TEXT SUMMARZATION APPLICATION

AMIN AHMED MOHAMMEDELHASSAN
ALAWAD
1191302190

BACHELOR OF COMPUTER SCIENCE
B.CS (HONS) DATA SCIENCE

FACULTY OF COMPUTING AND
INFORMATICS MULTIMEDIA UNIVERSITY

JULY 2024

TEXT SUMMARZATION APPLICATION

BY

AMIN AHMED MOHAMMEDELHASSAN ALAWAD

PROJECT REPORT SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR OF COMPUTER SCIENCE
B.CS (HONS) DATA SCIENCE

FACULTY OF COMPUTING AND INFORMATICS

MULTIMEDIA UNIVERSITY
MALAYSIA

JULY 2024

DECLARATION

I hereby declare that the work have been done by myself and no portion of the work contained in this thesis has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

[illegible]

Name: Amin Ahmed Alawad
Student Id: 1191302190
Faculty of Computing and Informatics
Multimedia University
Date: 19 February 2024

ACKNOWLEDGEMENT

First and foremost, I would like to extend my deepest gratitude to my supervisor, Dr. Goh Chien Le, for his invaluable guidance, support, and encouragement throughout this Final Year Project. His expertise and insights have been instrumental in shaping the direction and success of this project. I am also thankful to my faculty members at MMU for their continuous support and for providing a strong educational foundation that has been critical in completing this project.

Additionally, I would like to express my appreciation to my friends, colleagues, and family for their unwavering support and understanding throughout my academic journey. Their constructive feedback, moral support, and belief in my abilities have been a constant source of motivation.

ABSTRACT

This project presents the development of a web-based application designed to address the growing need for efficient processing of scientific texts through automated summarization. Leveraging advanced natural language processing (NLP) techniques and machine learning algorithms, the application aims to transform lengthy scientific documents into concise, informative summaries. The core of this endeavor lies in the implementation of an AI-driven framework that not only captures the essence of complex scientific information but also retains the accuracy and coherence essential for academic and research purposes. By integrating user-friendly interfaces with robust backend algorithms, this application seeks to provide a valuable tool for researchers, educators, and students, enabling them to quickly assimilate and navigate through the vast expanse of scientific literature. The findings highlight the application's efficacy in streamlining research workflows, enhancing accessibility to scientific knowledge, and fostering a more efficient dissemination of information within the scientific community.

Contents

List of Figures

List of Tables

1. Introduction

In today's era of information overload, text summarization has emerged as a crucial tool to streamline the arduous task of sifting through vast amounts of data. By distilling essential content from expansive texts, it preserves key information and overall meaning, becoming indispensable across various domains for effective decision-making and knowledge dissemination. The evolution of summarization from ancient scholarly practices to today's AI-driven techniques marks a significant leap, leveraging advanced algorithms to transform extensive documents into concise, informative summaries. This journey from manual efforts to computational methods underscores a shift towards efficiency and accessibility in information management.

The challenge of summarization extends beyond extracting key information; it involves maintaining the original context and meaning, balancing extractive and abstractive approaches. Extractive summarization focuses on selecting significant sentences directly from the source, while abstractive summarization generates new sentences, requiring advanced understanding and paraphrasing capabilities. This balance is critical in navigating the subtleties of language and ensuring accuracy and coherence.

Despite advancements, the practical adoption of text summarization faces hurdles, including technology limitations and a reluctance to rely on automated summaries for critical decisions. This highlights a gap between potential and practical application, underscoring the need for more reliable, user-friendly solutions.

This project introduces a web-based application tailored for the automated summarization of scientific texts, leveraging the precision of Large Language Models (LLMs) and the innovative Low-Rank Adaptation (LoRA) technique. Focused specifically on the domain of Networking and Internet Architecture in computer science, this approach enhances our model's ability to process complex scientific information, ensuring summaries are both concise and accurate. Through this application, we aim to bridge the gap in text summarization technology, offering a tool that not only advances the field but also meets the practical needs of researchers, educators, and students in navigating the dense landscape of scientific literature.

1.1. Problem Statement:

The exponential growth in scientific publications presents a formidable challenge for researchers across disciplines. Keeping abreast of the latest developments, theories, and experimental outcomes is increasingly daunting due to the sheer volume of information. This deluge of data necessitates efficient mechanisms for distilling complex studies into digestible summaries without sacrificing depth or accuracy. Traditional summarization tools often fall short in addressing the nuanced demands of scientific literature, leaving a significant gap in resources for effectively managing and assimilating scholarly information.

1.2. Objective

The primary objective of this project is the development of an advanced web-based application, leveraging Large Language Models (LLMs) and incorporating Low-Rank Adaptation (LoRA), designed specifically for the summarization of scientific texts in the domain of Networking and Internet Architecture in computer science. This application aims to revolutionize the literature review process by significantly enhancing the efficiency and breadth of knowledge accessibility. By providing precise, coherent summaries of complex research papers, this tool endeavors to democratize access to scientific knowledge, catering not only to researchers but also to a wider audience without extensive expertise in the field. The ultimate goal is to facilitate a more informed and rapid assimilation of scientific advancements, thereby accelerating research and educational endeavors.

1.3. Target Audience

The target audience for this web application encompasses a broad spectrum of users within the scientific community and beyond. Primarily, it serves researchers and academics seeking to streamline their literature review processes and stay updated with the latest findings in their respective fields. Educators and students will also benefit significantly, as the tool provides an efficient means to access and understand complex scientific concepts and research outcomes. Moreover, industry professionals, policymakers, and the general public with interests in scientific advancements stand to gain from enhanced access to summarized scientific knowledge, making the application a versatile resource for anyone looking to engage with scientific literature more.

2. Literature review

In the ever-expanding universe of scientific knowledge, the ability to efficiently sift through, summarize, and comprehend voluminous research findings is indispensable. The advent and subsequent evolution of text summarization technologies have emerged as a cornerstone in the realm of information management, particularly within scientific research where the density and complexity of literature can be overwhelming. This literature review embarks on a journey through the development of text summarization techniques, with a spotlight on abstractive summarization, underscoring its significance and potential in transforming the accessibility of scientific literature.

The landscape of text summarization has been markedly reshaped by advances in artificial intelligence (AI) and natural language processing (NLP), transitioning from basic extractive methods to sophisticated abstractive approaches that promise a deeper comprehension and more nuanced condensation of texts. This evolution is crucial for the scientific domain, where the precise encapsulation of complex ideas and findings is paramount. Through the lens of seminal works such as " Vaswani et al. (2017) " and comprehensive surveys including "A comprehensive review of automatic text summarization methods, data, evaluation and" alongside "Automatic text summarization: A comprehensive survey," this review explores the trajectory of summarization technologies from their nascent stages to their current state-of-the-art incarnations.

By integrating insights from "Exploring the Landscape of Automatic Text Summarization: A Comprehensive Survey" and evaluating the impact of large language models as detailed in "Large Language Models for Scientific synthesis" and "Naveed et al. (2023)," we delve into the capabilities and challenges of applying these technologies to the summarization of scientific papers. The discourse extends to the examination of novel methodologies and evaluation frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and "Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing efforts to refine and enhance the effectiveness of summarization tools in meeting the specific demands of the scientific community.

This literature review aims to provide a structured overview of the field of text summarization, articulating the pivotal role of abstractive summarization in advancing the accessibility and comprehension of scientific research. Through this exploration, we underscore the necessity for continuous innovation and adaptation in summarization technologies to cater to the dynamic and diverse landscape of scientific literature.

2.1. Historical Evolution of Text Summarization:

The development of text summarization technologies has traversed a remarkable path, mirroring the broader trajectory of progress in computational linguistics and artificial intelligence. Initially grounded in manual efforts by scholars to distill lengthy texts into digestible summaries, the advent of digital computing ushered the field into an era of automation and innovation.

2.1.1. Early Beginnings and Extractive Summarization:

The genesis of automated text summarization can be traced back to the mid-20th century, with initial approaches focusing on extractive summarization. These methods relied on algorithmic extraction of key sentences or phrases based on statistical cues such as frequency and positional importance, as outlined in foundational surveys like "Cajueiro et al. (2023)". While effective in reducing text volume, extractive methods often struggled with coherence and the integration of text from disparate sections.

2.1.2. The Advent of NLP and Abstractive Summarization:

The field experienced a paradigm shift with the introduction of natural language processing (NLP) techniques, enabling more sophisticated analysis and processing of text. The publication of "Vaswani et al. (2017)" marked a significant milestone, introducing transformer models that became the backbone of modern NLP tasks, including abstractive summarization. Unlike extractive methods, abstractive summarization aims to generate new text that captures the essence of the source material, facilitating more nuanced and coherent summaries.

2.1.3. Influence of Large Language Models:

Recent years have witnessed the rise of large language models (LLMs), as detailed in "(Zheng et al., 2023)". These models, trained on extensive corpora, excel in generating text that mimics human writing, pushing the boundaries of abstractive summarization. Their ability to understand context and produce relevant, coherent summaries has opened new avenues for summarizing scientific literature, where accuracy and the preservation of nuanced information are paramount.

2.2. Overview of Text Summarization Techniques:

The landscape of text summarization is dominated by two principal approaches: extractive and abstractive summarization. Each method employs distinct strategies for distilling texts into concise summaries, leveraging advancements in computational linguistics and artificial intelligence.

2.2.1. Extractive Summarization:

Extractive summarization methods focus on identifying and extracting key sentences or fragments from the source text to assemble a summary. This approach relies on algorithms to determine the significance of text segments based on various features, such as term frequency, sentence positioning, and semantic relatedness. Extractive techniques are grounded in the principle that the most informative parts of a text can be directly compiled to form a coherent summary. Despite their relative simplicity and high fidelity to the original text, extractive methods may fall short in creating summaries that seamlessly integrate diverse pieces of information or adequately represent the source text's overarching themes. The comprehensive surveys "El-Kassas et al. (2021)"and "Abualigah et al. (2020)" detail the evolution and methodologies of extractive summarization, highlighting its applicability and limitations.

2.2.2. Abstractive Summarization:

In contrast, abstractive summarization aims to generate new text that captures the essence of the original content, often employing advanced natural language generation (NLG) techniques. This method requires a deeper understanding of the source material, enabling the summarization system to paraphrase, rephrase, and condense the original text into a novel summary that maintains its informational value and coherence. The breakthrough of transformer models, as introduced in “Vaswani et al. (2017)”, has significantly propelled the capabilities of abstractive summarization by facilitating better context understanding and text generation. Moreover, the development and application of large language models (LLMs), discussed in (Zheng et al., 2023) and Naveed et al. (2023) .have further advanced the field, enabling more accurate and contextually relevant summaries.

2.2.3. Justification for Choosing Abstractive Summarization:

The decision to employ abstractive summarization for scientific papers is driven by its potential to create coherent, integrated summaries that capture the complexity and nuance of scientific discourse. Scientific literature, characterized by its dense information, complex data, and specialized terminology, necessitates a summarization approach that can effectively synthesize and convey key findings and concepts in a comprehensible manner.

2.2.3.1. Complexity and Nuance:

Abstractive summarization's ability to generate new text makes it particularly suited for handling the intricate details and sophisticated language of scientific papers, providing summaries that are not only concise but also rich in context and meaning.

2.2.3.2. Technological Advancements:

The advent of transformer models and LLMs has dramatically enhanced the feasibility of producing high-quality abstractive summaries. These models' capacity for deep semantic understanding and language generation aligns well with the requirements for summarizing scientific content, as they can accurately interpret and articulate complex ideas.

2.2.3.3. Enhancing Accessibility:

By distilling essential information from scientific papers into accessible summaries, abstractive summarization serves as a bridge connecting researchers, practitioners, and the broader public to the latest scientific discoveries, facilitating greater understanding and engagement with research findings.

The overview of text summarization techniques and the rationale for selecting abstractive summarization underscore the dynamic evolution of the field and its critical role in advancing scientific communication. Abstractive methods, supported by groundbreaking AI technologies, offer promising solutions to the challenges of summarizing scientific literature, paving the way for broader access to and engagement with scientific knowledge.

2.3. Challenges in Summarizing Scientific Papers

Summarizing scientific literature poses distinct challenges that test the limits of current text summarization technologies. The inherent complexity of scientific texts, combined with the need for precision and comprehensiveness, requires advanced summarization approaches, particularly when employing abstractive methods.

- Complex Terminology and Concepts: Scientific papers are laden with specialized vocabulary and complex concepts unique to each field of study. Successfully summarizing such texts demands not only an understanding of the subject matter but also the ability to accurately convey specialized terms and ideas in the summary. This challenge is compounded by the diverse audience of scientific summaries, ranging from experts in the field to interdisciplinary researchers and the general public, each with different levels of familiarity with the topic.

- Data and Methodological Detail: Scientific studies often include detailed descriptions of methodologies, experimental designs, and data analyses. Summarizing this information without oversimplifying or omitting critical details is crucial for maintaining the integrity and utility of the summary. The challenge lies in distilling these details into a format that is both accessible and informative.

- Integrating Results and Discussions: A key aspect of scientific papers is the interpretation of results and their implications for the field. Abstractive summarization must effectively integrate these sections to provide a coherent narrative that captures the study's conclusions and its contribution to the broader scientific dialogue. This requires a nuanced understanding of the research and the ability to synthesize key points from various sections of the paper.

- Maintaining Factual Accuracy: Given the importance of precision in scientific communication, summaries must maintain the factual accuracy of the original text. This is particularly challenging for abstractive summarization, which generates new text based on the model's understanding of the source material. Ensuring that summaries accurately reflect the findings and claims of the original studies is paramount.

The documents “Cajueiro et al. (2023)” and “Cajueiro et al. (2023)” highlight the need for summarization approaches that address these challenges. Furthermore, “Adams, Suri, & Chali (2022)” illustrates the potential of leveraging cutting-edge AI models to overcome the difficulties inherent in summarizing scientific literature.

Addressing the challenges of summarizing scientific papers requires a careful balance between technological innovation and an in-depth understanding of the nuances of scientific discourse. The development of abstractive summarization tools that can navigate these complexities is essential for enhancing the accessibility and dissemination of scientific knowledge.

2.4. LLM explanation:

The landscape of Natural Language Processing (NLP) has undergone remarkable transformations, with each new model architecture pushing the boundaries of what machines can understand and generate in human language. The advent of the Transformer model, introduced in the groundbreaking paper "Vaswani et al. (2017)" by Vaswani et al. in 2017, marked a pivotal moment in this evolutionary journey. This model not only challenged the prevailing norms of sequence-to-sequence processing but also set a new standard for efficiency and effectiveness in a wide array of NLP tasks.

Prior to the Transformer, the field of NLP was predominantly navigated through the use of Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). These architectures, despite their successes, faced significant challenges, especially when dealing with long sequences of data. RNNs, for example, were inherently sequential, which limited the ability to parallelize operations—a critical bottleneck for training speed and efficiency. Moreover, they struggled with long-term dependencies due to issues like gradient vanishing or explosion. CNNs, while offering more parallelism, still could not fully capture the complexities of sequential data and long-range dependencies as effectively as needed for advanced NLP tasks.

The introduction of the Transformer model was a response to these challenges. By eschewing recurrence and convolution in favor of attention mechanisms, the Transformer was able to model dependencies regardless of their distance in the sequence. This was a significant shift, enabling unprecedented parallelization of training processes and dramatically improving the efficiency and scalability of model training for complex NLP tasks.

The objectives of this report are multifaceted. Firstly, it aims to elucidate the theoretical underpinnings and architectural nuances of the Transformer model. Through a detailed examination of its components—ranging from self-attention and multi-head attention to positional encoding and the encoder-decoder structure—this report seeks to provide a comprehensive understanding of why and how the Transformer model represents a quantum leap in NLP capabilities. Secondly, the report will explore the impact of this model on the field of NLP, highlighting its role in the development of subsequent models like BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer) series, which have further pushed the boundaries of what's possible in language understanding and generation. Lastly, by offering insights into the model's training methodologies and applications, this report aims to shed light on the practical implications of the Transformer architecture, setting the stage for discussions on its future directions and potential advancements.

2.4.1. Theoretical Foundations

To fully appreciate the innovation behind the Transformer model, it's crucial to understand the theoretical underpinnings that have guided the evolution of NLP models. This section delves into the limitations of traditional neural network architectures in handling language processing tasks and introduces the concept of attention mechanisms, setting the stage for the emergence of the Transformer model.

2.4.1.1. Neural Networks in NLP

Neural networks have been at the core of advancements in NLP, offering powerful tools for modeling complex patterns in language data. Initially, models like Recurrent Neural Networks (RNNs) and their more advanced variants, Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), dominated the scene. These architectures were designed to process sequential data, capturing temporal dependencies by passing hidden states from one time step to the next. However, RNNs and their ilk faced significant challenges, notably difficulty in parallelizing operations, which became a bottleneck in processing long sequences. Furthermore, they were prone to issues such as vanishing or exploding gradients, complicating the learning of long-range dependencies within text.

Convolutional Neural Networks (CNNs), known for their success in computer vision, were also adapted for NLP tasks. By applying convolutions across sequences, CNNs could capture local dependencies and were more amenable to parallelization than RNNs. Despite these advantages, CNNs inherently struggled to model longer-range dependencies within text, a limitation stemming from their fixed receptive field sizes.

2.4.1.2. Limitations of Previous Architectures

The key limitations of RNNs and CNNs in NLP can be summarized as follows:

- Sequential Processing: RNNs inherently process data sequentially, limiting the ability to parallelize training and leading to longer training times, particularly for long sequences.

- Long-Term Dependencies: Both RNNs and CNNs face challenges in capturing long-term dependencies within text. While RNNs theoretically can model such dependencies, in practice, they often fail due to vanishing or exploding gradients. CNNs, on the other hand, are constrained by their local receptive fields.

- Efficiency and Scalability: The sequential nature of RNNs impacts efficiency and scalability, making it difficult to leverage modern parallel computing architectures. CNNs, while more parallelizable, still require extensive computation to model longer sequences effectively.

2.4.1.3. **Introduction to Attention Mechanisms**

The concept of attention mechanisms emerged as a solution to the limitations of RNNs and CNNs, inspired by the human cognitive process of focusing on specific parts of input when processing information. Attention mechanisms allow the model to dynamically weigh the importance of different parts of the input data, focusing on relevant parts to make predictions.

In the context of NLP, attention mechanisms enable models to learn contextual relationships between words in a sentence, regardless of their positional distances. This is achieved by computing attention scores that represent the relevance of each word to the task at hand, allowing the model to focus on pertinent information and ignore the rest.

The introduction of attention mechanisms marked a significant shift towards more flexible and efficient modeling of sequences, paving the way for the development of the

Transformer model, which leverages attention as its core mechanism, eschewing traditional recurrence and convolution altogether.

2.4.2. Transformer Model Architecture

The Transformer model, introduced by Vaswani et al. in their seminal paper “Vaswani et al. (2017)” represents a departure from traditional neural network architectures used in NLP, such as RNNs and CNNs. At the heart of the Transformer's success is its innovative use of attention mechanisms, specifically designed to improve efficiency, scalability, and the ability to capture long-range dependencies in text. This section explores the core components and functionalities of the Transformer model architecture.

2.4.2.1. Overview

The Transformer architecture is based on an encoder-decoder structure, but unlike its predecessors, it relies entirely on self-attention mechanisms to process input data. This design choice allows for significant parallelization during training and inference, leading to substantial improvements in efficiency. The absence of recurrence and convolution in the Transformer's architecture eliminates the constraints that previously hindered the processing of long sequences.

2.4.2.2. Transformer Model Architecture

In the quest to understand the transformative nature of the Transformer model within the domain of Natural Language Processing (NLP), a critical examination of its architecture is indispensable. The accompanying schematic provides a visual representation of the Transformer's intricate structure, delineating the flow from input to output. In this section, we dissect the diagram to elucidate the functionality and synergy of the various components that comprise the Transformer model.

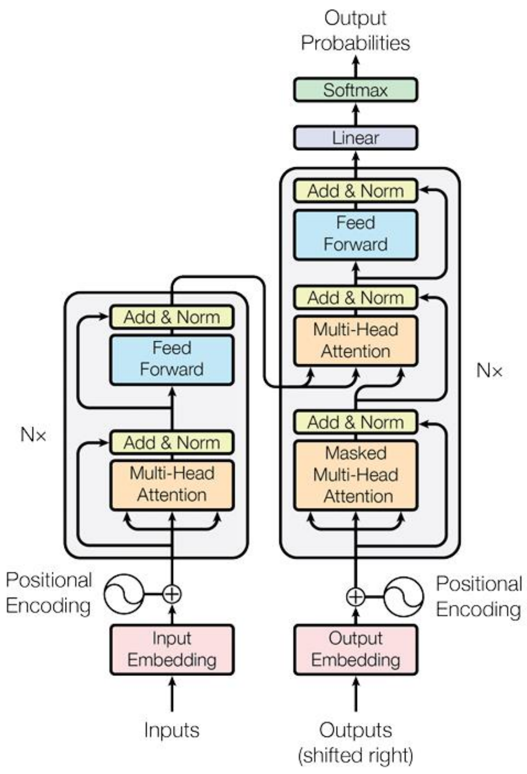


Figure 1: Transformer Model Architecture which has Encoder and Decoder. Vaswani et al. (2017)

2.4.2.3. Encoder:

2.4.2.3.1. Input Processing:

- Input Embedding: The initial step in the Transformer's processing pipeline involves translating discrete input tokens into continuous vectors. This embedding phase is crucial as it provides a dense representation of the input tokens, encapsulating them in a format amenable to the model's subsequent layers.
- Positional Encoding: Following embedding, positional encodings are infused into the input vectors. This step is vital in a model devoid of recurrence, as it imbues the sequence with awareness of the order of tokens—a fundamental aspect of language understanding.

2.4.2.3.2. Layer Architecture (Repeated N Times):

- Multi-Head Self-Attention: A pivotal feature of the Transformer, the multi-head self-attention mechanism, allows the model to process tokens in parallel. This sub-layer facilitates the simultaneous capture of different dependencies and features from various representation subspaces.

- Add & Norm: Post attention, the model employs a residual connection that adds the input of the sub-layer to its output before normalization. This design choice combats the vanishing gradients dilemma and promotes training stability across the model's depth.
- Feed-Forward Network: Subsequent to attention and normalization, a position-wise feed-forward network is applied. Through linear transformations and non-linear activation functions, this network further processes each token's representation independently across the sequence.

2.4.2.4. Decoder:

2.4.2.4.1. Output Processing:

- Output Embedding: Analogous to the encoder, the decoder embeds its output tokens. However, a crucial distinction lies in the rightward shift of these tokens, a measure taken to ensure causality in the model's predictions.
- Positional Encoding: The decoder also integrates positional encodings into its embeddings, maintaining consistency with the encoder and preserving the sequence's positional information.

2.4.2.4.2. Layer Architecture (Repeated N Times):

- Masked Multi-Head Self-Attention: The decoder's initial attention mechanism is 'masked' to preclude future positions from influencing the prediction of the current token. This ensures that the generation of each token can only be contingent upon the known preceding tokens.
- Add & Norm: As in the encoder, a residual connection followed by normalization is utilized. This consistency in architecture facilitates the model's ability to learn and adapt during training.
- Multi-Head Attention (with Encoder Outputs): This sub-layer performs attention over the outputs from the encoder stack, allowing the decoder to focus on relevant segments of the input sequence while generating the translation or response.

- Add & Norm: Another round of residual connections and normalization follows, further refining the information flow.
- Feed-Forward Network: The decoder mirrors the encoder with its inclusion of a position-wise feed-forward network, essential for complex pattern learning in the output sequence.

2.4.2.4.3. Final Output Generation:

- Linear: The decoder's output undergoes a linear transformation, projecting the processed token representations into a higher-dimensional space corresponding to the model's vocabulary.
- Softmax: The linearly transformed outputs are converted into a probability distribution using a softmax layer. This distribution indicates the likelihood of each vocabulary token being the next element in the output sequence.

2.4.2.5. Scaled Dot-Product Attention:

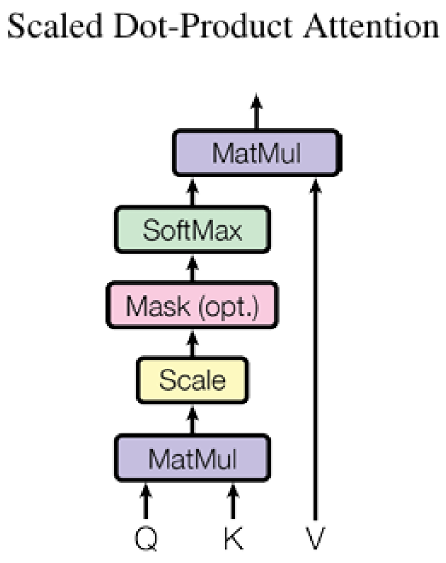


Figure 2: Scaled Dot-Product Attention structure. Vaswani et al. (2017)

At the core of the self-attention mechanism is the scaled dot-product attention, which computes attention scores based on the dot products of queries and keys associated with the input. These scores are then scaled down to prevent the softmax function from entering regions where gradients are too small, improving the stability of the learning process. The attention scores determine how much focus should be given to each part of the input sequence when constructing the output of the attention layer.

2.4.2.6. Multi-Head Attention:

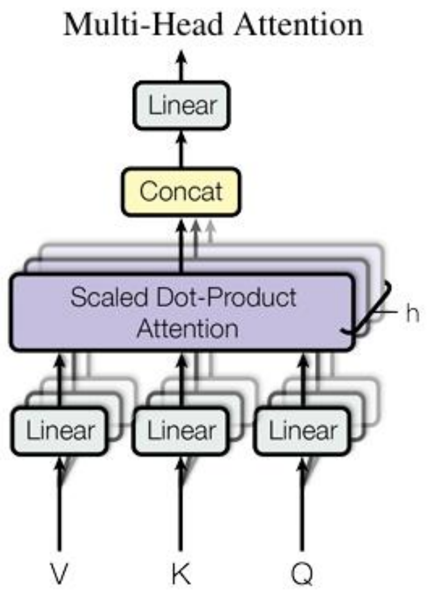


Figure 3: Scaled Dot-Product Attention structure. Vaswani et al. (2017)

To enhance the model's ability to capture various aspects of the data, the Transformer employs a multi-head attention mechanism. This approach splits the attention process into multiple "heads," allowing the model to attend to different parts of the sequence simultaneously. By doing so, the Transformer can capture a diverse range of relationships within the data, such as syntactic and semantic connections, from different representation subspaces.

2.4.2.7. Positional Encoding

Given its lack of recurrence or convolution, the Transformer model requires a method to incorporate the order of the sequence into its processing. Positional encoding serves this purpose by adding information about the position of each word in the sequence to its representation. This enables the model to maintain awareness of the sequence order, a critical aspect of understanding language.

2.4.3. Feed-Forward Networks

In addition to the attention mechanisms that form the core of the Transformer's innovative approach, the model also incorporates position-wise feed-forward networks (FFNs) within each encoder and decoder layer. These FFNs play a crucial role in the model's ability to process and interpret the input data, further contributing to its remarkable performance across various NLP tasks.

2.4.3.1. Structure and Purpose

Each layer in both the encoder and decoder of the Transformer contains a feed-forward network that consists of two linear transformations with a ReLU activation in between. Unlike the attention mechanisms that allow each position in the sequence to interact with one another, the FFNs operate independently on each position. This means that the same feed-forward network is applied to each position separately and identically, ensuring that the model can capture complex data representations while maintaining the sequence's order and structure.

The primary purpose of these FFNs is to introduce non-linearity into the model, enabling it to learn more complex patterns in the data. By processing each position's output from the attention mechanism, the FFNs contribute to refining the representations of each word or token in the sequence, enhancing the model's overall ability to understand and generate language.

2.4.3.2. Contribution to the Transformer's Capabilities

The inclusion of FFNs in the Transformer architecture underscores the model's capacity for deep representation learning. By applying these networks position-wise across the sequence, the Transformer can effectively model intricate relationships within the data, beyond what is captured through attention alone. This layered approach, combining multi-head attention with position-wise feed-forward networks, equips the Transformer with the flexibility and power needed to excel at a wide range of NLP tasks.

2.4.4. Training Methodology

Training the Transformer model effectively requires careful attention to several key aspects, including the choice of optimization algorithms, learning rate scheduling, and the handling of large datasets. The original "Vaswani et al. (2017)" paper introduced the concept of the "warm-up" learning rate scheduling strategy, which gradually increases the learning rate at the start of training, before decreasing it according to a predefined schedule. This approach helps to stabilize the training process, ensuring that the model learns efficiently without diverging.

Key Applications

- The Transformer's proficiency has been proven across various NLP tasks, making it a cornerstone for subsequent advancements in the field:
- Machine Translation: The Transformer has set new standards in machine translation due to its ability to capture the nuances of language and efficiently process long sequences.
- Text Summarization: The model's capacity for understanding context and generating coherent text makes it ideal for summarizing extensive documents succinctly and meaningfully
- Question Answering: The Transformer excels in question-answering tasks by effectively synthesizing and retrieving relevant information from large corpora.
- Text Generation: The architecture's deep contextual understanding has led to breakthroughs in text generation, producing coherent and contextually relevant passages of text.

2.5. Fine tuning: Low-Rank Adaptation (LoRA) of Large Language Models

The surge in the deployment of large pre-trained language models (LLMs) across a spectrum of applications has brought about a need for effective fine-tuning methods that allow for task-specific customization without compromising on model performance or efficiency. In this context, the paper "LoRA: Low-Rank Adaptation of Large Language Models" (https://ar5iv.labs.arxiv.org/html/2106.09685) introduces a novel approach that addresses this need. This literature review will synthesize the key findings of the paper, contextualizing its significance within the broader research landscape.

2.5.1. The Challenge of Fine-Tuning LLMs

As the scale of LLMs has grown, the traditional fine-tuning approach, which involves updating all the parameters of the model, has become increasingly computationally expensive and less feasible. This method demands substantial resources, both in terms of the time required for training and the computational power. Moreover, it may lead to overfitting, especially when adapting large models to small datasets.

2.5.2. LoRA: A Parameter-Efficient Approach

LoRA, which stands for Low-Rank Adaptation, proposes an alternative fine-tuning method that targets a small subset of the model's parameters. Instead of updating the entire set of weights within the model's layers, LoRA focuses on adapting the self-attention and feed-forward layers by introducing low-rank matrices that capture the adaptations necessary for the new tasks. This method significantly reduces the number of trainable parameters, resulting in a more efficient and streamlined adaptation process.

2.5.3. Core Mechanisms of LoRA

The methodology behind LoRA involves inserting trainable low-rank matrices into the weight matrices of a pre-trained model's attention and feed-forward networks. These low-rank matrices, which are much smaller in size compared to the original weight matrices, effectively capture the salient features required for the new task while keeping the original pre-trained parameters frozen. The low-rank structure inherently acts as a regularizer, mitigating overfitting risks and promoting generalization.

2.5.4. Comparative Analysis

In comparison to other parameter-efficient adaptation techniques, such as adapters and prompt tuning, LoRA distinguishes itself by its integration into the model's existing architecture, as opposed to adding external modules or prompts. This seamless integration is posited to allow LoRA to leverage the pre-trained parameters more effectively, tapping into the rich representational power of LLMs without the need for extensive retraining.

