Text Summarization and Question Answering System in Finance

Candidate Number: BRBG4

Supervisor: Prof. Graham Roberts External supervisor: Joseph Connor

COMP0073

MSc Computer Science

September 18, 2023

Department of Computer Science

University College London

Abstract

In the financial field, there is a large amount of text data, such as annual reports, financial news, etc., and it is a heavy job to read and analyze these texts in detail. Because these texts tend to be very long. For example, an annual report, usually published in PDF format, may exceed 100 pages. This requires the use of new technologies to improve work efficiency. Summarization and Question answering (QA) are two important tasks in Natural language processing (NLP). This project attempts to apply these two NLP techniques to the financial field. This project aims to develop a text summarization and question answering system using BART model and GPT-3.5 model. The system can generate a summary of financial news or the qualitative part of an annual report (i.e. without tables). In addition, the system can answer user questions based on the uploaded PDF of the annual report. Hugging face, LangChain, Gradio and other tools are used in this system. Fine-tuning of the BART model was also implemented during development to make this pre-trained model better suited for the task of summarization of annual reports. Individuals in the financial sector can use the system to quickly access key information in annual reports or financial news, and thus make quick judgments about the market.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

The financial domain depends on text data. A large amount of data is created by different firms, which comes in different forms such as annual financial reports, press releases (El-Haj et al., 2018)[1]. Natural Language Processing (NLP) technology can efficiently process text data, and NLP has a wide range of applications in the financial domain. The amount of financial information generated every day is huge. It is time-consuming and laborious to accurately find relevant articles and documents from the vast information database, and to read and analyze each important piece of content. Automatic text summarisation and question answering technologies seem to solve this problem.

Text summarization is one of the most challenging NLP tasks, requiring the comprehension of long paragraphs and condensing them into shorter versions, with the benefit of reducing the burden of detailed reading of long documents by domain experts and increasing productivity.

There are two main types of summarization: extractive and abstract. Extractive summarization extracts salient sentences directly from the source document and combines them into a summary (Koh et al., 2022)[2]. Thus, the essence of the original text remains unchanged, only highlighting the key points. Abstract summarization, a method that requires an in-depth understanding of the core ideas of the original text and the preparation of a new summary based on these ideas, is a

more advanced process that aims to convey the same information in a more concise way (Nantasenamat, 2023)[3].

This project focuses on abstract summaries because it is more flexible and can generate more diverse summaries. This project uses the BART model and the GPT-3.5 model to summarise financial news and annual reports. Generative Pre-trained Transformer 3.5 (GPT-3.5) is a large language model released by OpenAI that performs well in various natural language processing tasks. The BART model is a large language model that can be used for text generation and is suitable for the text summarization task. In this project, financial news and annual reports are considered as long documents because most existing pre-trained models can only handle 512 to 1,024 lexical tokens, and the number of lexical tokens in financial news and annual reports often exceeds this range (Koh et al., 2022)[2]. Long document summarization is more challenging than short document summarization. Because the number of lexical tokens and the breadth of content differ significantly between short and long documents. The longer the document, the more critical content it contains, making it difficult for automated summarization models to cover all the important information within the restricted output length (Koh et al., 2022)[2].

It is worth mentioning that since annual reports are not only longer (typically more than 100 pages) but also mostly published in PDF format compared to financial news, there may be greater challenges in text summarization of them using large language models. The qualitative content of annual reports, on the other hand, is an important basis used by financial market participants to make investment decisions, and there is great value in implementing automated text summarization of them (Hsieh and Hristova, 2022)[4]. In addition, the absence of a golden dataset of financial documents and their associated summaries is a major limitation when applying NLP technologies for summarization (Abdaljalil and Bouamor, 2021)[5]. The unstructured nature of annual report documents and the lack of a dataset consisting of annual reports as well as their corresponding summaries make the processing of financial texts with existing text summarization models potentially face performance drawbacks. For the specific object of annual report long documents, this project

utilised the EDGAR-CORPUS dataset, an annual report dataset. A new dataset consisting of the text of annual reports and the corresponding summaries was created based on this dataset, which contained 1000 text-summary pairs, and then the facebook/bart-large-cnn model was fine-tuned using the created dataset with the aim of improving the performance of the BART model when summarising annual reports.

Question Answering is also a key NLP technique, where the user asks questions in natural language and the model performs in-depth semantic analyses to better understand the user's intent and to quickly and accurately access the information in the knowledge base, which usually consists of documents. The difference between QA systems is in how they generate the answers. In extractive QA, the model extracts answers directly from the given context to present them to the user, which is often handled by models such as BERT. Generative QA, on the other hand, constructs textual answers on its own based on the context, which relies heavily on text generation models (Chiusano, 2022)[6]. This project focuses on generative QA because this approach provides more flexible and natural answers. This project uses the GPT-3.5 model and the annual report uploaded by the user as a knowledge base, which allows the user to quickly and easily access the key information in the report by using the prompt without having to read through a lengthy document.

1.2 Project Goals

The main goal of this project is to develop and test a system that integrates text summarization with QA functions. The system can be applied to process text data in the financial sector, mainly used to process long documents such as financial news and annual reports, helping users to quickly understand and analyse these informations. The user interacts with the system through the Gradio user interface. The main features of the final product are described below.

- Requirements gathering: Gradio UI to collect user requirements for the format of the final generated summary report file, TXT file or PDF file. Through the Gradio UI to obtain the user's preference for a large language model,

BART model or GPT-3.5 model. Let the user input the URL of the financial news to be processed as well as upload the PDF file of the annual report through Gradio UI.

- Data collection and pre-processing: Use Beautiful Soup to scrape the data from the specified URL and load the annual reports uploaded by users using the PDF document loader in the LangChain framework. Data cleaning of the captured text data using NLTK toolkit and regular expressions and custom functions. Segmentation and chunking of financial news and annual reports using text splitters in the LangChain framework and appropriate tokenizer.

- Summarization: Text summarization based on the large language model selected by the user. For the BART model, the chunked text is encoded and sent to the model for prediction, then decoded, and each chunk of generated text is concatenated to get the complete summary. For the GPT-3.5 model, the text chunks are summarised using the summary chain in the LangChain framework.

- Q&A: After chunking the annual report using the text splitter in the LangChain framework, embed the content of each chunk, then store the embedding and the document in the vector store, and use the embedding to index the document. Using similarity search to retrieve splits related to the user question. Use the chain component of the LangChain framework to provide the user question as well as the retrieved documents to the LLM for answer distillation.

- Output presentation and storage: Display the generated summary in the Gradio front-end. Display the answer to the user question and the document related to the question asked in the Gradio front-end. Generate a PDF file or TXT file of the summary, then store the generated file in Google Cloud Storage and output a public URL for user access.

1.3 Personal Aims

- Take an in-depth look at natural language processing technology and understand their application in the financial sector.
- Learn how to develop applications with specific functionalities using large language models.
- Improve Python programming skills.
- Gain experience in developing applications independently.

1.4 Overview of the report

There are seven chapters in this report and the following is an outline of the seven chapters.

Chapter 1 Introduction

The report begins with a problem statement that describes the motivation for applying automated text summarisation and Q&A techniques to the financial domain, before outlining the project goals and personal aims to be achieved.

Chapter 2 Background

This chapter introduces the relevant theories involved in the project and outlines the core software and tools used in this project.

Chapter 3 Requirements and Analysis

This chapter describes all the requirements of the developed system and identifies the main use cases that the final delivered system should have.

Chapter 4 Fine-tuning

This chapter details the process of fine-tuning the BART model, including dataset preparation, and training the BART model.

Chapter 5 Design and Implementation

This chapter describes the structure of the system and the core components, and describes the implementation details of the components.

Chapter 6 Testing

This chapter outlines the testing strategy for the application and describes the process and results of the different testings performed on the system.

Chapter 7 Conclusion

This chapter summarises the results of the project and discusses future plans to further improve the project.

Chapter 2

Context

2.1 Natural language processing (NLP)

Natural language processing is a subfield at the intersection of computer science and linguistics that involves the use of rule-based or probabilistic machine learning methods to process natural language datasets, such as text corpora ('Natural language processing', 2023)[7]. The goal of NLP tasks is not only to understand individual words in isolation, but also to be able to understand, analyse, and process these words in context. NLP tasks involve summarization, translation, and Q&A, and this report only discusses summarization and Q&A in relation to this project. Computers process information differently from humans. It is difficult for Machine Learning (ML) models to understand human language, which has complex rules and other characteristics. Therefore text data needs to be processed in some way that the model can learn from it.

2.2 Neural networks

A neural network is a mathematical or computational model that simulates the way biological neurons communicate with each other. The human nervous system consists of more than 100 billion cells called neurons, which communicate through electrical signals; within a neuron, when a dendrite receives a signal, if the signal is strong enough, it may pass on to the axon and then to the terminal button; if the signal reaches the terminal button, they send out a signal that emits chemicals called neurotransmitters, which travel through the intercellular spaces (called synapses) to

communicate with other neurons (Walinga and Stangor, 2014)[8].

Similarly, a neural network consists of an input layer, one or more hidden layers, and an output layer. Individual nodes are connected to another node or nodes, and the nodes assign relevant weights and thresholds to incoming data; if the output of any individual node is above the specified threshold, that node is activated and sends the data to the next layer of the network; otherwise, no data is passed on to the next layer of the network (Hardesty, 2017)[9].

Common types of neural networks are feed-forward networks, which is the type involved in the large language model used in this project. As illustrated in Figure 2.1, in a feed-forward network, information always moves in one direction, usually only from layer n to layer n+1; the input is passed forward layer by layer in layer 0, and the neurons in each layer perform some computation before passing the information on to the next layer; and through the use of weights, biases, and activation functions, the network learns to perform a set of mappings from vector X to vector Y (Ciliberto, 2023)[10].

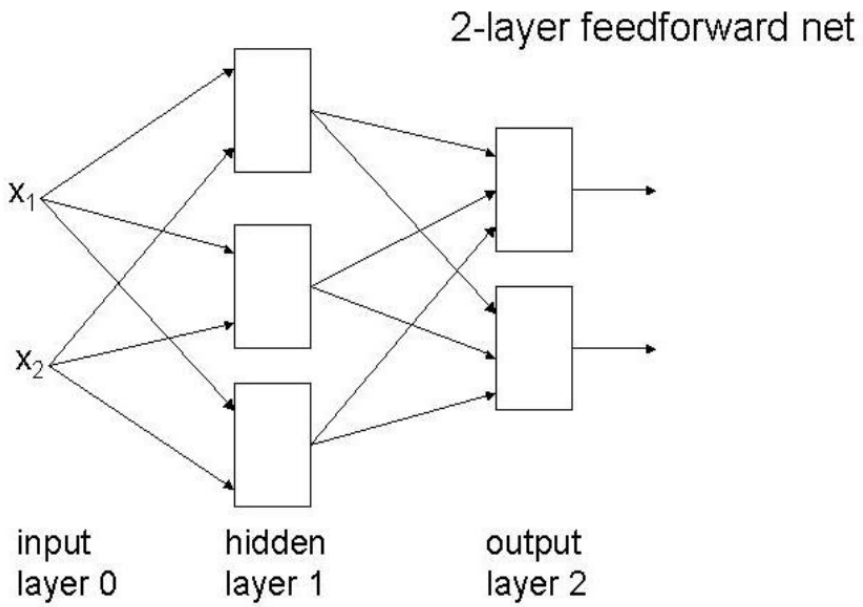

Figure 2.1: Feed-forward network

2.3 Transformer

Transformer is a neural network architecture for language understanding based on the mechanism of self-attention. All transformer models have the same main components: a tokenizer, which converts text into tokens; an embedding layer, which converts tokens into semantically meaningful representations; and a transformer layer, which performs inference functions ('Transformer (machine learning model)', 2023)[11]. The transformer layer can be of two types: encoder and decoder, based on which transformer models can be classified into three categories: auto-regressive transformer models (decoder-only models, e.g., GPT-3.5), auto-encoding transformer models (encoder-only models), and sequence to sequence transformer models (models that use both encoder and decoder, e.g., BART). The encoder accepts an input sequence (x1, ... , xn) and transforms it into a continuous representation of the form z = (z1, ... , zn), given z, the decoder generates an output sequence (y1, ..., ym) of symbols one element at a time (Vaswani et al., 2017)[12].

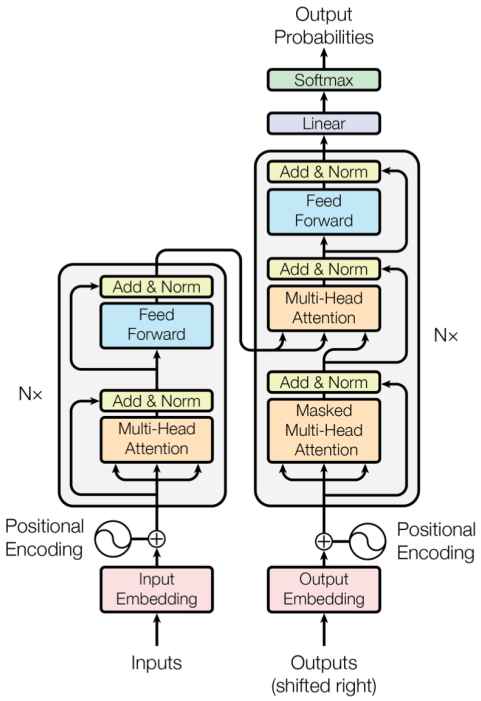

Figure 2.2: The Transformer - model architecture

As shown in Figure 2.2 (Vaswani et al., 2017)[12], the encoder consists of two main parts: the self-attention mechanism and the feed-forward neural network. The

decoder consists of three main parts: the self-attention mechanism, the encoding attention mechanism and the feed-forward neural network.

A key feature of the transformer model is built from the attention layer, where self-attention will take into account an element's relationship with other elements in the sequence when processing it, capturing richer contextual information. In the self-attention mechanism, each input element (e.g., a word in a sentence) generates a Query, a Key, and a Value, and the output is computed as a weighted sum of the values, where the weight assigned to each value is computed by the compatibility function of the query with the corresponding key (Vaswani et al., 2017)[12].

2.4 Fine-tuning

The transformer model (GPT-3.5, BART) used in this project has been pre-trained with a large amount of raw text in a self-supervised manner. Self-supervised learning allows the model to automatically generate training goals based on its inputs. Pre-training refers to training the model from an initial state where the model weights start from random values with no prior knowledge. Such training is often performed based on large-scale datasets. Although such models are able to statistically understand the language in which they are trained, they may not be particularly effective on specific tasks. As a result, these pre-trained models typically go through a transfer learning phase where the model is fine-tuned to task-specific labelled data. Fine-tuning is a strategy of transfer learning that allows a pre-trained model to adjust its weights based on new data ('Fine-tuning (deep learning)', 2023)[13].

In order to perform fine-tuning, a pre-trained language model is first available, followed by further training using task-specific data. This pre-trained model has been previously trained on data similar to the fine-tuned data, so the fine-tuning can take advantage of what the model has learnt in the pre-training phase. Moreover, given that the pre-trained model has been trained on a large amount of data, fine-tuning can be achieved with less data.

2.5 Evaluation Metrics: ROUGE

In order to measure the performance of the fine-tuned BART model, an evaluation metric is needed. The efficient ROUGE metric has long been the standard method for comparing the performance of summary models, and the basic idea behind it is to compare automatically generated summaries with a reference summary, which measures the word overlap between the reference and generated summaries, with common n-gram measures being the unigram (ROUGE-1), bigram (ROUGE-2) and longest common sub-sequence (ROUGE-L) (Koh et al., 2022)[2]. ROUGE is based on calculating recall, precision and F1 score. For ROUGE, recall measures how much of the reference summary is captured by the generated summary, precision measures the proportion of information in the generated summary that is shared with the reference summary, and F1 score is the harmonic mean of precision and recall.

For example, the reference summary and generated summary are: I love watching the Harry Potter films. I absolutely love watching the Harry Potter films. The number of words they overlap is 7.

$$\text{ROUGE-1 recall} = \frac{\text{Number of overlapping words}}{\text{Total number of words in reference summary}} = 1$$

$$\text{ROUGE-1 precision} = \frac{\text{Number of overlapping words}}{\text{Total number of words in generated summary}} = 0.875$$

$$\text{ROUGE-1 F1} = \frac{2\times(\text{Precision}\times\text{Recall})}{\text{Precision}+\text{Recall}} = 0.93$$

ROUGE-2 measures binary syntax overlap. Here is a reference summary and a generated summary. Reference Summary: The dog sits on the sofa. Generated Summary: The dog is on the sofa.

