary for the search to proceed. However, proceeding with a search without setting a sort type is allowed, and it performs as designed, sorting results by default date as expected. Similar tests are applied when entering keywords for marital status searches that are out of defined range and when the number of results per page is not a positive number, with good outcomes. Besides alert testing, directly clicking to switch result pages or entering a page number to jump to is feasible. Every button included on the page responds correctly, whether exporting search results in Excel format or switching pages through navigation points or scroll down buttons.

Figure 5.9: Alert of Validating Query Type

Performance Analysis of ARO Intelligent The initial performance analysis starts with response times. Sorting by date or frequency is typically fast, usually completed within ten seconds, because it only involves simple dictionary generation and retrieval sorting. This is why the GPT-3.5 API was chosen. Since model processing is required, choosing similarity for sorting takes a bit longer than ordinary date or frequency sorting, but generally does not exceed three minutes. Another factor affecting response speed is the number of synonyms and variants generated by the predefined prompt in the GPT-3.5 API; if set to a high value, it increases response

times but also broadens the range of search results, significantly saving time of users compared to their manual search and compilation of search dictionaries. This is much more efficient than fast sorting by date and frequency because users can first browse through entries with higher similarity, finding entries relevant to the intended meaning of their search keywords, not just spelling similarities.

The second important performance aspect of ARO Intelligent is its vocabulary expansion. During previous function testing, it was found that the synonym dictionary generated by the GPT API might differ upon re-searching, possibly adding or replacing a word. This can be seen as both a disadvantage and an advantage. The downside is that it cannot guarantee that users will receive the same search results every time, meaning that if users do not save or record their search results, they might not be able to reproduce the same search. However, the advantage is that it provides more reference possibilities for users, as each search might yield new results not found in previous searches, which they can later compile. As with the longer response times for similarity sorting, at least this significantly improves efficiency and time over manual searching and filtering. This is precisely the main purpose of developing this search tool: to leverage more advanced tools and technology to increase the possibilities of search results, reducing the difficulty and barriers to researching the ARO corpus.

5.1.5 Known Issues

During the design phase, I experimented with implementing a sliding location feature to ensure that the position remains unchanged when paging through or refreshing the page. However, because I did not anticipate future upgrades and redesigns of the page layout during the design of this search function, I did not design the two pages separately. Moreover, this decision was also based on the desire not to disrupt the user's seamless experience due to page navigation. The problem that arose is that when users try to paginate or click buttons, the page does not return to the results page, requiring them to manually scroll down again. Another issue is that the principle behind the implementation of the similarity function differs from the other two categorization functions, leading to malfunctions when exporting data to Excel. As a result, users can only view the results sorted by similarity on the web page.

Chapter 6

Discussion & Limitations

6.1 Discussion

The introduction of the NLP-powered search tool has significantly enhanced the accessibility and analysis of historical documents in the Aberdeen Registers Online (ARO). The implementation of semantic search technologies, particularly through the XLM-RoBERTa model, allowed for more nuanced and context-aware search capabilities.

The tool demonstrated a higher precision in returning relevant documents compared to traditional keyword-based search systems. Enhanced User Experience: Users reported an easier and more efficient search process, facilitated by the ability of the tool to understand and process complex queries involving historical language and spelling variants.

Comparison with Existing Literature The effectiveness of NLP models in handling linguistic complexities aligns with existing research that underscores the potential of AI in enhancing digital humanities. Similar studies have indicated that semantic technologies can bridge the gap between digital archives and end-users by simplifying interactions with complex datasets. However, this project uniquely applied these models to a multilingual historical database, extending the utility of NLP in historical research beyond what was previously documented.

Implications of the Research

Demonstrates a viable pathway for integrating advanced NLP techniques into historical research, suggesting a shift from manual archival methods to more sophisticated, technology-driven approaches. Offers a blueprint for other digital humanities projects that aim to enhance the accessibility of historical documents.Provides insights into the adaptability of NLP models in dealing with non-standard language and multilingual content, which could influence future developments in the field of computational linguistics.