2.5.5. Contributions to the Field

LoRA's approach contributes to the field of NLP by offering a balance between adaptation power and parameter efficiency. It enables the fine-tuning of LLMs in a resource-conscious manner, making it particularly valuable for scenarios where computational resources are limited or when rapid deployment is necessary.

2.5.6. Implications and Future Directions

The success of LoRA as a fine-tuning technique opens up new avenues for research, particularly in exploring how low-rank structures can be exploited to adapt LLMs for a wider array of tasks. It also sparks interest in investigating how such techniques can be combined with other methods of transfer learning to further enhance the efficiency and effectiveness of fine-tuning LLMs.

2.6. Applications of Text Summarization in Various Fields

Text summarization technologies, bolstered by the advancements in Large Language Models (LLMs) and natural language processing (NLP), have found applications across a broad spectrum of fields beyond scientific literature. These technologies are transforming how information is accessed, processed, and utilized, offering streamlined workflows and enhanced knowledge dissemination across industries.

- Healthcare and Medical Research: In the healthcare sector, text summarization tools are instrumental in managing the vast amounts of medical literature and patient data. Summarization helps clinicians and researchers quickly grasp the findings of clinical studies, review patient histories, and stay updated with the latest treatment protocols. This application is critical in a field where timely access to information can directly impact patient care and outcomes

- Legal Industry: The legal domain benefits significantly from text summarization technologies, especially in managing case files, legal precedents, and legislation. Summarization tools can distill lengthy legal documents into concise summaries, aiding lawyers and legal researchers in their case preparation and research activities. This not only saves time but also ensures that critical information is highlighted and readily accessible.

- Finance and Business: In finance and business, executive summaries generated by text summarization tools provide quick insights into market trends, financial reports, and business news. This application supports decision-making processes by condensing complex financial data and analyses into digestible formats, enabling executives and analysts to make informed decisions swiftly.

- Academic and Educational Use: Beyond scientific research, text summarization aids in academic learning and teaching by providing summaries of textbooks, lectures, and scholarly articles. This enhances the learning experience by offering students and educators a way to quickly review key concepts and theories, facilitating a better understanding and retention of knowledge.

- News and Media: The news and media industry utilizes text summarization to generate news digests and article summaries, allowing readers to quickly catch up on important events and stories. This application caters to the growing demand for concise, accurate news content in an era of information overload, helping readers stay informed without dedicating extensive time to reading full articles.

The widespread applications of text summarization underscore its versatility and potential to impact various aspects of society positively. By enabling quicker access to and comprehension of large volumes of text, summarization technologies are playing a pivotal role in information management across fields, driving efficiency, and fostering informed decision-making.

2.7. Applications of LLMs in Scientific Summarization

The advent of Large Language Models (LLMs) has opened new horizons in the field of scientific summarization, offering sophisticated tools capable of understanding and generating complex, nuanced summaries of scientific literature. These models, trained on extensive corpora of text, leverage their vast knowledge base and deep learning capabilities to produce summaries that are not only concise but also rich in content and context.

- Transforming Scientific Literature Access: LLMs have the potential to significantly enhance the accessibility of scientific knowledge. By providing succinct, accurate summaries of lengthy research papers, these models make it easier for researchers, practitioners, and the general public to quickly grasp essential findings and implications without delving into the full text. This capability is particularly beneficial in fields where staying abreast of the latest research is crucial but challenging due to the sheer volume of publications.

- Facilitating Cross-disciplinary Research: One of the standout applications of LLMs in scientific summarization is their ability to bridge knowledge gaps between disciplines. Summaries generated by LLMs can be tailored to different expertise levels, enabling researchers from various fields to understand and leverage findings from outside their areas of specialization. This cross-pollination of knowledge fosters innovation and encourages a more integrative approach to solving complex scientific problems.

- Enhancing Research Efficiency: LLMs streamline the literature review process, a task that is both time-consuming and critical to the research workflow. By quickly summarizing key aspects of related studies, LLMs allow researchers to identify relevant research, understand broader trends, and pinpoint gaps in the literature more efficiently. This not only speeds up the research process but also improves the quality of literature reviews by ensuring comprehensive coverage of existing work.

- Customization and Hyperparameter Tuning: The effectiveness of LLMs in scientific summarization can be further enhanced through customization and hyperparameter tuning, as discussed previously. By adjusting model parameters such as learning rate, batch size, and attention heads, developers can optimize LLMs for specific summarization tasks, improving their performance in terms of accuracy, coherence, and relevance. This tailored approach ensures that the summaries meet the exact needs of the target audience, whether for detailed analysis or broad overview.

The integration of LLMs into web applications for scientific summarization represents a significant advancement in making scientific knowledge more accessible and understandable. As these technologies continue to evolve, their application in summarizing scientific literature is poised to become an indispensable tool in the research community, democratizing access to information and facilitating the rapid dissemination of scientific discoveries.

2.8. Evaluation of LLM Text Summarization

Evaluating the performance of Large Language Models (LLMs) in text summarization is pivotal to ensure the generation of accurate, coherent, and relevant summaries. Various metrics have been developed to assess different aspects of summaries, broadly categorized into reference-based and context-based metrics.

Reference-based metrics evaluate the quality of generated summaries by comparing them to one or more human-written reference summaries. The most widely used reference-based metric is ROUGE (Recall-Oriented Understudy for Gisting Evaluation), which measures the overlap of n-grams between the generated and reference texts (Lin, 2004).

ROUGE-1 quantifies the recall of unigram (single-word) overlap between a reference text and the generated text.

$$\text{Rouge} - 1\ (\text{Recall}) \ = \frac{unigram\ matches}{unigram\ in\ reference}$$

$$\text{Rouge} - 1\ (\text{Precision}) \ = \frac{unigram\ matches}{unigram\ in\ output}$$

$$\text{Rouge} - 1\ (\text{F1}) = 2\ \text{x}\ \frac{Precision\ x\ Recall}{Precision+Recall}$$

Figure 4: ROUGE 1

ROUGE-L, however, contrasts with ROUGE-1 and ROUGE-2 scores, which are based on unigram or bigram overlaps. Instead, it compares the Longest Common Subsequence (LCS) between the reference and human-generated text.

$$\text{Rouge} - \text{L}\ (\text{Recall}) = \frac{Length\ of\ LCS}{unigrams\ in\ reference}$$

$$\text{Rouge} - \text{L}\ (\text{Precision}) = \frac{Length\ of\ LCS}{unigrams\ in\ output}$$

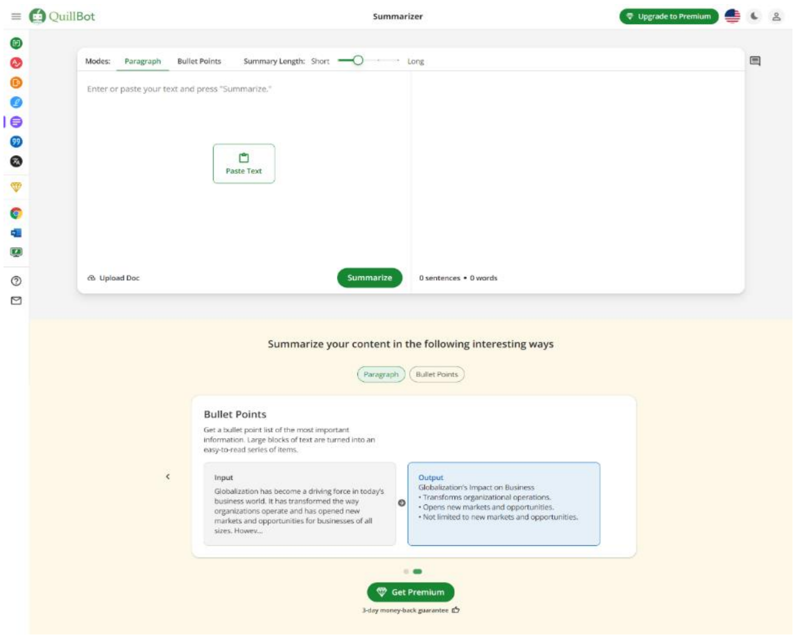
$$\text{Rouge} - \text{L}\ (\text{Precision}) = 2\ \text{x}\ \frac{Precision\ x\ Recall}{Precision + Recall}$$

Figure 5: ROUGE-L

2.9. Similar Applications in Text Summarization:

The landscape of text summarization technology is rich and varied, with numerous applications designed to serve different needs across academic research, business, legal, and casual reading. Below are some notable examples of applications that leverage text summarization to enhance efficiency and information accessibility:

- QuillBot Summarizer: QuillBot provides a summarization tool that helps condense long articles, papers, and documents into concise summaries. It offers flexibility in summary generation, allowing users to choose between key sentences or a concise paragraph format. This tool is particularly useful for students, researchers, and professionals looking to extract vital information quickly from extensive texts. QuillBot's application of AI and NLP technologies ensures that the summaries are accurate and maintain the original text's context.



Enter or paste your text and press "Summarize."

Summarize your content in the following interesting ways

Bullet Points

Get a bullet point list of the most important information. Large blocks of text are turned into an easy-to-read series of items.

Input
Globalization has become a driving force in today's business world. It has transformed the way organizations operate and has opened new markets and opportunities for businesses of all sizes. Howev...

Output
Globalization's Impact on Business
• Transforms organizational operations.
• Opens new markets and opportunities.
• Not limited to new markets and opportunities.

Figure 6: QuillBot Summarizer website

- Paraphraser Text Summarizer: In exploring existing applications that align with the objectives of our project, we have identified "Paraphraser text summarizer" as a notable example in the domain of text processing. This web-based application serves as a dual-purpose tool, offering both paraphrasing and summarization services to its users. This application utilizes advanced algorithms to rephrase text, providing alternatives that maintain the original meaning while changing the structure and wording. This is particularly useful for avoiding plagiarism and enhancing clarity. Alongside paraphrasing, it offers a summarization feature that condenses articles, research papers, and other forms of text into shorter versions, highlighting key points and essential information.

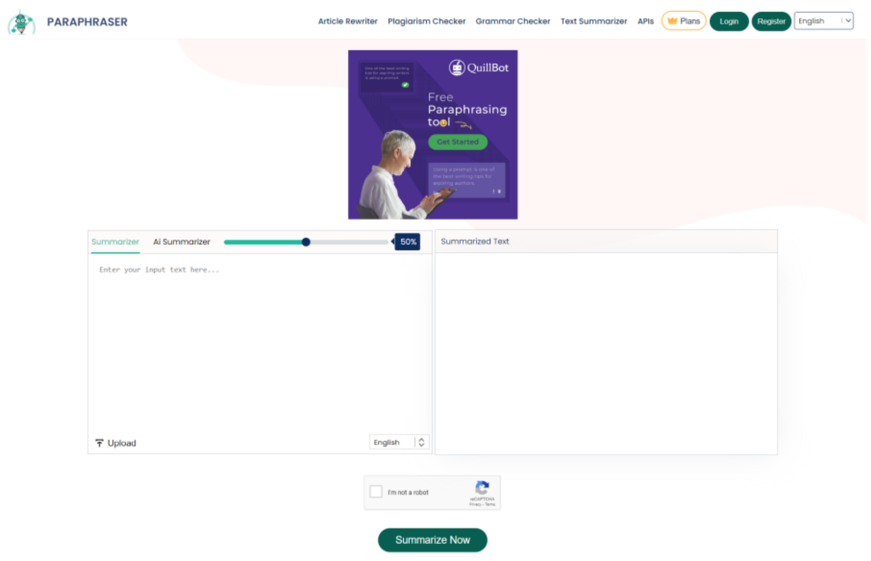


Figure 7: Paraphraser Text Summarizer

2.10. Future Directions:

- Enhanced Personalization: Future developments could focus on personalizing summarization outputs to cater to individual preferences, expertise levels, and specific informational needs. This would involve leveraging user feedback and AI to adapt summarization styles and content, making information even more accessible and relevant to diverse audiences.

- Interdisciplinary Integration: There is significant potential in integrating text summarization technologies with other AI domains, such as image and speech recognition, to create multimodal summarization tools. These integrated systems could offer comprehensive summaries that include text, visual elements, and audio, enhancing understanding and engagement across various content types.

- Advancements in Explainability: As LLMs and other AI models become more complex, increasing their explainability and transparency becomes crucial. Research into making these models more interpretable to users, explaining how summaries are generated and on what basis, could increase trust and reliability in AI-generated summaries.

- Real-time Summarization: The ability to provide real-time summarization of live content, such as conferences, lectures, and news broadcasts, represents an exciting frontier. This would require significant advancements in processing speed and model efficiency but could transform how information is consumed and disseminated in various contexts.

- Ethical Considerations and Bias Mitigation: Future research must also address the ethical implications of text summarization, including issues of data privacy, consent, and the potential for bias in summarization outputs. Developing frameworks and methodologies for identifying and mitigating bias in AI models will be essential to ensure equitable and fair use of summarization technologies.

The journey of text summarization from its early days to the advent of sophisticated LLM-driven approaches marks a significant evolution in the field of natural language processing. Today, text summarization technologies offer invaluable tools across various sectors, from scientific research to healthcare, legal, business, and beyond, facilitating access to and comprehension of vast amounts of information. As we look to the future, the continued advancement of these technologies promises even greater capabilities and broader applications, underscoring the importance of ongoing research, ethical considerations, and interdisciplinary collaboration.

3. System requirements

This chapter discusses the methodology followed in developing a Scientific AI Summarizer. It lays out installing the environment, data preprocessing steps, model training process, and visualizing results in a human-readable format. Detailed explanations of the tasks and processes in each section were made to give full clarity on how this was done.

3.1. System Requirements

Several system requirements were identified to build an efficient and effective Scientific AI Summarizer. These requirements have been divided into Functional, Non-Functional, and Technical Requirement Categories.

3.1.1. Functional Requirements

- Text Input Handling: The system should be able to accept research papers in several forms from the user, including plain text, PDF, and Word documents.
- Text Summarization: The system should create a smaller version of the text, which is known as summaries, from the trained model.
- User Interface: The web interface should be intuitively designed and better suited for text-input to summary-output relationship.
- File Handling: This system must support file uploads via drag and drop from the directory and also allow manual file selection.
- Word Count Display: The system shall indicate the original text word count.
- File Name Display: Where applicable, the system shall display the uploaded file name.
- Download Summary: Users can download the full summary as a text file.
- Copy Summary: Users should be able to copy the summary by clicking a button.
- Themes: The system will incorporate two themes—light and dark, allowing users to switch between them.

3.1.2. Non-Functional Requirements

- Performance: The system should be able to process and summarize text within a maximum of 30–60 seconds.
- Scalability: The system should scale with multithreading to handle multiple request processing simultaneously.
- Usability: The system should have an easy-to-use interface that is user-friendly, even for novice users.
- Reliability: The system should provide a summary that is both valid and correctly conveys the most important aspects.

3.1.3. Technical Requirements

- Software: For the project, various Python libraries need to be installed, including transformers, datasets, accelerate, evaluate, bitsandbytes, peft, rouge-score, tensorboard, py7zr, Flask, and Ngrok.

- Hardware: The training was performed using Google Colab Pro with an A100 GPU on this machine:

  RAM: 85 GB

  GPU RAM (vRAM): 40.0 GB

  Disk Space: 100 GB

3.1.4. Screen Design

The design and layout of the user interface are important to provide a well-structured summary accessible by end-users on how they interact with the Scientific AI Summarizer. The interface is intuitive, and responsive, and offers a seamless workflow for text summarization tasks. This section gives a deep dive into the interface design with screenshots.

3.1.5. Interface Overview

The interface contains the following main components:

- File Upload and Text Input: Users can either upload a document or paste text directly into the text area.
- Word Count Display: The user interface displays the word count live as text is inputted.
- File Name Display: It shows the file name after uploading, right under the word count.
- Summarize Button: When clicked, this button will initiate the summarization process.
- Summary Output: The generated summary will appear in a separate output area.

- Copy to Clipboard: Users can copy the summary to the clipboard with a click.
- Download as TXT: It allows users to download the summary as a file in TXT format.
- Theme Toggle: Light and Dark modes can be used by the users for this interface.

3.1.6. Interface Screenshots

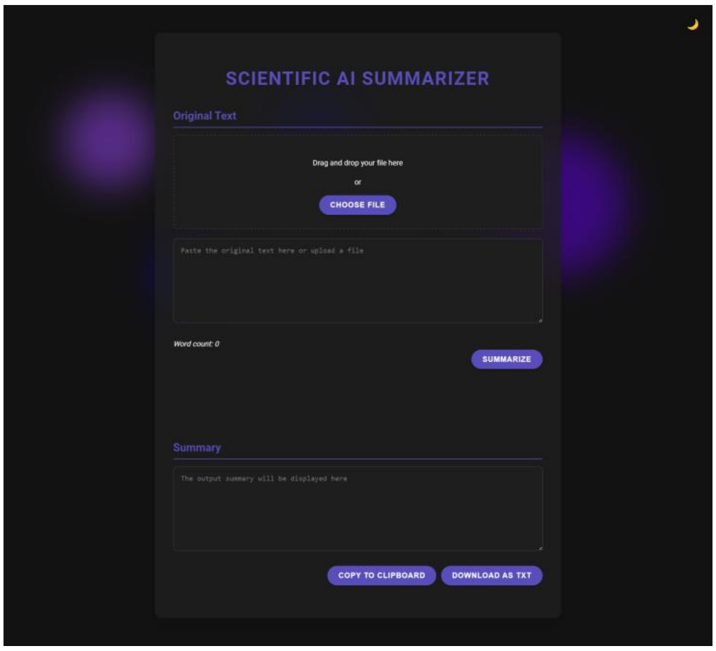

SCIENTIFIC AI SUMMARIZER
Original Text
Drag and drop your file here
or
CHOOSE FILE
Paste the original text here or upload a file
Word count: 0
SUMMARIZE
Summary
The output summary will be displayed here
COPY TO CLIPBOARD
DOWNLOAD AS TXT

Figure 8: Initial State Dark mode

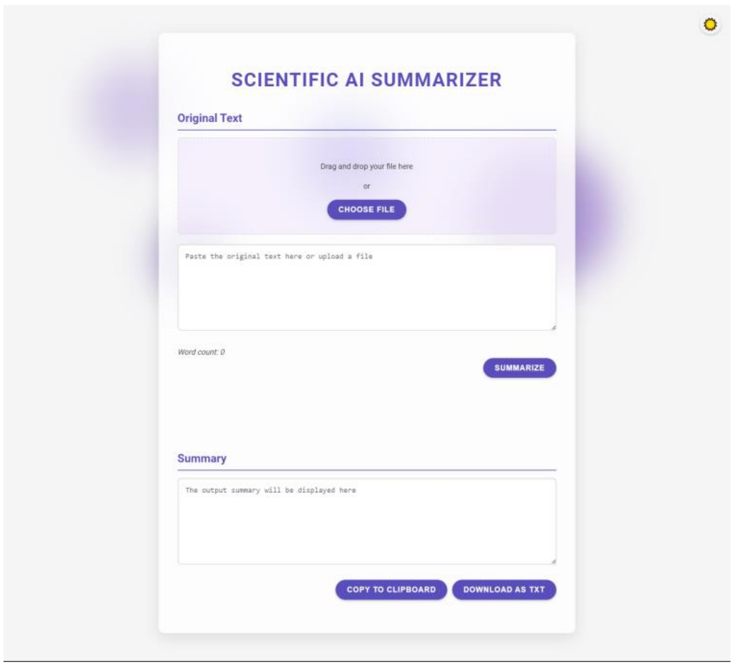
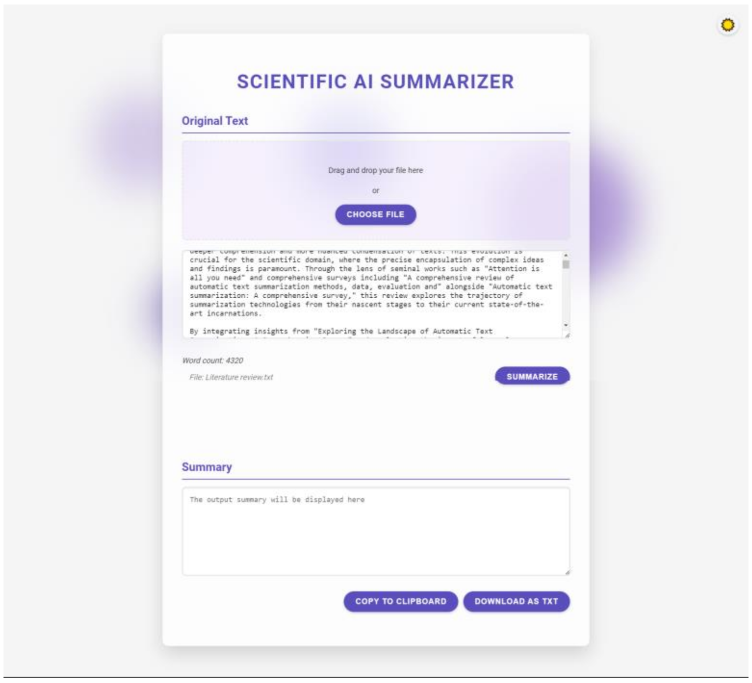

SCIENTIFIC AI SUMMARIZER
Original Text
Drag and drop your file here
or
CHOOSE FILE
Paste the original text here or upload a file
Word count: 0
SUMMARIZE
Summary
The output summary will be displayed here
COPY TO CLIPBOARD
DOWNLOAD AS TXT

Figure 9: Initial State Light mode

Description: The screenshot shows the initial state of the Scientific AI Summarizer interface. The interface offers options to either drag and drop a file or choose a file using the "Choose File" button. The "Summarize" button is available for generating summaries, and there are options to copy the summary to the clipboard or download it as a TXT file. At this stage, no text has been inputted, and the word count is zero.



crucial for the scientific domain, where the precise encapsulation of complex ideas and findings is paramount. Through the lens of seminal works such as "Attention is all you need" and comprehensive surveys including "A comprehensive review of automatic text summarization methods, data, evaluation and" alongside "Automatic text summarization: A comprehensive survey," this review explores the trajectory of summarization technologies from their nascent stages to their current state-of-the-art incarnations.

By integrating insights from "Exploring the Landscape of Automatic Text

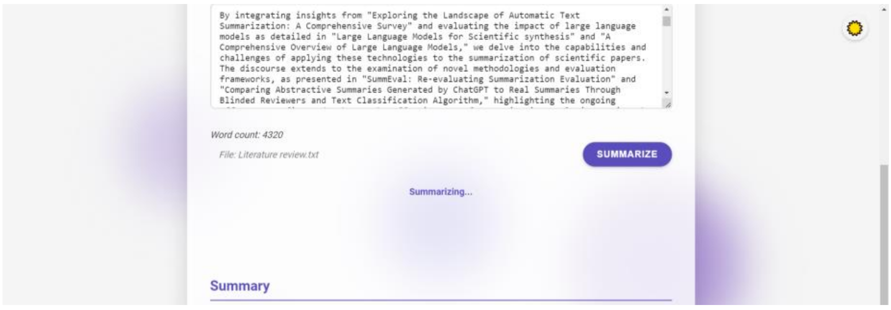
Word count: 4320

File: Literature review.txt

The output summary will be displayed here

Figure 10: Interface with Uploaded Text

This screenshot shows the light mode interface with text uploaded into the text area. The word count has been updated to reflect the input text, and the file name "Literature review.txt" is displayed below the word count. The "Summarize" button is available for generating the summary.

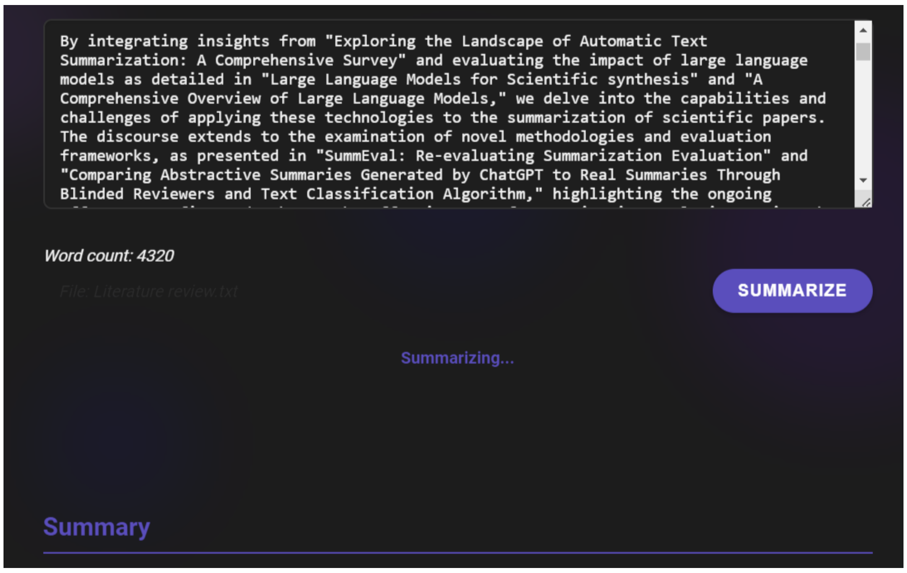


By integrating insights from "Exploring the Landscape of Automatic Text Summarization: A Comprehensive Survey" and evaluating the impact of large language models as detailed in "Large Language Models for Scientific synthesis" and "A Comprehensive Overview of Large Language Models," we delve into the capabilities and challenges of applying these technologies to the summarization of scientific papers. The discourse extends to the examination of novel methodologies and evaluation frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and "Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing

Word count: 4320

File: Literature review.txt

Figure 11: Summarization in Progress light mode



```
By integrating insights from "Exploring the Landscape of Automatic Text
Summarization: A Comprehensive Survey" and evaluating the impact of large language
models as detailed in "Large Language Models for Scientific synthesis" and "A
Comprehensive Overview of Large Language Models," we delve into the capabilities and
challenges of applying these technologies to the summarization of scientific papers.
The discourse extends to the examination of novel methodologies and evaluation
frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and
"Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through
Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing
```

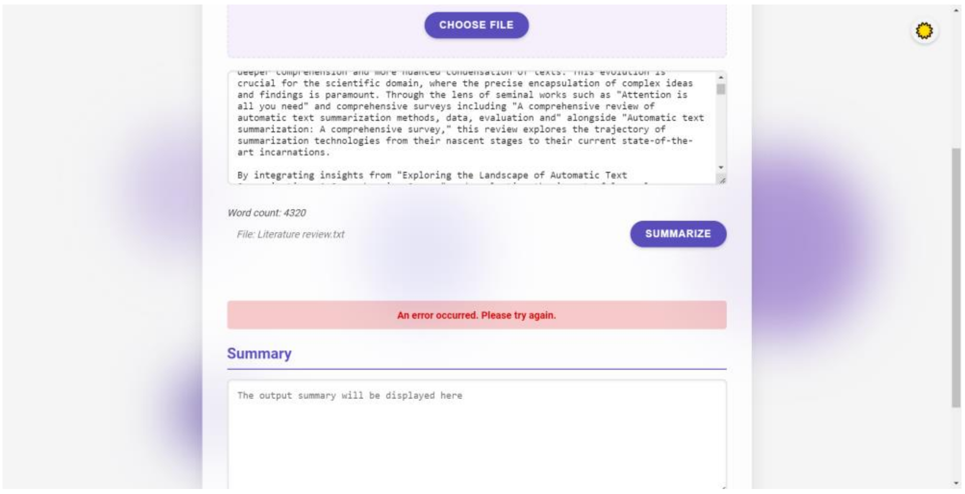
Word count: 4320

File: Literature review.txt

Summarizing...

Figure 12: Summarization in Progress Dark mode

This screens shows the interface while the summarization process is ongoing. The text "Summarizing..." is displayed, indicating that the system is processing the input text.



```
crucial for the scientific domain, where the precise encapsulation of complex ideas
and findings is paramount. Through the lens of seminal works such as "Attention is
all you need" and comprehensive surveys including "A comprehensive review of
automatic text summarization methods, data, evaluation and" alongside "Automatic text
summarization: A comprehensive survey," this review explores the trajectory of
summarization technologies from their nascent stages to their current state-of-the-
art incarnations.

By integrating insights from "Exploring the Landscape of Automatic Text
```

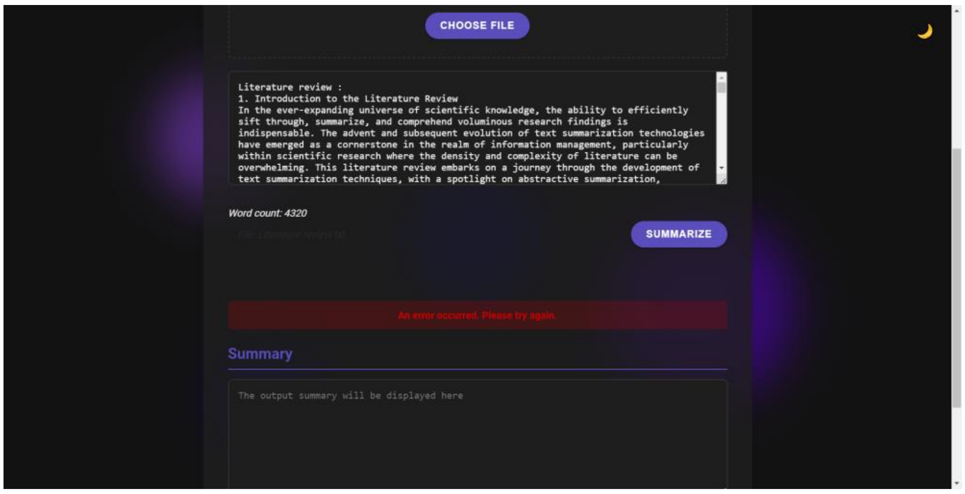
Word count: 4320

File: Literature review.txt

An error occurred. Please try again.

The output summary will be displayed here

Figure 13: Error Message Display light mode



CHOOSE FILE

```
Literature review :
1. Introduction to the Literature Review
In the ever-expanding universe of scientific knowledge, the ability to efficiently
sift through, summarize, and comprehend voluminous research findings is
indispensable. The advent and subsequent evolution of text summarization technologies
have emerged as a cornerstone in the realm of information management, particularly
within scientific research where the density and complexity of literature can be
overwhelming. This literature review embarks on a journey through the development of
text summarization techniques, with a spotlight on abstractive summarization,
```

Word count: 4320

File: Literature review.txt

SUMMARIZE

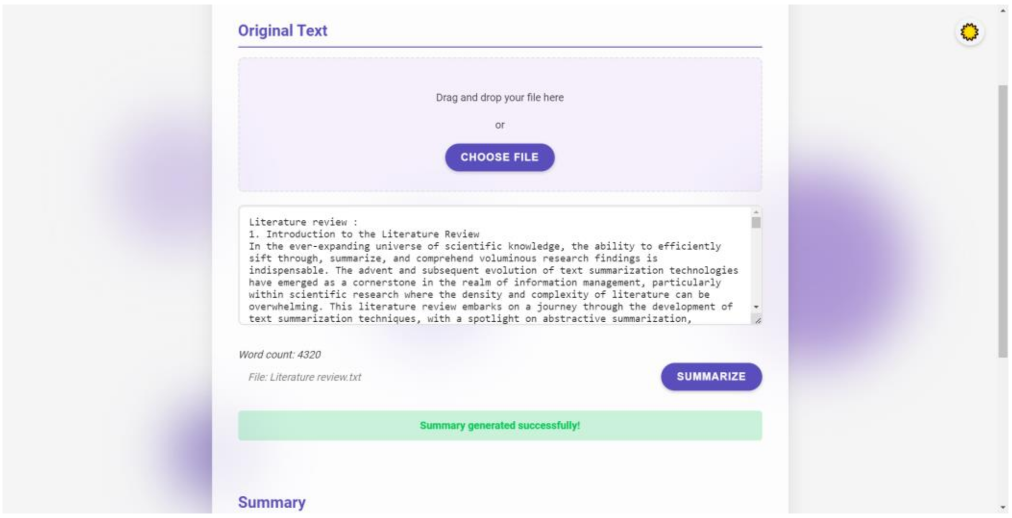
An error occurred. Please try again.