The reference summary bigrams and generated summary bigrams are: (“The”, “dog”), (“dog”, “sits”), (“sits”, “on”), (“on”, “the”), (“the”, “sofa”), (“The”, “dog”), (“dog”, “is”), (“is”, “on”), (“on”, “the”), (“the”, “sofa”)
The shared binary group: (“The”, “dog”), (“on”, “the”), (“the”, “sofa”)

$$\text{ROUGE-2 recall} = \frac{\text{Number of shared binary group}}{\text{Total number of binary group in the reference summary}} = 0.6$$

$$\text{ROUGE-2 precision} = \frac{\text{Number of shared binary group}}{\text{Total number of binary group in the generated summary}} = 0.6$$

$$\text{ROUGE-2 F1} = \frac{2\times(\text{Precision}\times\text{Recall})}{\text{Precision}+\text{Recall}} = 0.6$$

Still using the two summaries above, the longest common subsequence (LCS) between the reference summary and the generated summary is "The dog on the sofa", which is 5 in length.

$$\text{ROUGE-L recall} = \frac{\text{Longest common sequence length}}{\text{Total number of words in reference summary}} = 0.83$$

$$\text{ROUGE-L precision} = \frac{\text{Longest common sequence length}}{\text{Total number of words in generated summary}} = 0.83$$

$$\text{ROUGE-L F1} = \frac{2\times(\text{Precision}\times\text{Recall})}{\text{Precision}+\text{Recall}} = 0.83$$

2.6 Tokenization

By tokenization, text is split into words (or parts of words, punctuation marks, etc.) called tokens, and the GPT and BART series of models use tokens to process text. Tokenization is divided into three main categories: word tokenization (splitting the original text into words), character tokenization (splitting the text into characters) and subword tokenization. For example, "Cute dog" can use spaces to tokenize the text into words, ["cute", "dog"] is the word token. Character tokenization is: ["c", "u", "t", "e", "d", "o", "g", "d"]. The subword tokenization algorithm breaks down rare words into meaningful subwords, and frequently used words are not split into smaller subwords, which mean the stem. For example, "tokenization" is split into "token" and "ization".

2.7 Embedding and cosine similarity

In natural language processing (NLP), word embeddings are vector representations of words, and embeddings are used in text analysis such as search (where results are ranked by relevance to the query string) ('Word embedding', 2023)[14]. Cosine similarity is a calculation that measures the similarity of two nonzero vectors in the inner product space based on the cosine of the Angle between the vectors. In the field of information retrieval, each term is assigned a unique coordinate, and the document is described by a vector that represents the frequency of each term in the document. This method can effectively measure how similar two documents are in terms of topics, independent of the length of the documents. ('Cosine similarity', 2023)[15]. Cosine similarity ranges from 0 to 1. The closer the cosine value is to 1, the more similar the two vectors are. This project involves embedding and cosine similarity in the retrieval part of the question answering task and in the testing phase of the system.

2.8 Core software and tools

2.8.1 Gradio

Gradio is an open-source Python package that quickly generates visual interfaces for machine learning models. It enables access to machine learning models through public urls, sharing machine learning models with anyone. With Gradio, researchers can quickly create beautiful user interfaces around machine learning models, allowing domain experts to perform input operations to interact with the model and some Python functions (Abid et al., 2019)[16]. Gradio is used in this project to build an application based on the BART and GPT3.5 models.

2.8.2 Replit

Replit is an online integrated development environment (IDE) that supports a variety of programming languages such as Python. Its main feature is collaborative programming, which allows users to share the Repl with others, view edits across files in real time, exchange information, and collaborate on code debugging ('Replit',

2023)[17]. At the request of the external supervisor Prof. Joseph Connor, this project used Replit for development. Using Replit helps Prof. Joseph Connor to update and debug the code according to real-time requirements.

2.8.3 Hugging Face

Hugging Face is an open source community and platform for the field of Natural Language processing (NLP), providing a rich set of pre-trained models, tools, and resources to help researchers and developers achieve better results on NLP tasks. The following are some of the main components and features of the Hugging Face platform.

Pre-trained Models: Hugging Face offers a number of advanced pre-trained language models such as the BART model used in this project. These models can be used directly, or they can be fine-tuned to fit a specific task. In this project, fine-tuning the facebook/bart-large-cnn model was implemented to fit the summary task of the annual report.

Transformers Library: Hugging Face's Transformers library is an open-source Python library that provides tools and interfaces for loading, fine-tuning, and applying pretrained models. This library makes it easy to work with pre-trained models. This project used this library to call the pre-trained model and its corresponding tokenizer.

Model Repository: The Hugging Face Model Repository is a centralized place to store various Natural Language Processing (NLP) pre-trained models, making it easy for users to access and use them. This project upload the fine-tuned BART model to the Hugging Face model repository with the aim of facilitating subsequent calls.

Datasets: Hugging Face provides a repository of datasets containing a wide variety of NLP datasets for training and evaluating models. These datasets can be easily loaded and used. This project created a data set of text summaries of annual reports based on the Edger-Corpus dataset on Hugging Face to prepare for fine-tuning the facebook/bart-large-cnn model.

2.8.4 LangChain

LangChain is a framework for developing applications powered by language models. It has 2 main capabilities: it can connect LLM models with external data sources, and it allows interaction with LLM models. In the part of data collection and preprocessing, model summary, and model question answering, this project used the relevant components in the LangChain framework.

2.8.5 NLTK

NLTK is a widely used natural language processing (NLP) toolkit, a library for the Python programming language. "NLTK" provides various features including text processing, tokenization, part-of-speech tagging, syntactic analysis, sentiment analysis, semantic analysis, corpus access, and more. It has powerful text processing capabilities and is suitable for a variety of natural language processing tasks. This project used NLTK for text preprocessing such as text cleaning and chunking.

2.8.6 Google Cloud Storage

Google Cloud Storage is a service of Google Cloud Platform for storing and managing large amounts of structured and unstructured data. It provides a scalable and highly usable storage solution that can be used to store various types of data including images, audio, video, text files, etc. This project used it to store summary files.

2.8.7 Google Colab

Google Colab is a free cloud-based laptop environment launched by Google for data analytics, machine learning, and deep learning. It is based on Jupyter Notebook, enabling users to write and run code in the cloud. Google Colab provides free GPU and TPU (Tensor Processing unit) acceleration, which can greatly speed up the training of deep learning models. This project used the tool for fine-tuning tasks. In addition, the project also used Google Colab in test phase.

2.8.8 openai-python

The OpenAI Python library provides easy access to the OpenAI API from applications written in the Python language. It contains a set of predefined API resource classes that dynamically initialize themselves based on API responses, which makes it compatible with various versions of the OpenAI API. This project used the library to connect an application with OpenAI's models for an automatic text summarization task and a question answering task.

2.9 Summary

This chapter provides an overview of the theoretical knowledge related to this project (e.g., transformer architecture for large language models, etc.) and some of the core software and tools used (e.g., Hugging face, etc.).

Chapter 3

Requirements analysis

3.1 Requirements gathering

During the requirements gathering phase, the external supervisor, Prof. Joseph Connor, provided a project guidance document that outlined his requirements for the system to be developed. In addition, weekly meetings were held with Prof. Joseph Connor to discuss various aspects of the project and further ask about his requirements. These requirements were divided into functional and non-functional.

3.2 Problem Statement

Employees in the financial field usually need to process and analyze a large amount of text data, such as annual reports, financial news and other long documents. However, reading and analyzing these text data is very time-consuming and laborious. Therefore, there is an urgent need for an application to quickly extract information from annual reports, financial news, to help practitioners in the financial field to quickly make analysis, judgment. This text summarization and question answering system utilizes two large language models, the BART model and the GPT-3.5 model. The system has three interfaces, the functions of the three interfaces are: summarizing the financial news, summarizing the annual report, and answering questions based on the PDF file of the annual report uploaded. In interface 1, the user can enter a URL of financial news, choose to use BART model or GPT-3.5 model according to preference and choose the type of summary file generated (txt or pdf file). The system can present the summary text of the financial news and a

link to the summary file, which is stored in Google Cloud storage. In Interface 2, the user can upload a local PDF file of the annual report and set a page number range for the summary, choose to use the BART model or the GPT-3.5 model according to preference, and choose the type of summary file generated (txt or pdf file). The system can present the summary text of the annual report within the specified range and a link to the corresponding summary file, which is stored in Google Cloud Storage. In Interface 3, the user can upload a local PDF file of the annual report. After that, the user can enter a question related to that annual report (e.g., what was the net profit of the company?). Afterwards, the system can present an answer (e.g., the net profit of the company was $A4,777.) as well as the part of the uploaded PDF document that most closely resembles the question and its page number.

3.3 Personas

After an initial meeting with the external supervisor, it was determined that the main users of the system would be investment analysts who have to deal with annual reports and financial news.

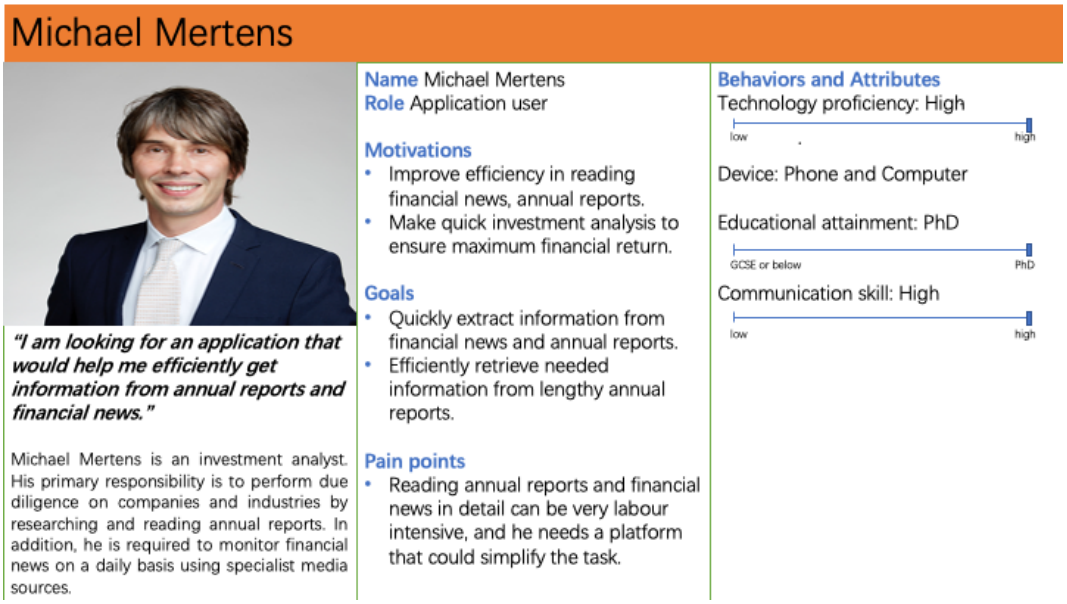

Figure 3.1: Persona of the system

3.4 MoSCoW Requirement List

The MoSCoW method is a prioritization technique used in software development to determine the importance of various features, requirements, and tasks. In this technique, requirements labelled as "must" are critical to the current delivery cycle; requirements labelled as "should" are important but not necessary; requirements labelled as "could" are desired but not necessary; and requirements labelled as "won't" are those that are considered by stakeholders to be the least important or least rewarding items ('MoSCoW method', 2023)[18].

ID	**Functional Requirements**	**Priority**
RQ1	Users can upload a existing local PDF file of the annual report.	Must have
RQ2	Users can enter the web address of the financial news.	Must have
RQ3	The user interface can display the summary text generated.	Must have
RQ4	Users can choose the language model (BART or GPT-3.5).	Must have
RQ5	Users can choose the file format (txt or pdf) of the generated summary report file.	Should have
RQ6	The system can produce a file with the generated summary text and save it in the Google Cloud Storage.	Should have
RQ7	The system should provide the user with an access link to the summary report file.	Should have
RQ8	When performing text summarization on a PDF file of the annual report, the user can specify a range of pages.	Should have
RQ9	When performing the question answering task based on the PDF file of the annual report uploaded, the user can input a question related to that annual report in the user interface.	Must have
RQ10	When performing the question answering task based on the PDF file of the annual report uploaded, the answer generated by the system can be displayed in the user interface.	Must have

RQ11	When performing the question answering task based on the PDF file of the annual report uploaded, the system should display the most relevant part of the uploaded PDF document to the question.	Should have
RQ12	If the user has done something wrong or has missed steps or has entered a URL that is not accessible, the system could display a notification.	Could have
RQ13	The system can be accessed directly through the URL.	Could have
ID	**Non-Functional Requirements**	**Priority**
RQ14	The system should facilitate handover to the external supervisor and subsequent maintenance.	Must have
RQ15	The user's interaction with the system should be smooth.	Should have
RQ16	The system should be accessible and easy to learn.	Should have

Table 3.1: MoSCoW Requirement List

3.5 Use Cases

This section elaborates on how the system interacts with the user to achieve a specific goal by creating a use case diagram and a use case list. Figure 3.5 shows a use case diagram. The list of use cases defines in detail the steps and actions for each use case. Table 3.3 is a summary table, and a full description of the use cases is given in Appendix C.

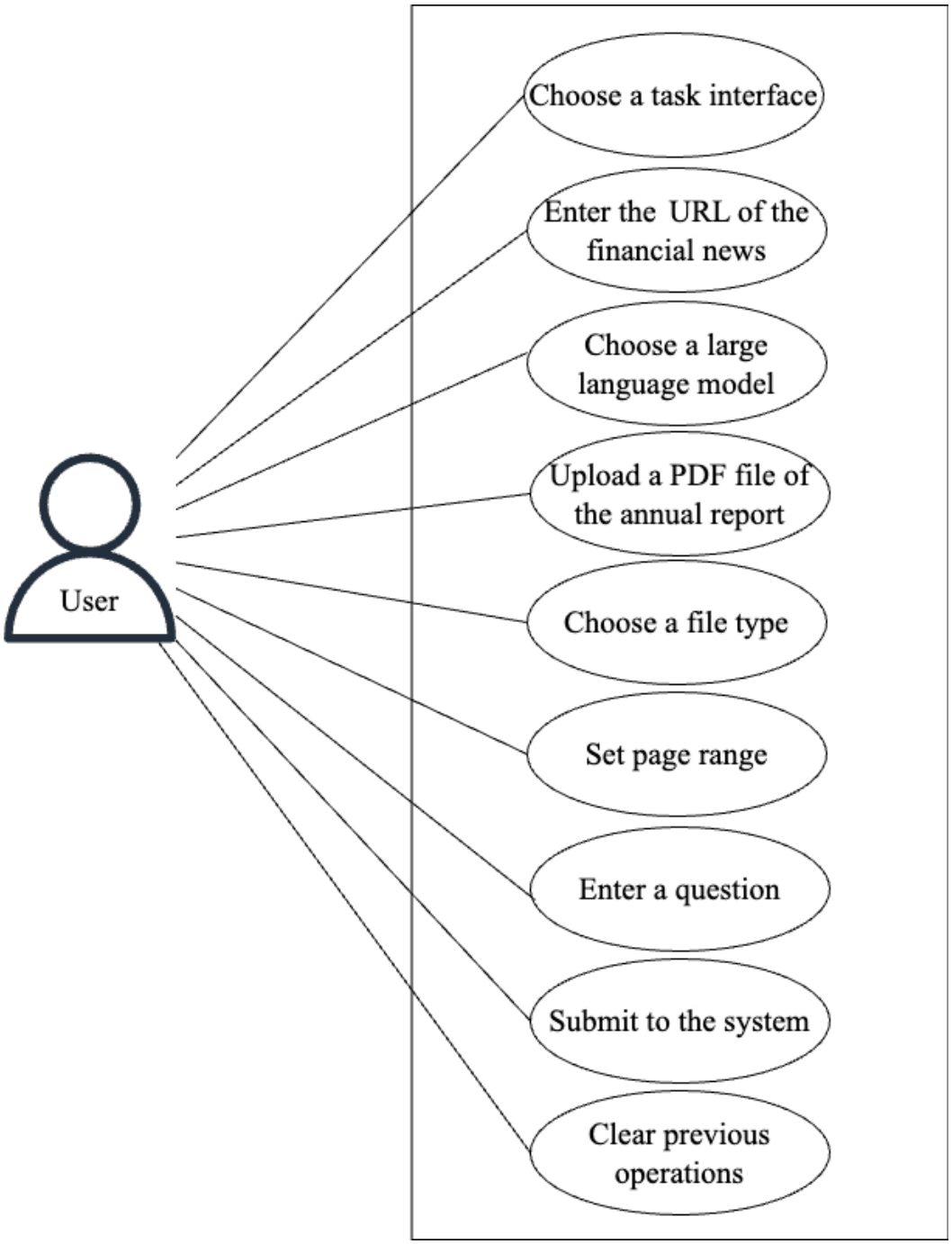

Figure 3.2: Use case diagram

ID	Description
UC1	The user chooses a interface.
UC2	The user inputs a financial news link.
UC3	The user selects a large language model (BART or GPT-3.5).
UC4	The user selects a file type (pdf or txt).
UC5	The user submits to generate a summary and the corresponding summary report file link.
UC6	The user uploads the PDF file of the annual report.
UC7	The user sets the range of page numbers.
UC8	The user enters a question.
UC9	The user submits to generate an answer.
UC10	The user clears the previous operations.