6.2 Limitations And Future Work

In the example provided by previous researchers, nowt was classified as cow in the search results. It is derived from Middle English nowte, a dialect of Scotland and northern England. Most dictionaries will give the translation result as "nothing", although it has the meaning of "an ox" and "a herd of cattle". It is more commonly used to describe people who are like stubbron, crass, or clumsy like cattle. , resulting in the inability to be classified as a synonym of cow. Its alternative form and nolt also appear in the corpus, but it is considered to be dialectal or obsolete. Without in-depth study of its etymology, it cannot be connected with cow. It is more directly translated as a human name. . Words like this that are "hidden" due to time changes are difficult to find and classify using existing technology. Because the spelling of Scots was still very much in flux from the late fourteenth to the early sixteenth century.

In the functionality of sorting based on similarity, although it achieves good sorting effects, helping users search for content containing desired keywords more quickly and accurately, it can effectively sort in most cases. However, it is difficult to avoid a few instances of unexpectedly poor sorting results. For instance, when searching for "bull," even though the embedding values of entries containing "ane bull" (a bull) showed higher similarity to the search query than entries where "bull" is used as a surname during the model fine-tuning and analysis process, normally, the same sorting results should occur when the model is integrated into this tool. Yet, there have been one or two instances where entries with "bull" as a surname were ranked above "ane bull." Thus, the sorting by similarity can achieve smarter searching and sorting than existing tools to some extent, but its effectiveness is unstable. This instability is the result of not considering more scenarios during the model fine-tuning, which will require further analysis of the corpus content and more targeted fine-tuning of the model in the future. It may also be necessary to consider several different scenarios again and modify the settings of the prompt to make it more comprehensive to handle various situations.

Scalability needs to be enhanced as this tool cannot be applied to other corpora. Although it provides a new perspective for the development of search tools, in addition to continuing to fine-tune the model, there is a hope to make the search function more flexible, not limited to pre-defining the type of keywords to search. Consider adding a feature that allows users to enter prompts, such as wanting to search for content with "Lamb" as a surname, users can inform the system that the entered keyword should be a personal name, rather than just limited to searching for animals, names, and marital relations. Users could even enter prompts themselves, using more advanced technology to extract, for example, the keywords and their attribute classifications contained in a sentence entered by the user. This process can be completed automatically based on the input of user, without needing to be predefined. This was an aspect not considered during the development of this tool because the initial client requirement was simply to search

for animal-related terms for research work. Moreover, due to time constraints, it was desired that the tool be delivered promptly, so more complex and broader query processing was not considered.

If more accurate searching is desired, one might consider the supervised or semi-supervised learning mentioned in the background introduction on semantic search, by annotating the corpus to help the machine understand the content of the corpus more quickly and accurately. However, this might require a significant amount of time and human resources. It cannot be automatically recognized by existing annotation tools due to many spelling mistakes and missing words in the corpus. Although it has been initially annotated using existing tools, it still requires manual re-checking if a higher accuracy rate is desired.

Chapter 7

Conclusion

7.1 Project Summary

In this project, I successfully developed a search tool designed to improve the retrieval of historical documents from the Aberdeen Registers Online ARO. The tool incorporates advanced NLP technologies, including XLM-RoBERTa models, to effectively handle the complexities of historical texts, which often contain varied spellings and are written in multiple languages.

Key accomplishments of the project include:

- Implementation of Semantic Search: By integrating semantic search capabilities, the tool allows users to conduct searches based not only on specific keywords but also on the contextual meanings of words, enhancing the relevance of search results.

- Adaptation to Historical Variabilities: The tool was specifically tailored to address the unique challenges posed by historical documents, adapting to linguistic variations without requiring precise input from users.

- User-Centric Design: I focused on creating a user-friendly interface that simplifies the search process, making historical research more accessible and efficient for historians and scholars.