Summary

The output summary will be displayed here

Figure 14: Error Message Display Dark mode

This screenshot captures the interface in light mode after an unsuccessful attempt to summarize the uploaded text. An error message is displayed in red, indicating that an error occurred during the summarization process.



Original Text

Drag and drop your file here

or

CHOOSE FILE

```
Literature review :
1. Introduction to the Literature Review
In the ever-expanding universe of scientific knowledge, the ability to efficiently
sift through, summarize, and comprehend voluminous research findings is
indispensable. The advent and subsequent evolution of text summarization technologies
have emerged as a cornerstone in the realm of information management, particularly
within scientific research where the density and complexity of literature can be
overwhelming. This literature review embarks on a journey through the development of
text summarization techniques, with a spotlight on abstractive summarization,
```

Word count: 4320

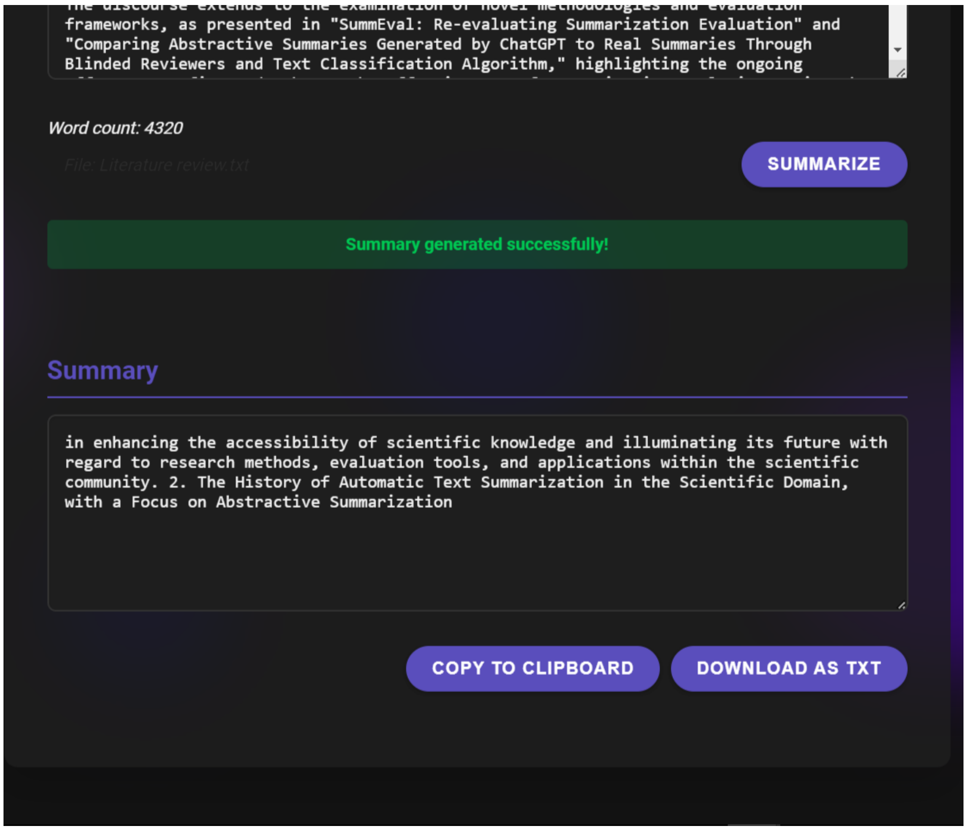
File: Literature review.txt

SUMMARIZE

Summary generated successfully!

Summary

Figure 15: Successful Summarization light mode



```
The discourse extends to the examination of novel methodologies and evaluation
frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and
"Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through
Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing
```

Word count: 4320

File: Literature review.txt

Summary generated successfully!

Summary

```
in enhancing the accessibility of scientific knowledge and illuminating its future with
regard to research methods, evaluation tools, and applications within the scientific
community. 2. The History of Automatic Text Summarization in the Scientific Domain,
with a Focus on Abstractive Summarization
```

Figure 16: Successful Summarization Dark mode

This screenshot shows the interface with a successfully generated summary. The original text is displayed in the text area, with the word count and file name displayed below. A success message is shown, indicating the summary was generated successfully. The summary output area is ready to display the generated summary.

3.2. Principles of Design

The following design elements guide the interface:

- User-Friendly: The interface is simple and easy to navigate, with clear instructions and feedback.
- Responsive Design: The layout adapts to different screen sizes, ensuring it works on every device.
- Visual Appeal: The integration of modern typefaces, color schemes, and animations makes the interface visually appealing.
- Accessibility: The design includes features like dark mode and clear text to ensure accessibility for all users.

3.2.1. User Interaction

File Upload and Text Input

- Users can upload a file by dragging and dropping it into the designated area or by clicking the "Choose File" button.
- Alternatively, users can paste text directly into the text area provided.

Word Count and File Name Display

- The interface dynamically updates the word count as text is inputted.
- When a file is uploaded, its name is displayed below the word count, providing users with immediate feedback.

Summarize Button

- Clicking the "Summarize" button initiates the summarization process. A loading indicator is displayed while the system processes the input text.

Summary Output

- The generated summary is displayed in the output area. Users can then copy the summary to the clipboard or download it as a TXT file using the provided buttons.

Theme Toggle

- Users can switch between light and dark modes using the theme toggle button located at the top right corner of the interface.

3.3. Theoretical Framework

3.3.1. Literature Review

The Scientific AI Summarizer is developed based on advancements in large language models (LLM), particularly in text summarization. Text summarization methods have evolved significantly from traditional extractive algorithms to advanced abstractive models.

Key Literature and Models:

- **Extractive Summarization:** Early methods focused on identifying and extracting key sentences from the original text to create a summary. Popular techniques included frequency-based methods, graph-based methods like TextRank, and machine learning classifiers.
- **Abstractive Summarization:** This approach generates new sentences that convey the meaning of the original text, similar to human summarization. Abstractive methods use sequence-to-sequence models, such as those based on Transformer architecture.
- **Transformer Models:** Introduced by Vaswani et al. (2017), Transformer models revolutionized NLP with attention mechanisms, allowing models to capture long-range dependencies more effectively than traditional RNN-based models.
- **Pre-trained Language Models:** Models like BERT, GPT, and T5 have shown remarkable performance in various NLP tasks, including summarization. These models are pre-trained on large corpora and fine-tuned for specific tasks.
- **Evaluation Metrics:** ROUGE scores are the standard for evaluating the quality of summaries, comparing the overlap of n-grams, word sequences, and word pairs between the generated summary and the reference summary.

3.3.2. Model Selection

The google/flan-t5-xxl model was chosen for the Scientific AI Summarizer due to its state-of-the-art performance in text summarization tasks.

Reasons for Selecting google/flan-t5-xxl:

- State-of-the-art Performance: The flan-t5-xxl model has demonstrated superior performance in generating coherent and concise summaries across various domains.
- Scalability: Its architecture allows for handling large-scale datasets and generating high-quality summaries for lengthy scientific texts.
- Flexibility: The model's ability to be fine-tuned on specific datasets makes it adaptable to the nuances of scientific literature.

3.3.3. Algorithm and Techniques

The implementation of the Scientific AI Summarizer involves several critical steps, leveraging LLM techniques and algorithms.

Key Algorithms and Techniques:

- Tokenization: Utilizing the AutoTokenizer from the Hugging Face library, the text data is tokenized to convert the input text into a format suitable for the model. This includes handling out-of-vocabulary words and managing input sequence lengths.
- Parameter-Efficient Fine-Tuning (PEFT): The peft library is employed for parameter-efficient fine-tuning, allowing the model to adapt to the specific characteristics of scientific text without overfitting.
- Low-Rank Adaptation (LoRA): The LoRA technique reduces the number of trainable parameters by projecting the weights of the neural network into a lower-dimensional space. This approach significantly reduces computational cost and memory usage while maintaining high performance.
- Training Configuration: Training is conducted using the Seq2SeqTrainer from the Hugging Face library with the following parameters:
  - Learning Rate: 1e-5
  - Number of Epochs: 3
  - Mixed Precision Training (FP16): Utilized to optimize training time and computational resources.
  - Batch Size: Automatically determined using auto_find_batch_size.
- Checkpointing and Logging: Configured to save training progress and facilitate monitoring, ensuring the training process is transparent and adjustable based on performance metrics.

3.3.4. Evaluation Metrics

The performance of the Scientific AI Summarizer is evaluated using the ROUGE metric, a set of metrics for evaluating automatic summarization and machine translation.

ROUGE Metrics Used:

- ROUGE-1: Measures the overlap of unigrams (single words) between the generated summary and the reference summary.
- ROUGE-2: Measures the overlap of bigrams (two consecutive words) between the generated summary and the reference summary.
- ROUGE-L: Measures the longest common subsequence (LCS) between the generated summary and the reference summary, reflecting fluency and coherence.
- ROUGE-Lsum: A variant of ROUGE-L that measures the LCS at the summary level.

By integrating these metrics, the summarizer performance is quantitatively assessed, ensuring that the generated summaries are concise while maintaining the essential information and coherence of the original texts.

The comprehensive approach outlined in this theoretical framework section ensures a robust foundation for the Scientific AI Summarizer, leveraging cutting-edge techniques in LLM and machine learning to achieve high-quality summarization of scientific texts.

3.4. Technical Specifications

This section provides an in-depth explanation of the technical specifications and configurations used in the development of the Scientific AI Summarizer. This includes environment details, such as hardware, software, and specific libraries and frameworks essential for the project.

3.4.1. Hardware Environment

Due to the computational requirements needed to train and run the AI model, the hardware environment is crucial. The project made use of cloud resources to ensure scalability and efficiency.

- Google Colab Pro: Utilized for its high-performance GPUs, specifically the NVIDIA A100, which provides the necessary computational power for training large-scale AI models.
- Local Development Machine: A machine with at least 16GB RAM and a multi-core CPU was used for initial development and testing. However, the primary

training and processing were offloaded to the cloud environment to leverage superior GPU capabilities.

3.4.2. Software Environment

The software environment refers to the operating system, programming languages, libraries, and frameworks used throughout the project. This setup ensured compatibility and efficiency in development and deployment.

- Operating System: Initial development was conducted on a Windows 10 machine, with cloud resources running on Linux-based systems in Google Colab.
- Programming Languages:
  - Python 3.8+: The core programming language used for backend development, data processing, and AI model training.
  - HTML/CSS/JavaScript: Used for developing the user interface and ensuring a responsive web design.

3.4.3. Libraries and Frameworks

Several libraries and frameworks were employed to facilitate different aspects of the project, from data processing to model training and web development.

- Data Processing:
  - Pandas: For data manipulation and analysis.
  - Requests: For making API calls to fetch data.
  - XML.etree.ElementTree: For parsing XML responses from the arXiv API.
  - Re: For regular expression operations in text cleaning.
  - Pdfminer.six and PyMuPDF (Fitz): For extracting and cleaning text from PDF files.

- AI Model and Training:
  - Transformers: Hugging Face's Transformers library for leveraging the T5 model and related utilities.
  - Datasets: Used for handling and processing large datasets efficiently.

  - Seq2SeqTrainer: Part of the Transformers library, used for training the sequence-to-sequence model.
  - Bitsandbytes: For efficient 8-bit optimization during model training.
  - Accelerate: For managing multi-GPU training and optimizing performance.
  - LoRA (Low-Rank Adaptation): A technique used for parameter-efficient fine-tuning of the AI model.

- Web Development:
  - Flask: A lightweight web framework for developing the backend server.
  - Ngrok: For exposing the local Flask server to the internet, enabling easy testing and development.
  - pdf.js: A library for extracting text from PDF files within the web interface.
  - Mammoth: For extracting text from Word documents.

3.4.4. Configuration and Optimization

To achieve fast and efficient model training and deployment, optimization techniques were applied across all components, including the user interface.

- Mixed Precision Training: A technique to speed up training by using lower precision where available.
- Regular Checkpointing: Implemented to save model states at intervals, allowing for recovery in case of interruptions.
- Parameter Tuning: Learning rate, batch size, and other hyperparameters were fine-tuned to achieve the best performance.
- Frontend Optimization: Ensured quick loading times and a smooth user experience through efficient coding practices and asynchronous processing.

The technical specifications outlined above were carefully chosen and configured to support the development and deployment of the Scientific AI Summarizer. These requirements allowed the system to handle large-scale data processing and model training efficiently while providing an accessible platform for summarizing scientific papers.

4. Design

This chapter describes the design and research methodology, including environment setup, data preparation, model training, evaluation, and user interface creation for the Scientific AI Summarizer. The design process involved selecting tools, frameworks, and technologies to build a robust solution, integrating components for optimized data flow and processing.

For text summarization, advanced transformer models were used, including steps like model selection and fine-tuning, and backend integration to handle user inputs and generate summaries. The interface design focused on usability, offering features such as word count display, file information, and options to copy or download summaries, using modern web technologies for the frontend and Flask for server-side processing. Challenges like handling diverse file formats, optimizing model performance, and ensuring a responsive interface were addressed with practical solutions implemented throughout the project.

4.1. Data Collection

This project collects data by fetching and preprocessing full-text papers from the arXiv API using the metadata available in the initial dataset, cs_papers_api.csv. This dataset includes metadata for various computer science papers, such as paper IDs, titles, abstracts, publication years, primary categories, and additional categories. Below is a detailed description of the data collection process:

4.1.1. Original Dataset:

cs_papers_api.csv: This dataset contains six columns: paper_id, title, abstract, year, primary_category, and categories. There are 200,094 entries, providing a substantial base for analysis.

Columns:

- paper_id: Unique identifier for each paper.
- title: Title of the paper.
- abstract: Abstract of the paper.
- year: Year of publication.
- primary_category: Main category of the paper.
- categories: Other categories related to the paper.
- Fetching Metadata:

4.1.2. Fetching Metadata:

Metadata for each paper is fetched using the paper ID through the arXiv API. This step involves making HTTP requests to the arXiv API to obtain the latest metadata for each paper. The metadata includes essential information such as the paper's title, abstract, authors, and the link to the full-text PDF.

4.1.3. Fetching Full Text:

Full-text papers are downloaded using the metadata retrieved from the arXiv API. The PDF links provided in the metadata are used to download the full-text PDFs of the papers. This involves handling various file formats and ensuring that the text content is accurately extracted from the PDFs.

4.1.4. Cleaning Text:

- The downloaded text is cleaned by removing unnecessary elements such as HTML tags, extra spaces, and specific keywords. This cleaning process involves several steps:
    - Removing multiple whitespace characters and replacing them with a single space.
    - Removing text within square brackets and parentheses, which often contain references or additional notes.
    - Removing HTML tags to ensure only the plain text content is retained.
    - Removing URLs and other non-essential information.
- Additionally, the abstract text is removed from the full text to avoid redundancy.

4.1.5. Final Dataset:

full_data.csv: The cleaned dataset contains full-text papers ready for analysis. This dataset includes the original columns from cs_papers_api.csv along with the cleaned full text. The cleaned data ensures that the text is in a usable format for natural language processing and further analysis.

Below is the visual representation of the data collection process:

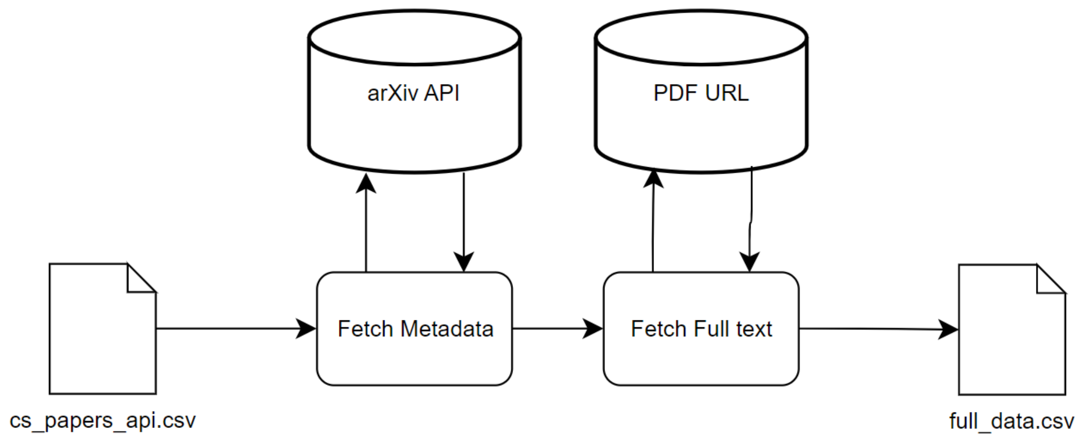


Figure 17: Data Collection Process

This diagram illustrates the flow of data from the initial dataset through the fetching and cleaning processes, resulting in the final dataset. By following this structured approach, the data is accurately collected and prepared for subsequent analysis and model training. The use of the arXiv API ensures that the most current and comprehensive data is utilized, enhancing the quality and reliability of the project's outcomes.

4.2. Solution Approach

The solution approach to develop the Scientific AI Summarizer was systematic and well-structured, ensuring every aspect of the project was meticulously planned and executed. The main components of this approach are detailed in the following sections:

4.2.1. Environment Setup

To create a conducive environment for model training and deployment, Google Colab Pro with an A100 GPU was utilized, providing significant computational power. The setup included installing necessary libraries such as Hugging Face's transformers, datasets, accelerate, and bitsandbytes for efficient model training. Additionally, Flask and Ngrok were installed to facilitate the development of the web interface.

4.2.2. Data Preparation

Data preparation was a crucial step, involving the collection and preprocessing of scientific papers. The dataset included titles, abstracts, and cleaned full texts of the papers. The data was split into training and testing sets, with 80% used for training and 20% for testing. Tokenization was performed using Hugging Face's AutoTokenizer, ensuring the data was in a suitable format for model training.

4.2.3. Data Collection

The original dataset, cs_papers_api.csv, contained 200,094 entries with columns including paper_id, title, abstract, year, primary_category, and categories. This dataset provided the basis for further data collection. Using the arXiv API, additional full-text data was fetched for the category 'Networking and Internet Architecture'. The collected data was then cleaned and processed to remove unnecessary elements, ensuring high-quality input for model training.

4.2.4. Model Selection and Training

The google/flan-t5-xxl model was selected for its superior performance in text summarization tasks. To make the training process more efficient, the LoRA (Low-Rank Adaptation) technique was employed for parameter-efficient fine-tuning. The training process involved configuring the Seq2SeqTrainer from Hugging Face with appropriate training parameters such as learning rate, number of epochs, and mixed precision training. Regular monitoring and checkpointing were implemented to track the training progress and ensure optimal performance.

4.2.5. Model Evaluation

The model's performance was evaluated using the ROUGE metric, which measures the quality of the generated summaries. The evaluation involved generating summaries for the test dataset and comparing them with the reference abstracts. Various configurations of maximum and minimum token lengths were tested to identify the best-performing model. The evaluation provided insights into the model's summarization capabilities and areas for further optimization.

4.2.6. User Interface Development

The user interface was developed to provide a seamless experience for users to interact with the summarization model. The frontend was designed with features such as drag-and-drop file upload, word count display, file information display, and options to copy or download summaries. The backend was built using Flask, handling user inputs, processing them through the model, and returning the generated summaries. The interface was designed to be user-friendly, with a focus on usability and aesthetics.

4.2.7. Handling File Formats

The interface supports multiple file formats, including plain text, PDF, and Word documents. To extract text from these formats, libraries such as pdf.js and Mammoth were integrated. This ensured that users could easily upload their documents in various formats and obtain summaries without any hassle.

4.2.8. Optimization and Performance

Throughout the development process, various optimization techniques were applied to improve the model's performance and ensure a responsive user interface. This included fine-tuning the model parameters, optimizing the data processing pipeline, and enhancing the frontend's responsiveness. The goal was to ensure that the system could process and summarize text in less than 30 seconds, providing a quick and efficient solution for users.

By following this structured solution approach, the project aimed to develop a robust and efficient AI summarization system capable of generating accurate and concise summaries of scientific papers, thereby enhancing the accessibility and comprehension of scientific knowledge.

4.3. System Architecture

The system architecture for the Scientific AI Summarizer is designed to efficiently handle user inputs, perform text summarization using an AI model, and return results through a user-friendly interface. The architecture is divided into several key components, each responsible for a specific part of the workflow. Below is a detailed description of each component, followed by a visual representation of the overall system architecture.

4.3.1. User Interface

The User Interface (UI) is the primary point of interaction for users. It allows users to input text or upload documents for summarization. The UI is designed to be intuitive and user-friendly, providing the following features:

- Drag-and-Drop File Upload: Users can easily upload documents by dragging and dropping them into the designated area.
- Text Area for Direct Input: Users can paste text directly into a text area for summarization.
- Word Count Display: The UI dynamically displays the word count of the input text.
- File Information Display: Information about the uploaded file, such as its name and size, is displayed to the user.
- Options to Copy or Download Summary: Users can copy the generated summary to the clipboard or download it as a text file.

4.3.2. Web Interface

The web interface is developed using HTML, CSS, and JavaScript. It handles user requests and communicates with the web server. The web interface ensures a smooth and interactive user experience, facilitating the input of text and documents, and displaying the generated summaries. It provides the following functionalities:

- Handling User Inputs: The web interface captures user inputs and sends them to the web server for processing.
- Displaying Results: The generated summaries are displayed in the UI, allowing users to view, copy, or download them.

- Error Handling: Any errors encountered during processing are displayed to the user, ensuring transparency and user awareness.

4.3.3. Web Server

The web server is built using Flask and Ngrok. It is responsible for handling user requests, routing them to the appropriate services, and returning the results. The web server acts as a bridge between the user interface and the backend processing components. Its responsibilities include:

- Routing Requests: The web server routes incoming user requests to the appropriate preprocessing and AI model components.
- Handling Responses: Once the processing is complete, the web server sends the results back to the web interface for display.
- Managing Sessions: The web server manages user sessions to ensure continuity and state management across interactions.

4.3.4. Preprocessing

The preprocessing component handles the initial processing of user inputs. This includes handling different file formats (plain text, PDF, DOCX), extracting text, and performing tokenization. The preprocessing ensures that the input data is in a suitable format for the AI model. Key tasks include:

- File Handling: Extracting text from various file formats using libraries like pdf.js for PDFs and Mammoth for Word documents.
- Text Cleaning: Removing unwanted characters, formatting issues, and irrelevant content to prepare clean input text.
- Tokenization: Converting the input text into tokens that the AI model can process. This is done using Hugging Face's AutoTokenizer.

4.3.5. File Handling

Within preprocessing, the file handling subcomponent deals with different types of documents. It extracts text from plain text files, PDFs, and DOCX files using appropriate libraries and methods. The extracted text is then cleaned and prepared for further processing. Specific functions include:

- Plain Text Handling: Reading and cleaning text from plain text files.
- PDF Handling: Extracting text from PDF documents using pdf.js and handling any format-specific issues.
- DOCX Handling: Extracting text from Word documents using Mammoth and ensuring the extracted text is properly formatted.

4.3.6. Tokenization

Tokenization is the process of converting the input text into tokens that the AI model can understand. This step is crucial for ensuring that the text is in a format suitable for the summarization model. The tokenization process includes:

- Sentence Splitting: Dividing the text into individual sentences for better processing.
- Token Generation: Converting sentences into tokens that can be fed into the AI model for summarization.

4.3.7. Model Initialization

Model initialization involves loading the T5 model and LoRA parameters into memory. This step is performed when the server starts, ensuring that the model is ready for processing user requests. Key activities include:

- Loading Pre-trained Model: Loading the pre-trained T5 model from the Hugging Face library.
- Applying LoRA Fine-Tuning: Applying LoRA parameters to the pre-trained model to enhance its performance for the specific summarization task.

4.3.8. AI Model Processing

The AI model processing component is responsible for generating summaries from the tokenized text. It uses the google/flan-t5-xxl model, fine-tuned with LoRA (Low-Rank

Adaptation) for parameter-efficient training. The model generates concise summaries based on the input text. The process involves:

- Loading the Model: The AI model and its parameters are loaded into memory for efficient processing.
- Generating Summaries: The tokenized text is fed into the model, which generates summaries based on the input content.

4.3.9. Summary Generation

The summary generation component takes the output from the AI model and formats it as a summary.

4.3.10. Output Handling

The final component handles the output, returning the generated summary to the web server, which then sends it back to the user interface. The user can then view, copy, or download the summary as needed. Key functions include:

- Returning Results: Sending the generated summaries back to the web server for delivery to the user interface.
- User Actions: Allowing users to copy the summary to the clipboard or download it as a text file.

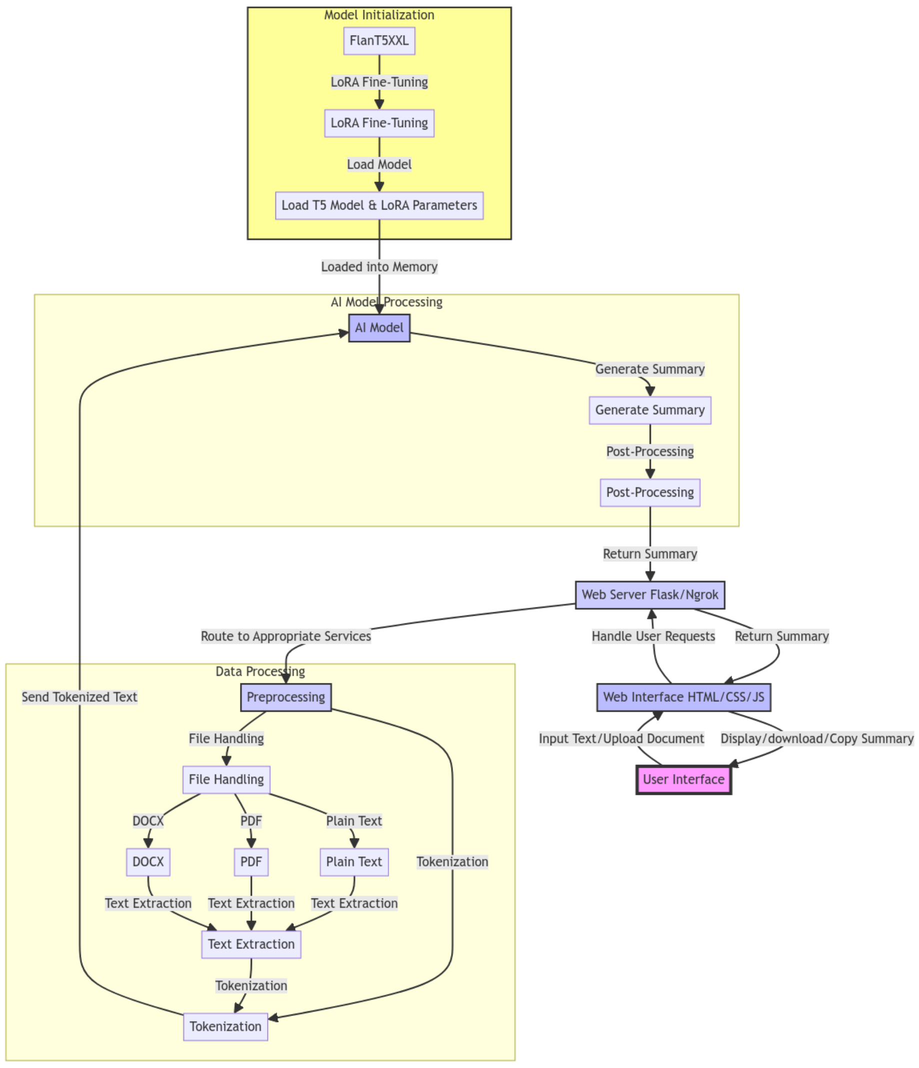


Figure 18: System Architecture

This architecture ensures a seamless and efficient workflow for summarizing scientific papers, from user input to summary generation and output. By modularizing the components, the system is designed to be scalable, maintainable, and easy to understand.

4.4. Technical Representations

4.4.1. Use Case Diagram

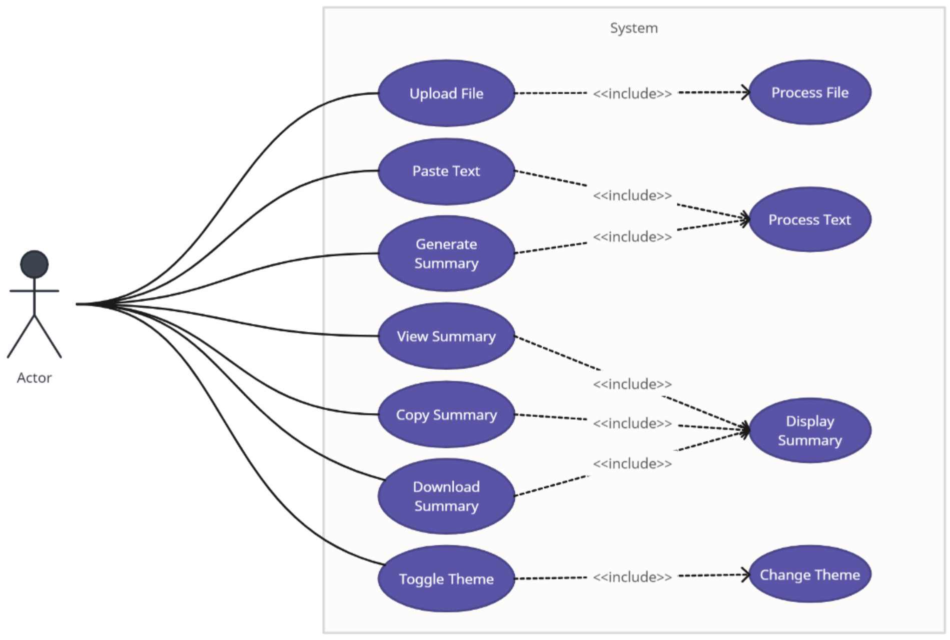


Figure 19: Use Case Diagram

This diagram illustrates the interactions between the user (actor) and the Scientific AI Summarizer system. It includes the following use cases:

- **Upload File**: The user uploads a file to the system. This action triggers the "Process File" use case, which handles the extraction and processing of text from the uploaded file.
- **Paste Text**: The user pastes text directly into the system. This action triggers the "Process Text" use case, which processes the pasted text to prepare it for summarization.
- **Generate Summary**: The user initiates the generation of a summary from the processed text. This use case involves the system's core summarization functionality.