Table 3.2: Use cases list

3.6 Wireframes

Below are 3 wireframes of the design. The application has 3 different interfaces that the user can switch between via the navigation bar.

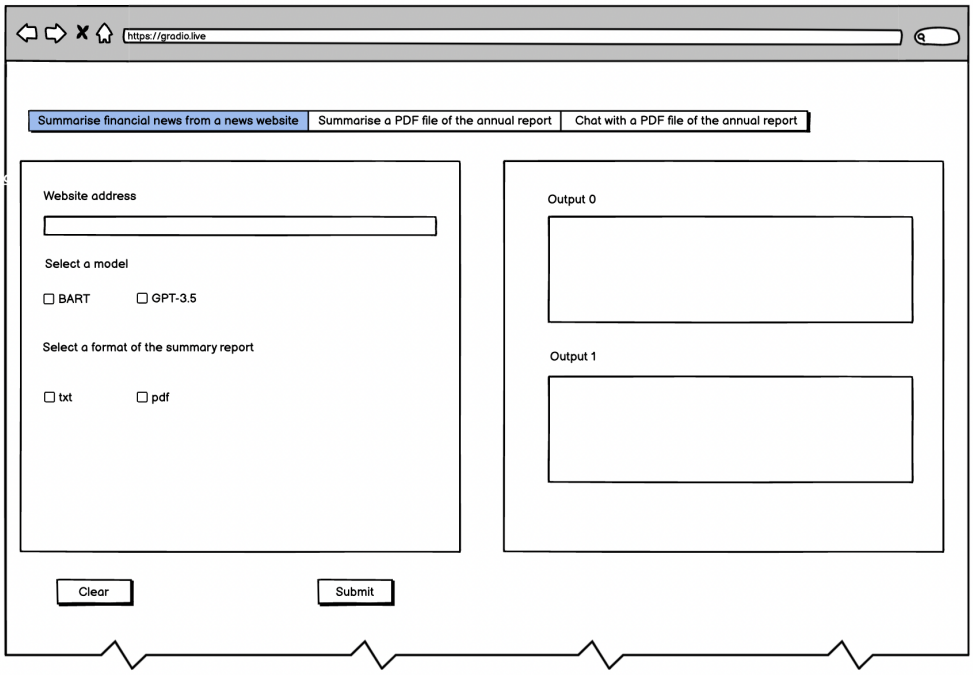

Figure 3.3: The interface 1

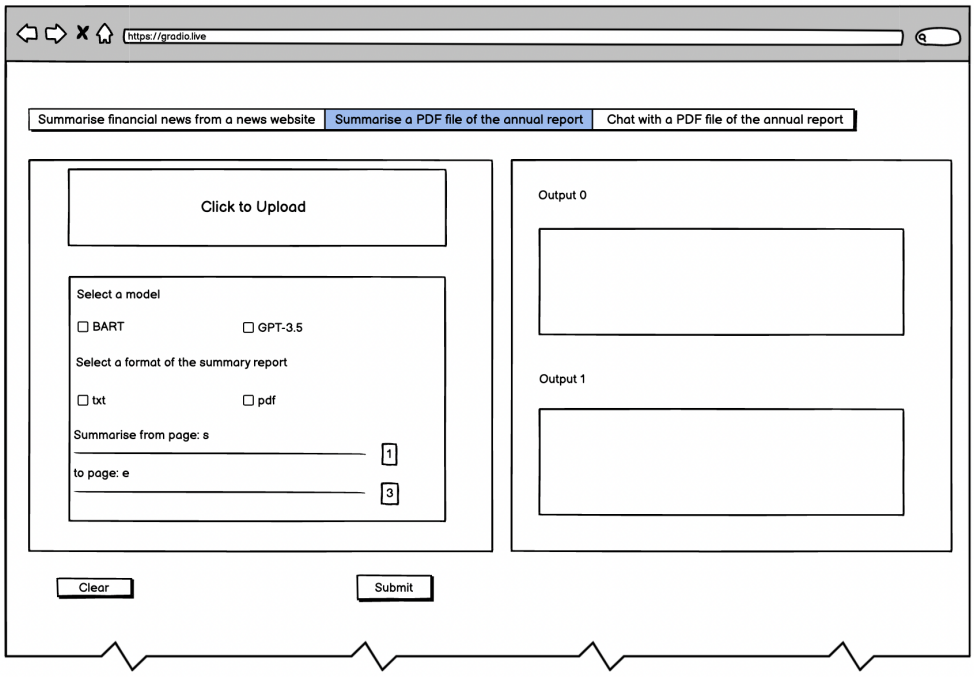

Figure 3.4: The interface 2

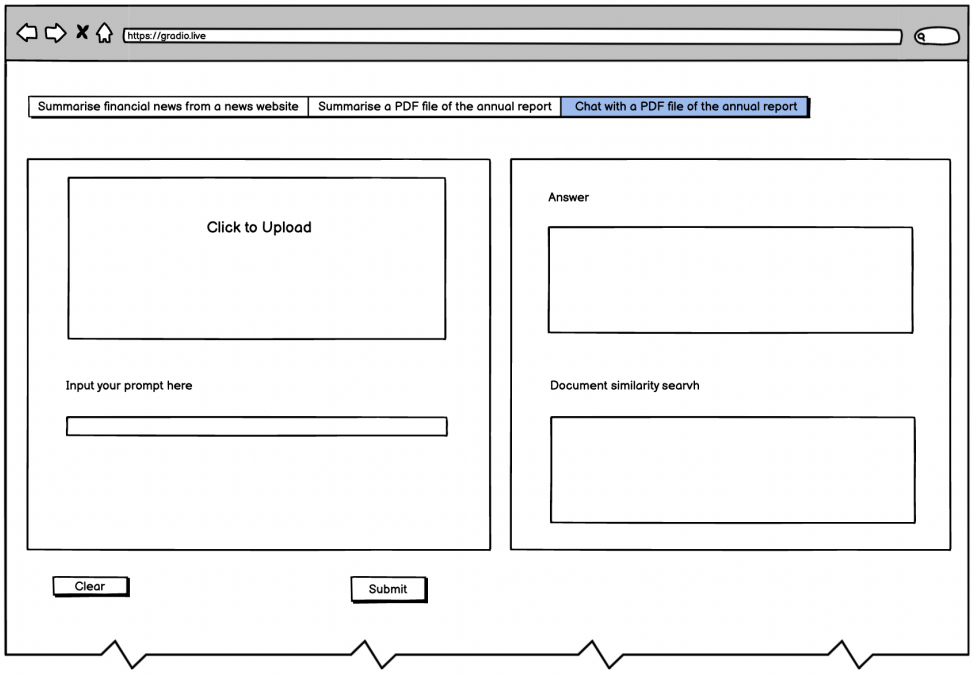

Figure 3.5: The interface 3

3.7 Summary

This chapter covers the requirements analysis phase of the project. This chapter begins with a description of the requirements gathering, the background for developing the application, and then shows a persona of the application, prioritization of the requirements, use cases, and wireframes.

Chapter 4

Fine-tuning

4.1 Overview

This chapter describes how the large language model has been fine-tuned, such that the model can better adapt to the text summary task when working with annual reports. Since no fine-tuning function was introduced for GPT-3.5 series models when this project was developed, only the BART model to be used was fine-tuned in this project. The facebook/bart-large-cnn model is a pre-trained NLP model and fine-tuned on the CNN Daily Mail dataset for text generation tasks, such as summary generation, and has been published on hugging face. The project has fine-tuned the model to help it adapt to text data of annual reports. This project used hugging face's Transformers library for fine-tuning the facebook/bart-large-cnn model , which provided a Trainer class. The Trainer class provides an API for fully functional training on most standard models. This project used the Trainer.train() method for fine-tuning training, but this was very slow on CPU. Due to the lack of high performance computers and graphics cards, this project fine-tuned the BART model on Google Colab, which provided GPU resources. The fine-tuned model was published on hugging face under the name wyx-ucl/bart-EDGAR-CORPUS, so that the fine-tuned model can be easily used through hugging face's relevant library.

4.2 Preparing the Dataset

Due to the lack of a dataset of text summary pairs that have been processed about annual reports, this project created a dataset of text summary pairs based on

the EDGAR-CORPUS dataset for fine-tuning the facebook/bart-large-cnn model. The EDGAR-CORPUS dataset has been published on Hugging face, below is a link to access it. https://huggingface.co/datasets/eloukas/edgar-corpus

EDGAR-CORPUS is a new corpus in the financial field that contains annual reports (10-K filings) from all publicly listed companies in the US over 25 years, all in a clean, easy-to-use JSON format. The 10-K report not only provides a comprehensive account of the company's economic activities, risks, liabilities and operations during the corresponding financial year, but also covers the company's contracts and agreements (Loukas, 2021)[19]. The 10-K report is divided into 4 parts and 20 different items. The structure of the 10-k report is shown in the figure below.

	Item	Section Name
Part I	Item 1	Business
	Item 1A	Risk Factors
	Item 1B	Unresolved Staff Comments
	Item 2	Properties
	Item 3	Legal Proceedings
	Item 4	Mine Safety Disclosures
Part II	Item 5	Market
	Item 6	Consolidated Financial Data
	Item 7	Management's Discussion and Analysis
	Item 7A	Quantitative and Qualitative Disclosures about Market Risks
	Item 8	Financial Statements
	Item 9	Changes in and Disagreements With Accountants
	Item 9A	Controls and Procedures
	Item 9B	Other Information
Part III	Item 10	Directors, Executive Officers and Corporate Governance
	Item 11	Executive Compensation
	Item 12	Security Ownership of Certain Beneficial Owners
	Item 13	Certain Relationships and Related Transactions
	Item 14	Principal Accounting Fees and Services
Part IV	Item 15	Exhibits and Financial Statement Schedules Signatures

Figure 4.1: The structure of the 10-k report

This project focused on textual data, and after examining these items, the following five items: Item 1, Item 1A, Item 3, Item 7, Item 7A were chosen, because they did not contain a tabular section.

This project downloaded the annual reports of 2000 (9,488 annual reports), 2019 (6,693 annual reports) and 2020 (6,944 annual reports), and extracted a part from the annual reports of each of the three years. Through the custom function, items 1, Item 1A, item 3, item 7, item 7A were extracted from each selected annual report. Then the extracted item 1, item 1A, item 3, item 7, and item 7A were used to

generate new JSON files, resulting in a total of about 24,586 JSON files, each file contains only a key-value pair, that is, the item name and the corresponding text content.

Considering that the BART model can only handle the largest sequence of 1024 tokens, this project picked 1000 JSON files of size 3KB-4KB from 24,586 JSON files, this is because the vocabulary size contained in the files in this size range happens to be 800-1000. In order to facilitate subsequent data processing, the key name of each JSON file was uniformly changed to “section”.

```
import os
import json
import openai
openai.api_key = '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'

# Get all the json files in the folder
folder_path = "/Users/wangyaoxuan/Downloads/final_version"
json_files = [pos_json for pos_json in os.listdir(folder_path) if pos_json.endswith('.json')]

# Each json file is processed
for file in json_files:
    file_path = os.path.join(folder_path, file)

    # Reading a json file
    with open(file_path, 'r', encoding='utf-8') as json_file:
        data = json.load(json_file)

    # Get the text that we want to summarize
    text = data.get('section')

    # GPT-3.5-turbo was used to generate summaries
    prompt = "I need a summary of the following text consisting of some complete sentences:\n\n" + text
    response = openai.Completion.create(engine="text-davinci-003", prompt=prompt, temperature=0,
    max_tokens=250)
    summary = response.choices[0].text.strip()

    # Add the summary to the original json data
    data['summary'] = summary
    # Write back to the json file
    with open(file_path, 'w', encoding='utf-8') as json_file:
        json.dump(data, json_file, ensure_ascii=False)
```

Figure 4.2: Generate reference summaries

As shown in figure 4.2, the OpenAI API was then called to generate summaries of the text contents in these 1000 JSON files using GPT-3.5-turbo model, and then each summary was reviewed to ensure that the summary contained important information from the original text. It is worth noting that the process of calling the OpenAI API incurs a cost. Therefore, only 1000 JSON files were processed for this project. After creating 1000 reference summaries, these summaries were added to the JSON

files of the corresponding original text, and then 1000 JSON files containing text-summary pairs were obtained.

These 1000 JSON files were used to generate a JSONL file where each line contained a text-summary pair. At this point, the new dataset was created. At the same time, in order to allow subsequent fine-tuning to proceed smoothly and not take too much time (e.g., more than 10 hours), only a portion of the new dataset was used for fine-tuning , based on the availability of computing resources.

The created dataset has been uploaded to hugging face and it can be accessed via the following link: `https://huggingface.co/datasets/wyx-ucl/SUM-DATASET-BASED-EDGAR-CORPUS`

4.3 Training process on Google Colab

This section details how to complete the fine-tuning of the facebook/bart-large-cnn model on Google Colab.