The development process followed an agile methodology, enabling rapid iterations based on user feedback and continuous testing. This approach ensured that the tool not only met the immediate needs of researchers but also remained adaptable for future enhancements.

The project has significantly streamlined the process of accessing and analyzing historical documents, demonstrating the potential of NLP and semantic search technologies to transform historical research.

7.2 Personal Reflection

Reflecting on my experience with this project, I've learned a ton about using new technologies and tools—skills I picked up from scratch, especially in natural language processing, an area I had not touched before. At the beginning, I immersed myself in extensive research, getting to grips with cutting-edge knowledge that was entirely new to me. However, my unfamiliarity made me hesitant to apply what I had learned in practice. This hesitation, unfortunately, led to a crunch in development time later on.

Additionally, I also started learning Django from the ground up for this project. The learning curve was steep, but it was thrilling to piece together how everything works in building a functional application. I now realize that being proactive in applying new knowledge, even if not perfectly mastered, is crucial. It will be better to experiment early and iterate rather than wait to feel fully prepared, which might delay the whole process.

As I dove into new technologies, I encountered numerous bugs and setbacks. One particular challenge was integrating the NLP tools with the Django framework, which required aligning the back-end logic with front-end presentation. It was a trial-and-error process that sometimes felt like two steps forward, one step back. However, each setback was a learning opportunity, pushing me to understand the systems deeper and refine my problem-solving skills.

Another lesson I learned was the value of iterative testing and user feedback. Early in the project, I delayed seeking feedback on the tool's usability, which later resulted in some user experience issues that were hard to fix at the advanced stages of development. Going forward, I will incorporate regular feedback loops into my development process to catch and address issues early.If I get another chance to work on a similar project, I would definitely start applying what I learn much earlier.

Bibliography

Armengol-Estapé, J., Carrino, C. P., Rodriguez-Penagos, C., Bonet, O. d. G., Armentano-Oller, C., Gonzalez-Agirre, A., Melero, M., and Villegas, M. (2021). Are multilingual models the best choice for moderately under-resourced languages? a comprehensive assessment for catalan. *arXiv preprint arXiv:2107.07903*.

Aroca-Ouellette, S. and Rudzicz, F. (2020). On losses for modern language models. *arXiv preprint arXiv:2010.01694*.

Bhagdev, R., Chapman, S., Ciravegna, F., Lanfranchi, V., and Petrelli, D. (2008). Hybrid search: Effectively combining keywords and semantic searches. In *The Semantic Web: Research and Applications: 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008 Proceedings 5*, pages 554–568. Springer.

Capra, R., Arguello, J., Crescenzi, A., and Vardell, E. (2015). Differences in the use of search assistance for tasks of varying complexity. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, page 23–32, New York, NY, USA. Association for Computing Machinery.

Chowdhury, G. G. (2010). *Introduction to modern information retrieval*. Facet publishing.

Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., Grave, E., Ott, M., Zettlemoyer, L., and Stoyanov, V. (2019). Unsupervised cross-lingual representation learning at scale. *arXiv preprint arXiv:1911.02116*.

Degani, T. and Tokowicz, N. (2010). Ambiguous words are harder to learn. *Bilingualism: Language and Cognition*, 13(3):299–314.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dwivedi, Y. K., Kshetri, N., Hughes, L., Slade, E. L., Jeyaraj, A., Kar, A. K., Baabdullah, A. M., Koohang, A., Raghavan, V., Ahuja, M., Albanna, H., Albashrawi, M. A., Al-Busaidi, A. S., Balakrishnan, J., Barlette, Y., Basu, S., Bose, I., Brooks, L., Buhalis, D., Carter, L., Chowdhury, S., Crick, T., Cunningham, S. W., Davies, G. H., Davison, R. M., Dé, R., Dennehy, D., Duan, Y., Dubey, R., Dwivedi, R., Edwards, J. S., Flavián, C., Gauld, R., Grover, V., Hu,