- **View Summary**: The user views the generated summary. This action is linked to the "Display Summary" use case, which handles the presentation of the summary to the user.
- **Copy Summary**: The user copies the generated summary to the clipboard. This action is also linked to the "Display Summary" use case, ensuring the summary is presented and ready for copying.
- **Download Summary**: The user downloads the generated summary as a text file. This use case involves providing the summary in a downloadable format.
- **Toggle Theme**: The user switches between light and dark themes for the user interface. This action triggers the "Change Theme" use case, which adjusts the visual presentation of the interface.

Each use case is connected to the corresponding system processes via include relationships, indicating the internal operations required to fulfill each user action. This diagram provides a clear overview of the system's functionality and user interactions.

4.4.2. Sequence Diagram

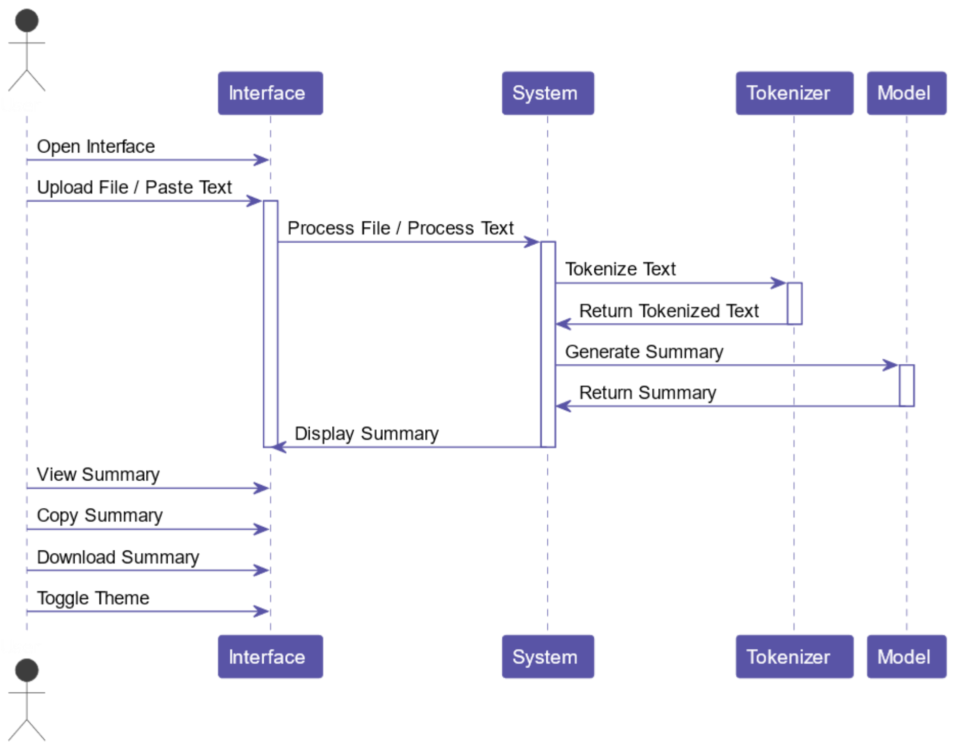


Figure 20: Sequence Diagram

This sequence diagram illustrates the interactions between the user, the interface, and the underlying system components in the Scientific AI Summarizer.

- User Actions:
  - The user opens the interface of the Scientific AI Summarizer.
  - The user uploads a file or pastes text into the interface.
  - The user interacts with the interface to view the summary, copy the summary, download the summary, and toggle the theme.
- Interface Actions:
  - The interface processes the file or text input by the user.
  - The interface communicates with the system to process the text.
  - The interface displays the generated summary to the user.
- System Actions:
  - The system processes the file or text input from the interface.
  - The system tokenizes the text using the Tokenizer component.
  - The system generates a summary using the Model component.
  - The system returns the generated summary to the interface for display.
- Tokenizer Actions:
  - The tokenizer tokenizes the input text received from the system.
  - The tokenizer returns the tokenized text to the system.
- Model Actions:
  - The model generates a summary from the tokenized text.
  - The model returns the generated summary to the system.

This sequence diagram encapsulates the entire process of generating a summary from user input, highlighting the interactions between different components and ensuring a seamless user experience.

4.4.3. Data Flow Diagrams (DFD)

4.4.3.1. Level 0: Context Diagram

The context diagram provides a high-level view of the system, illustrating the interaction between the user and the Scientific AI Summarizer.

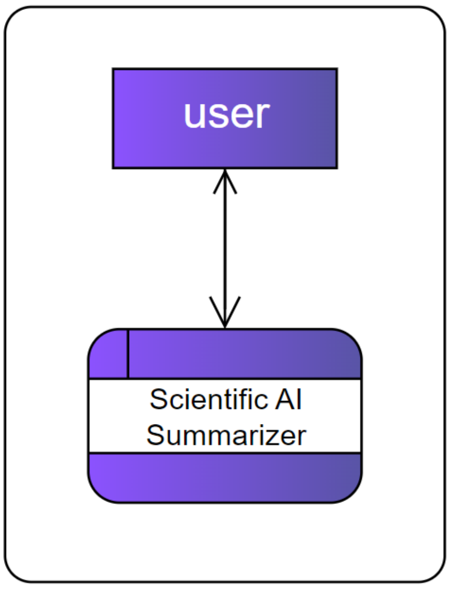


Figure 21: Level 0 Context Diagram

This figure depicts the user interacting with the Scientific AI Summarizer. The user provides input (text or file) to the system, which then processes the input and generates a summary as output.

4.4.3.2. Level 1: Top-Level Data Flow Diagram

This top-level diagram breaks down the main components of the system into four processes: Input Handling, Text Processing, Summarization, and Output Handling.

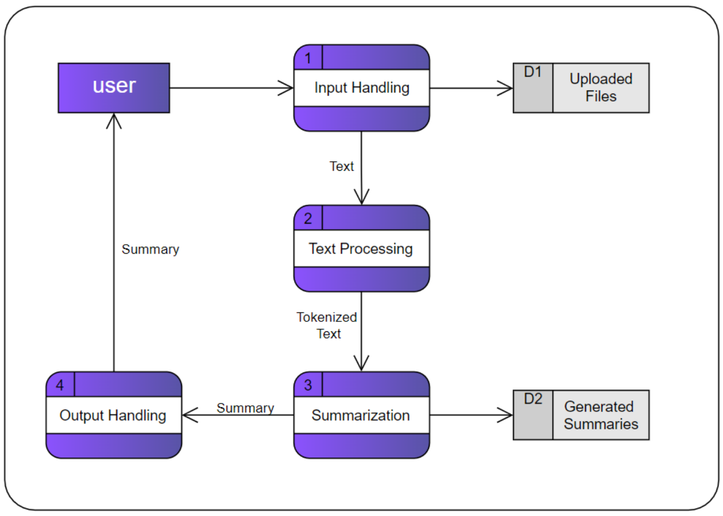


Figure 22: Level 1 Data Flow Diagram

This figure shows the data flow between the main processes of the system. The user interacts with the Input Handling process, which manages text input and file uploads. The Text Processing process prepares the input for summarization. The Summarization process generates the summary, which is then managed by the Output Handling process for display, copying, or downloading.

4.4.3.3. Level 2: Input Handling

This diagram details the processes involved in handling user input, either through pasting text or uploading files.

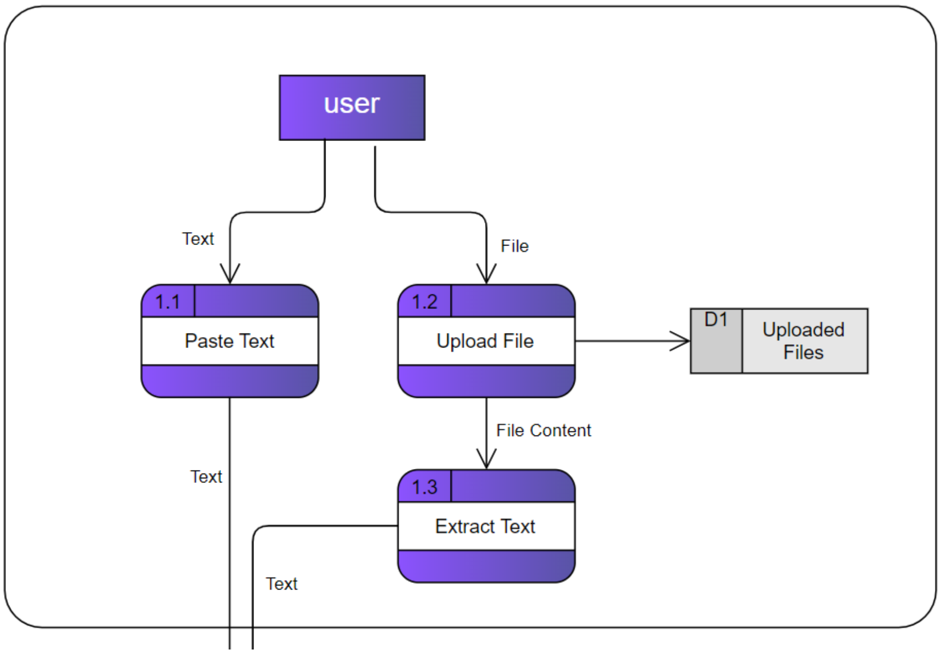


Figure 23: Level 2 Input Handling

This figure breaks down the Input Handling process into three sub-processes: Paste Text, Upload File, and Extract Text. The user can either paste text directly or upload a file. If a file is uploaded, the system extracts text from it.

Level 2: Tokenization

This diagram focuses on the Tokenization process, detailing how the system tokenizes the text for summarization.

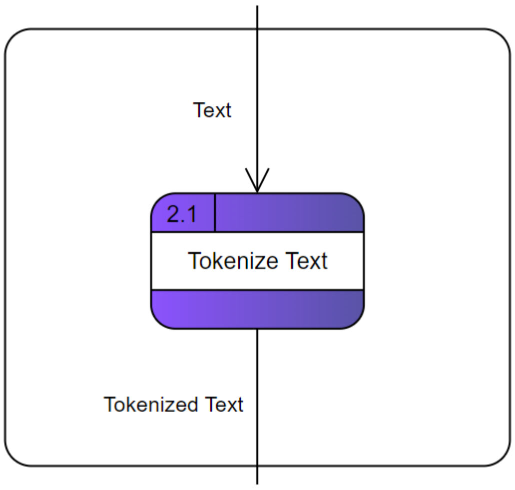


Figure 24: Level 2 Tokenization

This figure illustrates the Tokenization process, where the extracted text is tokenized for further processing in the Summarization process.

Level 2: Summarization

This diagram focuses on the Summarization process, detailing how the system generates summaries from the processed text.

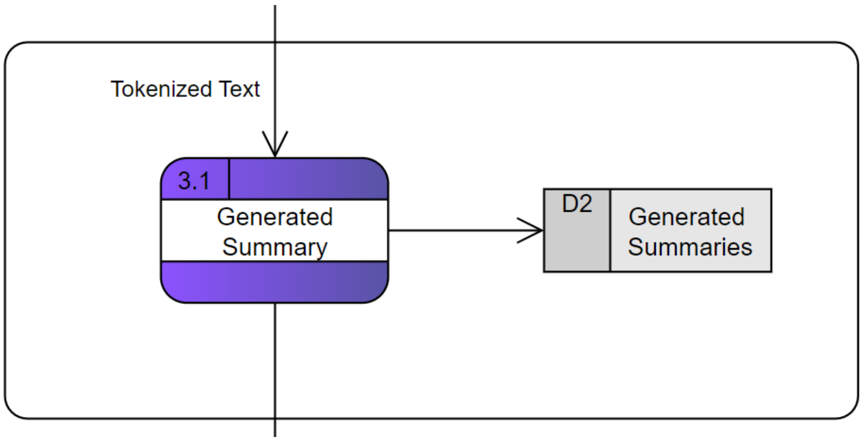


Figure 25:Level 2 Summarization

This figure illustrates the Summarization process, where tokenized text is processed to generate a summary. The generated summaries are stored in the Generated Summaries data store.

Level 2: Output Handling

This diagram details the processes involved in handling the output summary, including displaying, copying, and downloading the summary.

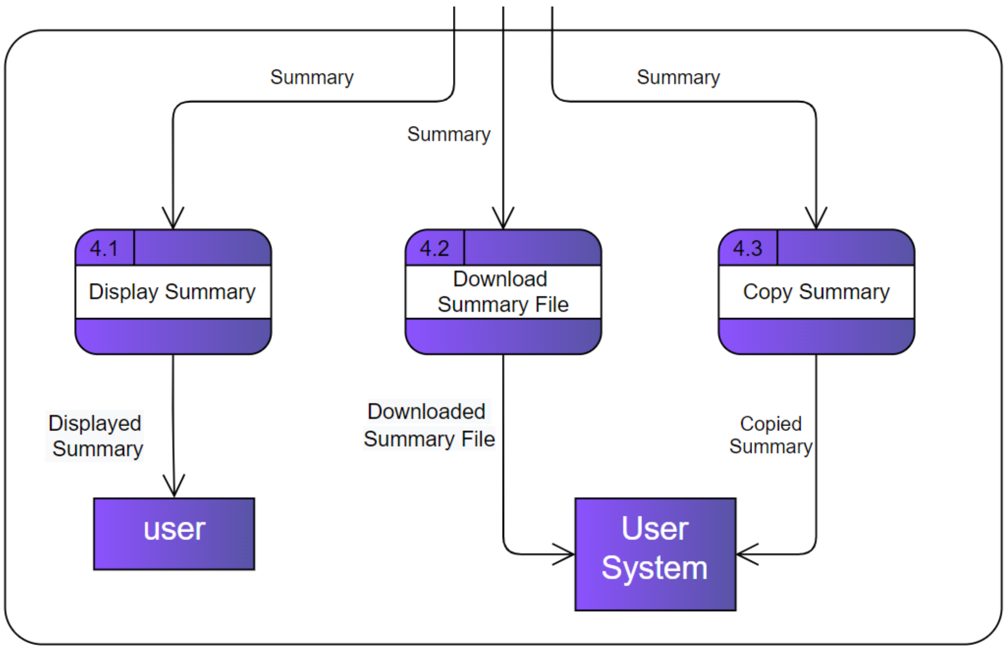


Figure 26: Level 2 Output Handling

This figure shows the sub-processes of Output Handling: Display Summary, Download Summary File, and Copy Summary. The summary can be displayed to the user, copied to the clipboard, or downloaded as a text file.

5. Implementation Plan

The implementation of the model for the Scientific AI Summarizer involves a series of steps starting from data collection to model evaluation. This chapter outlines the detailed process of implementing the model, including setting up the environment, collecting and preprocessing data, training the model, and evaluating its performance.

5.1. Environment Setup

To ensure a robust and efficient development environment, several libraries and tools were installed. The training was conducted on Google Colab Pro, utilizing an A100 GPU with the following specifications:

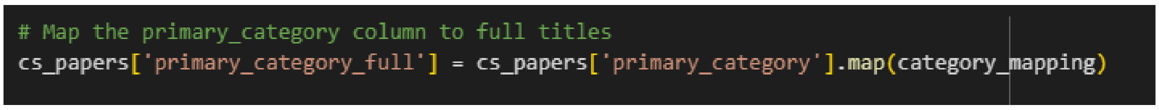
- RAM: 84 GB
- GPU RAM: 40.0 GB
- Disk Space: 100 GB
- The necessary libraries for this project included:
    - Hugging Face Libraries: transformers, datasets, accelerate, and evaluate
    - PEFT: peft library for parameter-efficient fine-tuning
    - Additional Dependencies: bitsandbytes, rouge-score, tensorboard, and py7zr

5.2. Data Collection

The data collection process involved fetching full-text papers from the arXiv API based on their metadata. Here is a detailed description of the process:

5.2.1. Loading the Dataset:

- The initial dataset, cs_papers_api.csv, containing columns such as paper_id, title, abstract, year, primary_category, and categories, was loaded into a pandas DataFrame.
- A mapping dictionary was created to map primary category codes to their full titles, which were then added to the DataFrame.

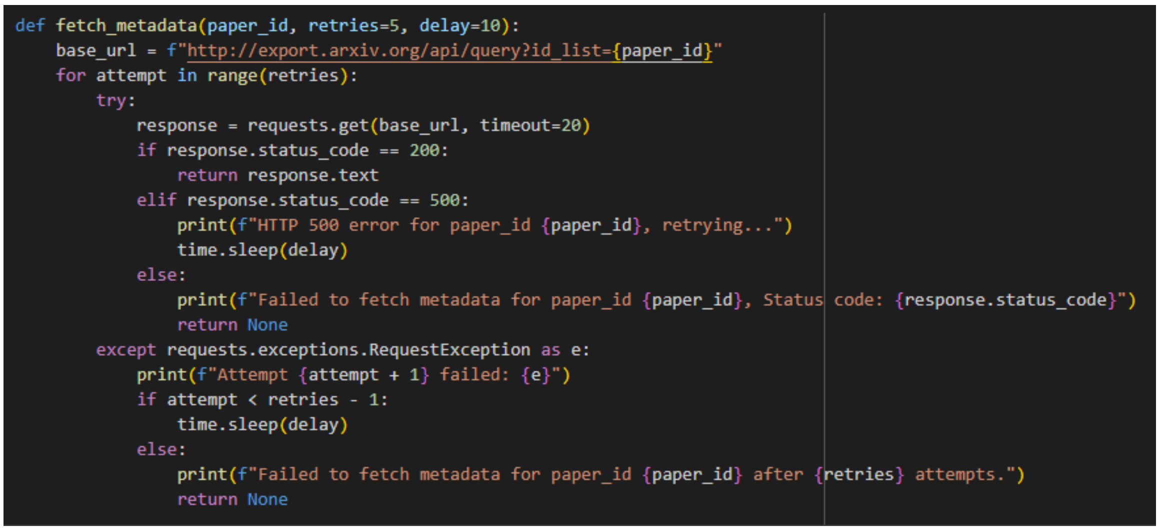


```
# Map the primary_category column to full titles
cs_papers['primary_category_full'] = cs_papers['primary_category'].map(category_mapping)
```

Figure 27: Loading the Dataset

5.2.2. Fetching Metadata:

The fetch_metadata function was implemented to fetch metadata for each paper using its paper ID from the arXiv API. The function included retry logic to handle intermittent failures and delays.

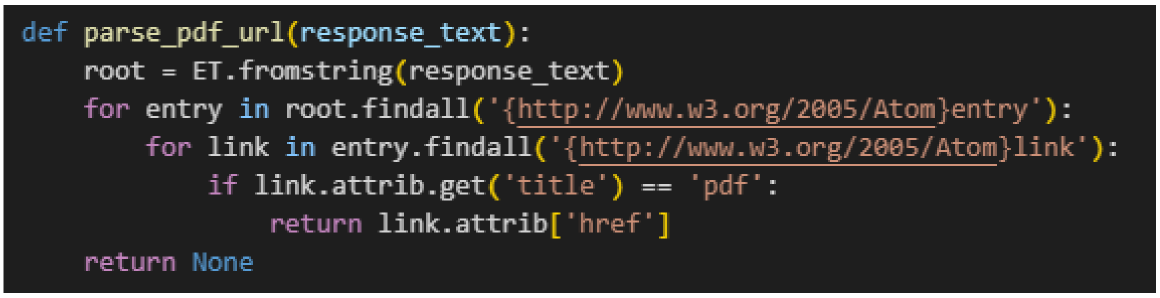


```python
def fetch_metadata(paper_id, retries=5, delay=10):
    base_url = f"http://export.arxiv.org/api/query?id_list={paper_id}"
    for attempt in range(retries):
        try:
            response = requests.get(base_url, timeout=20)
            if response.status_code == 200:
                return response.text
            elif response.status_code == 500:
                print(f"HTTP 500 error for paper_id {paper_id}, retrying...")
                time.sleep(delay)
            else:
                print(f"Failed to fetch metadata for paper_id {paper_id}, Status code: {response.status_code}")
                return None
        except requests.exceptions.RequestException as e:
            print(f"Attempt {attempt + 1} failed: {e}")
            if attempt < retries - 1:
                time.sleep(delay)
            else:
                print(f"Failed to fetch metadata for paper_id {paper_id} after {retries} attempts.")
                return None
```

Figure 28: Fetching Metadata

5.2.3. Parsing PDF URLs:

The parse_pdf_url function parsed the API response to extract the PDF URL of each paper.

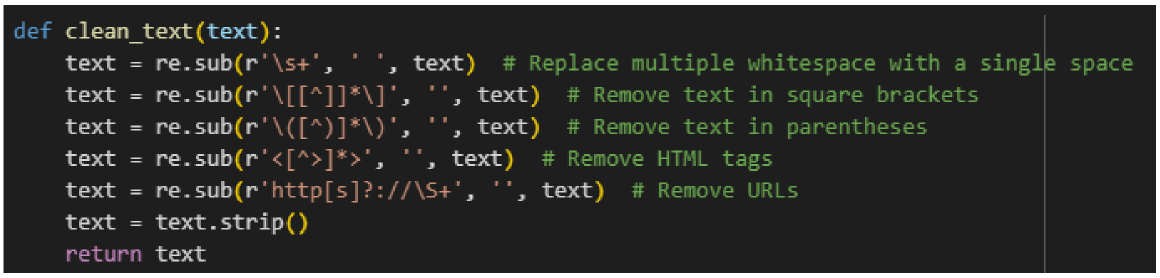


```python
def parse_pdf_url(response_text):
    root = ET.fromstring(response_text)
    for entry in root.findall('{http://www.w3.org/2005/Atom}entry'):
        for link in entry.findall('{http://www.w3.org/2005/Atom}link'):
            if link.attrib.get('title') == 'pdf':
                return link.attrib['href']
    return None
```

Figure 29: Parsing PDF URLs

5.2.4. Cleaning Text:

The clean_text function cleaned the extracted text by removing extra whitespace, text in square brackets, text in parentheses, HTML tags, and URLs.

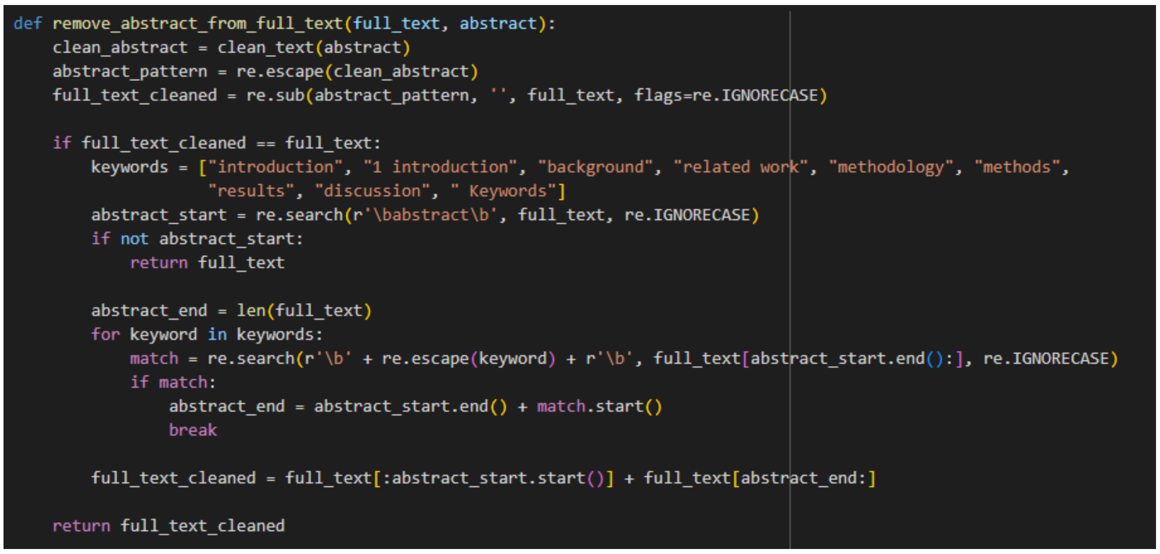


```python
def clean_text(text):
    text = re.sub(r'\s+', ' ', text)  # Replace multiple whitespace with a single space
    text = re.sub(r'\[[^]]*\]', '', text)  # Remove text in square brackets
    text = re.sub(r'\([^)]*\)', '', text)  # Remove text in parentheses
    text = re.sub(r'<[^>]*>', '', text)  # Remove HTML tags
    text = re.sub(r'http[s]?://\S+', '', text)  # Remove URLs
    text = text.strip()
    return text
```

Figure 30: Cleaning Text

5.2.5. Removing Abstracts from Full Text:

The remove_abstract_from_full_text function removed the abstract section from the full text to avoid redundancy in the summarization process.

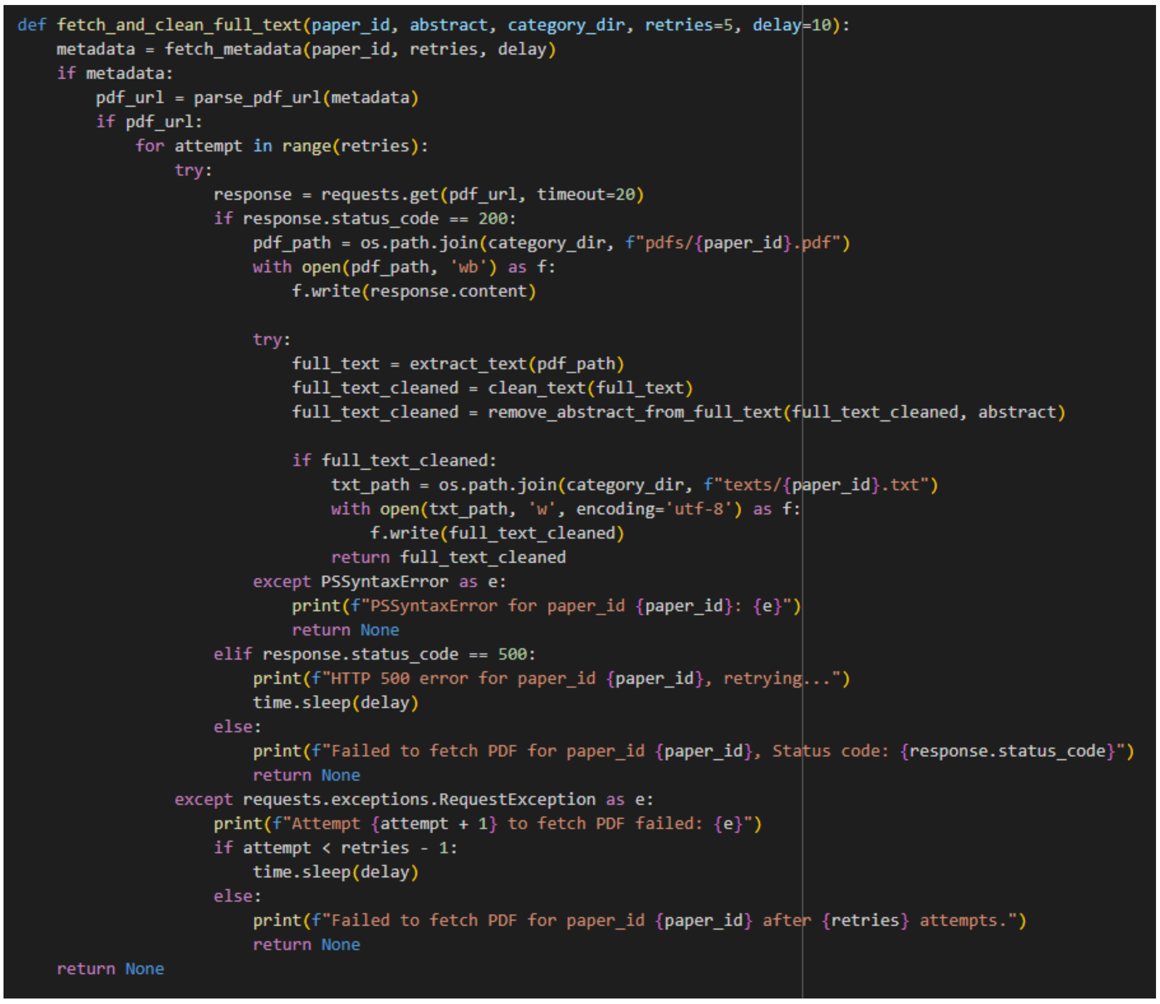


```python
def remove_abstract_from_full_text(full_text, abstract):
    clean_abstract = clean_text(abstract)
    abstract_pattern = re.escape(clean_abstract)
    full_text_cleaned = re.sub(abstract_pattern, '', full_text, flags=re.IGNORECASE)

    if full_text_cleaned == full_text:
        keywords = ["introduction", "1 introduction", "background", "related work", "methodology", "methods",
                    "results", "discussion", " Keywords"]
        abstract_start = re.search(r'\babstract\b', full_text, re.IGNORECASE)
        if not abstract_start:
            return full_text

        abstract_end = len(full_text)
        for keyword in keywords:
            match = re.search(r'\b' + re.escape(keyword) + r'\b', full_text[abstract_start.end():], re.IGNORECASE)
            if match:
                abstract_end = abstract_start.end() + match.start()
                break

        full_text_cleaned = full_text[:abstract_start.start()] + full_text[abstract_end:]

    return full_text_cleaned
```

Figure 31: Removing Abstracts from Full Text

5.2.6. Fetching and Cleaning Full Text:

The fetch_and_clean_full_text function combined the above functions to fetch the PDF, extract its text, clean it, and save the cleaned text to a file. It handled various exceptions and included retry logic.