As shown in figure 4.3, first we need to install some Python packages: “transformers”, “datasets”, “numpy”, “nltk”, “rouge_score”, “torch”, “accelerate”. “transformers”, “datasets”, “accelerate” are all libraries developed by Hugging Face. The “transformers” library provides a number of pre-trained models, including the facebook/bart-large-cnn model, along with tools and API related to these models. It also provides tools and API for fine-tuning models. The “datasets” library provides a convenient way to download and use datasets and is used in this project to quickly access and process the previously created dataset of text summary pairs. “numpy” is a library for working with large, multidimensional arrays and matrices, and provides a large number of mathematical functions to operate on these arrays. It is needed to process and manipulate the data when fine-tuning the model. “nltk” is a natural language processing library that provides tools for text processing and a corpus of English language text. The “rouge_score” library provides tools for calculating the ROUGE score, which is used to evaluate the performance of the fine-tuned model. “torch” is an open-source deep learning library that provides a set of high-level API to build and train models. The BART model is built based on PyTorch, so this li-

brary is required. The “accelerate” library provides a simplified way to speed up the training of the model.

```
[ ] !pip install transformers datasets numpy nltk rouge_score torch accelerate -U
```

Figure 4.3: Install dependencies

After installing the above Python packages, we can import them or the functions in them.

```
[ ] from transformers import BartForConditionalGeneration, BartTokenizer, Seq2SeqTrainer, Seq2SeqTrainingArguments
    from transformers import DataCollatorForSeq2Seq
    import torch
    import numpy as np
    import nltk
    from nltk.tokenize import sent_tokenize
    from datasets import load_dataset, load_metric
    nltk.download('punkt')
```

Figure 4.4: Import libraries

Figure 4.5 shows the code for loading the model, the tokenizer, the dataset and splitting the dataset. The BartForConditionalGeneration.from_pretrained() and BartTokenizer.from_pretrained() methods are used to load the facebook/bart-large-cnn model and the tokenizer associated with the pretrained model, and the load_dataset function is used to load the previously created dataset of text summary pairs. The fine-tuning process uses a portion of the new data set and uploads it to the Google Colab notebook at “/content/ Combination.jsonl”. The loaded dataset is split into a training set and a test set using the train_test_split() method. The parameter “test_size=0.1” instructs to use 10% of the data as the test set and the remaining 90% as the training set.

```
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
dataset = load_dataset("json", data_files="/content/combine.jsonl")
train_dataset = dataset['train']
dataset = train_dataset.train_test_split(test_size=0.1)
```

Figure 4.5: Load model, tokenizer, dataset

After that, the contents of section and summary in the dataset are tokenized and

encoded, as shown in figure 4.6. A function named preprocess_function is defined. The first "inputs" list in the function contains all the documents obtained from the "section" key in the loaded dataset, and the first "outputs" list contains all the documents obtained from the "summary" key. The contents of section and summary in the dataset have the potential to exceed the maximum context size of the model. Therefore, the inputs and outputs lists need to be processed by the loaded tokenizer to tokenize and truncate the documents in each list. The parameters "max_length=1024" and "max_length=300" limit the maximum length to 1024 and 300, respectively, and the parameter "truncation=True" indicates that parts of the document that exceed the maximum length should be truncated. Next, a new key, "labels", is added to the "inputs" dictionary with the value taken from the "input_ids" key in the "outputs" dictionary, and tokenizer.eos_token_id (for "end") is added at the end. During training, the model will use these "labels" to calculate the loss between its prediction and the reference summary. The preprocess_function is applied to each element of the dataset through the map() method. The parameter "remove_columns=dataset["train"].column_names" allows us to remove all columns from the original dataset during processing.

```
def preprocess_function(examples):
    inputs = [doc for doc in examples['section']]
    outputs = [doc for doc in examples['summary']]
    inputs = tokenizer(inputs, truncation=True, max_length=1024)
    outputs = tokenizer(outputs, truncation=True, max_length=300)
    inputs['labels'] = [x + [tokenizer.eos_token_id] for x in outputs['input_ids']]
    return inputs

tokenized_dataset = dataset.map(preprocess_function, batched=True,remove_columns=dataset["train"].column_names)
```

Figure 4.6: Preprocess dataset

Figure 4.7 illustrates the creation of the evaluation function. The rouge metric library is loaded through the load_metric() method to calculate the ROUGE score between the model-generated summary and the reference summary. A function called compute_metrics is defined to evaluate the model during training. It takes a single argument eval_pred. The two elements of eval_pred are assigned to "predictions" and "labels". "predictions" contains the predictions of the model. "labels" contains the tokenized version of the reference summary. The numpy library is used to replace elements with a value of "-100" in labels with the padding token ID of

the tokenizer object, indicating that these labels should be ignored when calculating the loss. The tokenizer object's batch_decode() method is used to decode the token list of predictions and the token list of labels into lists of strings. The parameter "skip_special_tokens=True" indicates that special tokens (such as start and end tokens) should be ignored. Decoded predictions and labels are split into sentences using NLTK's sent_tokenize() method and combined using newline characters. The ROUGE library is then used to compute the ROUGE score between the decoded prediction and the label, with "use_stemmer=True" indicating that stemming should be used when calculating the score. Finally, the median score is extracted from the results and a dictionary is returned with ROUGE scores rounded to four decimal places.

```
metric = load_metric("rouge")

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    # Decode generated summaries into text
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
    # Replace -100 in the labels as we can't decode them
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    # Decode reference summaries into text
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
    # ROUGE expects a newline after each sentence
    decoded_preds = ["\n".join(sent_tokenize(pred.strip())) for pred in decoded_preds]
    decoded_labels = ["\n".join(sent_tokenize(label.strip())) for label in decoded_labels]
    # Compute ROUGE scores
    result = metric.compute(
        predictions=decoded_preds, references=decoded_labels, use_stemmer=True
    )
    # Extract the median scores
    result = {key: value.mid.fmeasure * 100 for key, value in result.items()}
    return {k: round(v, 4) for k, v in result.items()}
```

Figure 4.7: Evaluation function

Figure 4.8 shows instantiating the DataCollatorForSeq2Seq collator of the transformers library to dynamically pad the inputs and labels and defining the parameters associated with the training loop by the Seq2SeqTrainingArguments class. The parameter "predict_with_generate=True" indicates that summaries should be generated during evaluation in order to compute ROUGE scores for each epoch. The parameter "output_dir" is required to write the "bart-EDGAR-CORPUS" directory used to hold model predictions and checkpoints. A checkpoint holds a model and its associated configuration files for a given epoch. The parameter "evaluation_strategy" is used to set the evaluation strategy during training, and "epoch" means that the evaluation is performed at the end of each epoch. The parameter

"save_strategy" is used to set the saving strategy for checkpoints during training, with "epoch" meaning that saving is done at the end of each epoch. The parameter "learning_rate" controls the updating speed of model parameters. If the learning rate is too small, the convergence speed will be greatly reduced and the training time will be increased. By default, torch's multi-GPU mode is automatically enabled. The parameter "per_device_train_batch_size" sets the number of samples on each gpu used for training. The parameter "per_device_eval_batch_size" sets the number of samples per gpu for evaluation. The parameter "weight_decay" is used to make the model parameters as small as possible and as compact as possible. The parameter "save_total_limit" is used to limit the number of checkpoints in the output directory, and older checkpoints in the output directory are removed. The parameter "num_train_epochs" sets the total number of training epochs to perform. Enabling the parameter "load_best_model_at_end" will preserve the best checkpoint in the output directory. The parameter "logging_steps" is used to set the number of update steps between two logs.

```
data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)
batch_size = 16
num_train_epochs = 1
# Show the training loss with every epoch
logging_steps = len(tokenized_dataset["train"]) // batch_size
args = Seq2SeqTrainingArguments(
    output_dir=f"bart-EDGAR-CORPUS",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=5e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    weight_decay=0.01,
    save_total_limit=3,
    num_train_epochs=num_train_epochs,
    predict_with_generate=True,
    logging_steps=logging_steps,
    load_best_model_at_end=True
)
```

Figure 4.8: Set parameters

All the ingredients needed for training have been prepared. The final step involves instantiating the trainer using the standard parameters and then starting the training run.

```
trainer = Seq2SeqTrainer(model=
    model,
    args=args,
    train_dataset=tokenized_dataset["train"],
    eval_dataset=tokenized_dataset["test"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)


trainer.train()
```

Figure 4.9: instantiate the trainer

Once training is complete, run Trainer.evaluate() to see the final ROUGE score. The notebook used by this project to fine-tune the facebook/bart-large-cnn model can be accessed via the following link `https://colab.research.google.com/drive/17CuyYWezM2s2RbcsOUheaJjf8EWPHga3?usp=sharing` The fine-tuned model has been uploaded to hugging face and the model card can be accessed via the following link `https://huggingface.co/wyx-ucl/bart-EDGAR-CORPUS`

4.4 Summary

This chapter describes in detail the process of fine-tuning the BART model to be used in the application. In this project, the fine-tuning process consists of preparing the dataset and training the model on Google Colab.

Chapter 5

Design and implementation

This chapter describes the design of the system architecture and then describes the implementation details of the system components. It should be noted that this system is designed to run on the Replit cloud and therefore requires an internet connection. In addition, the user of this system is an external supervisor, Prof. Joseph Connor, so the case of having multiple concurrent users is not considered. In this chapter, starting with section 5.2, implementation details of various components are described.

5.1 System Architecture

The system has three interfaces. Figure 5.1 shows the workflow of the first interface. Interface 1 is responsible for the summary of financial news.

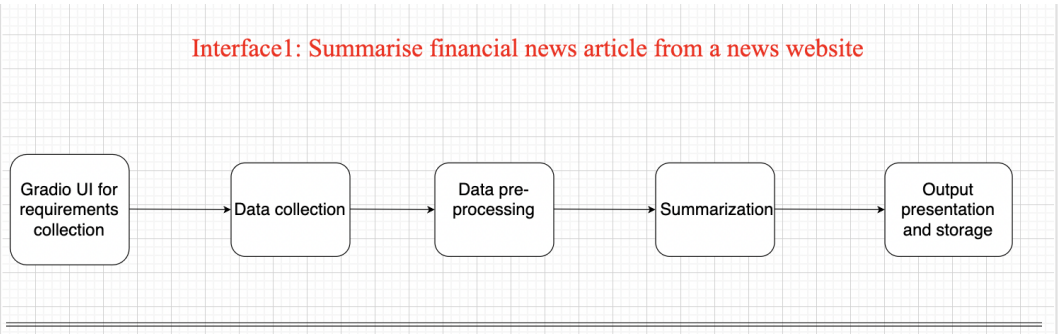

Figure 5.1: The workflow of interface 1

Figure 5.2 shows the workflow of the second interface. Interface 2 is responsible for the summary of PDF documents of annual reports of companies.

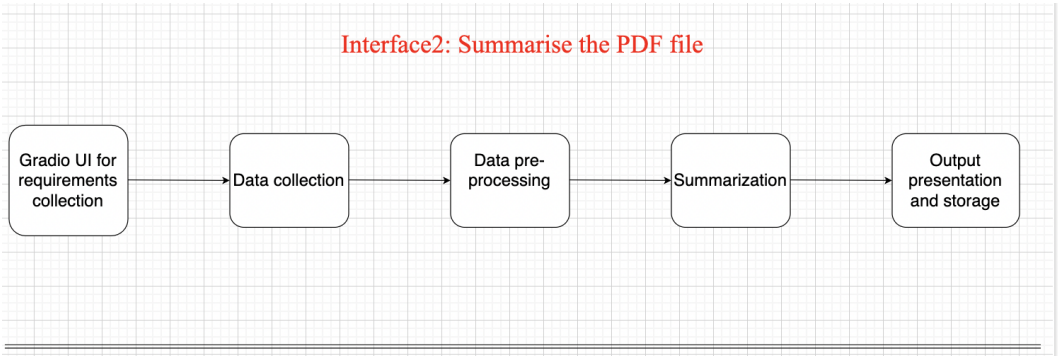

Figure 5.2: The workflow of interface 2

Figure 5.3 shows the workflow of the third interface. Interface 3 undertakes the question answering task based on the uploaded PDF file of the annual report.

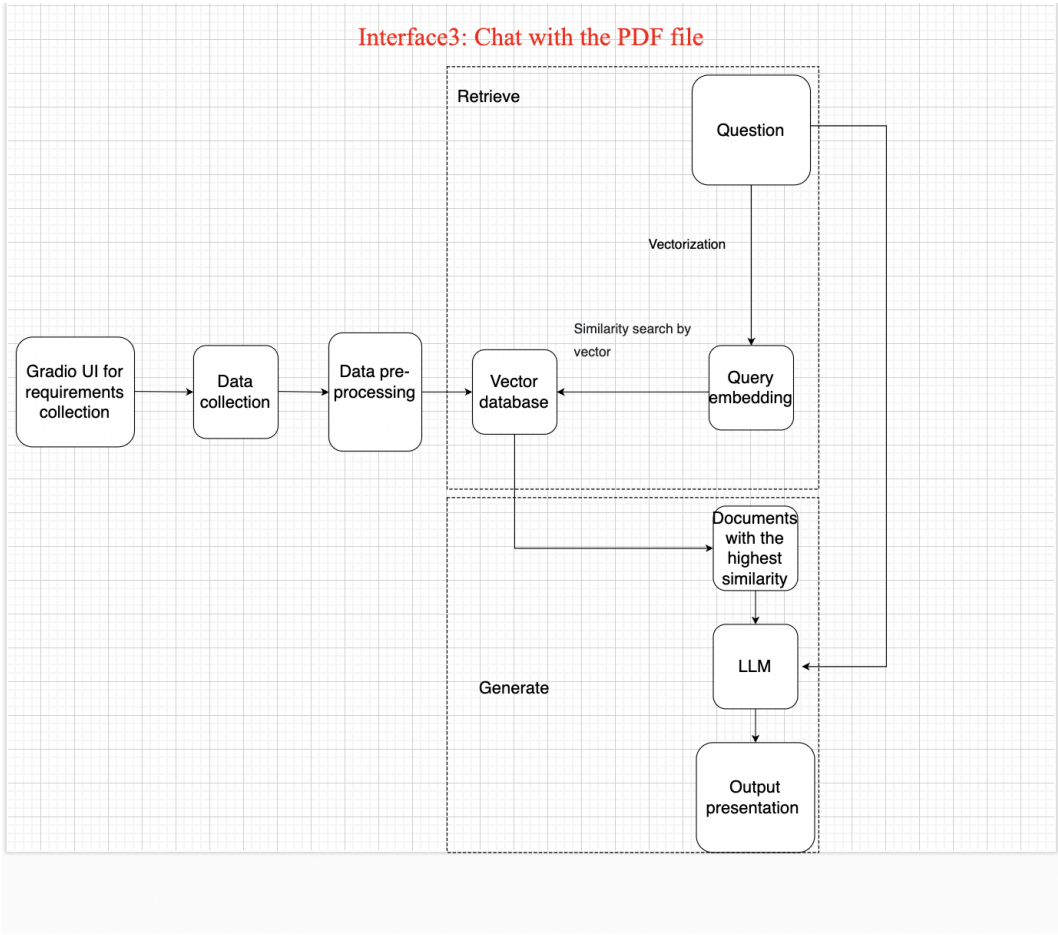

Figure 5.3: The workflow of interface 3

Interface 1 and Interface 2 have the same workflow. That is, they all consist of four components: Gradio UI for requirements collection, data collection, data pre-

processing, summarization, output presentation and storage. The workflow of interface 3 is somewhat different from the previous two interfaces. It basically consists of five components: Gradio UI for requirements collection, data collection, data pre-processing, retrieve and generate, three of which are the same as the first two interfaces.

5.1.1 Gradio UI for requirements collection

In interface 1, this component is used to collect the financial news links to be processed by the user, the user's preference for the large language model, and the user's requirement for the format type of the output summary file. In interface 2, it is used to collect the PDF file to be processed, the user's preference for the large language model, the user's requirement for the format type of the output summary file, and the user's requirement for the scope of the PDF file to be summarised. In interface 3, this component is used to collect the PDF file that the user wants to chat with and the user's question.

5.1.2 Data collection

Once a user submits their list of requirements through the Gradio UI, the text data needs to be extracted first. This is achieved in interface 1 through the tool Beautiful Soup. In Interface 2 and Interface 3, the system uses the PDF document loader in LangChain framework to load the annual report uploaded by users and extract text data.

5.1.3 Data pre-processing

In NLP tasks, it is needed to pre-process the text data before entering it into a large language model, because there may be a variety of problems with the raw text data that can affect the performance and results of the model. The purpose of pre-processing is to clean, transform, and prepare data for input into a large language model so that the model can better understand and process the text. In interface 1, this process includes text data cleaning as well as chunking. It is worth noting that the text chunking will vary depending on the choice of model, as the maximum number of tokens that BART model can handle is 1024 and the maximum number

of tokens that the GPT-3.5 model can handle is 4097 (both input and output). In Interface 2, the process involves the splitting of text data, and for the same reason as before, the splitting step varies depending on the choice of model. In interface 3, this process only involves splitting the extracted PDF text data using recursive splitter by character under the LangChain framework.

5.1.4 Summarization

For the automatic summarization task, after obtaining the pre-processed text data, we can enter the data into the selected large language model for summarization. In interface 1, if the selected model is BART model, methods in hugging face's transformers library and tokenizer library are called for summary; if the model selected is a GPT-3.5 model, text is summarized using the chain component in the LangChain framework. In Interface 2, if the model selected is a BART model, the LLMs component in the LangChain framework is used to call the fine-tuned BART model hosted on the hugging face for summarization; if the selected model is GPT-3.5 model, the chain component in the LangChain framework is used as in Interface 1.

5.1.5 Output presentation and storage

In the automatic summarization part of the system, we need to present the summaries obtained through the large language model and the link to the generated summary report file in the Gradio front end, and store the generated summary report file in the Google cloud storage.

5.1.6 Retrieve

This component is used for the question answering part of the system, it vectorises the collected question to get the query embedding, and retrieves the relevant split of the question using similarity search. In the data preprocessing stage, we get multiple documents after splitting. The content of each document is embedded using OpenAI's text embedding model, and then the embedding and the document are stored in a vector store, and the embedding will be used for document indexing.