M.-C., Janssen, M., Jones, P., Junglas, I., Khorana, S., Kraus, S., Larsen, K. R., Latreille, P., Laumer, S., Malik, F. T., Mardani, A., Mariani, M., Mithas, S., Mogaji, E., Nord, J. H., O'Connor, S., Okumus, F., Pagani, M., Pandey, N., Papagiannidis, S., Pappas, I. O., Pathak, N., Pries-Heje, J., Raman, R., Rana, N. P., Rehm, S.-V., Ribeiro-Navarrete, S., Richter, A., Rowe, F., Sarker, S., Stahl, B. C., Tiwari, M. K., van der Aalst, W., Venkatesh, V., Viglia, G., Wade, M., Walton, P., Wirtz, J., and Wright, R. (2023). Opinion paper: "so what if chatgpt wrote it?" multidisciplinary perspectives on opportunities, challenges and implications of generative conversational ai for research, practice and policy. *International Journal of Information Management*, 71:102642.

Feng, F., Yang, Y., Cer, D., Arivazhagan, N., and Wang, W. (2020). Language-agnostic bert sentence embedding. *arXiv preprint arXiv:2007.01852*.

Ghayoomi, M. and Mousavian, M. (2022). Deep transfer learning for covid-19 fake news detection in persian. *Expert Systems*, 39(8):e13008.

Kalyan, K. S. and Sangeetha, S. (2021). A hybrid approach to measure semantic relatedness in biomedical concepts. *arXiv preprint arXiv:2101.10196*.

Knapp, B., Bardenet, R., Bernabeu, M. O., Bordas, R., Bruna, M., Calderhead, B., Cooper, J., Fletcher, A. G., Groen, D., Kuijper, B., Lewis, J., McInerny, G., Minssen, T., Osborne, J., Paulitschke, V., Pitt-Francis, J., Todoric, J., Yates, C. A., Gavaghan, D., and Deane, C. M. (2015). Ten simple rules for a successful cross-disciplinary collaboration. *PLOS Computational Biology*, 11(4):1–7.

Lassila, O., Hendler, J., and Berners-Lee, T. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Libovickỳ, J., Rosa, R., and Fraser, A. (2019). How language-neutral is multilingual bert? *arXiv preprint arXiv:1911.03310*.

Ma, H., Lin, Y., Miao, Z., Gao, J., and Lu, J. (2023). A financial derivatives related multi-label text classification algorithm based on financial knowledge graph. In *Third International Conference on Machine Learning and Computer Application (ICMLCA 2022)*, volume 12636, pages 1271–1278. SPIE.

Malmasi, S., Fang, A., Fetahu, B., Kar, S., and Rokhlenko, O. (2022). Multiconer: A large-scale multilingual dataset for complex named entity recognition. *arXiv preprint arXiv:2208.14536*.

Min, B., Ross, H., Sulem, E., Veyseh, A. P. B., Nguyen, T. H., Sainz, O., Agirre, E., Heintz, I., and Roth, D. (2023). Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Comput. Surv.*, 56(2).

Parsing, C. (2009). Speech and language processing. *Power Point Slides*.

Philips, J. P. and Tabrizi, N. (2020). Historical document processing: Historical document processing: A survey of techniques, tools, and trends.

Pimentel, J. F., Murta, L., Braganholo, V., and Freire, J. (2019). A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 507–517.

Rafiei-Asl, J. and Nickabadi, A. (2017). Tsake: A topical and structural automatic keyphrase extractor. *Applied soft computing*, 58:620–630.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Reimers, N. and Gurevych, I. (2020). Making monolingual sentence embeddings multilingual using knowledge distillation. *CoRR*, abs/2004.09813.

Robertson, S. E. and Jones, K. S. (1976). Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146.

Schütze, H., Manning, C. D., and Raghavan, P. (2008). *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge.

Seiler, A. (2020). How to catch your unicorn: Defining meaning in ælfric's glossary, the oxford english dictionary, and urban dictionary. *Dictionaries: Journal of the Dictionary Society of North America*, 41(2):245–276.