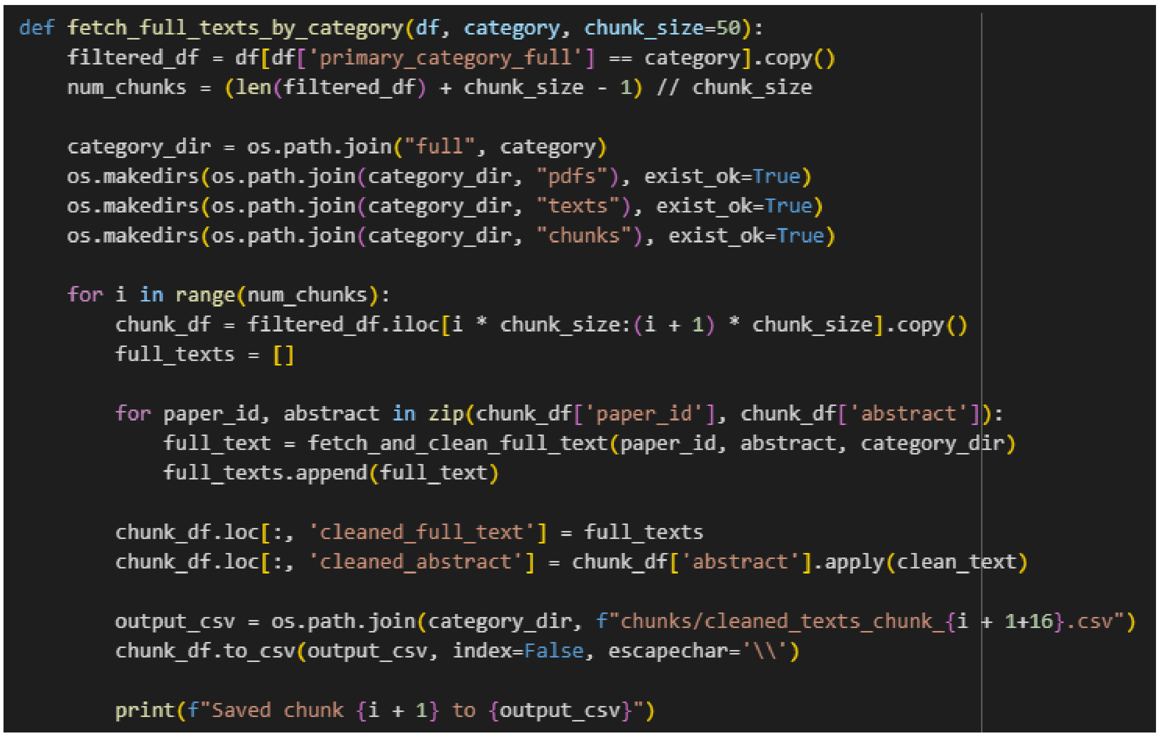


```python
def fetch_and_clean_full_text(paper_id, abstract, category_dir, retries=5, delay=10):
    metadata = fetch_metadata(paper_id, retries, delay)
    if metadata:
        pdf_url = parse_pdf_url(metadata)
        if pdf_url:
            for attempt in range(retries):
                try:
                    response = requests.get(pdf_url, timeout=20)
                    if response.status_code == 200:
                        pdf_path = os.path.join(category_dir, f"pdfs/{paper_id}.pdf")
                        with open(pdf_path, 'wb') as f:
                            f.write(response.content)

                        try:
                            full_text = extract_text(pdf_path)
                            full_text_cleaned = clean_text(full_text)
                            full_text_cleaned = remove_abstract_from_full_text(full_text_cleaned, abstract)

                            if full_text_cleaned:
                                txt_path = os.path.join(category_dir, f"texts/{paper_id}.txt")
                                with open(txt_path, 'w', encoding='utf-8') as f:
                                    f.write(full_text_cleaned)
                                return full_text_cleaned
                        except PSSyntaxError as e:
                            print(f"PSSyntaxError for paper_id {paper_id}: {e}")
                            return None
                    elif response.status_code == 500:
                        print(f"HTTP 500 error for paper_id {paper_id}, retrying...")
                        time.sleep(delay)
                    else:
                        print(f"Failed to fetch PDF for paper_id {paper_id}, Status code: {response.status_code}")
                        return None
                except requests.exceptions.RequestException as e:
                    print(f"Attempt {attempt + 1} to fetch PDF failed: {e}")
                    if attempt < retries - 1:
                        time.sleep(delay)
                    else:
                        print(f"Failed to fetch PDF for paper_id {paper_id} after {retries} attempts.")
                        return None
    return None
```

Figure 32: Fetching and Cleaning Full Text

5.2.7. Fetching Full Texts by Category:

The fetch_full_texts_by_category function iterated through the dataset by category, fetched and cleaned full texts, and saved them into chunks of 50 papers each. It also created necessary directories for storing PDFs and cleaned texts.



```python
def fetch_full_texts_by_category(df, category, chunk_size=50):
    filtered_df = df[df['primary_category_full'] == category].copy()
    num_chunks = (len(filtered_df) + chunk_size - 1) // chunk_size

    category_dir = os.path.join("full", category)
    os.makedirs(os.path.join(category_dir, "pdfs"), exist_ok=True)
    os.makedirs(os.path.join(category_dir, "texts"), exist_ok=True)
    os.makedirs(os.path.join(category_dir, "chunks"), exist_ok=True)

    for i in range(num_chunks):
        chunk_df = filtered_df.iloc[i * chunk_size:(i + 1) * chunk_size].copy()
        full_texts = []

        for paper_id, abstract in zip(chunk_df['paper_id'], chunk_df['abstract']):
            full_text = fetch_and_clean_full_text(paper_id, abstract, category_dir)
            full_texts.append(full_text)

        chunk_df.loc[:, 'cleaned_full_text'] = full_texts
        chunk_df.loc[:, 'cleaned_abstract'] = chunk_df['abstract'].apply(clean_text)

        output_csv = os.path.join(category_dir, f"chunks/cleaned_texts_chunk_{i + 1+16}.csv")
        chunk_df.to_csv(output_csv, index=False, escapechar='\\')

        print(f"Saved chunk {i + 1} to {output_csv}")
```

Figure 33: Fetching Full Texts by Category

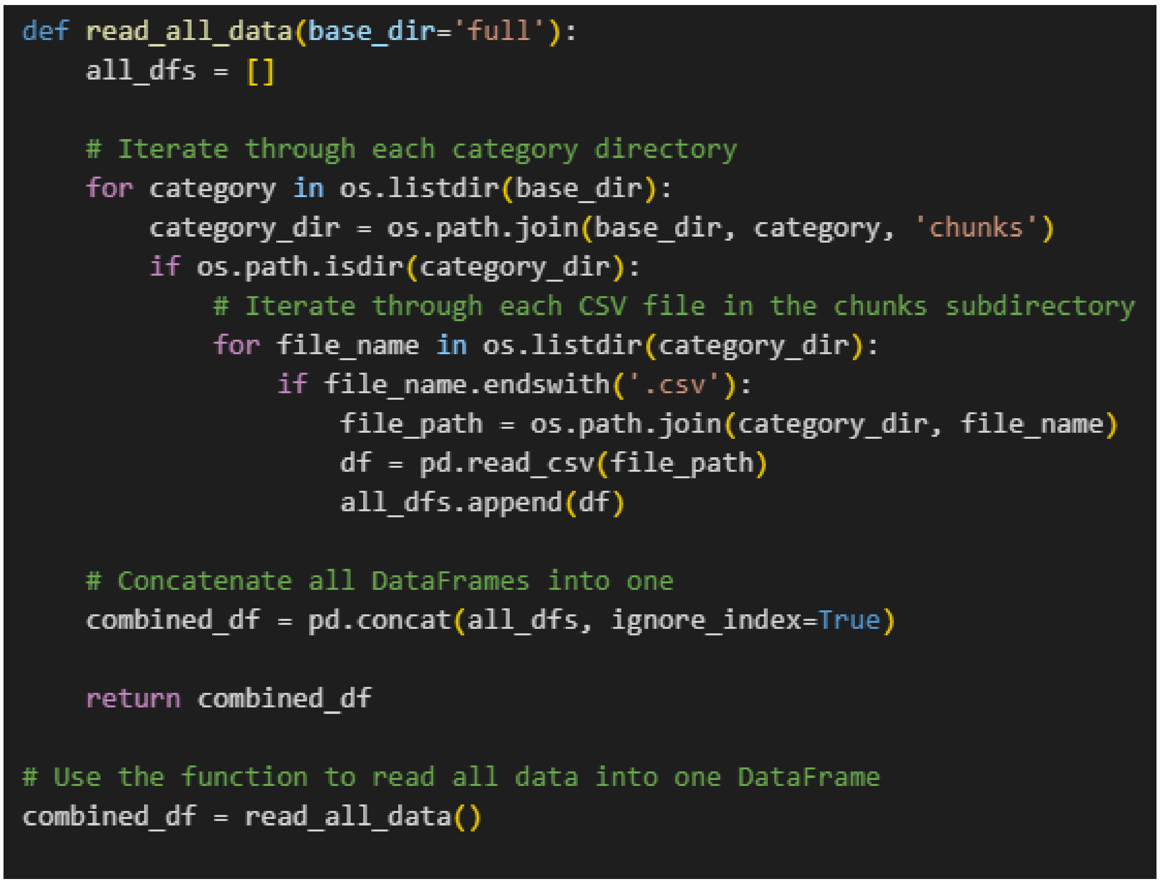
The collected data was then saved into separate CSV files for further processing.

5.3. Data Preparation

The data preparation phase involved combining and cleaning the collected datasets to create a final dataset suitable for model training:

5.3.1. Reading All Data:

The read_all_data function read all CSV files from the full directory, combined them into a single DataFrame, and returned it.



```python
def read_all_data(base_dir='full'):
    all_dfs = []

    # Iterate through each category directory
    for category in os.listdir(base_dir):
        category_dir = os.path.join(base_dir, category, 'chunks')
        if os.path.isdir(category_dir):
            # Iterate through each CSV file in the chunks subdirectory
            for file_name in os.listdir(category_dir):
                if file_name.endswith('.csv'):
                    file_path = os.path.join(category_dir, file_name)
                    df = pd.read_csv(file_path)
                    all_dfs.append(df)

    # Concatenate all DataFrames into one
    combined_df = pd.concat(all_dfs, ignore_index=True)

    return combined_df

# Use the function to read all data into one DataFrame
combined_df = read_all_data()
```

Figure 34: Reading All Data

5.3.2. Checking and Cleaning Data:

The combined DataFrame was checked for missing values and necessary columns. Missing values were handled, and columns were cleaned as needed.

5.3.3. Splitting Data:

The cleaned DataFrame was split into training and testing sets, with 85% used for training and 15% for testing. The split was stored as a DatasetDict for easy access.

5.3.4. Saving Data:

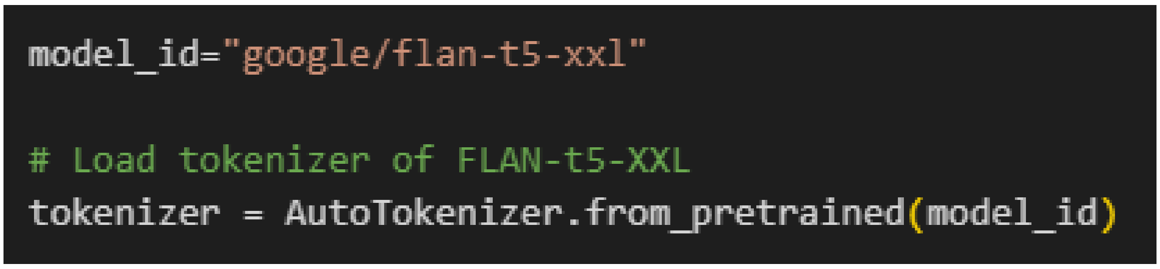
The final combined dataset was saved to disk for easy loading in subsequent steps.

5.4. Model Training

Model training was a critical phase involving tokenization, model preparation, and fine-tuning using LoRA (Low-Rank Adaptation):

5.4.1. Tokenization:

The AutoTokenizer from Hugging Face's transformers library was used to tokenize the full text and abstracts. The token lengths were calculated, and maximum lengths were set based on the 85th and 90th percentiles.

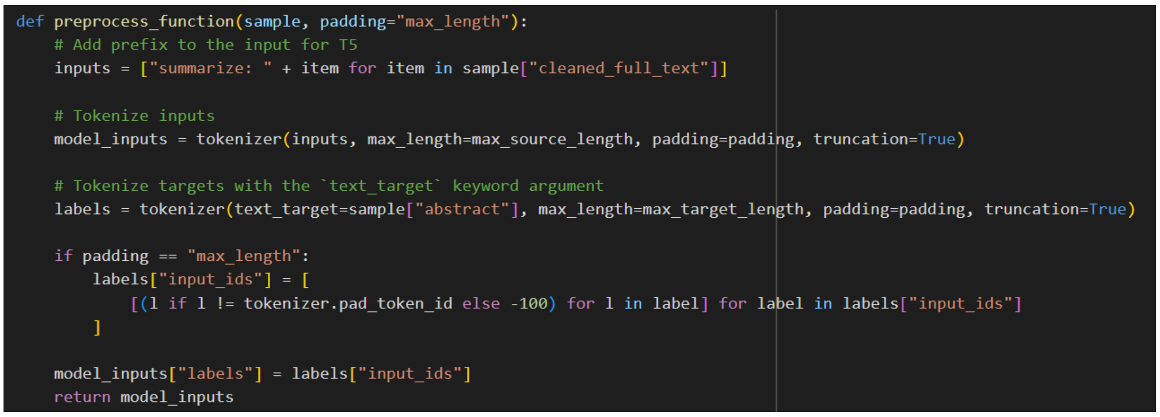


```python
model_id="google/flan-t5-xxl"

# Load tokenizer of FLAN-t5-XXL
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

Figure 35: Tokenization

5.4.2. Preprocessing:

The preprocess_function was defined to handle the tokenization process for both inputs and targets. It added a prefix to the input for T5, tokenized the inputs and targets, and handled padding appropriately. The function replaced padding token IDs with -100 to ignore padding in the loss calculation.

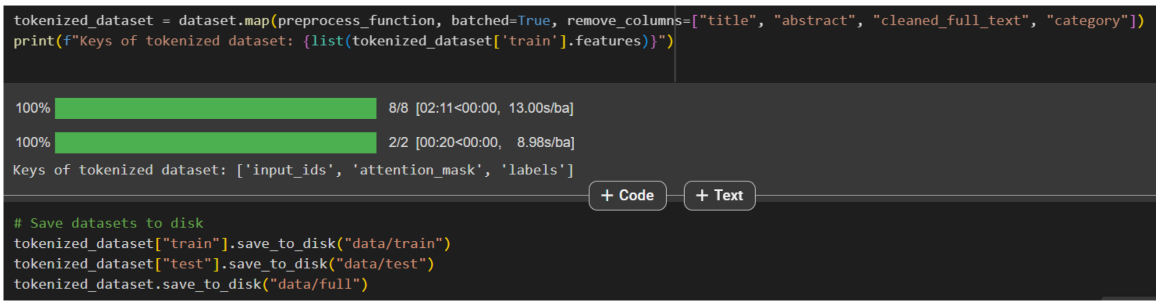


```python
def preprocess_function(sample, padding="max_length"):
    # Add prefix to the input for T5
    inputs = ["summarize: " + item for item in sample["cleaned_full_text"]]

    # Tokenize inputs
    model_inputs = tokenizer(inputs, max_length=max_source_length, padding=padding, truncation=True)

    # Tokenize targets with the `text_target` keyword argument
    labels = tokenizer(text_target=sample["abstract"], max_length=max_target_length, padding=padding, truncation=True)

    if padding == "max_length":
        labels["input_ids"] = [
            [(l if l != tokenizer.pad_token_id else -100) for l in label] for label in labels["input_ids"]
        ]

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs
```

Figure 36: Preprocessing Function

5.4.3. Loading and Tokenizing Data:

The dataset was loaded, tokenized using the preprocessing function, and saved to disk for later use.

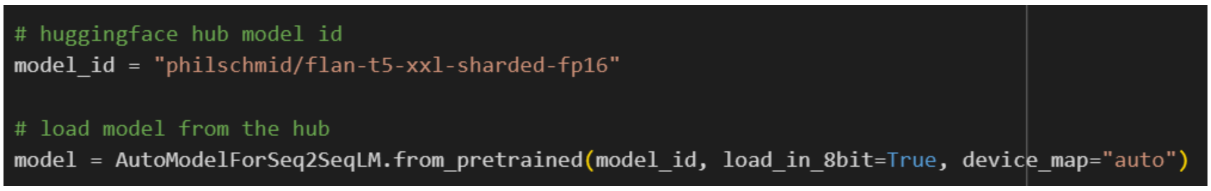


```
tokenized_dataset = dataset.map(preprocess_function, batched=True, remove_columns=["title", "abstract", "cleaned_full_text", "category"])
print(f"Keys of tokenized dataset: {list(tokenized_dataset['train'].features)}")

100% 8/8 [02:11<00:00, 13.00s/ba]
100% 2/2 [00:20<00:00,  8.98s/ba]
Keys of tokenized dataset: ['input_ids', 'attention_mask', 'labels']

# Save datasets to disk
tokenized_dataset["train"].save_to_disk("data/train")
tokenized_dataset["test"].save_to_disk("data/test")
tokenized_dataset.save_to_disk("data/full")
```

Figure 37: Loading and Tokenizing Data

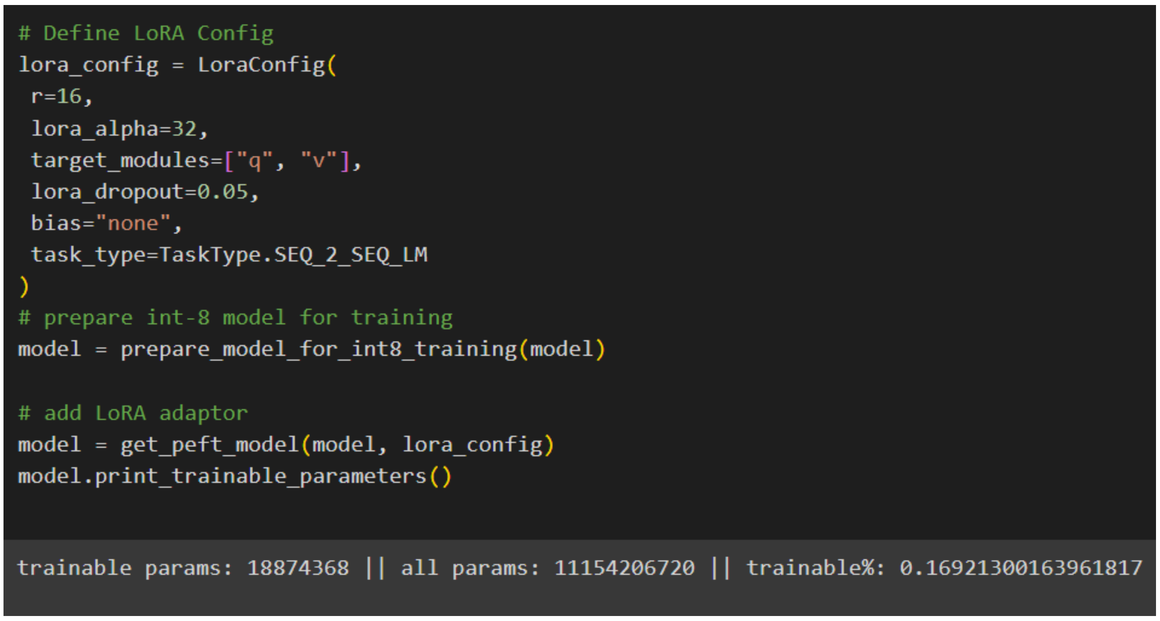
5.4.4. Model Configuration:

The google/flan-t5-xxl model was loaded and prepared for int-8 training. LoRA configuration was set up with specific parameters for fine-tuning.



```
# huggingface hub model id
model_id = "philschmid/flan-t5-xxl-sharded-fp16"

# load model from the hub
model = AutoModelForSeq2SeqLM.from_pretrained(model_id, load_in_8bit=True, device_map="auto")
```

Figure 38: Model Loading



```
# Define LoRA Config
lora_config = LoraConfig(
 r=16,
 lora_alpha=32,
 target_modules=["q", "v"],
 lora_dropout=0.05,
 bias="none",
 task_type=TaskType.SEQ_2_SEQ_LM
)
# prepare int-8 model for training
model = prepare_model_for_int8_training(model)

# add LoRA adaptor
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()

trainable params: 18874368 || all params: 11154206720 || trainable%: 0.16921300163961817
```

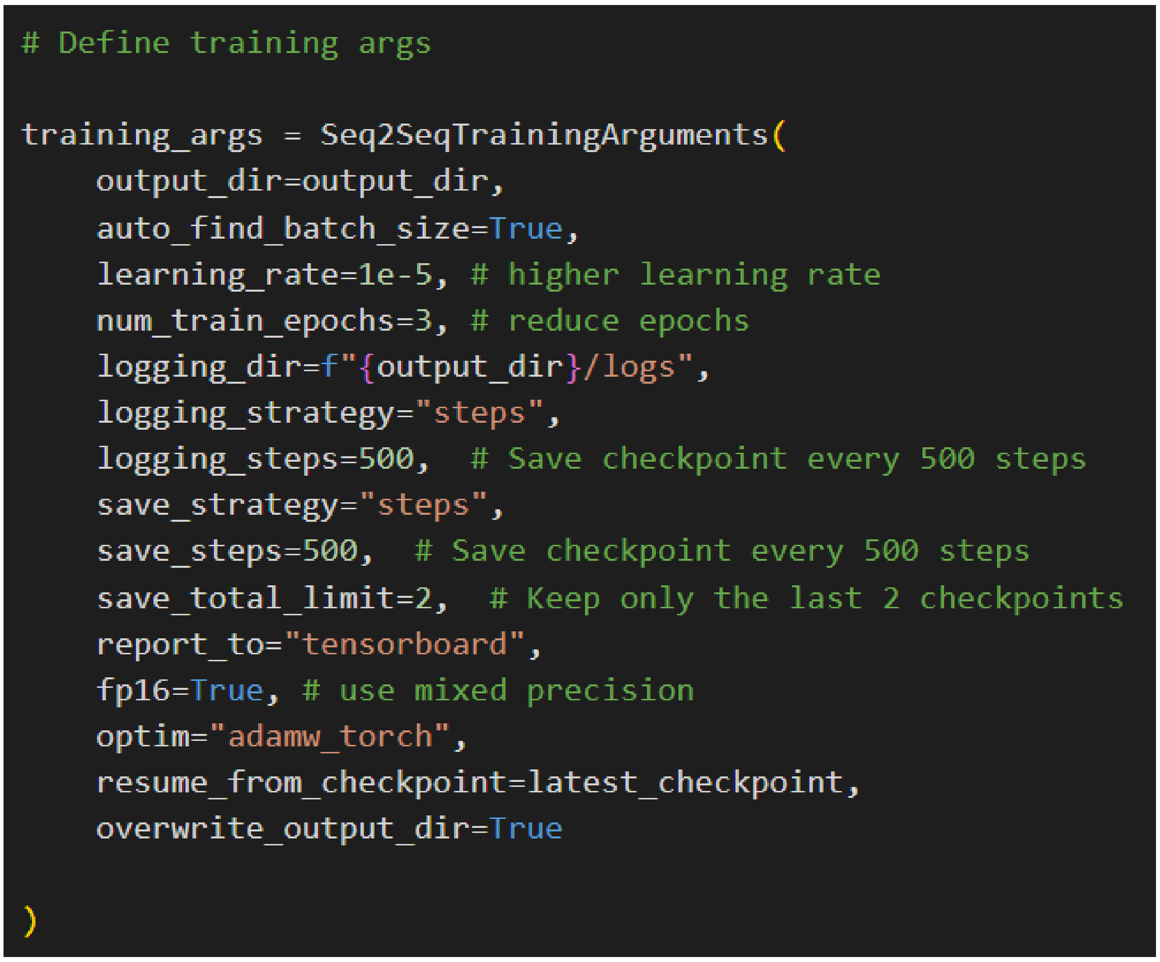
Figure 39: LoRA Configuration and setup

5.4.5. Training Configuration:

The training configuration was set using the Seq2SeqTrainingArguments class. Key parameters such as learning rate, number of epochs, batch size, logging strategy, and mixed precision training were defined to optimize the training process.

5.4.5.1. Define Training Arguments:

Parameters for training were set, including batch size, learning rate, and number of epochs.

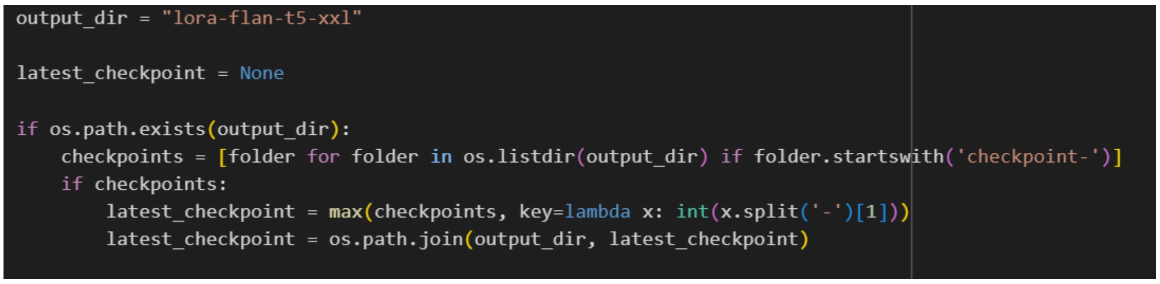


```
# Define training args

training_args = Seq2SeqTrainingArguments(
    output_dir=output_dir,
    auto_find_batch_size=True,
    learning_rate=1e-5, # higher learning rate
    num_train_epochs=3, # reduce epochs
    logging_dir=f"{output_dir}/logs",
    logging_strategy="steps",
    logging_steps=500,  # Save checkpoint every 500 steps
    save_strategy="steps",
    save_steps=500,  # Save checkpoint every 500 steps
    save_total_limit=2,  # Keep only the last 2 checkpoints
    report_to="tensorboard",
    fp16=True, # use mixed precision
    optim="adamw_torch",
    resume_from_checkpoint=latest_checkpoint,
    overwrite_output_dir=True

)
```

Figure 40: Define Training Arguments

5.4.5.2. Set Logging and Checkpointing:

Logging and checkpointing strategies were implemented to monitor training progress and save model checkpoints.



```python
output_dir = "lora-flan-t5-xxl"

latest_checkpoint = None

if os.path.exists(output_dir):
    checkpoints = [folder for folder in os.listdir(output_dir) if folder.startswith('checkpoint-')]
    if checkpoints:
        latest_checkpoint = max(checkpoints, key=lambda x: int(x.split('-')[1]))
        latest_checkpoint = os.path.join(output_dir, latest_checkpoint)
```

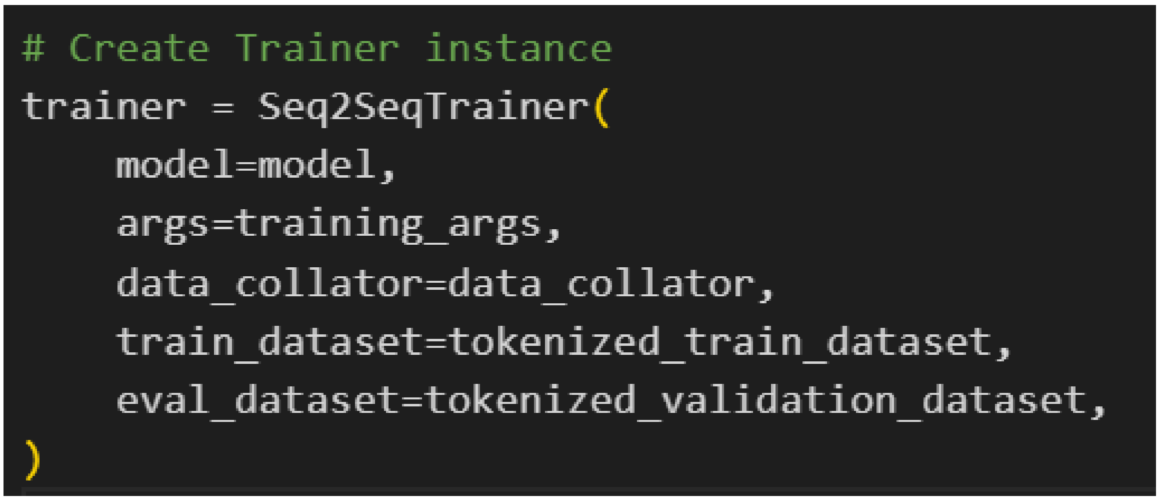
Figure 41: Set Logging and Checkpointing

5.4.6. Fine-Tuning the Model

The Seq2SeqTrainer class was used to fine-tune the model. The trainer was configured with the prepared model, training arguments, data collator, and datasets. The training process involved running multiple epochs, with regular logging and checkpointing to track progress.

5.4.6.1. Initialize Trainer:

The trainer was initialized with the model, training arguments, and data collator.

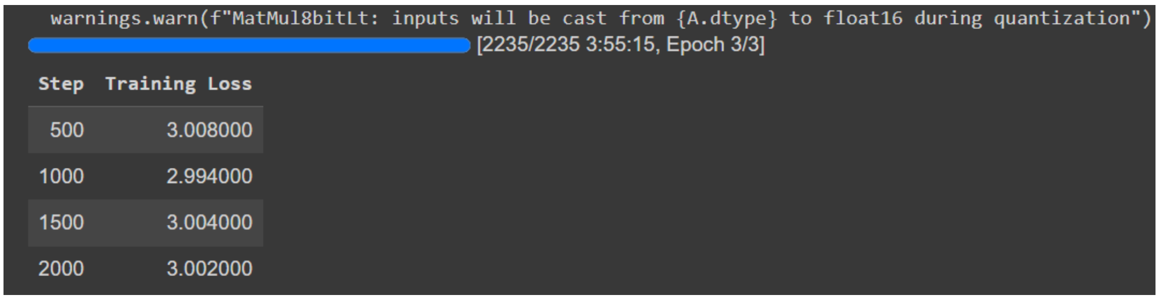


```python
# Create Trainer instance
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=tokenized_train_dataset,
    eval_dataset=tokenized_validation_dataset,
)
```

Figure 42: Initialize Trainer

5.4.6.2. Run Training:

The training process was executed, with the model fine-tuning over several epochs.



```
  warnings.warn(f"MatMul8bitLt: inputs will be cast from {A.dtype} to float16 during quantization")
                                        [2235/2235 3:55:15, Epoch 3/3]
Step  Training Loss
 500       3.008000
1000       2.994000
1500       3.004000
2000       3.002000
```

Figure 43: Model training

5.4.7. Saving the Final Model:

After training, the final model, its configuration, LoRA adapter, and tokenizer were saved to disk.

5.5. Model Evaluation

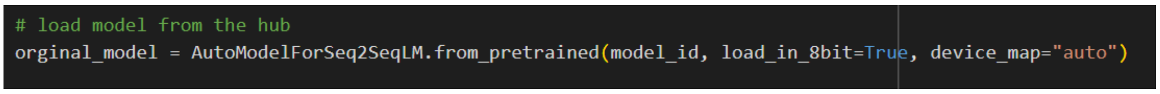
Model evaluation is a critical phase in the implementation process to measure the performance of the trained model and identify areas for further optimization. The evaluation process for the Scientific AI Summarizer involved loading the trained model, testing it on the prepared dataset, and analyzing its performance using various metrics and visualizations.

5.5.1. Loading the Model and Tokenizer

The first step in the evaluation process was to load the trained model and tokenizer from disk. The AutoModelForSeq2SeqLM from Hugging Face's transformers library was used to load the model, and the AutoTokenizer was used to load the tokenizer. Additionally, the LoRA adapter was added to the model to enable parameter-efficient fine-tuning. The following snippet provides an overview of how the models and tokenizers were loaded and prepared:

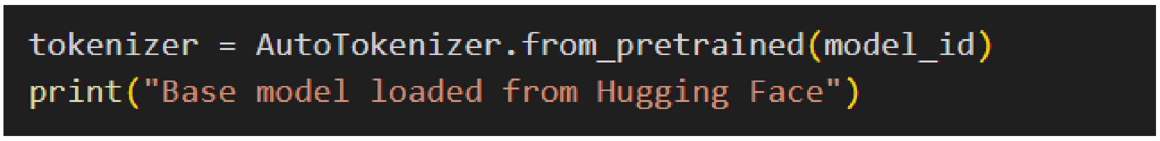
5.5.1.1. Loading the Base Model and Tokenizer:

The base model and tokenizer are loaded from the Hugging Face model hub.