5.1.7 Generate

This is the last component of the question answering part of the system. The documents retrieved and the question are sent to the GPT-3.5 model to generate an answer. The generated answer and the content of the retrieved document with the highest similarity to the question are presented in the Gradio front-end.

5.2 Gradio UI

This section describes the main components of 3 interfaces and their functionality. As shown in Figure 5.4, the user can summarize the financial news through interface 1. The user can input the web address of the financial news through the text box component of Gradio, and can select a large language model to use and select the file type of the output summary report through the check box group component of Gradio. After completing the above input and preference selection, the user can use the submit button to let the system run and display the summary and a link to the summary file stored in google cloud storage in two text box components.

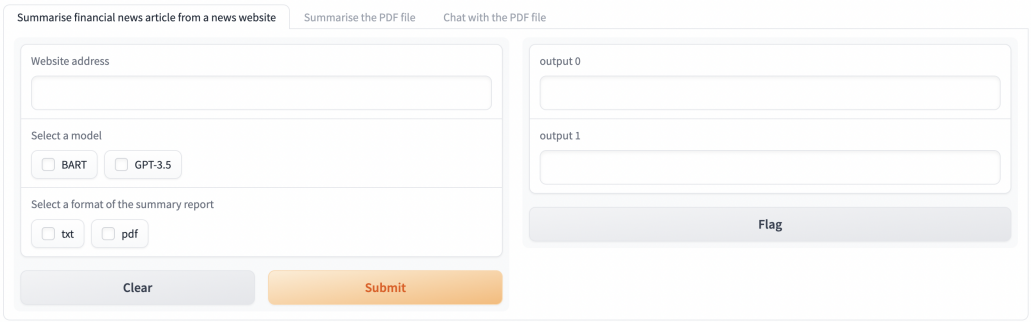

Figure 5.4: Interface 1

As shown in Figure 5.5, in interface 2, the user can upload a PDF file of an annual report through the file component of Gradio and summarize the contents of some pages in it. Similar to Interface 1, the user can select a large language model and select the file type of the output summary report through the checkbox group component of Gradio. The difference is that the user can specify a range of page numbers through the slider component of Gradio and let the system summarize the

contents of the selected range. After uploading the file and selecting preferences, the user can use the submit button to get the system running and display the summary and a link to the summary file stored in google cloud storage in two text box components.

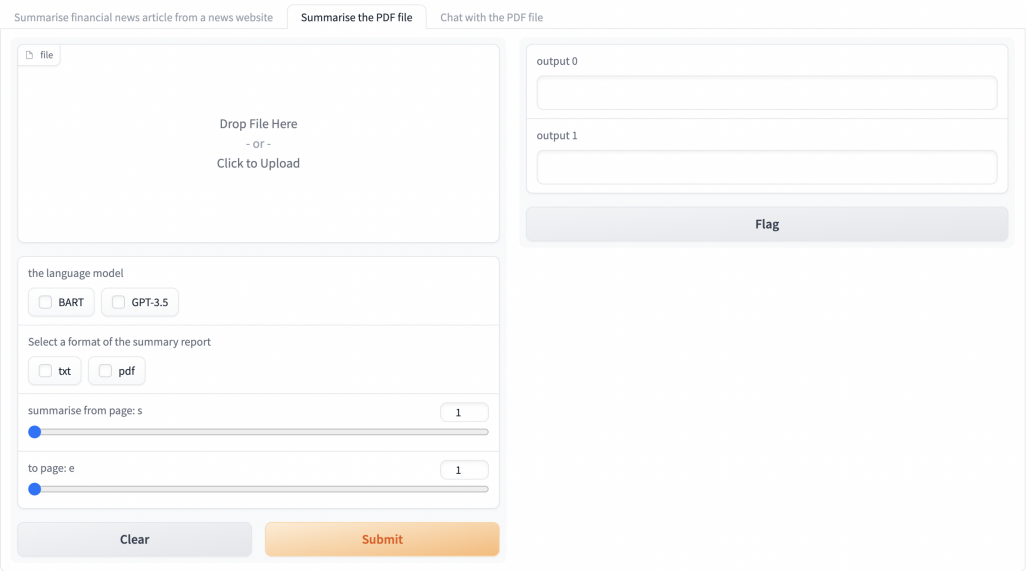

Figure 5.5: Interface 2

As shown in Figure 5.6, in interface 3, the user can upload a PDF file of an annual report through Gradio's file component, and then enter a question related to the annual report through Gradio's text box component for GPT-3.5 model to answer. The answers will be displayed in the text box named "Answer". And in the text box named "Document Similarity Search", the contents of the uploaded annual report with the highest similarity to the question will be displayed. There is a clear button in each interface. This button can let the user clear all the operations in the current interface.

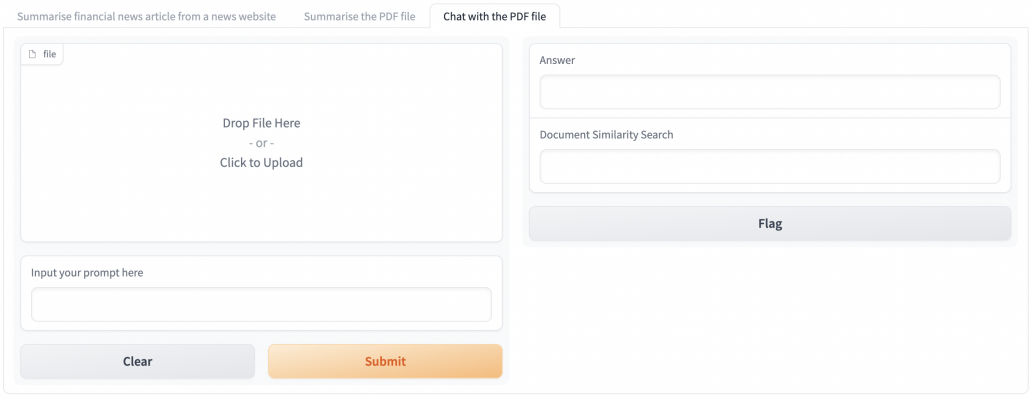

Figure 5.6: Interface 3

5.3 Data collection

In interface 1, after the user clicks "Submit" button, the system backend will first check whether the URL is entered. If the URL is empty, the prompt "please input a valid URL" is returned and displayed on the Gradio front end. If the URL entered is valid, a GET request is made to the given URL using the requests library and the response is parsed using BeautifulSoup. In the parsed HTML content, all "figure" tags are looked for and cleared, this is done to remove unnecessary content or media from the body of the article. If the URL entered is invalid, it will still return "please input a valid URL". Here are the steps to capture the article content: initialise a string article_Body to store the content of the article, then retrieve the content of the "article" tag from the parsed HTML and store it in the body variable, look for all "p" tags within the "article" tag, then retrieve the text of these paragraphs and append them to the article_Body.

```
def scrape_data(url,mo,file_type):
    if not url:
        output="please input a valid URL"
        link ="please input a valid URL"
        return output,link
    try:
        response = requests.get(url)
        soup = BeautifulSoup(response.content, 'html.parser')
        for tag in soup.find_all("figure"):
            tag.clear()
        article_Body = " "
        body = soup.article

        for data in body.find_all("p"):
            article_Body += data.get_text()

```

Figure 5.7: Data collection of interface 1

When the user uses interface 2 or 3, the system backend will first check whether the file exists. If no file is provided, the system will return a message prompting the user to upload the file. In particular, if the user does not enter a question in interface 3, the system will return a message prompting the user to enter a question. The system backend loads and processes the PDF file, using the LangChain framework's document loader to load the data from the source file as a document, which is a piece of text and associated metadata. Since the file type of the uploaded file is PDF, we use PyPDFLoader. Here "file.name" is the path to the file. After that, the load_and_split() method is called to load the contents of the PDF file and split it into multiple documents, where each document contains one page of the source file and metadata with the page number. The result is stored in the "pages" variable. The advantage of this approach is that documents can be retrieved using page numbers.

```
def extract_text_from_pdf(file, mod,file_type, startpage, endpage,):
    if file is None:
        output2="Please upload a file"
        link="Please upload a file"
        return output2,link

    loader = PyPDFLoader(file.name)
    pages = loader.load_and_split()

def chat(file, prompt):
    if file is None:
        return "Please upload a file","Please upload a file"
    if not prompt:
        return "Please enter a question","Please enter a question"
    loader = PyPDFLoader(file.name)
    pages = loader.load_and_split()
```

Figure 5.8: Data collection of interface 2, 3

5.4 Data pre-processing

After collecting the text data, we need to pre-process it. This is reflected in the back end of interface 1 as text cleaning and text chunking. We first use the NLTK library's sent_tokenize method to split text into sentences. We then iterate over each sentence, replacing hyphens and newlines with spaces using regular expressions, processing abbreviations using the contractions library (for example, replacing "it's" with "it is"), removing all parentheses and their contents, and converting the sentence to lowercase. Finally, the processed sentences are added to the processed_sentences list. For each sentence in processed_sentences, check if it contains URLs or email addresses, and if not, add the sentence to the arr2 list.

We iterate over each sentence in arr2 and do the following: split the sentence

into words using nltk.word_tokenize, check the length of each word; if the word is longer than 12 characters, use a custom function infer_spaces to split the word and add it to words_list2; otherwise, just add the word to words_list2. The processed sentence is added to the processed_sentences2 list, and all the sentences in processed_sentences2 are merged into a single text, which is stored in the "article" variable.

```
def text_cleaning(text)
    z = nltk.sent_tokenize(text)
    processed_sentences = []
    processed_sentences2 = []
    arr2 = []
    for sentence in z:
        sentence = re.sub(r'-[\n\r]+', "", sentence)
        sentence = re.sub(r'-', ' ', sentence)
        sentence = re.sub(r'\n', ' ', sentence)
        sentence = contractions.fix(sentence)
        sentence = re.sub(r'\([^()]*\)', '', sentence)
        sentence = sentence.lower()
        processed_sentences.append(sentence)
    for s in processed_sentences:
        m = re.search(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+', s)
        n = re.search(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', s)
        if m:
            pass
        elif n:
            pass
        else:
            arr2.append(s)
    for sen in arr2:
        words_list = nltk.word_tokenize(sen)
        words_list2 = []
        for word in words_list:
            if len(word) > 12:
                w = infer_spaces(word)
                u = w.split()
                for q in u:
                    words_list2.append(q)
            else:
                words_list2.append(word)
        recombined_string = ' '.join(words_list2)
        processed_sentences2.append(recombined_string)
    article = ' '.join(processed_sentences2)
    return article
```

Figure 5.9: Text cleaning

After the text has been cleaned, the backend will split the text into chunks. As shown in figure 5.10, if the user has chosen the BART model in interface 1, the backend do this via the preprocess_text function, which uses the NLTK sentence tokenizer and the BART tokenizer to process and segment the input text. The aim is to ensure that the number of tokens per text chunk does not exceed 980. Because the BART model has a token limit (usually 1024 tokens). There is some room to avoid any problems, so a limit of 980 is set here.

```
def preprocess_text(text):
    # Segmenting Text with NLTK's Sentence Segmenter
    tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
    sentences = nltk.sent_tokenize(text)
    chunks = []
    chunk = ""
    for sentence in sentences:
        temp_chunk = chunk + " " + sentence if chunk else sentence
        if len(tokenizer.tokenize(temp_chunk)) <= 980:
            chunk = temp_chunk
        else:
            chunks.append(chunk)
            chunk = sentence
    if chunk:
        chunks.append(chunk)
    return chunks
```

Figure 5.10: Chunking of interface 1(BART)

LangChain has a number of built-in document tranformers that make it easy to split documents. As shown in figure 5.11, if the user selects the GPT-3.5 model in interface 1, the backend uses RecursiveCharacterTextSplitter to split the text. The text splitter takes a list of characters. It tries to create chunks based on the split of the first character, but if any chunk is too big, it moves to the next character, and so on. Here "chunk_size=1000" refers that the maximum chunk size is 1000 characters, and "chunk_overlap=0" refers that the maximum overlap between chunks is 0.

```
    # chunking
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    texts = text_splitter.split_text(text)

```

Figure 5.11: Chunking of interface 1 (GPT-3.5)

As shown in figure 5.12, if the user chooses to use the BART model in interface 2, the model's tokenizer is loaded first, and a CharacterTextSplitter object is initialised using that tokenizer. This object is used to split the document into small chunks so that each chunk is no more than 980 tokens and the maximum overlap between chunks is 20 tokens.

```
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
text_splitter = CharacterTextSplitter.from_huggingface_tokenizer(tokenizer, chunk_size=980, chunk_overlap=20)
split_docs = text_splitter.split_documents(docs)
```

Figure 5.12: Chunking of interface 2

As shown in figure 5.13, if the user selects the GPT-3.5 model in interface 2, then RecursiveCharacterTextSplitter is used to split the text by character. Similarly, in the backend of interface 3, the backend only uses RecursiveCharacterTextSplitter

to split the text. Here “chunk_size=1000” refers that the maximum chunk size is 1000 characters, and “chunk_overlap=0” refers that the maximum overlap between chunks is 0.

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
split_docs = text_splitter.split_documents(docs)
```

Figure 5.13: Chunking of interface 2, 3

5.5 Summarization

As shown in figure 5.14, in interface 1, for the summarization task using the BART model, we define the “summarize” function, which uses the pre-trained BART model (facebook/bart-large-cnn) for summarization. The BART tokenizer and the “facebook/bart-large-cnn” model are first loaded. The text is split into chunks using the previously defined preprocess_text function to ensure that each chunk can be processed by the BART model. The encode method of BART’s tokenizer is used to convert each text chunk into a numerical code (also known as token IDs), which is acceptable to the model. The parameter return_tensors=‘pt’ instructs the tokenizer to return a PyTorch tensor. The generate() method of the BART model is used to generate a summary for the input (the encoded text chunk), and then the decode() method of the BART tokenizer is used to convert the generated summary (currently saved as a digital encoding) back to human readable text. The parameter “summary_ids[0]” instructs to take the first element of the summary tensor. Finally, the summaries of text chunks are concatenated into a string and returned.

```
def summarize(text):
    tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
    model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')
    chunks = preprocess_text(text)
    summaries = []
    for chunk in chunks:
        # Generate a summary for each chunk
        inputs = tokenizer.encode(chunk, return_tensors='pt')
        summary_ids = model.generate(inputs, num_beams=2, min_length=0, max_length=150)
        summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
        summaries.append(summary)

    # Stringing together the generated summaries
    return " ".join(summaries)
```

Figure 5.14: Summary in interface 1 (BART)

As shown in figure 5.15, in Interface 1, for the summarisation task using the GPT-3.5 model, we use the load_summarize_chain provided by LangChain to directly

summarise the preprocessed text chunks. And the MapReduce document chain first applies the LLM chain to each chunk individually, treating the chain output as a new document. It then passes all new documents to a separate combined documents chain for a single output.

```
def summarize_with_gpt3(text):
    llm = OpenAI(temperature=0, openai_api_key=OPENAI_API_KEY, model_name="text-davinci-003")
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    texts = text_splitter.split_text(text)
    docs = [Document(page_content=t) for t in texts]
    chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=False)
    response = chain.run(docs)
    return response

```

Figure 5.15: Summary in interface 1 (GPT-3.5)

As shown in figure 5.16, in Interface 2, for the summarization task using the BART model, we connect to the previously fine-tuned BART model in the Hugging Face Hub, and then call the predict() method to summarise each preprocessed chunk of text, and finally concatenate these summaries.

```
def summarize_docs2(docs):
    llm2 = HuggingFaceHub(repo_id="wyx-ucl/bart-EDGAR-CORPUS", model_kwargs={"max_length": 200})
    text_splitter = CharacterTextSplitter.from_huggingface_tokenizer(tokenizer, chunk_size=980, chunk_overlap=20)
    split_docs = text_splitter.split_documents(docs)
    w = []
    for doc in split_docs:
        page_contents = doc.page_content
        v = llm2.predict(page_contents)
        w.append(v)
    return " ".join(w)
```

Figure 5.16: Summary in interface 2 (BART)

As shown in figure 5.17, in Interface 2, for the summarization task using the GPT-3.5 model, we also use the load_summarize_chain provided by LangChain to directly summarise the preprocessed text chunks.