Shi, P. and Lin, J. (2019). Simple bert models for relation extraction and semantic role labeling. *arXiv preprint arXiv:1904.05255*.

Trott, S. and Bergen, B. (2023). Word meaning is both categorical and continuous. *Psychological Review*.

van Otterlo, M. (2018). Ethics and the value(s) of artificial intelligence. *Nieuw Archief voor Wiskunde (Serie V)*, 5/19(3).

.1 User Manual

- When run ARO Intelligent for the first time, database needs to be set up first. You should open the Settings.py file under the 'search_project' as shown in Figure 1.

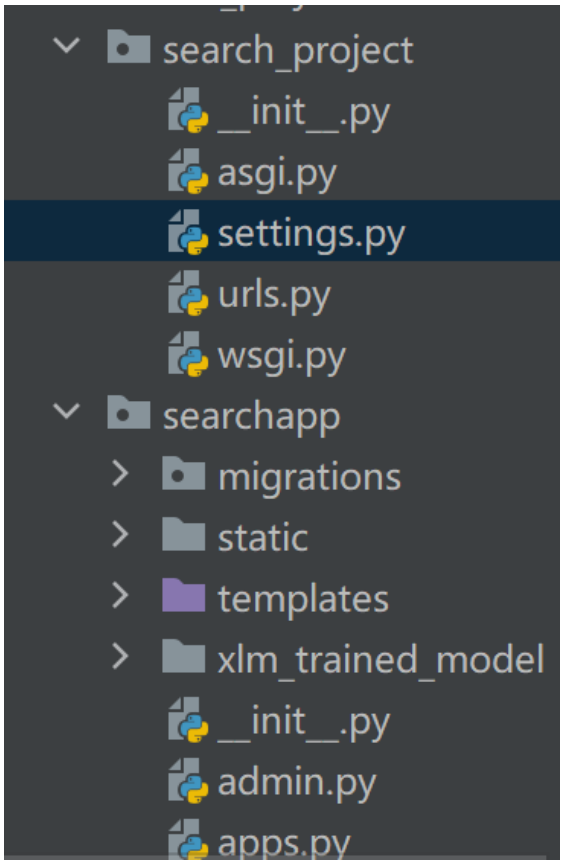


Figure 1: settings.py

- When you open the file, find DATABASES setting and change it to USER, PASSWORD, HOST, PORT according to your database.

- You can refer to this database setting guidance. Click the '+' to create a connection and set up the connection name as you like. Remember your USERNAME, HOSTNAME, PORT. And then clcik 'test connection'. When the box shown in 5. Connection is successful! Click of to exit.