```
# load model from the hub
orginal_model = AutoModelForSeq2SeqLM.from_pretrained(model_id, load_in_8bit=True, device_map="auto")
```

Figure 44: Loading the Base Model

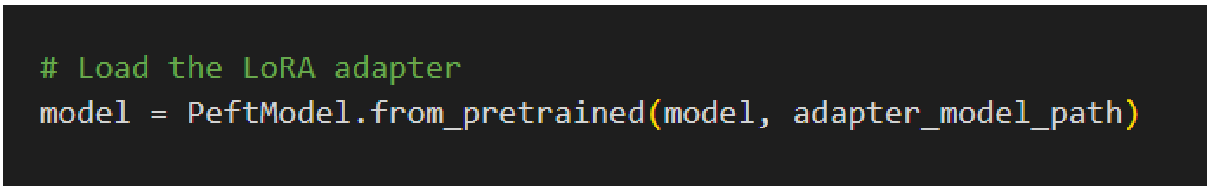


```
tokenizer = AutoTokenizer.from_pretrained(model_id)
print("Base model loaded from Hugging Face")
```

Figure 45: Loading Tokenizer

5.5.1.2. Loading the LoRA Adapter:

The LoRA adapter, which allows for parameter-efficient fine-tuning, is loaded and applied to the base model.



```
# Load the LoRA adapter
model = PeftModel.from_pretrained(model, adapter_model_path)
```

Figure 46: Loading the LoRA Adapter

5.5.1.3. Setting Model to Evaluation Mode:

The model is set to evaluation mode to ensure it is ready for inference.

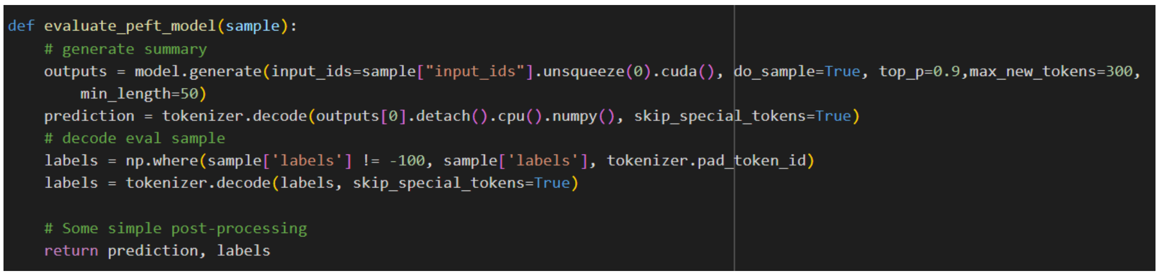
5.5.2. Loading the Test Dataset

The test dataset, saved earlier during the data preparation phase, was loaded using Hugging Face's load_from_disk function. This dataset was formatted for PyTorch to ensure compatibility with the evaluation process. The dataset was split into a smaller test subset to speed up the evaluation process while maintaining a representative sample.

5.5.3. Evaluating the Model

The evaluation function was defined to generate summaries for each sample in the test dataset. The function used the model to generate outputs based on the input text and compared the generated summaries with the reference abstracts. The evaluate_peft_model function handled the generation of summaries and the comparison with reference texts.

- **Generate Summaries:** The function generates summaries for each input text using the fine-tuned model.
- **Decode Predictions and Labels:** The generated summaries (predictions) and reference abstracts (labels) are decoded for comparison.
- **Post-Processing:** Simple post-processing is applied to clean up the predictions and labels.



```
def evaluate_peft_model(sample):
    # generate summary
    outputs = model.generate(input_ids=sample["input_ids"].unsqueeze(0).cuda(), do_sample=True, top_p=0.9,max_new_tokens=300,
        min_length=50)
    prediction = tokenizer.decode(outputs[0].detach().cpu().numpy(), skip_special_tokens=True)
    # decode eval sample
    labels = np.where(sample['labels'] != -100, sample['labels'], tokenizer.pad_token_id)
    labels = tokenizer.decode(labels, skip_special_tokens=True)

    # Some simple post-processing
    return prediction, labels
```

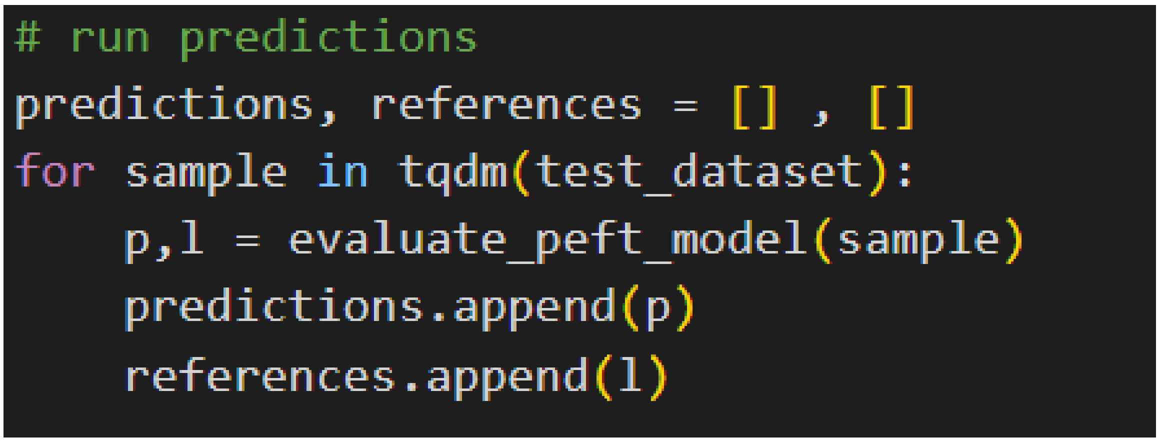
Figure 47: Evaluate Peft Model

5.5.4. Running the Evaluation

The model was evaluated using different configurations of maximum and minimum token lengths to determine the best settings. The evaluation involved generating summaries for the test samples and calculating the ROUGE scores for each configuration. This process provided insights into the model's summarization capabilities and helped identify the optimal settings.

5.5.4.1. Run Predictions:

The model generates predictions for the test dataset.

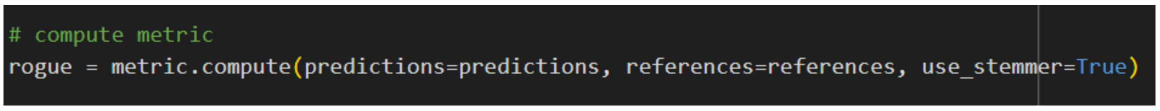


```
# run predictions
predictions, references = [] , []
for sample in tqdm(test_dataset):
    p,l = evaluate_peft_model(sample)
    predictions.append(p)
    references.append(l)
```

Figure 48: Run Predictions

5.5.4.2. Compute ROUGE Metrics:

The ROUGE metric is used to evaluate the quality of the generated summaries by comparing them with the reference abstracts.

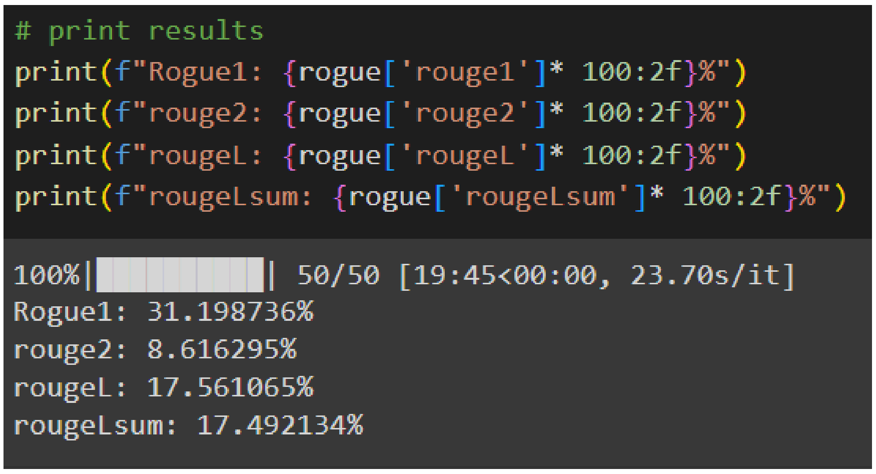


```
# compute metric
rogue = metric.compute(predictions=predictions, references=references, use_stemmer=True)
```

Figure 49: Compute ROUGE

5.5.4.3. Print Results:

The ROUGE scores are printed to provide a quantitative measure of the model's performance.



```
# print results
print(f"Rogue1: {rogue['rouge1']* 100:2f}%")
print(f"rouge2: {rogue['rouge2']* 100:2f}%")
print(f"rougeL: {rogue['rougeL']* 100:2f}%")
print(f"rougeLsum: {rogue['rougeLsum']* 100:2f}%")
```

```
100%|██████████| 50/50 [19:45<00:00, 23.70s/it]
Rogue1: 31.198736%
rouge2: 8.616295%
rougeL: 17.561065%
rougeLsum: 17.492134%
```

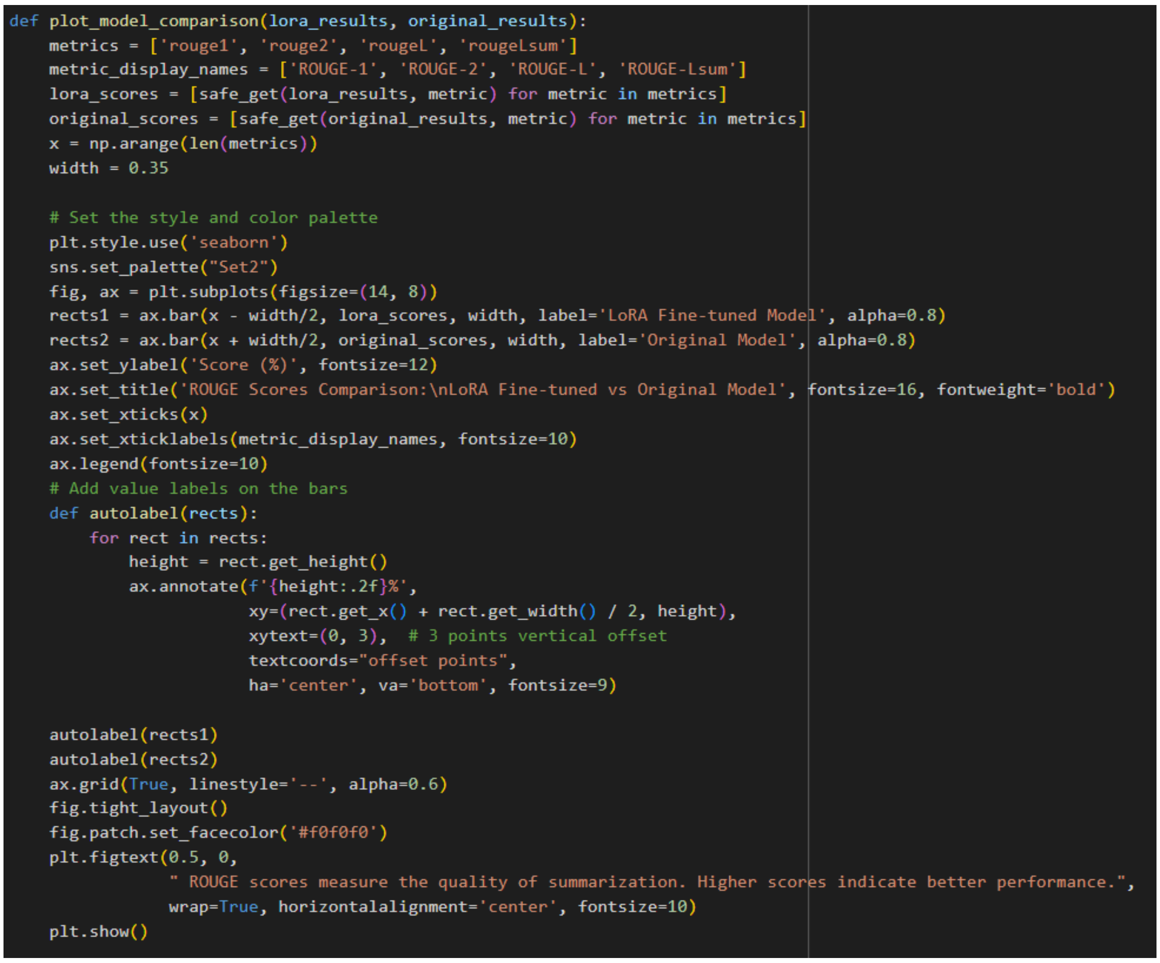
Figure 50: ROUGE Results

5.5.5. Visualizing the Results

The evaluation results were visualized using line plots to show the performance of different configurations. The ROUGE-1, ROUGE-2, ROUGE-L, and ROUGE-Lsum scores were plotted against various configurations of maximum and minimum token lengths, and the fine-tuned model against the base model. This visualization helped in understanding the impact of different configurations on the model's performance.

5.5.5.1. Plot ROUGE Scores:

The ROUGE scores for original model and the fine-tuned model are plotted.

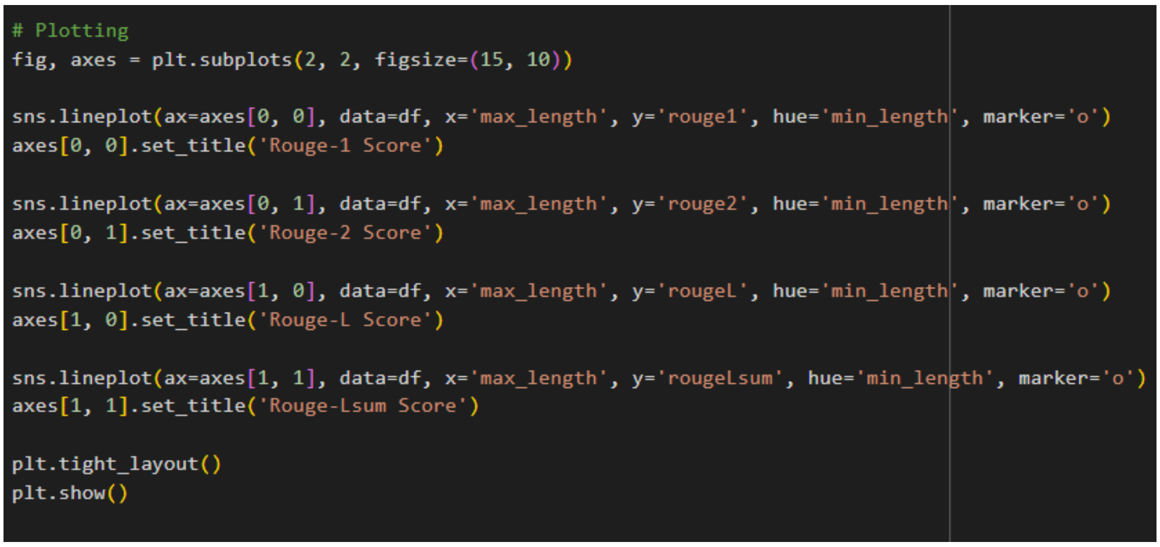


```python
def plot_model_comparison(lora_results, original_results):
    metrics = ['rouge1', 'rouge2', 'rougeL', 'rougeLsum']
    metric_display_names = ['ROUGE-1', 'ROUGE-2', 'ROUGE-L', 'ROUGE-Lsum']
    lora_scores = [safe_get(lora_results, metric) for metric in metrics]
    original_scores = [safe_get(original_results, metric) for metric in metrics]
    x = np.arange(len(metrics))
    width = 0.35

    # Set the style and color palette
    plt.style.use('seaborn')
    sns.set_palette("Set2")
    fig, ax = plt.subplots(figsize=(14, 8))
    rects1 = ax.bar(x - width/2, lora_scores, width, label='LoRA Fine-tuned Model', alpha=0.8)
    rects2 = ax.bar(x + width/2, original_scores, width, label='Original Model', alpha=0.8)
    ax.set_ylabel('Score (%)', fontsize=12)
    ax.set_title('ROUGE Scores Comparison:\nLoRA Fine-tuned vs Original Model', fontsize=16, fontweight='bold')
    ax.set_xticks(x)
    ax.set_xticklabels(metric_display_names, fontsize=10)
    ax.legend(fontsize=10)
    # Add value labels on the bars
    def autolabel(rects):
        for rect in rects:
            height = rect.get_height()
            ax.annotate(f'{height:.2f}%',
                        xy=(rect.get_x() + rect.get_width() / 2, height),
                        xytext=(0, 3),  # 3 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom', fontsize=9)

    autolabel(rects1)
    autolabel(rects2)
    ax.grid(True, linestyle='--', alpha=0.6)
    fig.tight_layout()
    fig.patch.set_facecolor('#f0f0f0')
    plt.figtext(0.5, 0,
                " ROUGE scores measure the quality of summarization. Higher scores indicate better performance.",
                wrap=True, horizontalalignment='center', fontsize=10)
    plt.show()
```

Figure 51: Plot ROUGE Scores function

5.5.5.2. Plot ROUGE Scores:

The ROUGE scores for different configurations are plotted to visualize the performance of the model under various settings.



```python
# Plotting
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

sns.lineplot(ax=axes[0, 0], data=df, x='max_length', y='rouge1', hue='min_length', marker='o')
axes[0, 0].set_title('Rouge-1 Score')

sns.lineplot(ax=axes[0, 1], data=df, x='max_length', y='rouge2', hue='min_length', marker='o')
axes[0, 1].set_title('Rouge-2 Score')

sns.lineplot(ax=axes[1, 0], data=df, x='max_length', y='rougeL', hue='min_length', marker='o')
axes[1, 0].set_title('Rouge-L Score')

sns.lineplot(ax=axes[1, 1], data=df, x='max_length', y='rougeLsum', hue='min_length', marker='o')
axes[1, 1].set_title('Rouge-Lsum Score')

plt.tight_layout()
plt.show()
```

Figure 52: Plot ROUGE Scores

5.5.6. Test Case Evaluation

In parallel to the general evaluation, a test case evaluation was performed on a subset of papers selected for detailed analysis. The evaluation process involved generating summaries for these test cases and comparing them with the reference abstracts to ensure the model's effectiveness in real-world scenarios.

- **Load Test Cases:** Choose a subset of papers for detailed evaluation.
- **Generate Summaries for Test Cases:** The model generates summaries for the test cases.
- **Save Summaries:** The generated summaries are saved alongside the original abstracts for further analysis.

By following these steps, the model was thoroughly evaluated, and its performance was analyzed. The insights gained from this evaluation helped in refining the model and ensuring it met the desired performance criteria for summarizing scientific papers.

5.6. Implementation Plan for the Interface

The implementation plan for the interface involves creating a user-friendly web application that interacts with the fine-tuned model to generate summaries of scientific papers. The interface provides functionalities for users to upload text files or directly input text, and it returns the generated summaries.

5.6.1. Environment Setup

The interface was developed using Flask, a lightweight WSGI web application framework in Python. Flask was chosen for its simplicity and flexibility. Additionally, Ngrok was used to create a secure tunnel to the local server, making the interface accessible via a public URL during development and testing.

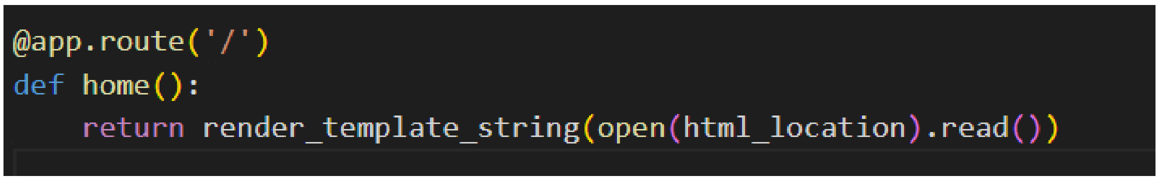
- **Install Required Libraries:** The necessary libraries such as Flask, Flask-Ngrok, Transformers, and other dependencies were installed.
- **Setup Ngrok:** Ngrok was configured with an authentication token to facilitate secure access to the local server.

5.6.2. Flask Application Setup

The Flask application serves as the backend of the interface. It handles user requests, processes inputs, and returns the generated summaries. The application consists of several routes to handle different functionalities.

5.6.2.1. Home Route:

Serves the HTML interface to the user.

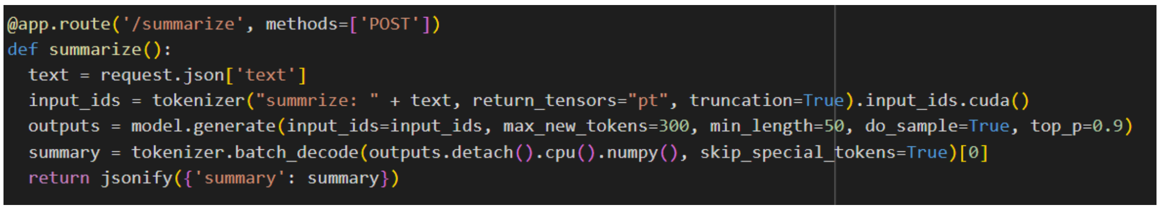


```
@app.route('/')
def home():
    return render_template_string(open(html_location).read())
```

Figure 53: Home Route

5.6.2.2. Summarize Route:

Receives text input from the user, processes it through the model, and returns the generated summary.



```
@app.route('/summarize', methods=['POST'])
def summarize():
  text = request.json['text']
  input_ids = tokenizer("summrize: " + text, return_tensors="pt", truncation=True).input_ids.cuda()
  outputs = model.generate(input_ids=input_ids, max_new_tokens=300, min_length=50, do_sample=True, top_p=0.9)
  summary = tokenizer.batch_decode(outputs.detach().cpu().numpy(), skip_special_tokens=True)[0]
  return jsonify({'summary': summary})
```

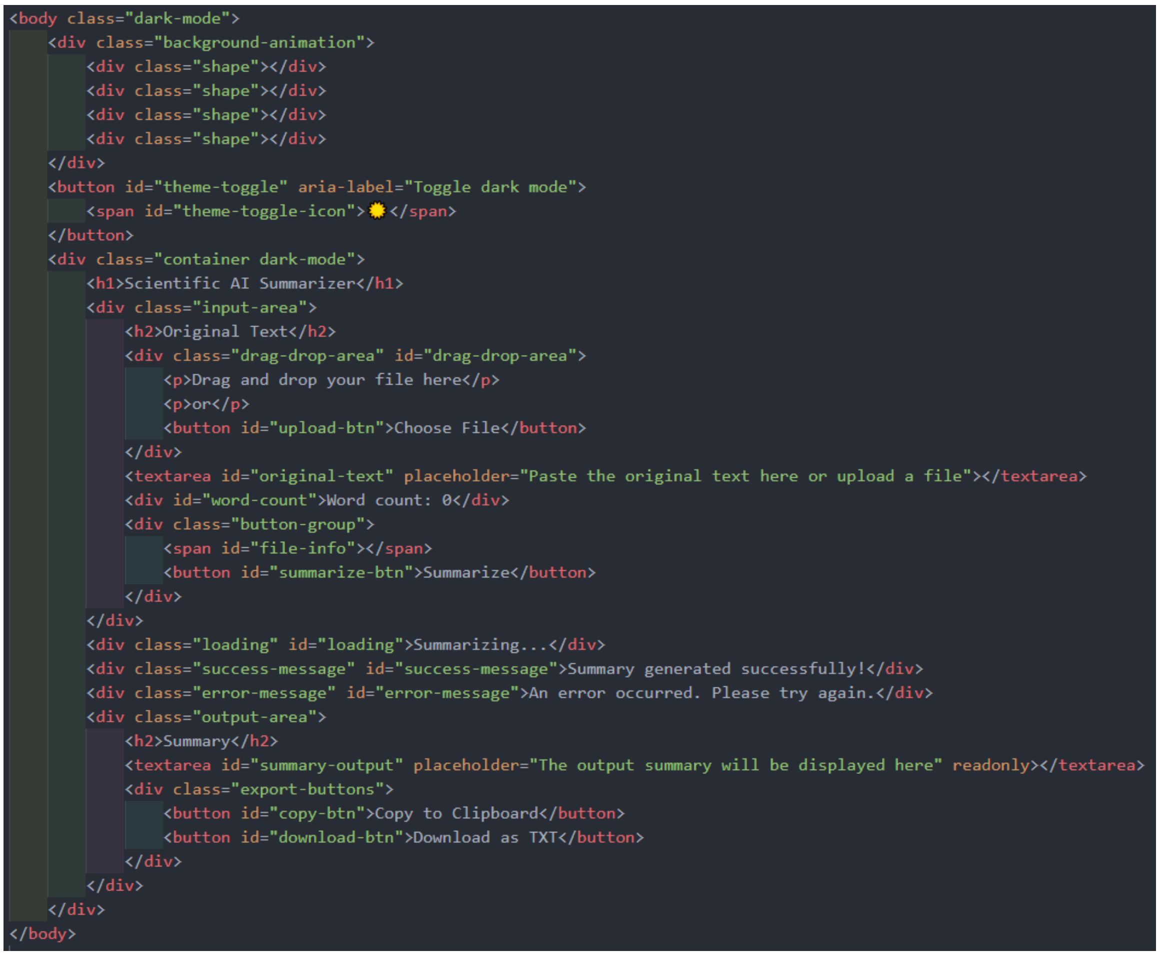
Figure 54: Summarize Route

5.6.3. HTML Interface

The HTML interface provides a user-friendly frontend where users can input text or upload files. It includes features like drag-and-drop file upload, real-time word count display, and options to copy or download the generated summaries.

- **File Upload Handling:** Users can upload text files, PDFs, or Word documents. The interface uses JavaScript libraries like pdf.js and mammoth.js to extract text from these files.
- **Text Input Area:** Provides a textarea for users to paste text directly.
- **Summarize Button:** A button that sends the input text to the Flask backend for summarization.
- **Output Area:** Displays the generated summary returned by the backend.

HTML Structure:



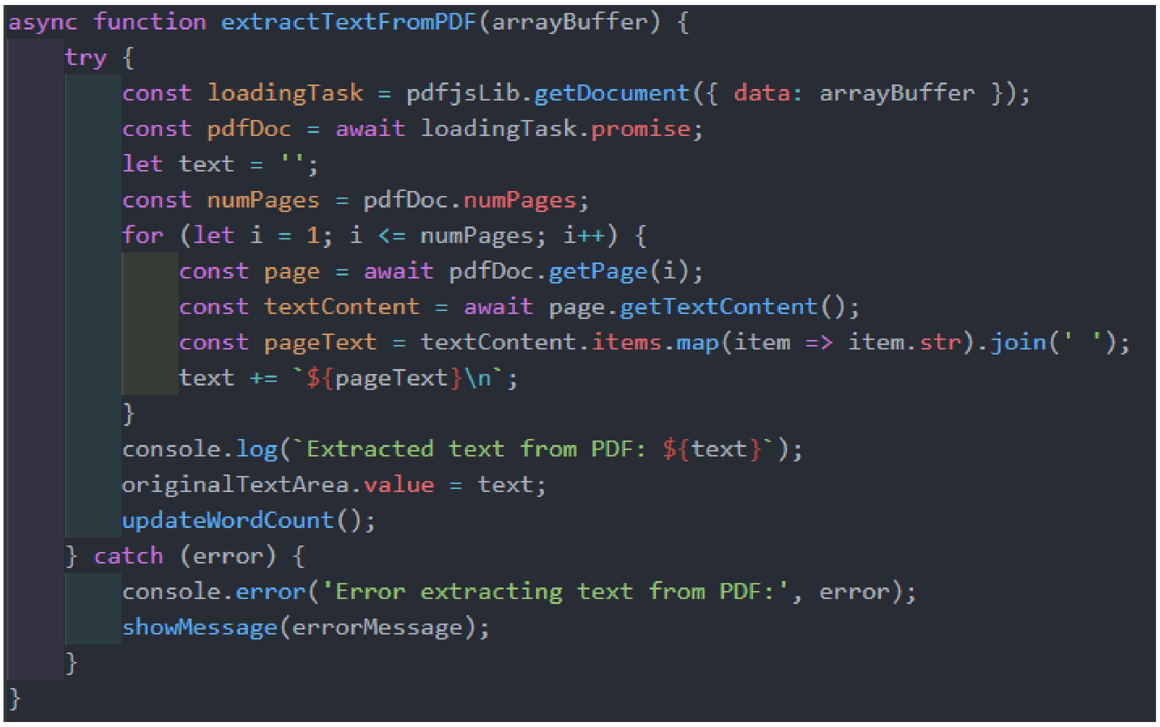
```html
<body class="dark-mode">
    <div class="background-animation">
        <div class="shape"></div>
        <div class="shape"></div>
        <div class="shape"></div>
        <div class="shape"></div>
    </div>
    <button id="theme-toggle" aria-label="Toggle dark mode">
        <span id="theme-toggle-icon">🌞</span>
    </button>
    <div class="container dark-mode">
        <h1>Scientific AI Summarizer</h1>
        <div class="input-area">
            <h2>Original Text</h2>
            <div class="drag-drop-area" id="drag-drop-area">
                <p>Drag and drop your file here</p>
                <p>or</p>
                <button id="upload-btn">Choose File</button>
            </div>
            <textarea id="original-text" placeholder="Paste the original text here or upload a file"></textarea>
            <div id="word-count">Word count: 0</div>
            <div class="button-group">
                <span id="file-info"></span>
                <button id="summarize-btn">Summarize</button>
            </div>
        </div>
        <div class="loading" id="loading">Summarizing...</div>
        <div class="success-message" id="success-message">Summary generated successfully!</div>
        <div class="error-message" id="error-message">An error occurred. Please try again.</div>
        <div class="output-area">
            <h2>Summary</h2>
            <textarea id="summary-output" placeholder="The output summary will be displayed here" readonly></textarea>
            <div class="export-buttons">
                <button id="copy-btn">Copy to Clipboard</button>
                <button id="download-btn">Download as TXT</button>
            </div>
        </div>
    </div>
</body>
```

Figure 55: Complete HTML code

5.6.4. JavaScript Functions:

JavaScript functions handle user interactions such as file uploads, text summarization requests, and UI updates.

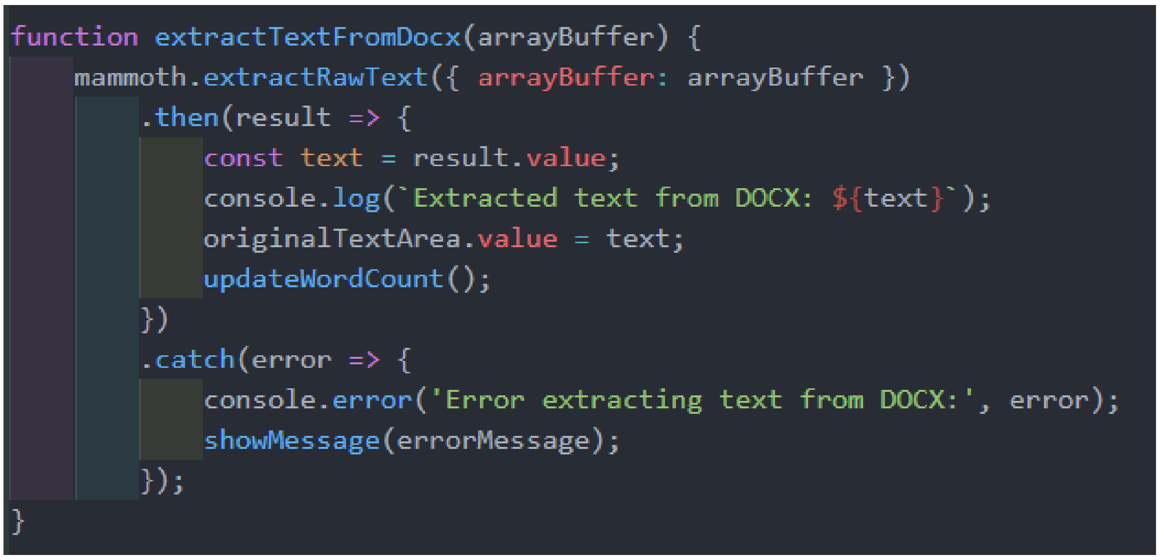
5.6.4.1. **PDF Files:** Use pdf.js to extract text.