```
def summarize_docs(docs):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    split_docs = text_splitter.split_documents(docs)
    chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=False)
    response = chain.run(input_documents=split_docs)
    return response
```

Figure 5.17: Summary in interface 2 (GPT-3.5)

5.6 Output presentation and storage

As shown in figure 5.18, when the user selects the txt file type, the temporary file is first given a unique filename and then the summary text is written to the temporary file. The backend then initializes an object that communicates with the Google

Cloud Storage service, fetches the bucket, and uploads the file to Google Cloud Storage. Finally, a public link is generated, and the user can download the summary report file through the link. In the case where the user selects a pdf file type, a PDF file with a unique name is first created, and then the summary text is written to that file. Similarly, the backend then initialises an object that communicates with the Google Cloud Storage service, acquires the bucket, and then uploads the file to Google Cloud Storage. Finally, a public link is generated through which the user can download the summary report file.

```
        if 'txt' in file_type and len(file_type) == 1:
            le = str(hash(url))
            typ = str(file_type[0])
            filename = "summary_gpt" + le + "." + typ
            with open(filename, 'w') as f:
                f.write(output)
                # Create a storage client
            storage_client = storage.Client()
            # Get the bucket that the file will be uploaded to
            bucket = storage_client.get_bucket('0073')
            # Create a new blob and upload the file's content
            blob = bucket.blob(filename)
            blob.upload_from_filename(filename)
            # Make the blob publicly viewable
            blob.make_public()
            link = blob.public_url

        elif 'pdf' in file_type and len(file_type) == 1:
            le = str(hash(url))
            typ = str(file_type[0])
            filename = "summary_gpt" + le + "." + typ
            pdf = PDFDocument(filename)
            pdf.init_report()
            pdf.h1('Summary')
            pdf.p(output)
            pdf.generate()
            # Create a storage client
            storage_client = storage.Client()
            # Get the bucket that the file will be uploaded to
            bucket = storage_client.get_bucket('0073')
            # Create a new blob and upload the file's content
            blob = bucket.blob(filename)
            blob.upload_from_filename(filename)
            # Make the blob publicly viewable
            blob.make_public()
            link = blob.public_url
```

Figure 5.18: Google Cloud Storage

5.7 Retrieve

As shown in figure 5.19, firstly, we need to set the environment variable which contains the key of OpenAI. OpenAIEmbeddings is a class used to communicate with OpenAI and get the embedding representation of the text, through the embedding class, we can create embeddings for all the chunks of text obtained after the data preprocessing. Then they are stored in a vector database. The vector database used in this project is Chroma, which is based on its lightweight, small memory, and easy-to-use features.

“Chroma.from_documents” indicates that the from_documents method of the Chroma class is being called, the purpose of this method is to build a vector store from the supplied collection of documents. “split_docs” is the first parameter to the from_documents method, which is a collection of text blocks previously obtained in the data preprocessing stage. “embeddings” is the second parameter, which is the openai embedding model defined earlier. “collection_name=‘annualreport’” is a named parameter passed to the method, which names the vector repository “annualreport” to make it easier to retrieve or manipulate this particular collection later.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma

os.environ['OPENAI_API_KEY'] = "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"
OPENAI_API_KEY = os.environ['OPENAI_API_KEY']
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
vectorstore = Chroma.from_documents(split_docs, embeddings, collection_name='annualreport')
docs = vectorstore.similarity_search(prompt, 3, include_metadata=True)
```

Figure 5.19: Chroma

Next the similarity search method is used to find documents relevant to the question entered, the “docs” variable will store the retrieved documents. “vectorstore.similarity_search” indicates that the similarity_search method of the vectorstore object is being called, which searches the stored document collection for the documents most similar to a given prompt. “prompt” is the first parameter passed to similarity_search method, which is the text or query to be compared with the documents in the vector store, and which is manually entered by the user in the front-end. “3” is the second parameter passed, which indicates the number of most similar documents obtained from the search. “include_metadata=True” indicates that in addition to returning documents that are similar to prompt, metadata associated with those documents (i.e., the source of the document and the page number where it is located) is also returned.

5.8 Generate

After retrieval, the 3 documents that are most similar to the user’s question can be obtained, and these 3 documents along with the question are passed to the GPT-3.5 model to get an answer. As shown in figure 5.20, firstly, the language model used

to answer the question is initialised, here an object called llm is created which is an instance of the OpenAI class. “temperature=0” is a parameter that is usually used to control the randomness of the model’s output, a value of 0 means that the output will be very deterministic. The higher the value, the more random the output will be. The temperature is set to 0 in order to get the highest fidelity, most reliable answer. “openai_api_key=OPENAI_API_KEY” passes the OpenAI API key to the model so that the model can access OpenAI’s services. “model_name=“text-davinci-003” ” specifies the version of the model to be used.

```
from langchain.chains.question_answering import load_qa_chain

llm = OpenAI(temperature=0, openai_api_key=OPENAI_API_KEY, model_name="text-davinci-003")
chain2 = load_qa_chain(llm, chain_type="stuff", verbose=False)
re = chain2.run(input_documents=docs, question=prompt)
first_page_content = docs[0].page_content
first_source = docs[0].metadata['page']
```

Figure 5.20: Generate

“load_qa_chain” is the chain used for dialogue based on retrieved documents. The previously created llm is used as its first parameter. “chain_type=“stuff” ” specifies the type of QA chain to be loaded. The stuff documents chain is the most straightforward document chain, inserting the document collection and questions into the prompt and passing the prompt to the LLM. This chain is ideal for applications where the documents are small and most calls pass only a few documents. “verbose=False” is a parameter that controls the detail of the output, setting it to False means that the log will not be displayed.

By using the run method to perform the actual question and answer operation. “input_documents=docs” provides the set of documents to be considered. The docs variable comes from the previous similarity search and contains documents that are similar to the question. “question=prompt” passes the question that the QA chain will use to find the answer in the provided document collection. “first_page_content” variable is used to store the text content of the first document previously retrieved. And “first_source” is used to store the page number of the first document previously retrieved. The purpose of this is to allow the user to see the text content of the original document with the highest relevance to the question and the page it is on in the front end.

Chapter 6

Testing

Testing an application before it is released is critical to reveal bugs and deficiencies in the code and to ensure that the application functions correctly and meets the expected requirements. There are four levels of testing: unit testing, integration testing, system testing, and acceptance testing. The goal of unit testing is to test whether a single unit of software is working as designed, in this case a unit is usually a function. Unit testings are usually followed by integration testings, which combine individual units and test them as a group in order to check for possible point failures in the interactions between units. System testing aims to assess the compliance of the system to specified requirements, testing complete and integrated software. Acceptance testing is the final phase of software testing, in which actual users test system acceptability against specifications to ensure that the system can handle the required tasks in realistic scenarios.

6.1 Unit testing and integration testing

Python unittest unit test framework provides a rich set of tools for building and running tests. This project used the unittest module for unit testing and integration testing. The basic building blocks of the system are some functions. The unittest module is used to test each fairly independent function that does not depend on other functions to ensure that they work properly, that is, unit testing. A module consisting of several functions is then tested, which is an integration testing.

```python
import nltk
from transformers import BartTokenizer, BartForConditionalGeneration
nltk.download('punkt')
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')
def preprocess_text(text):
    # Segmenting Text with NLTK's Sentence Segmenter
    sentences = nltk.sent_tokenize(text)
    chunks = []
    chunk = ""
    for sentence in sentences:
        temp_chunk = chunk + " " + sentence if chunk else sentence
        if len(tokenizer.tokenize(temp_chunk)) <= 980:
            chunk = temp_chunk
        else:
            chunks.append(chunk)
            chunk = sentence
    if chunk:
        chunks.append(chunk)
    return chunks

def summarize(text):
    chunks = preprocess_text(text)
    summaries = []
    for chunk in chunks:
        # Generate a summary for each chunk
        inputs = tokenizer.encode(chunk, return_tensors='pt')
        summary_ids = model.generate(inputs, num_beams=2, min_length=0, max_length=150)
        summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
        summaries.append(summary)

    # Stringing together the generated summaries
    return " ".join(summaries)
```

Figure 6.1: preprocess_text and summarize functions

```python
import unittest
import nltk
from my_module import preprocess_text,tokenizer
class TestPreprocessText(unittest.TestCase):
    def setUp(self):
        nltk.download('punkt')  # Ensure the punkt tokenizer is available for nltk.sent_tokenize

    def test_single_sentence(self):
        text = "This is a single sentence."
        result = preprocess_text(text)
        self.assertEqual(len(result), 1)
        self.assertEqual(result[0], text)

    def test_large_text_splitting(self):
        text = "This is sentence one. " * 500 + "This is sentence two. " * 500
        result = preprocess_text(text)
        # Ensure the large text is split into chunks
        self.assertTrue(len(result) > 1)

    def test_chunk_size(self):
        text = "This is a test sentence. " * 1000
        result = preprocess_text(text)
        for chunk in result:
            # Ensure every chunk has a token count less than or equal to 980
            self.assertLessEqual(len(tokenizer(chunk)), 980)

    def test_chunk_boundary(self):
        text = "This is sentence one. " * 980 + "This is sentence two."
        result = preprocess_text(text)
        # The last sentence should be in a separate chunk due to token limit
        self.assertEqual(len(result), 2)
        self.assertTrue("This is sentence two." in result[1])

if __name__ == '__main__':
    unittest.main()
```

Figure 6.2: Unit testing performed on the preprocess_text function

For example, Figure 6.1 shows the preprocess_text and summarize functions. The preprocess_text function, mentioned in section 5.4, will be used to split the text into chunks if the user selects the BART model in interface 1. The preprocess_text function in the system is used to split the text content that has been captured from the news website and has gone through the text data cleaning step. After processing by the preprocess_text function, the long text is split into chunks with a token size of no more than 980. If the user selects the BART model in Screen 1, the summary function mentioned in Section 5.5 will be used to summarise and merge summaries of text blocks separately. The summarize function depends on the result of preprocess_text function. First, the preprocess_text function is tested to make sure it can split text of different lengths according to a set maximum token length. The summarize function is then tested, which is equivalent to integration testing, because

the summarize function calls the preprocess_text function. And the integration testing ensures that it works correctly with its dependency, the preproces_text function. The purpose of integration testing is to check that the overall flow is working as expected, not just to test individual functions. Figure 6.2 shows the unit testing performed on the preprocess_text function, this project created several test cases to test the preprocess_text function:

- Test whether a single sentence returns a chunk.
- Test whether the large text is split into multiple chunks.
- Test whether the size of each chunk meets the requirements.
- The chunk boundary condition is tested to ensure that new sentences will be put into a new chunk when the limit of 980 tokens is approached.

```python
import unittest
import nltk
from transformers import BartTokenizer, BartForConditionalGeneration
from my_module import preprocess_text
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')

def summarize(text):
    chunks = preprocess_text(text)
    summaries = []
    for chunk in chunks:
        # Generate a summary for each chunk
        inputs = tokenizer.encode(chunk, return_tensors='pt')
        summary_ids = model.generate(inputs, num_beams=2, min_length=0, max_length=150)
        summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
        summaries.append(summary)

    # Stringing together the generated summaries
    return " ".join(summaries)

class TestSummarizeIntegration(unittest.TestCase):
    def setUp(self):
        nltk.download('punkt')

    def test_integration_with_preprocess(self):
        # A long text that exceeds the BART tokenizer limit
        long_text = "This is a test sentence. " * 1000
        summary = summarize(long_text)
        # Ensure a summary is returned
        self.assertIsNotNone(summary)
        # The summary should be shorter than the input
        self.assertLess(len(summary), len(long_text))

if __name__ == '__main__':
    unittest.main()
```

Figure 6.3: Integration testing of the summarize function

Figure 6.3 shows an integration testing of the summarize function. We created a test case where the input is a long text that exceeds the maximum processing length of the BART model to test whether the summarize function is able to summarise it successfully. After thorough unit testing and integration testing, we evaluated the main functions and ran at least one test case for each of them.

6.2 System Testing

We need to test whether the three interfaces of the system work properly and evaluate the quality of the summaries obtained when using the system for summarization tasks. We use cosine similarity as a criterion to measure the degree of thematic similarity between the generated summaries and the original text, where the closer the cosine similarity between the generated summaries and the original text is to 1 the greater the degree of thematic similarity. In this project, we used Promptfoo for system testing, which is a tool for testing and evaluating the quality of LLM outputs, allowing systematic testing of the programme against predefined test cases. We can generate a matrix view via promptfoo and to compare the summary results of BART model and GPT-3.5 model side by side, as well as evaluate the quality of the summary results. Figure 6.4 shows the Promptfoo workflow.

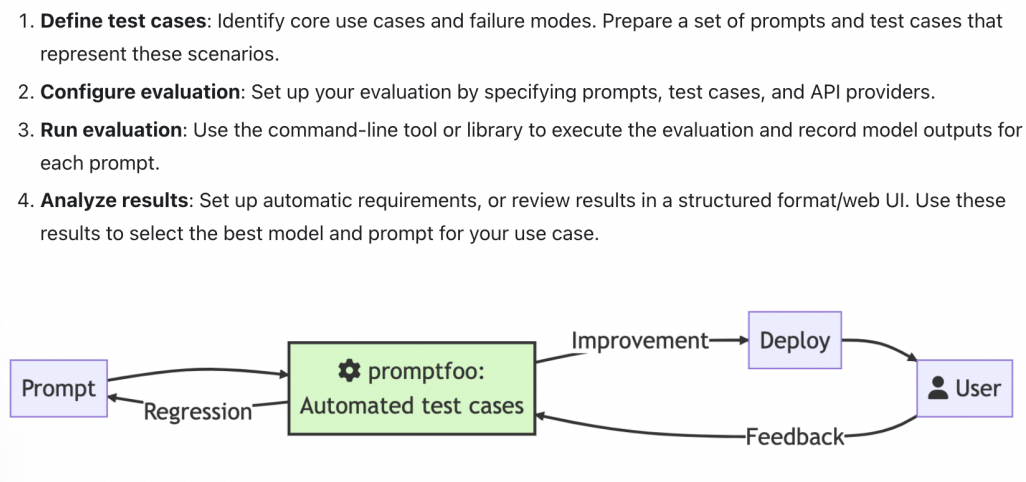

Figure 6.4: The Promptfoo workflow

Promptfoo has been integrated with Python Notebook and we have tested using

promptfoo in Google Colab. The following describes how to use promptfoo for system testing in Google Colab. Before the formal test we need to set up and configure promptfoo. First, the Node.js dependencies are installed, as shown in Figure 6.5.

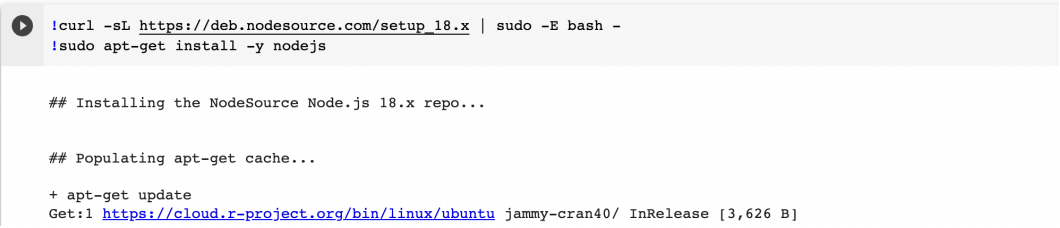

Figure 6.5

And next, as shown in Figure 6.6, we install and initialize promptfoo.

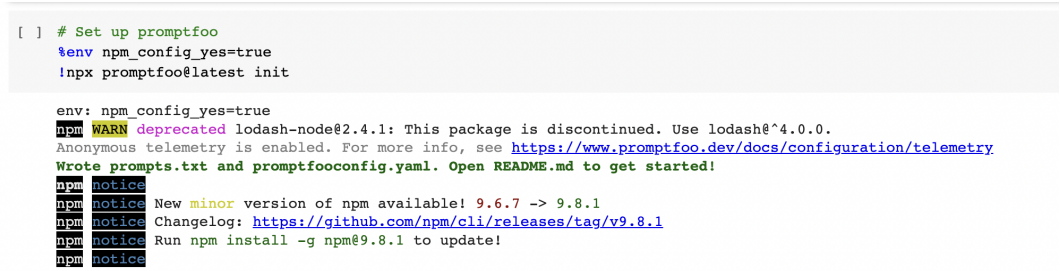

Figure 6.6

After setting up promptfoo, we can configure promptfoo. As shown in Figure 6.7, first, we set up the prompt, OpenAI API key, and install the libraries needed for subsequent python scripts.