- After setting up connection, back to the main page to open the connection you created. Choose import Data Import in Server and upload the corpus.sql file choosing import from self-contained file. Clcik 'start import'.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'corpus',
        'USER': 'root',
        'PASSWORD': 'Rlc0913$',
        'HOST': '127.0.0.2',
        'PORT': '3306',
    }
}
```

Figure 2: Database Setting

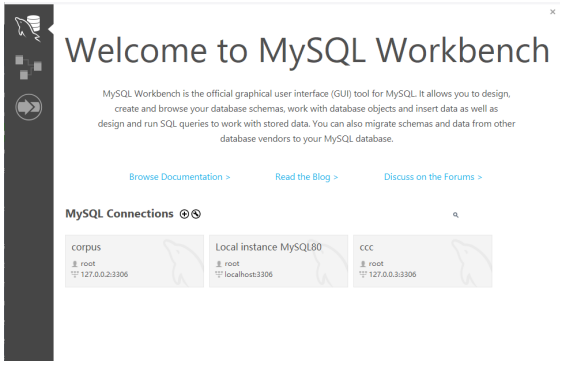


Figure 3: Click '+' to create a connection

- After importing the corpus, back to the setting.py and change the HOST, NAME and PORT.
- Next step is to migrate the data. Open the terminal and run 'python manage.py migrate'. After migration, you will get a prompt of successfully migration.
- Next step enter python manage.py runserver. Please mind that the path should be the same as the manage.py. Otherwise it will not be able to run successfully.
- When you see an url shown in the terminal, click on it and you can open the ARO Intelligent successfully! Congrats! Feel free to explore it!

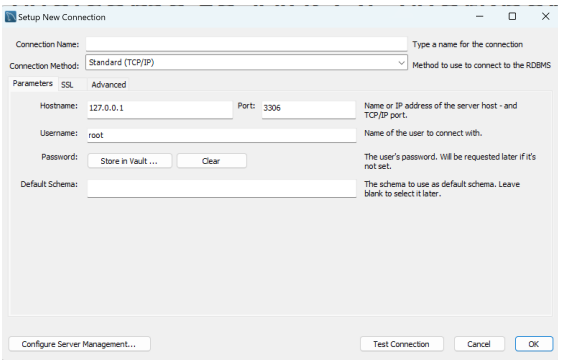


Figure 4: New Connection Setting

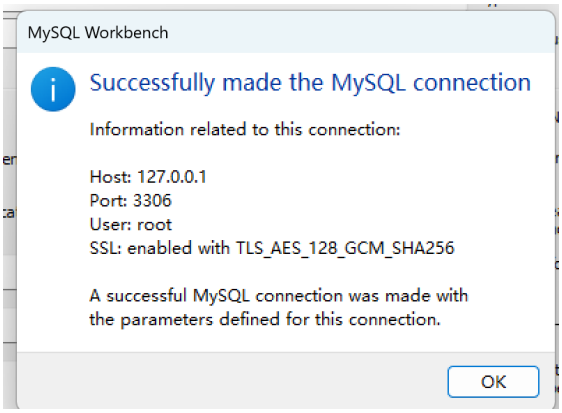


Figure 5: Test Connection

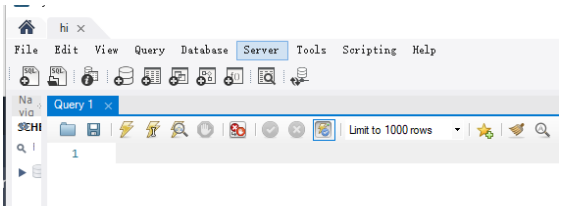


Figure 6: Server

```
h_project\search_project> python manage.py runserver
```

Figure 7: python manage.py runserver

```
System check identified no issues (0 silenced).
May 03, 2024 - 23:04:31
Django version 5.0.4, using settings 'search_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Figure 8: Running Successfully