```
async function extractTextFromPDF(arrayBuffer) {
    try {
        const loadingTask = pdfjsLib.getDocument({ data: arrayBuffer });
        const pdfDoc = await loadingTask.promise;
        let text = '';
        const numPages = pdfDoc.numPages;
        for (let i = 1; i <= numPages; i++) {
            const page = await pdfDoc.getPage(i);
            const textContent = await page.getTextContent();
            const pageText = textContent.items.map(item => item.str).join(' ');
            text += `${pageText}\n`;
        }
        console.log(`Extracted text from PDF: ${text}`);
        originalTextArea.value = text;
        updateWordCount();
    } catch (error) {
        console.error('Error extracting text from PDF:', error);
        showMessage(errorMessage);
    }
}
```

Figure 56: PDF to text

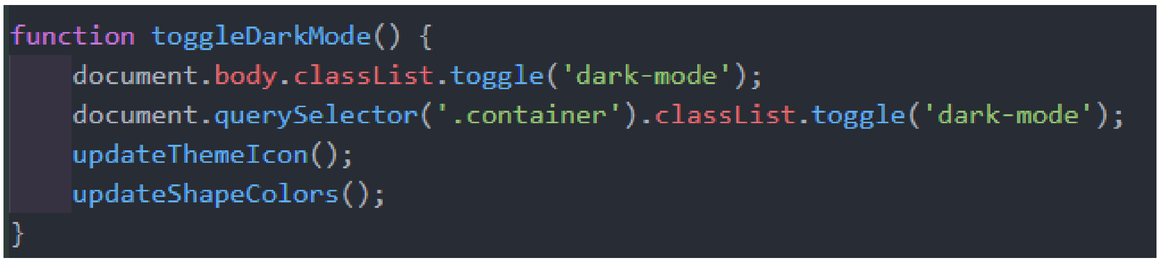
5.6.4.2. **Word Documents:** Use mammoth.js to extract raw text.



```
function extractTextFromDocx(arrayBuffer) {
    mammoth.extractRawText({ arrayBuffer: arrayBuffer })
        .then(result => {
            const text = result.value;
            console.log(`Extracted text from DOCX: ${text}`);
            originalTextArea.value = text;
            updateWordCount();
        })
        .catch(error => {
            console.error('Error extracting text from DOCX:', error);
            showMessage(errorMessage);
        });
}
```

Figure 57: Word Document to text

5.6.4.3. **Dark Mode**: Users can switch between light and dark modes, with the interface adapting accordingly.

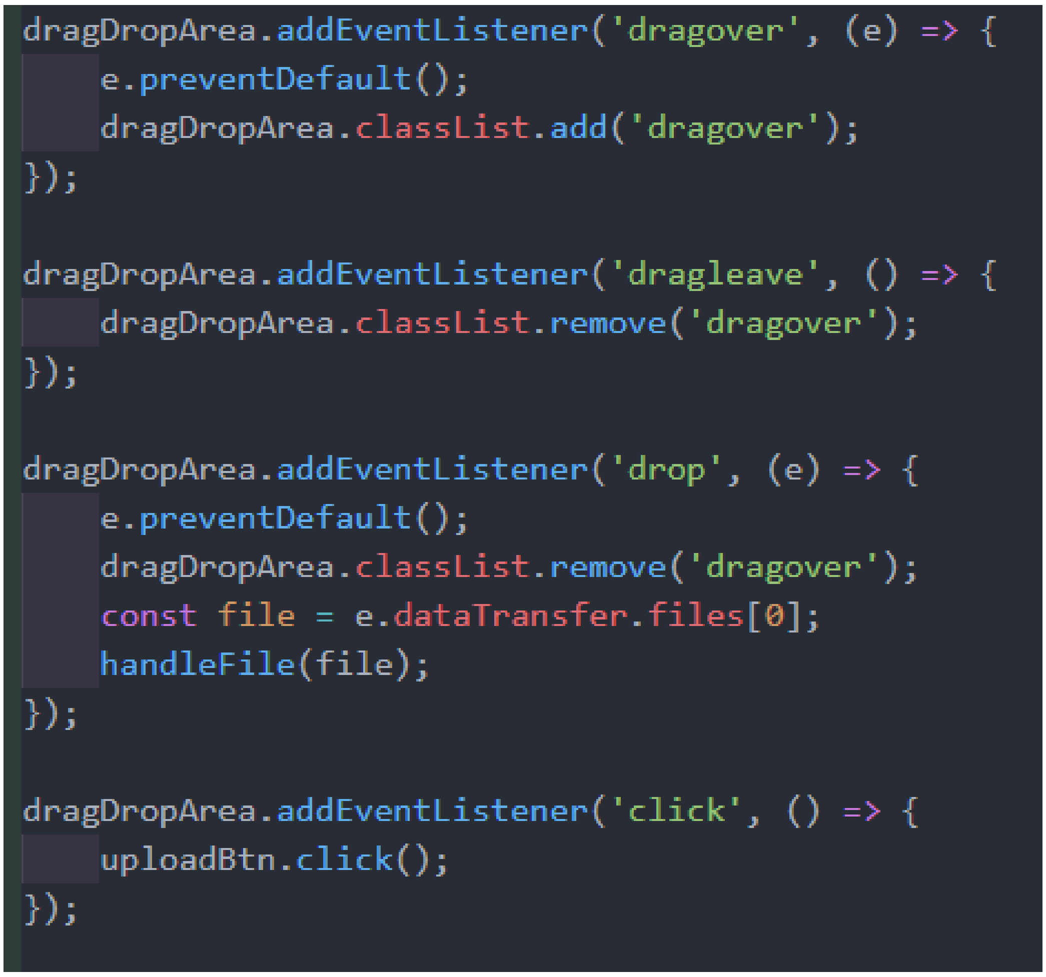


```
function toggleDarkMode() {
    document.body.classList.toggle('dark-mode');
    document.querySelector('.container').classList.toggle('dark-mode');
    updateThemeIcon();
    updateShapeColors();
}
```

Figure 58: Dark Mode Toggle

5.6.4.4. Drag-and-Drop File Upload

The drag-and-drop functionality allows users to upload files easily by dragging them into a designated area. This feature enhances the user experience by providing a convenient way to input large texts or documents.

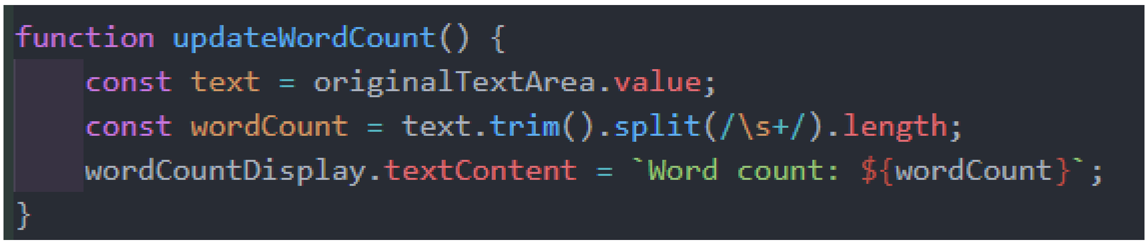


```
dragDropArea.addEventListener('dragover', (e) => {
    e.preventDefault();
    dragDropArea.classList.add('dragover');
});

dragDropArea.addEventListener('dragleave', () => {
    dragDropArea.classList.remove('dragover');
});

dragDropArea.addEventListener('drop', (e) => {
    e.preventDefault();
    dragDropArea.classList.remove('dragover');
    const file = e.dataTransfer.files[0];
    handleFile(file);
});

dragDropArea.addEventListener('click', () => {
    uploadBtn.click();
});
```

Figure 59: Drag-and-Drop File Upload

5.6.4.5. Real-Time Word Count

The real-time word count feature provides users with immediate feedback on the length of their input text. This feature helps users stay within any word limits and ensures they provide sufficient content for summarization.

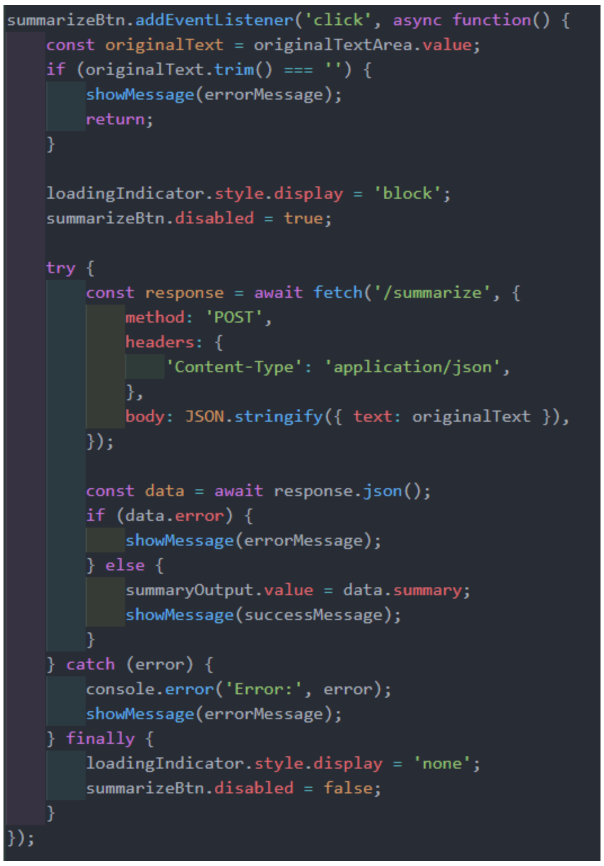


```
function updateWordCount() {
    const text = originalTextArea.value;
    const wordCount = text.trim().split(/\s+/).length;
    wordCountDisplay.textContent = `Word count: ${wordCount}`;
}
```

Figure 60: Update Word Count

5.6.4.6. Displaying the Summary

Once the summary is generated by the backend, it is displayed in a read-only text area. The interface also provides options to copy the summary to the clipboard or download it as a text file.

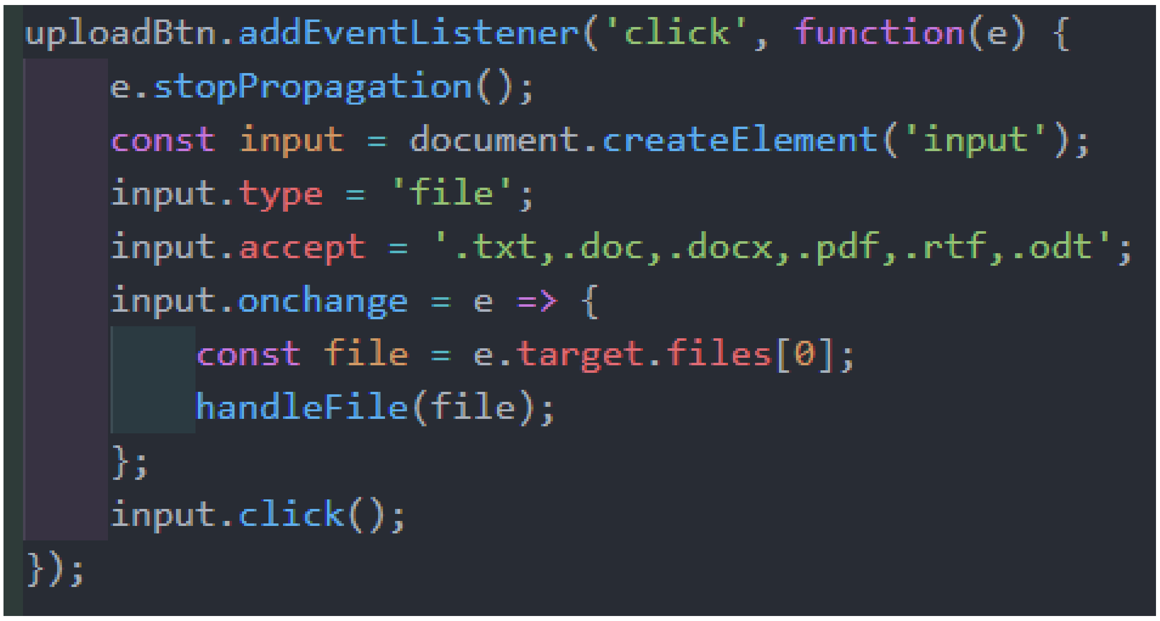


```
summarizeBtn.addEventListener('click', async function() {
    const originalText = originalTextArea.value;
    if (originalText.trim() === '') {
        showMessage(errorMessage);
        return;
    }

    loadingIndicator.style.display = 'block';
    summarizeBtn.disabled = true;

    try {
        const response = await fetch('/summarize', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ text: originalText }),
        });

        const data = await response.json();
        if (data.error) {
            showMessage(errorMessage);
        } else {
            summaryOutput.value = data.summary;
            showMessage(successMessage);
        }
    } catch (error) {
        console.error('Error:', error);
        showMessage(errorMessage);
    } finally {
        loadingIndicator.style.display = 'none';
        summarizeBtn.disabled = false;
    }
});
```

Figure 61: Displaying the Summary

5.6.4.7. Handling File Upload Button

The function to handle the file upload button click event, which allows users to choose files from their device.

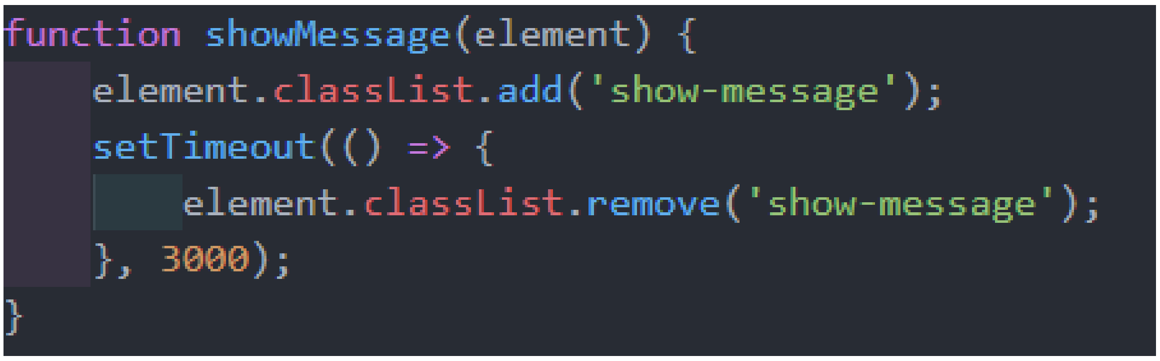


```
uploadBtn.addEventListener('click', function(e) {
    e.stopPropagation();
    const input = document.createElement('input');
    input.type = 'file';
    input.accept = '.txt,.doc,.docx,.pdf,.rtf,.odt';
    input.onchange = e => {
        const file = e.target.files[0];
        handleFile(file);
    };
    input.click();
});
```

Figure 62: Upload Button

5.6.4.8. Show Success and Error Messages

Functions to show success or error messages after certain actions, such as summarization completion or file upload issues.

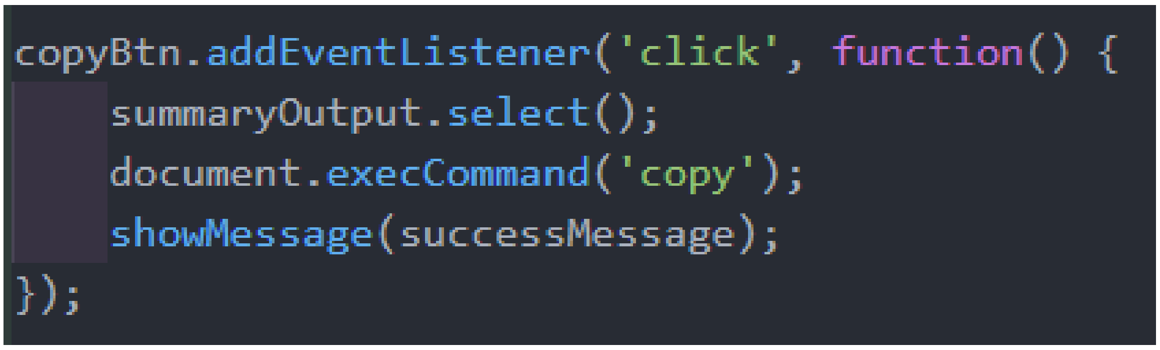


```
function showMessage(element) {
    element.classList.add('show-message');
    setTimeout(() => {
        element.classList.remove('show-message');
    }, 3000);
}
```

Figure 63: show Message

5.6.4.9. Copy Summary to Clipboard

Function to copy the generated summary to the clipboard.

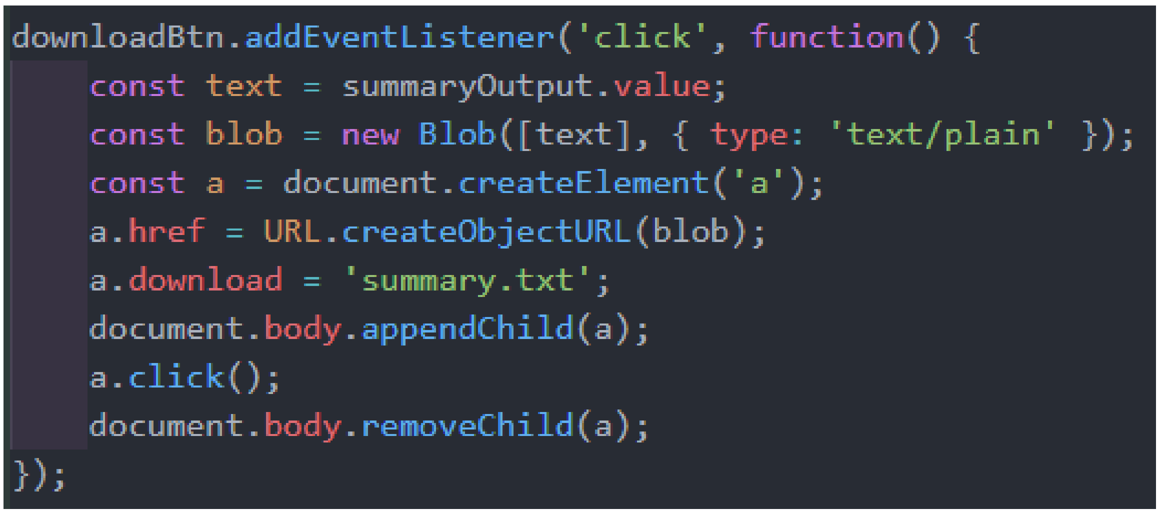


```
copyBtn.addEventListener('click', function() {
    summaryOutput.select();
    document.execCommand('copy');
    showMessage(successMessage);
});
```

Figure 64: copy to clipboard

5.6.4.10. Download Summary as Text File

Function to download the generated summary as a text file.

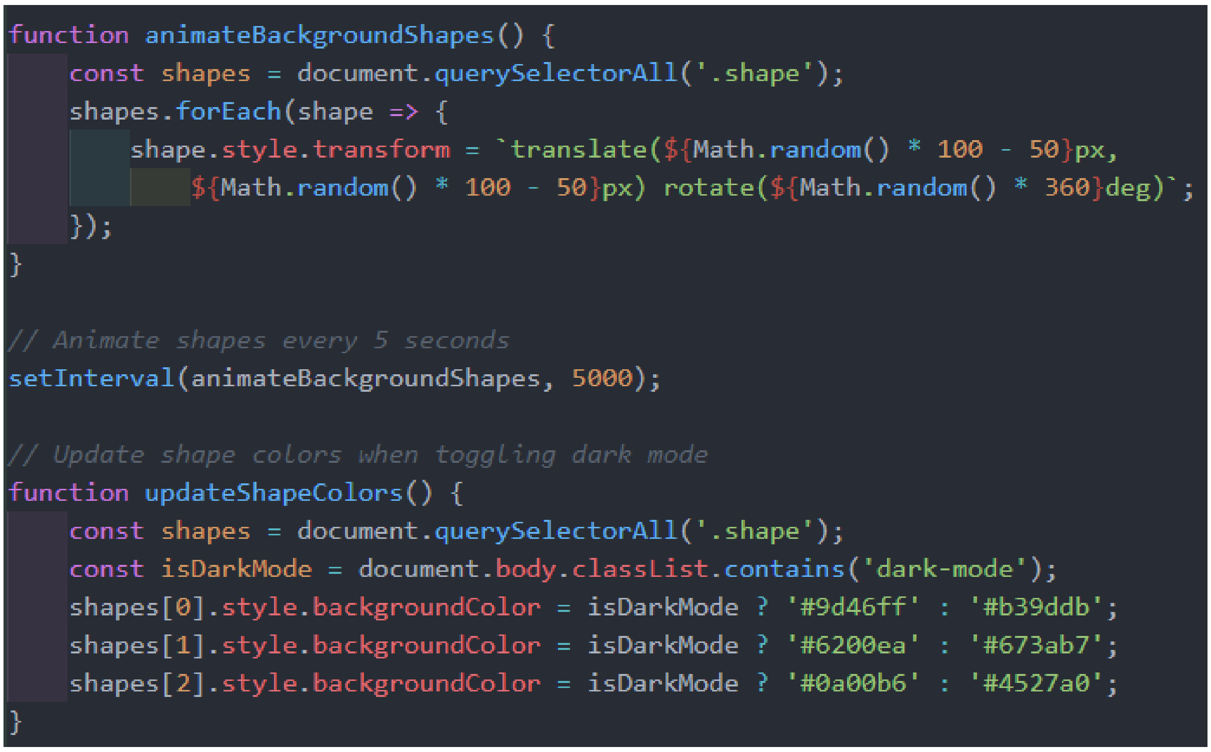


```
downloadBtn.addEventListener('click', function() {
    const text = summaryOutput.value;
    const blob = new Blob([text], { type: 'text/plain' });
    const a = document.createElement('a');
    a.href = URL.createObjectURL(blob);
    a.download = 'summary.txt';
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
});
```

Figure 65: Download Summary

5.6.4.11. Animate Background Shapes

Functions to animate background shapes and update their colors based on the current theme.



```javascript
function animateBackgroundShapes() {
    const shapes = document.querySelectorAll('.shape');
    shapes.forEach(shape => {
        shape.style.transform = `translate(${Math.random() * 100 - 50}px,
            ${Math.random() * 100 - 50}px) rotate(${Math.random() * 360}deg)`;
    });
}

// Animate shapes every 5 seconds
setInterval(animateBackgroundShapes, 5000);