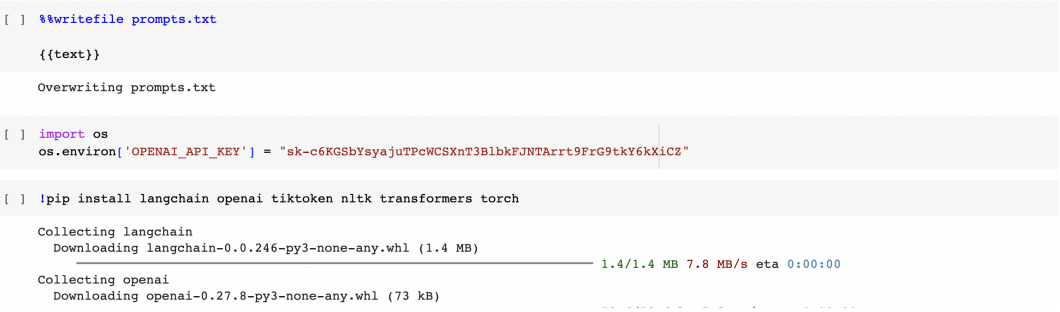

Figure 6.7

We then create two scripts that accept prompts and generate output to test and compare the summary results of financial news when using two different LLMs. The

script in Figure 6.8 uses the GPT-3.5 model , and the script in Figure 6.9 uses the BART model.

```
[ ] %%writefile langchain_example.py
    import os
    import sys
    from langchain.llms import OpenAI
    from langchain.docstore.document import Document
    from langchain.chains.summarize import load_summarize_chain
    from langchain.text_splitter import RecursiveCharacterTextSplitter
    def summarize_docs(text):
        text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
        texts = text_splitter.split_text(text)
        docs = [Document(page_content=t) for t in texts]
        llm = OpenAI(temperature=0, openai_api_key=os.getenv('OPENAI_API_KEY'), model_name="text-davinci-003")
        chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=False)
        response = chain.run(docs)
        return response
    prompt = sys.argv[1]
    o=summarize_docs(prompt)
    print(o)

    Writing langchain_example.py
```

Figure 6.8

```
[ ] %%writefile langchain_example2.py
    from transformers import BartForConditionalGeneration, BartTokenizer
    import nltk
    import sys

    def summarize(text):
        nltk.download('punkt')
        tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
        model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')
        sentences = nltk.sent_tokenize(text)
        chunks = []
        chunk = ""
        for sentence in sentences:
            temp_chunk = chunk + " " + sentence if chunk else sentence
            if len(tokenizer.tokenize(temp_chunk)) <= 510:
                chunk = temp_chunk
            else:
                chunks.append(chunk)
                chunk = sentence
        if chunk:
            chunks.append(chunk)
        summaries = []
        for chunk in chunks:
            # Generate a summary for each chunk
            inputs = tokenizer.encode(chunk, return_tensors='pt')
            summary_ids = model.generate(inputs, num_beams=2, min_length=0, max_length=150)
            summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
            summaries.append(summary)
        output = " ".join(summaries)
        return output
    prompt = sys.argv[1]
    o=summarize(prompt)
    print(o)
```

Figure 6.9

Next, we set up the test case, as shown in figure 6.10. the “text” and the “value” of the test case are written to the article content crawled from the news website, and “text” is used as input to get the summary via python script. The assertion checks whether the LLM output matches the given scoring criteria, i.e., the summary result is compared with the text data in “value” for cosine similarity. If the comparison

result is higher than the set threshold, i.e., 0.8, a “PASS” is displayed.

```
[ ] %%writefile promptfooconfig.yaml
    prompts: prompts.txt
    providers:
      - exec:python langchain_example.py
      - exec:python langchain_example2.py
    tests:
      - vars:
          text: In the past week, a combination of second-quarter earnings and newly released economic data have depicted a r
        assert:
          - type: similar
            value: In the past week, a combination of second-quarter earnings and newly released economic data have depicted
            threshold: 0.8

    Overwriting promptfooconfig.yaml
```

Figure 6.10

Finally, we run eval for comparison. Figure 6.11 shows the generated matrix view, the “text” column contains the original text, the second column is the summary generated using the BART model, and the third column is the summary generated using the GPT-3.5 model. Both generated summaries show “PASS”. This indicates that the values of cosine similarity between the two generated summaries and the original text are higher than the set threshold, 0.8. It means that the two summaries are highly similar to the original text in terms of content or context. There is reason to believe that summarising financial news using interface 1 produces reliable results.

```
[ ] !npx promptfoo@latest eval -c promptfooconfig.yaml --no-progress-bar
```

text	[exec:python langchain_example.py] {{text}}	[exec:python langchain_example2.py] {{text}}
In the past week, a combination of second-quarter earnings and newly released economic data have depicted a resilient American consumer, propelling stocks higher despite ongoing fears of a looming recession. The incoming data might even be bullish enough to assume a recessions isn't imminent at all."Consumption is 70% of the economy," Goldman Sachs Chief Economist Jan Hatzius said during a media roundtable on Wednesday. "So, you would generally expect these things to be fairly closely related, GDP growth versus consumer spending growth."Hatzius sees gross domestic product (GDP), the measurement used to indicate economic expansion or contraction, remaining positive. He predicts the growth will be "unspectacular," but it's growth nonetheless.Recent reports in retail sales and job additions back that opinion. In June, retail sales increased 0.2% from the month prior while the US economy added 209,000 jobs. Both metrics grew less than economists had projected. But when the discussion is centered around economic expansion versus contraction, growth matters.Earlier this week, Goldman Sachs pushed back its estimates for a recession. After seeing a 35% chance back in	[PASS] Goldman Sachs recentl y revised their estimates for a recession in the next 12 months from 35% to 20%, largely due to better-than-expected economic data and consumer spending. Real disposable income growth in May was 4%, and June retail sales control ...	[PASS] Goldman Sachs Chief E conomist Jan Hatzius sees growth in the US economy. HatZius sees real personal disposable income growth, a metric tracked by the Bureau of Economic Analysis, increasing by nearly 4%. Morgan Stanley boosted its fourth-quarter GDP ...

Figure 6.11: Evaluation results of interface 1

Through a similar process, we continue to test interface 2 to evaluate the quality of summaries obtained when the system summarizes the annual report. As shown in

Figure 6.12, we start as before by completing some setings.

Figure 6.12

After that, we extract the content of certain pages of the annual report of Tesla Inc. for the year 2022 through the load_pdf.py script, here we extract the content of page 5. The link to this annual report is given below. https://ir.tesla.com/_flysystem/s3/sec/000095017023001409/tsla-20221231-gen.pdf

```
%%writefile load_pdf.py
import re
from langchain.document_loaders import PyPDFLoader
def extract_text_from_pdf(file,startpage, endpage):
    loader = PyPDFLoader(file)
    pages = loader.load_and_split()
    s = startpage - 1
    e = endpage
    x=pages[s:e]
    return x
y=extract_text_from_pdf('tsla-20221231-gen.pdf',5,5)
z=y[0].page_content
s = re.sub(r'\t|\n', ' ', z)
t = re.sub(r':', '', s)
print(t)
```

```
Writing load_pdf.py
```

Figure 6.13

We then create two scripts that accept prompts and generate output to test and compare the results of the system summarizing the annual report when using two different LLMs. The script in Figure 6.14 uses the GPT-3.5 model , and the script in Figure 6.15 uses the fine-tuned BART model.

```
%%writefile langchain_example.py
import os
import sys
from langchain.llms import OpenAI
from langchain.docstore.document import Document
from langchain.chains.summarize import load_summarize_chain
from langchain.text_splitter import RecursiveCharacterTextSplitter
def summarize_docs(text):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    texts = text_splitter.split_text(text)
    docs = [Document(page_content=t) for t in texts]
    llm = OpenAI(temperature=0, openai_api_key=os.getenv('OPENAI_API_KEY'), model_name="text-davinci-003")
    chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=False)
    response = chain.run(docs)
    return response
prompt = sys.argv[1]
o=summarize_docs(prompt)
print(o)
```

```
Writing langchain_example.py
```

Figure 6.14

```
[ ] %%writefile langchain_example2.py
import sys
import os
from langchain.llms import HuggingFaceHub
from langchain.text_splitter import RecursiveCharacterTextSplitter
def summarize_docs2(text):
    os.environ['HUGGINGFACEHUB_API_TOKEN'] = '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'
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    texts = text_splitter.split_text(text)
    llm2 = HuggingFaceHub(repo_id="wyx-ucl/bart-EDGAR-CORPUS", model_kwargs={"max_length": 300})
    w = []
    for t in texts:
        v = llm2.predict(t)
        w.append(v)
    return " ".join(w)
prompt = sys.argv[1]
o=summarize_docs2(prompt)
print(o)
```

```
Overwriting langchain_example2.py
```

Figure 6.15

Finally, we set up the test case and run eval for comparison, as shown in Figure 6.16. The “text” and the “value” of the test case are written to the content of page 5 of the annual report of Tesla Inc. for the year 2022, and “text” is used as input to get the summary via python script. The assertion checks whether the LLM output matches the given scoring criteria, i.e., the summary result is compared with the text data in “value” for cosine similarity. If the comparison result is higher than the set threshold, i.e., 0.8, a “PASS” is displayed. The results show that there is a difference between the two summaries obtained using the two models, the GPT-3.5 model and the fine-tuned BART model. Both generated summaries show “PASS”. This indicates that the values of cosine similarity between 2 generated summaries and the original text are higher than the set threshold, 0.8. It means that the two summaries have high similarity to the original text in terms of content or context. There is reason to believe that summarising annual reports using interface 2 can give reliable results.

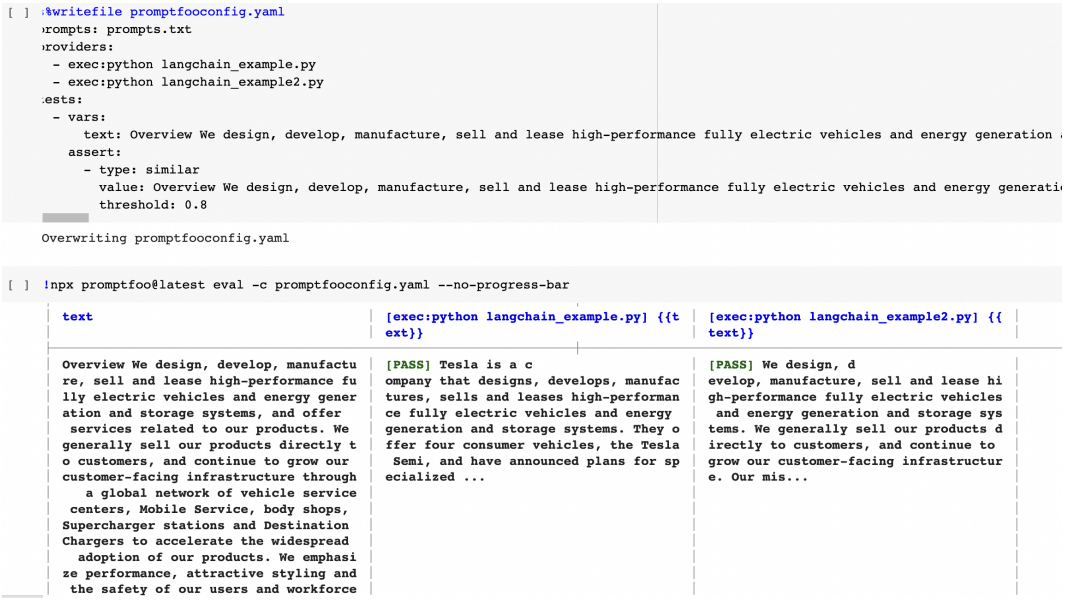

Figure 6.16: Evaluation results of interface 2

We additionally tested using several financial news and annual report texts, all of which yielded PASS results. Below are links to two Google Colab notebooks containing the relevant code for the testing process described above. The "text" is too long to be shown in a figure, and the full test cases and matrix views can be seen in the notebooks.

link 1:

`https://colab.research.google.com/drive/1UEC7HTnCphLa2Smb80oQunXAXV-usp=sharing`

link 2:

`https://colab.research.google.com/drive/1_ocEWnISY4C3AzCJysE18e41mq-usp=sharing`

6.3 Acceptance testing

In the final stage of testing, the client was given a video demonstration of the application's use process and an explanation of how the application worked. After watching the demonstration, the client actually used the application and pointed out that the application did not alert the user to incorrect operations. After making final adjustments to the code, the client expressed acceptance of the application.

Chapter 7

Conclusion, Evaluation and Future Work

7.1 Summary of Achievements

7.1.1 Project goals achieved

Based on the cases in Appendix D, it can be said that this project has achieved all the goals outlined in the introduction.

- Requirements gathering: The system successfully implements Gradio UI to gather users' requirements for the final summary report file (TXT file or PDF file) and to obtain users' preferences for large language models (BART model or GPT-3.5 model). The system successfully implements the Gradio UI to allow users to input the URL of the financial news to be processed and upload the PDF file of the annual report.

- Data collection and preprocessing: The system successfully scrapes data from the URL of financial news entered by users and loads the PDF file of annual report uploaded by users. The system successfully realizes data cleaning for scraped text data. The system successfully realizes word tokenization and chunking processing for financial news and annual reports.

- Summarization: The system successfully implements text summarization based on a large language model selected by the user.

- Question answering: The system successfully uses the LangChain framework to establish a knowledge base based on the PDF file of the annual report uploaded by the user, and realizes the question answering function based on the knowledge base.
- Output presentation and storage: The system successfully implements to display generated results (summary text, answers to user questions), document content related to the question, and links to PDF or TXT files of the summary in the Gradio front-end. The system successfully implements to store the PDF file or TXT file of the generated summary in Google Cloud storage.

7.1.2 Personal aims achieved

- Gained an in-depth understanding of two natural language processing tasks, summarization and question answering.
- Learned to build an application using the open source large Language Model.
- Became familiar with tools or softwares such as Replit, LangChain, Gradio, Hugging face, and Google Colab.
- Learned to fine-tune the large language model.
- Improved Python programming skills as well as gained project experience.

7.2 Critical Evaluation

The final delivery is an application for text summarization and question answering in the financial sector, which helps users to process annual reports and financial news more efficiently. All the requirements of Moscow are well fulfilled. And because unit testing and integration testing are carried out during development, potential logic errors or bugs for each feature are avoided. After the application initialization is complete, the result is presented to the client (external supervisor). The client is satisfied with the application, and an error notification feature has been added based on feedback. This project uses the mature and widely used Python language to build the application and is developed on Replit at the client's request,

which will help the client to carry out subsequent maintenance work. Regarding the user interface design of the application, this project follows the simple design principle to provide a simple interface to the user. This project focuses more on the functional implementation of the application, so in terms of aesthetics, the application may be flawed. In the phase of fine-tuning the large language model introduced in Chapter 4, this project only fine-tuned the facebook/bart-large-cnn model, and if possible, it could try to fine-tune the GPT-3.5 model. In addition, when creating the text summary pair dataset of the annual report text, only a dataset containing 1000 text summary pairs was created considering the expense. When fine-tuning in Google Colab, only a part of the dataset was used to fine-tune the facebook/bart-large-cnn model, considering the limitation of computational resources as well as the time cost.

7.3 Future Work

Although the application has met the requirements of the client, there are still some shortcomings. Therefore, given more time and computational resources, this project will fine-tune the facebook/bart-large-cnn model using the 1000 text summary pairs in the created dataset. In addition, this project will attempt to fine-tune the GPT-3.5 model using the created dataset. In terms of functionality, this project will add an output box in interface 3 to directly display the most relevant parts of the PDF file uploaded to the question entered in the form of images. Since the client of this application is external supervisor, Prof. Joseph Connor, and multiple users are not considered in this project. This project will enable Gradio's queue function to control the rate at which requests are processed, letting users know where they are in the queue. In terms of the aesthetics of the interface design, this project will utilize Gradio's theme engine to customize the styles of components, the background colors of the application, etc. Finally, this project will add some descriptive content or examples to the user interface to help the user understand the application.