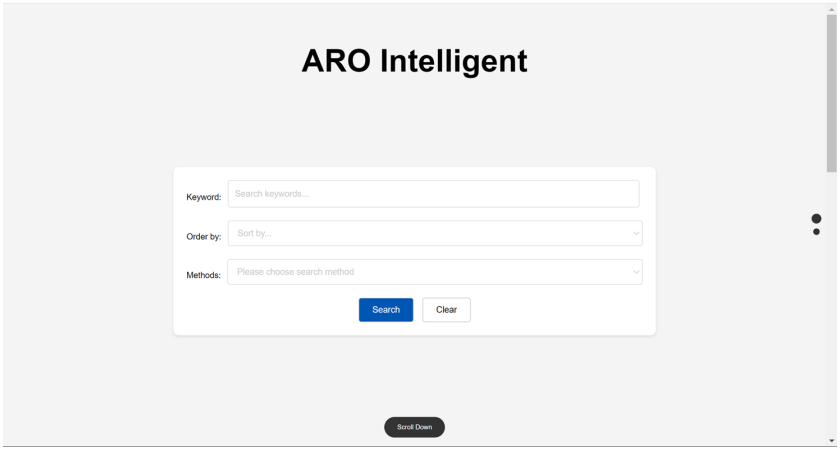


Figure 9: Search Page

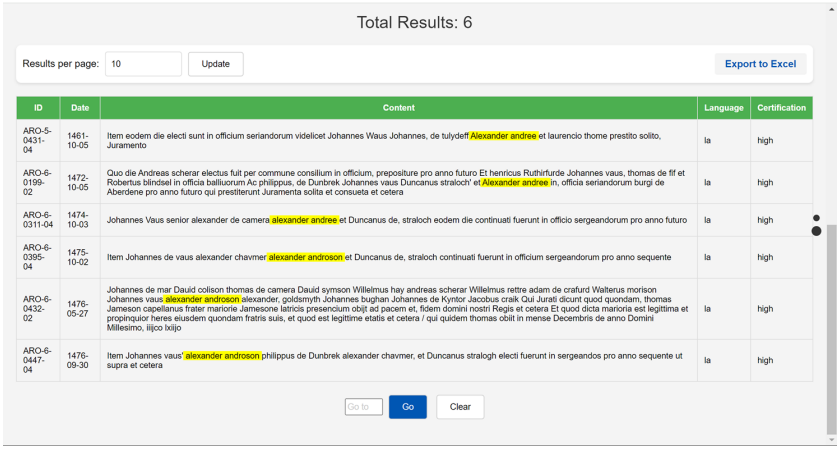


Figure 10: Results Page

.1 Maintenance Manual

- The first possible error is the data from previous search is cached so it will lead to searching results error. For example, you searched cow before and then searched lamb, you will find cow in the results list of searching lamb.
- Don't worry! You can try Ctrl+fn5 to refresh the page and clean the cache. And then click Ctrl+C in the terminal to stop. Try it again it should be working.
- If you are alerted that some packages are missing. All the packages you need is in the requirements.txt. You can enter pip install -r requirements.txt in the terminal, and all the packages needed to be download or update will be automatically installed.

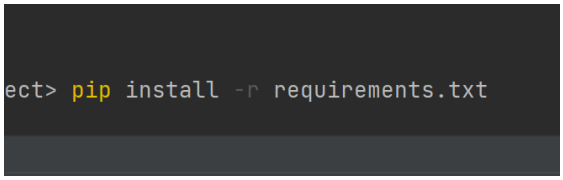


Figure 11: Requirements.txt command

- Also remember you should use python 3.10 or later version as your IDE. Otherwise it may have possibility that Some functions are not compatible.

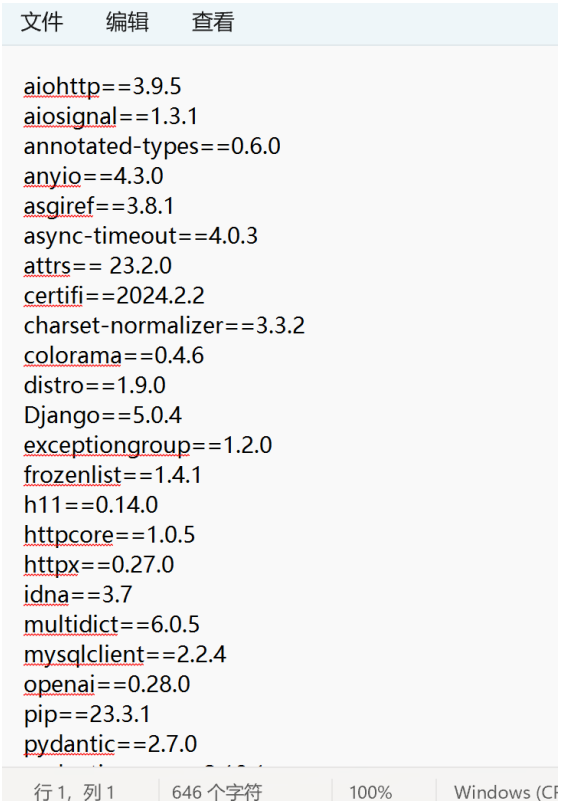


Figure 12: Requirements.txt