Bibliography

[1] Mahmoud El Haj, Paul Edward Rayson, Paulo Alves, and Steven Eric Young. Towards a multilingual financial narrative processing system. 2018.

[2] Huan Yee Koh, Jiaxin Ju, Ming Liu, and Shirui Pan. An empirical survey on long document summarization: Datasets, models, and metrics. *ACM computing surveys*, 55(8):1–35, 2022.

[3] Chanin Nantasenamat. *LangChain tutorial 3: Build a Text Summarization app*. Available at: `https://blog.streamlit.io/langchain-tutorial-3-build-a-text-summarization-app/`, 2023. (Accessed: 1 September 2023).

[4] Hsin-Ting Hsieh and Diana Hristova. Transformer-based summarization and sentiment analysis of sec 10-k annual reports for company performance prediction. 2022.

[5] Samir Abdaljalil and Houda Bouamor. An exploration of automatic text summarization of financial reports. In *Proceedings of the Third Workshop on Financial Technology and Natural Language Processing*, pages 1–7, 2021.

[6] Fabio Chiusano. *Two minutes NLP — Quick intro to Question Answering*. Available at: `https://medium.com/nlplanet/two-minutes-nlp-quick-intro-to-question-answering-124a0930577c`, 2022. (Accessed: 1 September 2023).

[7] 'Natural language processing' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Natural_language_processing. (Accessed: 1 September 2023).

[8] Charles Stangor and Jennifer Walinga. *Introduction to Psychology – 1st Canadian Edition*, chapter 4. BCcampus, Victoria, B.C., 1 edition, 2014. [Online]. Available from: https://opentextbc.ca/introductiontopsychology/chapter/3-1-the-neuron-is-the-building-block-of-the-nervous-system/.

[9] Larry Hardesty. *Explained: Neural networks*. Available at: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414, 2017. (Accessed: 1 September 2023).

[10] Carlo Ciliberto. (2023). 'comp0024: Neural computing'. *COMP0024: Artificial Intelligence and Neural Computing*. University College London. Unpublished.

[11] 'Transformer (machine learning model)' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Transformer_(machine_learning_model). (Accessed: 1 September 2023).

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[13] 'Fine tuning (deep learning)' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Fine-tuning_(deep_learning). (Accessed: 1 September 2023).

[14] 'Word embedding' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Word_embedding. (Accessed: 1 September 2023).

[15] 'Cosine similarity' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Cosine_similarity. (Accessed: 1 September 2023).

[16] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. Gradio: Hassle-free sharing and testing of ml models in the wild. *arXiv preprint arXiv:1906.02569*, 2019.

[17] 'Replit' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Replit. (Accessed: 1 September 2023).

[18] 'MoSCoW method' (2023). *Wikipedia*. Available at: https://en.wikipedia.org/wiki/MoSCoW_method. (Accessed: 1 September 2023).

[19] Lefteris Loukas, Manos Fergadiotis, Ion Androutsopoulos, and Prodromos' Malakasiotis. Edgar-corpus: Billions of tokens make the world go round. *arXiv preprint arXiv:2109.14394*, 2021.

Appendix A

System Manual

Requirements: Python 3.8+

Install the following packages.

contractions 0.1.73

pdfdocument 4.0.0

unstructured 0.8.1

transformers 4.30.2

sentence-transformers 2.2.2

gradio 3.35.2

requests 2.31.0

huggingface-hub 0.15.1

torch 2.0.1

pypdf 3.12.1

openai 0.9.81

tokenizers 0.13.3

chromadb 0.3.29

langchain 0.0.229

tiktoken 0.4.0

google-cloud-storage 2.10.0

nltk 3.8.1

beautifulsoup4 4.12.2

Appendix B

User Manual

The first way to use the application:

- Use the following link to access the project saved in replit `https://replit.com/join/bdqwfawpuq-yaoxuanwang`
- Click the Run button to run the main.py file to generate a public URL of the application.
- Run the application on public URL.

The second way to use the application:

- Access the github repository via the link `https://github.com/23yxw/summary-and-qa-System`
- Download all the files in the repository and open the downloaded folder through Pycharm or another code editor.
- Manually download python packages individually or install all requirements via a terminal command:
  pip install -r requirements.txt
- Replace the OPENAI API KEY in the sym.py file with your own.
- Run the sym.py file to generate a public URL of the application.
- Run the application on public URL.

Appendix C

Use Cases

ID	Description
UC1	The user chooses a interface.
UC2	The user inputs a financial news link.
UC3	The user selects a large language model (BART or GPT-3.5).
UC4	The user selects a file type (pdf or txt).
UC5	The user submits to generate a summary and the corresponding summary report file link.
UC6	The user uploads the PDF file of the annual report.
UC7	The user sets the range of page numbers.
UC8	The user enters a question.
UC9	The user submits to generate an answer.
UC10	The user clears the previous operations.

Table C.1: Use cases list

Use case	The user chooses a interface.
ID	UC1
Description	The user switches the interface with 3 buttons.
Primary Actor	User
Preconditions	None
Main Flow	1. The user clicks ”Summarise financial news article from a news website” button to go to the first interface. 2. The user clicks ”Summarise the PDF file” button to go to the second interface. 3. The user clicks ”Chat with the PDF file” button to go to the third interface.
Postconditions	None
Notes	None

Use case	The user inputs a financial news link.
ID	UC2
Description	The user inputs a news link through a text box.
Primary Actor	User
Preconditions	The user has entered the first interface.
Main Flow	1. The user only needs to copy the URL of the financial news into the text box.
Postconditions	None
Notes	None

Use case	The user selects a large language model.
ID	UC3
Description	The user selects a large language model via 2 check boxes.
Primary Actor	User
Preconditions	The user has entered the first interface or the second interface.
Main Flow	1. The user ticks the checkbox named BART or GPT-3.5.
Postconditions	None
Notes	None

Use case	The user selects a file type (pdf or txt).
ID	UC4
Description	The user selects a file type (pdf or txt) via 2 check boxes.
Primary Actor	User
Preconditions	The user has entered the first interface or the second interface.
Main Flow	1. The user ticks the checkbox named txt or pdf.
Postconditions	None
Notes	None

Use case	The user submits to generate a summary and the corresponding summary report file link.
ID	UC5
Description	The user clicks the submit button to generate a summary and the corresponding summary report file link.
Primary Actor	User
Preconditions	1. The user has entered the first interface or the second interface. 2. The user has entered a valid URL or uploaded a PDF file of the annual report. 3. The user has made a model selection and file type selection. 4. The user has set the page number range (when summarising the annual report in the second interface).
Main Flow	1. The user clicks the submit button.
Postconditions	None
Notes	None

Use case	The user uploads the PDF file of the annual report.
ID	UC6
Description	The user clicks the file upload box to start uploading the PDF file of the annual report.
Primary Actor	User
Preconditions	1. The user has entered the second interface or the third interface.
Main Flow	1. The user clicks the file upload box.
Postconditions	None
Notes	None

Use case	The user sets the range of page numbers.
ID	UC7
Description	The user sets the range of page numbers to be processed through two sliders.
Primary Actor	User
Preconditions	1. The user has entered the second interface. 2. The user has uploaded a PDF file of the annual report.
Main Flow	1. The user drags two sliders named ”s” and ”e” to set the start page and end page respectively.
Postconditions	None
Notes	None

Use case	The user enters a question.
ID	UC8
Description	The user enters a question via a text box.
Primary Actor	User
Preconditions	1. The user has entered the third interface. 2. The user has uploaded a PDF file of the annual report.
Main Flow	1. The user only needs to input a question related to the uploaded PDF file into the text box.
Postconditions	None
Notes	None

Use case	The user submits to generate an answer.
ID	UC9
Description	The user clicks the submit button to generate an answer.
Primary Actor	User
Preconditions	1. The user has entered the third interface. 2. The user has uploaded a PDF file of the annual report. 3. The user has input a question.
Main Flow	1. The user clicks the submit button.
Postconditions	None
Notes	None

Use case	The user clears the previous operations.
ID	UC10
Description	The user clicks the clear button to clear the previous operations on the current interface.
Primary Actor	User
Preconditions	1. The user has entered the first, second or third interface. 2. The user has performed at least one operation.
Main Flow	1. The user clicks the clear button.
Postconditions	None
Notes	None

Appendix D

Case study

Figure D.1 shows an example of financial news summarization using the GPT-3.5 model in interface 1, where the system returns the summary text as well as a link to the PDF file of the summary.

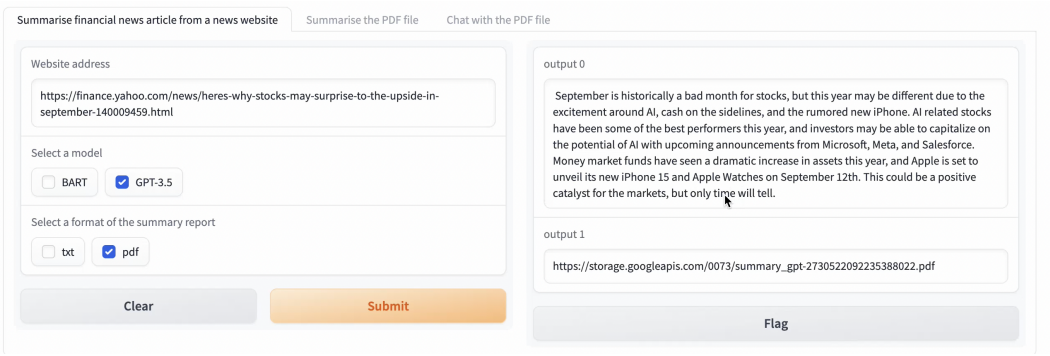

Figure D.1

Figure D.2 shows an example of the use of the BART model for financial news summarisation in interface 1, where the system returns the text of the summary and a link to the TXT file of the summary.

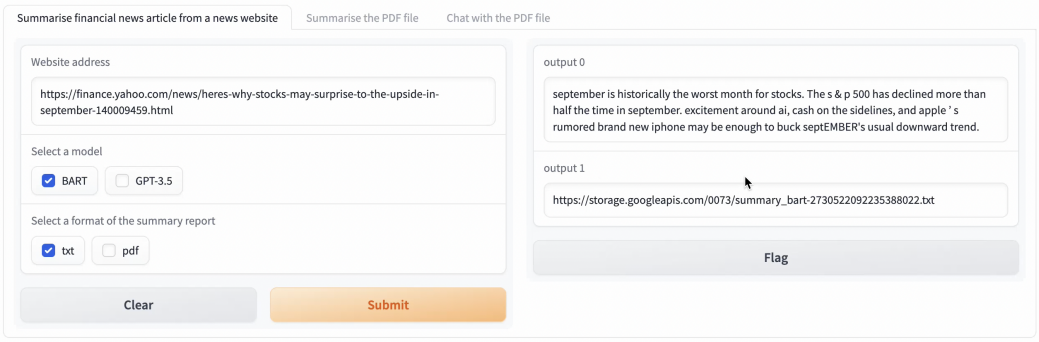

Figure D.2

Figure D.3 shows an example of using the GPT-3.5 model to summarise certain pages of an annual report in interface 2. The system returns the summary text and a link to the PDF file of the summary.

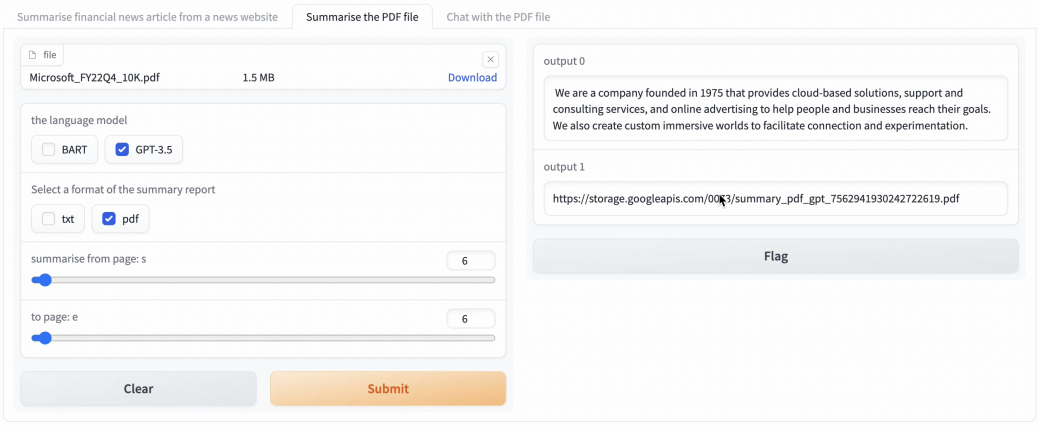

Figure D.3

Figure D.4 shows an example of the use of the fine-tuned BART model to summarise some pages of an annual report in interface 2. The system returns the summary text and a link to the TXT file of the summary.

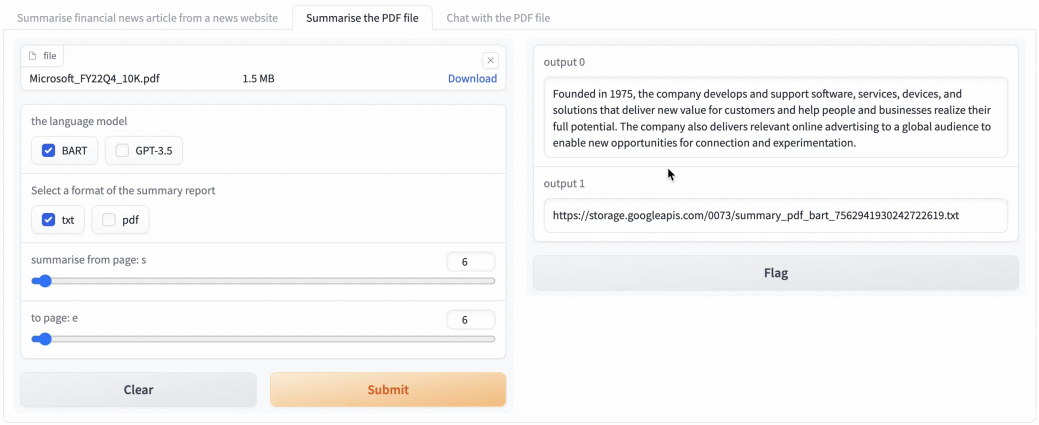

Figure D.4

Figure D.5 shows an example of a Q&A based on an uploaded annual report using the GPT-3.5 model in interface 3, where the system returns an answer and the content of the annual report that is most similar to the question.

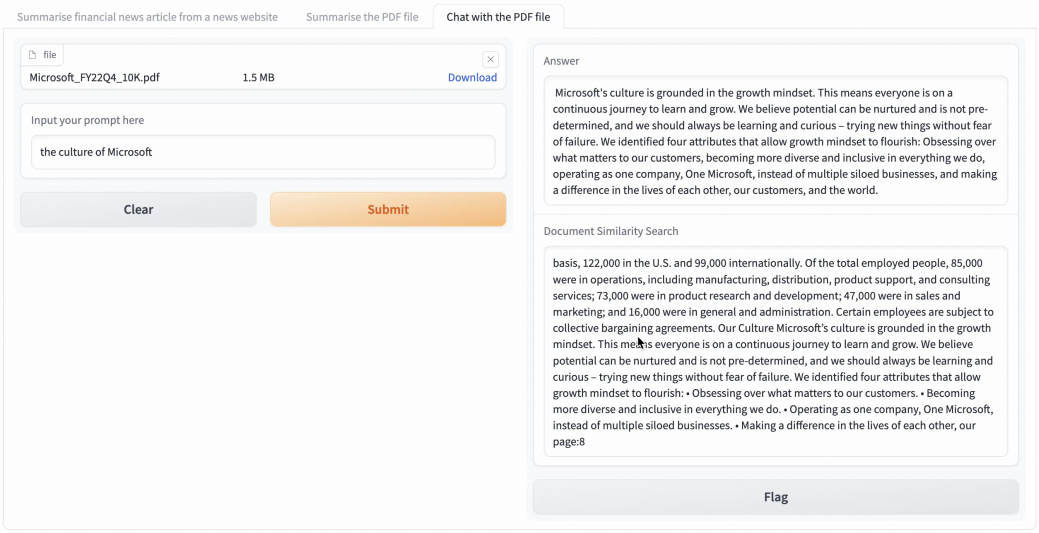

Figure D.5