// Update shape colors when toggling dark mode
function updateShapeColors() {
    const shapes = document.querySelectorAll('.shape');
    const isDarkMode = document.body.classList.contains('dark-mode');
    shapes[0].style.backgroundColor = isDarkMode ? '#9d46ff' : '#b39ddb';
    shapes[1].style.backgroundColor = isDarkMode ? '#6200ea' : '#673ab7';
    shapes[2].style.backgroundColor = isDarkMode ? '#0a00b6' : '#4527a0';
}
```

Figure 66: Animate Background Shapes

6. Evaluation and Results

In this chapter, we present a detailed evaluation of our summarization models, comparing the performance of the LoRA fine-tuned model with the original model. We use ROUGE scores as the primary evaluation metric and provide visualizations to illustrate the performance differences. This comprehensive analysis helps us understand the impact of fine-tuning on the model's ability to generate high-quality summaries.

6.1. Evaluation Metrics

To evaluate the performance of our models, we use the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) metrics. ROUGE is widely used in natural language processing for evaluating automatic summarization and machine translation systems. It measures the overlap between the system-generated summaries and reference summaries (ground truth). The following ROUGE metrics were used in our evaluation:

- ROUGE-1: Measures the overlap of unigrams (single words) between the system-generated summary and the reference summary. It provides a basic measure of content preservation.
- ROUGE-2: Measures the overlap of bigrams (two-word sequences) between the system-generated summary and the reference summary. This metric captures the preservation of basic contextual information.
- ROUGE-L: Measures the longest common subsequence (LCS) between the system-generated summary and the reference summary. ROUGE-L is sensitive to sentence-level structure and captures the fluency of the generated summary.
- ROUGE-Lsum: A variant of ROUGE-L, specifically designed for evaluating the quality of summaries by considering the summary as a whole.

These metrics provide a robust framework for evaluating the quality of generated summaries, offering insights into content preservation, contextual accuracy, and fluency.

6.2. Results and Analysis

6.2.1. ROUGE Scores Comparison

We begin by comparing the ROUGE scores of the LoRA fine-tuned model with those of the original model. The comparison is illustrated in Figure 6.1.

Figure 6.1: Comparison of ROUGE scores between the LoRA fine-tuned model and the original model.

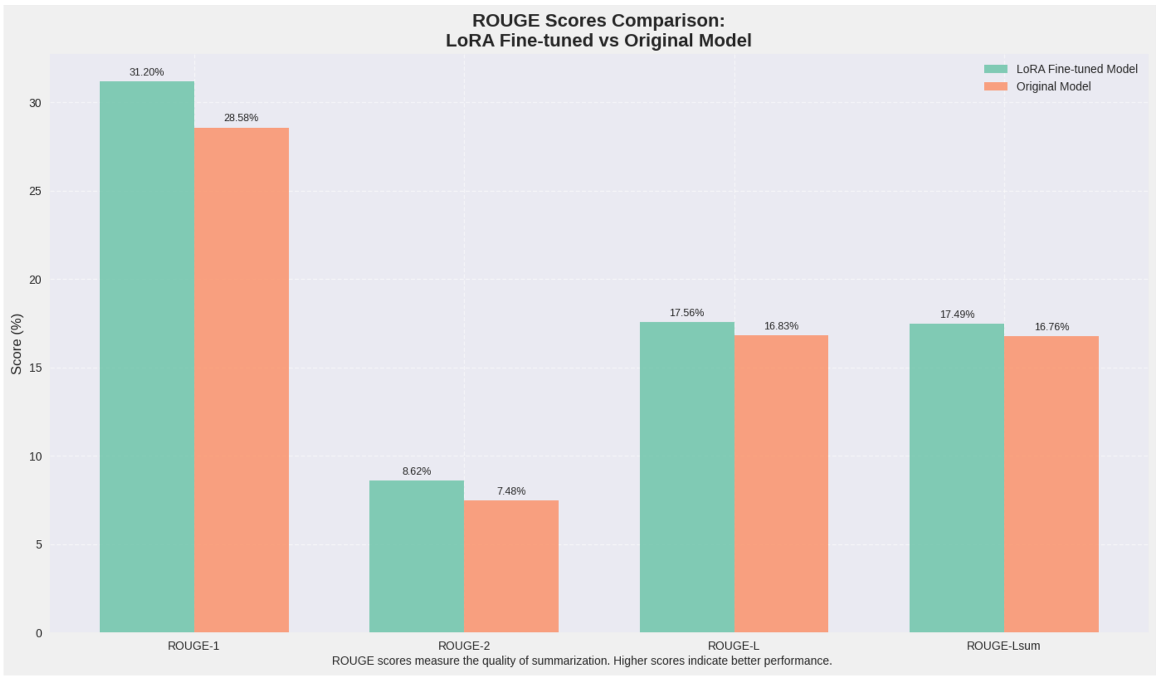


Figure 67: ROUGE Scores Comparison:

The results show that the LoRA fine-tuned model consistently outperforms the original model across all ROUGE metrics:

- ROUGE-1: The LoRA fine-tuned model achieves a ROUGE-1 score of 31.20%, compared to 28.58% for the original model. This indicates better preservation of individual words from the reference summary.
- ROUGE-2: The LoRA fine-tuned model achieves a ROUGE-2 score of 8.62%, compared to 7.48% for the original model. This demonstrates improved preservation of bigram sequences, capturing more contextual information.
- ROUGE-L: The LoRA fine-tuned model achieves a ROUGE-L score of 17.56%, compared to 16.83% for the original model. This highlights the improved fluency and sentence-level structure of the generated summaries.
- ROUGE-Lsum: The LoRA fine-tuned model achieves a ROUGE-Lsum score of 17.49%, compared to 16.76% for the original model. This indicates better overall summary quality.

These improvements demonstrate the effectiveness of the LoRA fine-tuning process in enhancing the model's summarization capabilities.

6.2.2. ROUGE Scores by Max and Min Lengths

We further analyze the model's performance by examining the ROUGE scores with varying maximum and minimum length constraints for the generated summaries. This analysis helps us understand how different length settings impact the quality of the summaries.

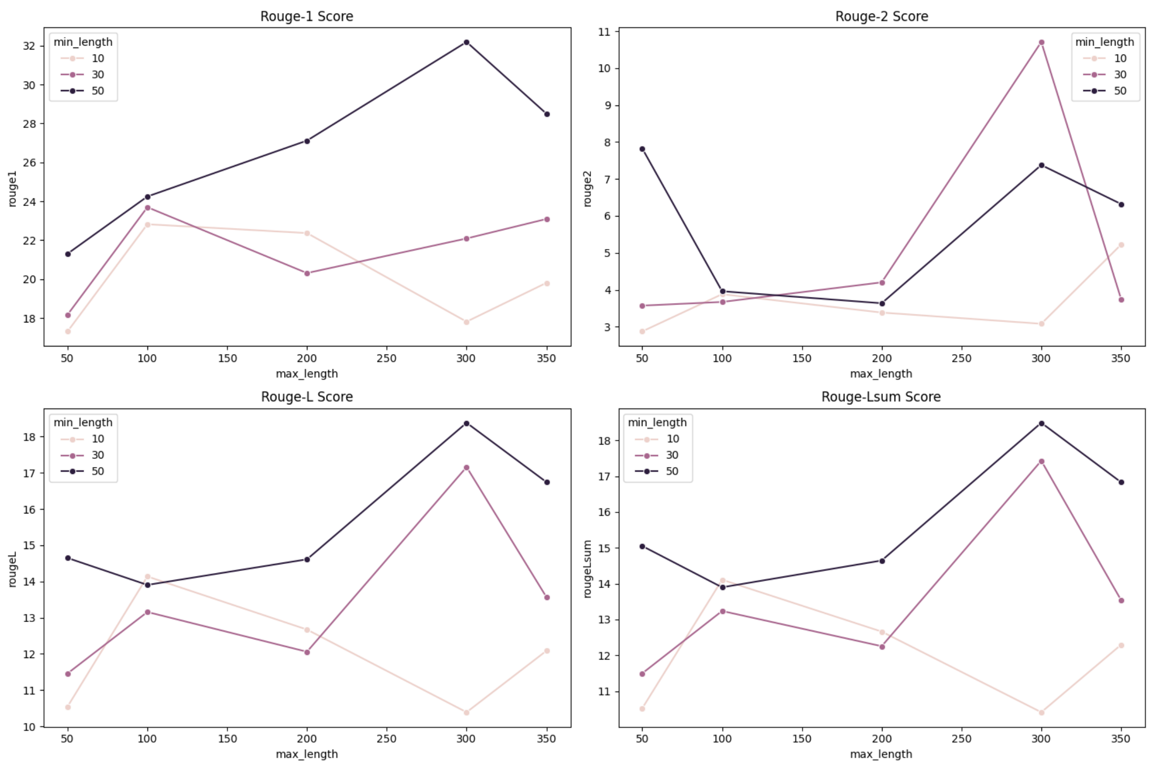


Figure 68: : ROUGE scores with varying max and min lengths

we observe the following trends:

- ROUGE-1 Scores: The scores vary significantly with changes in maximum and minimum lengths. The optimal settings for ROUGE-1 scores are observed at a max length of 300 and a min length of 50.
- ROUGE-2 Scores: The scores show a peak at a max length of 300 and a min length of 30. This indicates that preserving bigram sequences is more effective with these length constraints.
- ROUGE-L Scores: The scores are highest with a max length of 300 and a min length of 50, indicating better sentence-level structure and fluency under these settings.
- ROUGE-Lsum Scores: Similar to ROUGE-L, the best scores are observed with a max length of 300 and a min length of 50, reflecting the overall quality of the summaries.

These trends highlight the importance of tuning length constraints to optimize the performance of the summarization model.

6.2.3. Detailed Evaluation and Summarization Results

For a detailed evaluation, we generated summaries for a set of test papers and compared them with the reference abstracts. This evaluation provides insights into the strengths and weaknesses of the model, guiding further improvements. The following table summarizes the average ROUGE scores for the LoRA fine-tuned model and the original model across the test set:

Metric	LoRA Fine-tuned Model	Original Model
ROUGE-1	31.20%	28.58%
ROUGE-2	8.62%	7.48%
ROUGE-L	17.56%	16.83%
ROUGE-Lsum	17.49%	16.76%

Table 1: Average ROUGE scores for LoRA fine-tuned model and original model

The results clearly indicate that the LoRA fine-tuned model outperforms the original model across all metrics. The higher ROUGE scores suggest that the fine-tuning process enhances the model's ability to generate summaries that are more similar to the reference summaries.

6.3. Comparison of Human Evaluation with ROUGE Scores

In this section, we compare the average ROUGE scores of the LoRA model with the average human evaluation scores. This comparison provides a more nuanced understanding of the model's performance in generating summaries.

6.3.1. Average Human Evaluation Scores

The human evaluation scores were averaged across all the evaluated papers. Here is the summary of the average human evaluation scores:

Title	Overall Rating
Using Locality-sensitive Hashing for Rendezvous Search	4.0
Wirelessly-powered Sensor Networks: Power Allocation for Channel Estimation and Energy Beamforming	4.0
CABaRet: Leveraging Recommendation Systems for Mobile Edge Caching	1.0
This internet on the ground - Nick Merrill	4.0
Discrete-Time Analysis of Wireless Blockchain Networks	1.0
End-to-End Performance Analysis of Underwater Optical Wireless Relaying and Routing Techniques Under Location Uncertainty	4.0
An analysis of the Internet of Things in wireless sensor network technologies	3.5
Recent Results on Proportional Fair Scheduling for mmWave-based Industrial Wireless Networks	4.0
Aspects of Entertainment Distribution in an Intelligent Home Environment	2.5
Modeling and Analysis of MPTCP Proxy-based LTE-WLAN Path Aggregation	3.5

Table 2: Human Evaluation Scores

Average Human Evaluation Score:

(4 + 4 + 1 + 4 + 1 + 4 + 3.5 + 4 + 2.5 + 3.5) / 10 = 3.55

Average ROUGE Scores

The average ROUGE scores for the LoRA model are as follows:

Metric	Average Score
ROUGE-1	0.2927
ROUGE-2	0.0400
ROUGE-L	0.1606
ROUGE-Lsum	0.1606

Table 3: Model Average ROUGE scores

6.3.2. Comparative Analysis

The average human evaluation score is 3.05, which corresponds to a qualitative assessment of the summaries, considering aspects such as accuracy, coherence, and completeness. In contrast, the ROUGE scores provide a quantitative measure of the summaries by comparing the overlap of n-grams, word sequences, and word pairs between the generated summaries and the reference texts.

Key Observations:

6.3.2.1. Overall Evaluation:

- The average human evaluation score of 3.55 out of 5 indicates that the summaries generated by the LoRA model are generally acceptable but often lack depth and completeness.
- The ROUGE-1 score of 0.2927 suggests that the model captures about 29.27% of the unigrams present in the reference summaries, which aligns with the human evaluators' feedback on the summaries covering key points but lacking detail.

6.3.2.2. Detail and Completeness:

- The low ROUGE-2 score (0.0400) indicates that bigram overlaps are limited, reflecting the human evaluation's point that the summaries often miss detailed relationships and coherence between concepts.
- ROUGE-L and ROUGE-Lsum scores (both 0.1606) also show moderate performance in capturing longer sequences, which corresponds to the human feedback about missing technical details and deeper insights.

6.3.2.3. Human vs. ROUGE Scores:

- While the average human evaluation score is based on subjective judgment and qualitative aspects, the ROUGE scores are purely based on quantitative overlap metrics.

- The discrepancies between these scores highlight the need for a combined evaluation approach, incorporating both human judgments and automated metrics to gain a comprehensive understanding of summarization quality.

The comparison reveals that while the LoRA model performs reasonably well in generating summaries that capture the main points, there is room for improvement, particularly in capturing detailed and technical information. The combination of human evaluation and ROUGE scores provides a holistic view of the model's performance, suggesting areas for further enhancement to achieve more comprehensive and detailed summaries.

7. Conclusion and Future Work

7.1. Conclusion

7.1.1. Project Objectives and Accomplishments:

7.1.1.1. Developing a Text Summarization System for Scientific Literature:

Accomplished: Successfully developed a text summarization system tailored for scientific literature in the domain of Networking and Internet Architecture. The system utilizes the FLAN-T5 model fine-tuned with the LoRA approach, demonstrating improved performance in summarizing complex scientific texts.

7.1.1.2. Creating a Comprehensive Dataset:

Accomplished: Curated a comprehensive dataset from scientific papers specifically within the Networking and Internet Architecture domain. The dataset was meticulously pre-processed to ensure it was clean and suitable for model training and evaluation.

7.1.1.3. Implementing and Fine-Tuning Advanced Models:

Accomplished: Implemented the FLAN-T5 model and applied the LoRA fine-tuning technique to enhance its summarization capabilities. The model's performance was evaluated using standard metrics, showing significant improvement over the baseline model.

7.1.1.4. Evaluating Model Performance:

Accomplished: Rigorously evaluated the model's performance using ROUGE scores and human evaluation. The LoRA fine-tuned model outperformed the original model across all ROUGE metrics, with human evaluations validating its effectiveness.

7.1.1.5. Building an Interactive User Interface:

Accomplished: Developed an interactive user interface to facilitate easy input and output of text summaries. The interface supports multiple file formats and provides a seamless user experience for generating summaries.

7.1.2. Major Learnings from the Project:

7.1.2.1. Understanding LLM Techniques:

Gained in-depth knowledge of LLM techniques and models, particularly the FLAN-T5 model and LoRA fine-tuning.

7.1.2.2. Importance of Data Pre-processing:

Learned the critical importance of data pre-processing in ensuring the quality and effectiveness of model training.

7.1.2.3. Evaluation Metrics and Human Evaluation:

Understood the significance of using both quantitative metrics (ROUGE scores) and qualitative assessments (human evaluation) to comprehensively evaluate model performance.

7.2. Future Work and Enhancements:

7.2.1. Broader Dataset Inclusion:

Expand the dataset to include a wider range of categories within computer science and other scientific domains to enhance model generalizability and robustness.

7.2.2. Advanced Model Exploration:

Investigate more advanced transformer architectures or hybrid models to further improve summarization performance, and incorporate recent NLP advancements, such as prompt-based learning and few-shot learning.

7.2.3. User Interface Refinements:

Enhance the user interface by adding more features like interactive summarization, keyword extraction, and integration with reference management tools.

7.2.4. Real-Time and Cross-Language Summarization:

Develop capabilities for real-time summarization and support multiple languages to facilitate global research collaboration.

7.2.5. Scalability:

Scale the solution for deployment in large academic and research institutions, ensuring efficient handling of substantial data volumes.

7.3. Application of Program-Specific Skills and Knowledge:

- **Data Science and Machine Learning:** Applied data science and machine learning principles to preprocess data, train models, and evaluate their performance.
- **Natural Language Processing:** Leveraged advanced NLP techniques to develop a robust text summarization system tailored for scientific literature.
- **Software Development and User Interface Design:** Developed a user-friendly interface using software development skills, ensuring a seamless user experience.

- **Project Management:** Applied project management skills to plan, execute, and monitor the project, ensuring timely completion of objectives.

7.4. Final Remarks

This work provides an example of what is possible with advanced NLP models and how they can change the way we access large scientific bodies. Through further development and extension of this work, we hope to make scientific knowledge more accessible than ever before in the age of digital science.

8. Reference

- Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., & Mian, A. (2023). A comprehensive overview of large language models. arXiv. https://arxiv.org/abs/2307.06435
- Cajueiro, D. O., Nery, A. G., Tavares, I., De Melo, M. K., dos Reis, S. A., Weigang, L., & Celestino, V. R. R. (2023). A comprehensive review of automatic text summarization methods, data, evaluation, and coding. arXiv. https://arxiv.org/abs/2301.03403

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, CA, USA. arXiv. https://arxiv.org/abs/1706.03762

- El-Kassas, W. S., Salama, C. R., Rafea, A. A., & Mohamed, H. K. (2021). Automatic text summarization: A comprehensive survey. Expert Systems with Applications, 165, 113679. https://doi.org/10.1016/j.eswa.2020.113679

- Adams, D., Suri, G., & Chali, Y. (2022). Combining state-of-the-art models with maximal marginal relevance for few-shot and zero-shot multi-document summarization. arXiv. https://arxiv.org/abs/2211.10808

- Soni, M., & Wade, V. (2023). Comparing abstractive summaries generated by ChatGPT to real summaries through blinded reviewers and text classification algorithms. arXiv. https://arxiv.org/abs/2303.17650

- Khan, B., Shah, Z. A., Usman, M., Khan, I., & Niazi, B. (2023). Exploring the Landscape of Automatic Text Summarization: A Comprehensive Survey. IEEE Access, 11. https://doi.org/10.1109/ACCESS.2023.3322188

- Kocmi, T., & Federmann, C. (2023). Large language models are state-of-the-art evaluators of translation quality. arXiv. https://arxiv.org/abs/2302.14520v2

- Fabbri, A. R., Kryściński, W., McCann, B., Xiong, C., Socher, R., & Radev, D. R. (2021). SummEval: Re-evaluating summarization evaluation. Transactions of the Association for Computational Linguistics, 9, 391-409. https://doi.org/10.1162/tacl_a_00373

- Abualigah, L., Bashabsheh, M. Q., Alabool, H., & Shehab, M. (2020). Text summarization: A brief review. In M. Abd Elaziz et al. (Eds.), Recent Advances in NLP: The Case of Arabic Language Studies in Computational Intelligence (Vol. 874). Springer Nature Switzerland. https://doi.org/10.1007/978-3-030-34614-0_1

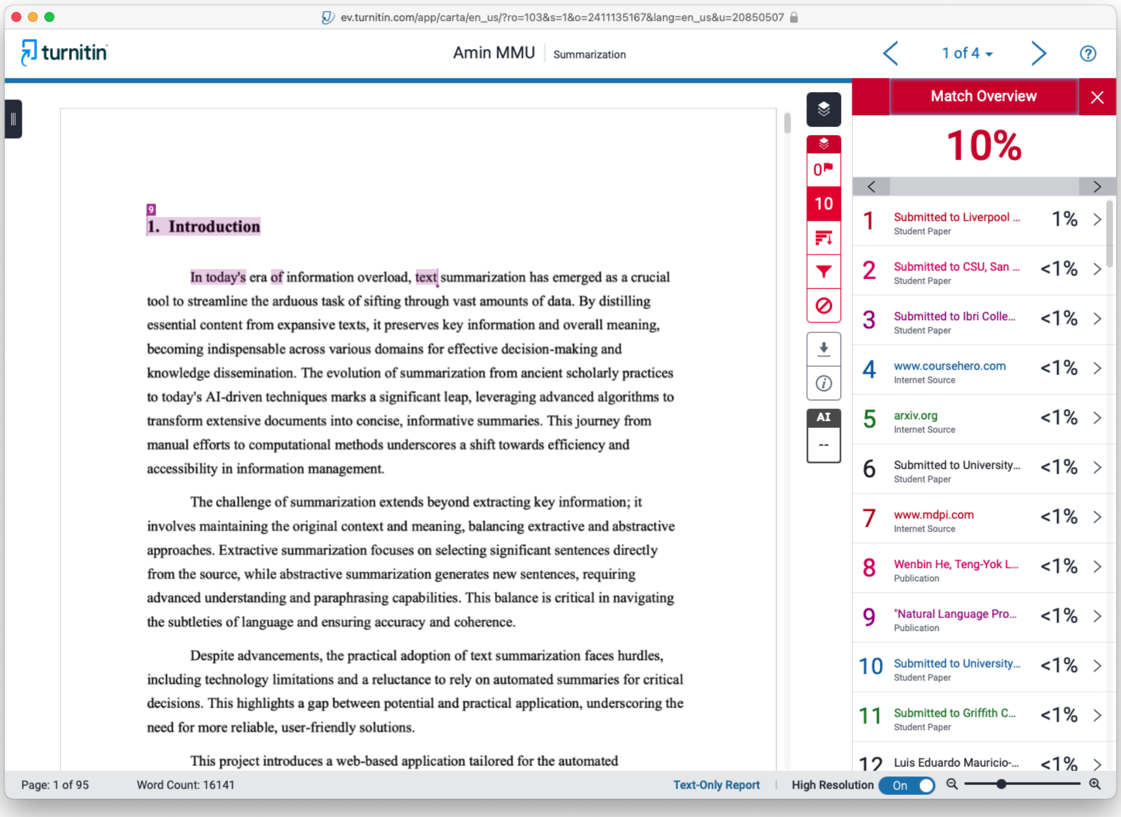
- Fabbri, A. R., Kryściński, W., McCann, B., Xiong, C., Socher, R., & Radev, D. (2021). SummEval: Re-evaluating Summarization Evaluation. Transactions of the

- Association for Computational Linguistics, 9, 391-409. https://doi.org/10.1162/tacl_a_00373

- Zheng, Y., Liu, H., & Co-authors. (2023). Large Language Models for Scientific Synthesis, Inference, and Explanation. arXiv. https://arxiv.org/abs/2310.07984

9. Appendix

Turnitin:



ev.turnitin.com/app/carta/en_us/?ro=103&s=1&o=2411135167&lang=en_us&u=20850507

turnitin Amin MMU | Summarization 1 of 4

1. Introduction

In today's era of information overload, text summarization has emerged as a crucial tool to streamline the arduous task of sifting through vast amounts of data. By distilling essential content from expansive texts, it preserves key information and overall meaning, becoming indispensable across various domains for effective decision-making and knowledge dissemination. The evolution of summarization from ancient scholarly practices to today's AI-driven techniques marks a significant leap, leveraging advanced algorithms to transform extensive documents into concise, informative summaries. This journey from manual efforts to computational methods underscores a shift towards efficiency and accessibility in information management.

The challenge of summarization extends beyond extracting key information; it involves maintaining the original context and meaning, balancing extractive and abstractive approaches. Extractive summarization focuses on selecting significant sentences directly from the source, while abstractive summarization generates new sentences, requiring advanced understanding and paraphrasing capabilities. This balance is critical in navigating the subtleties of language and ensuring accuracy and coherence.

Despite advancements, the practical adoption of text summarization faces hurdles, including technology limitations and a reluctance to rely on automated summaries for critical decisions. This highlights a gap between potential and practical application, underscoring the need for more reliable, user-friendly solutions.

This project introduces a web-based application tailored for the automated

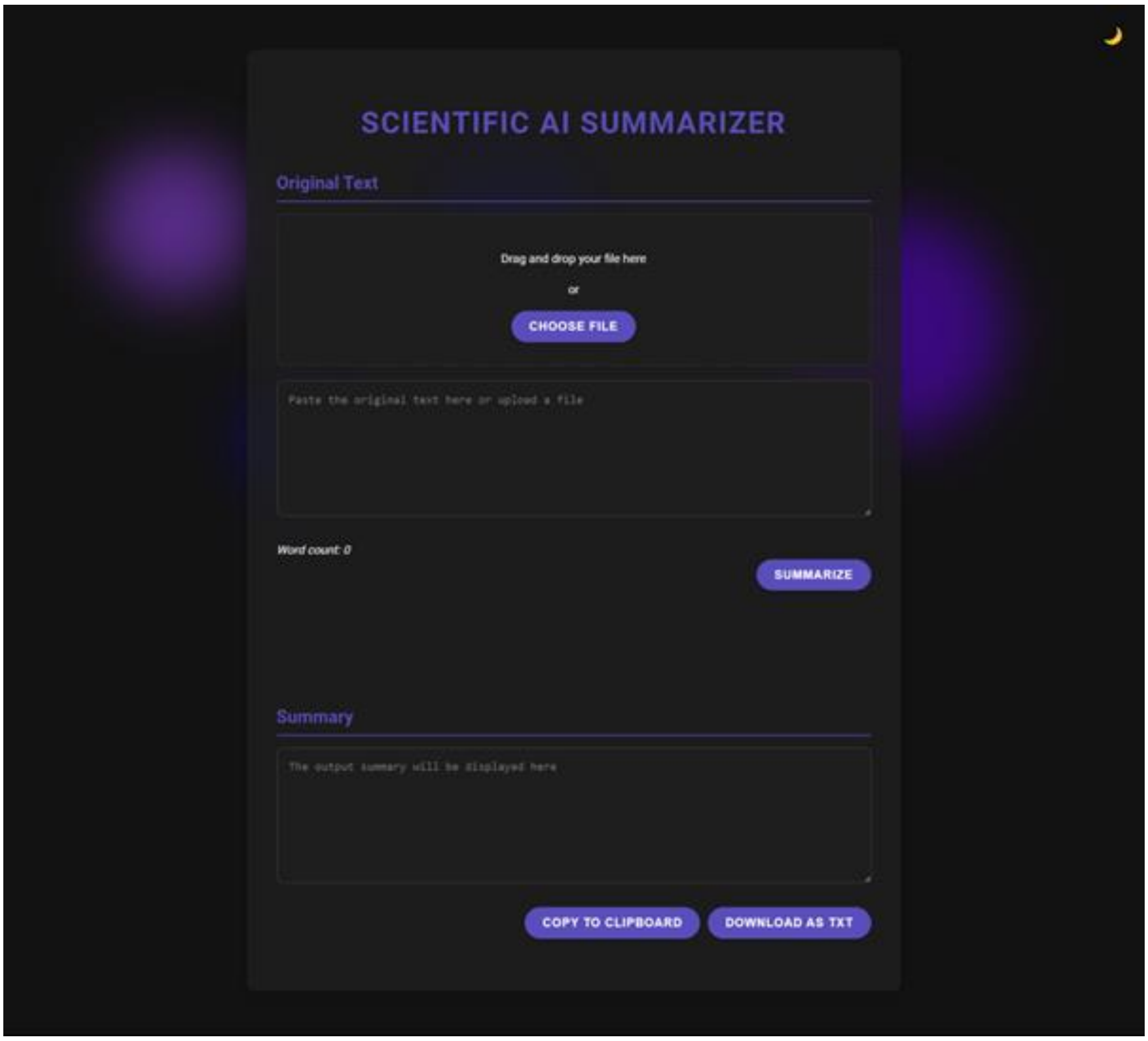
Match Overview

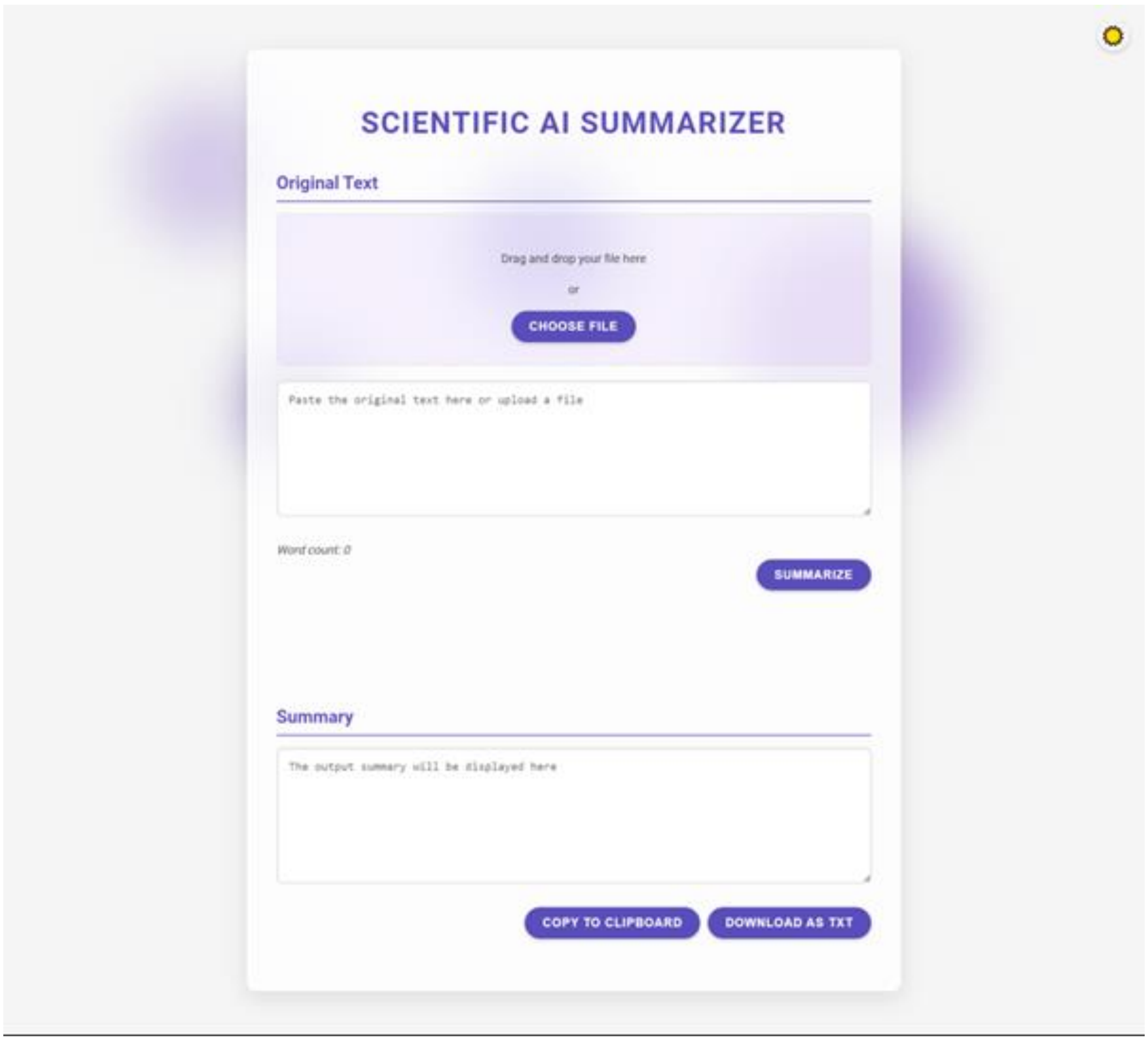
10%

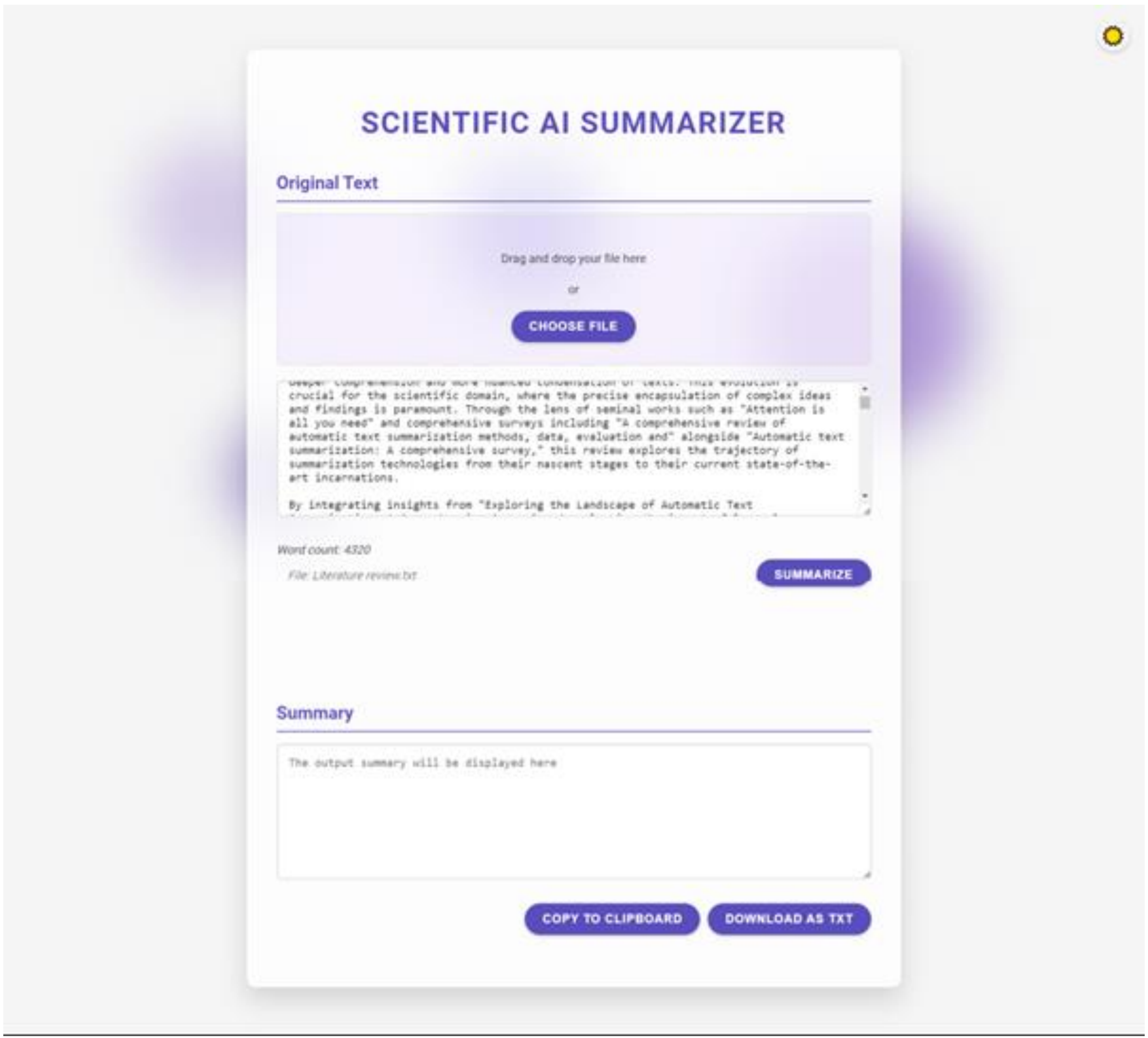
1	Submitted to Liverpool ... Student Paper	1%	>
2	Submitted to CSU, San ... Student Paper	<1%	>
3	Submitted to Ibri Colle... Student Paper	<1%	>
4	www.coursehero.com Internet Source	<1%	>
5	arxiv.org Internet Source	<1%	>
6	Submitted to University... Student Paper	<1%	>
7	www.mdpi.com Internet Source	<1%	>
8	Wenbin He, Teng-Yok L... Publication	<1%	>
9	"Natural Language Pro... Publication	<1%	>
10	Submitted to University... Student Paper	<1%	>
11	Submitted to Griffith C... Student Paper	<1%	>
12	Luis Eduardo Mauricio-...	<1%	>

Page: 1 of 95 Word Count: 16141 Text-Only Report | High Resolution On

Screenshots:


SCIENTIFIC AI SUMMARIZER
Original Text
Drag and drop your file here
or
CHOOSE FILE
Paste the original text here or upload a file
Word count: 0
SUMMARIZE
Summary
The output summary will be displayed here
COPY TO CLIPBOARD
DOWNLOAD AS TXT


SCIENTIFIC AI SUMMARIZER
Original Text
Drag and drop your file here
or
CHOOSE FILE
Paste the original text here or upload a file
Word count: 0
SUMMARIZE
Summary
The output summary will be displayed here
COPY TO CLIPBOARD
DOWNLOAD AS TXT



SCIENTIFIC AI SUMMARIZER

Original Text

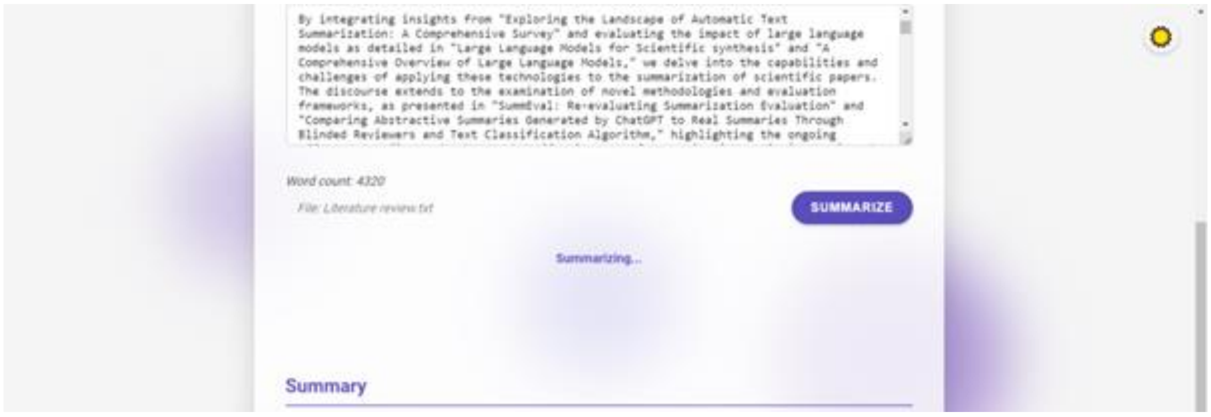
```
crucial for the scientific domain, where the precise encapsulation of complex ideas
and findings is paramount. Through the lens of seminal works such as "Attention is
all you need" and comprehensive surveys including "A comprehensive review of
automatic text summarization methods, data, evaluation and" alongside "Automatic text
summarization: A comprehensive survey," this review explores the trajectory of
summarization technologies from their nascent stages to their current state-of-the-
art incarnations.

By integrating insights from "Exploring the Landscape of Automatic Text
```

Word count: 4320

File: Literature review.txt

Summary

The output summary will be displayed here



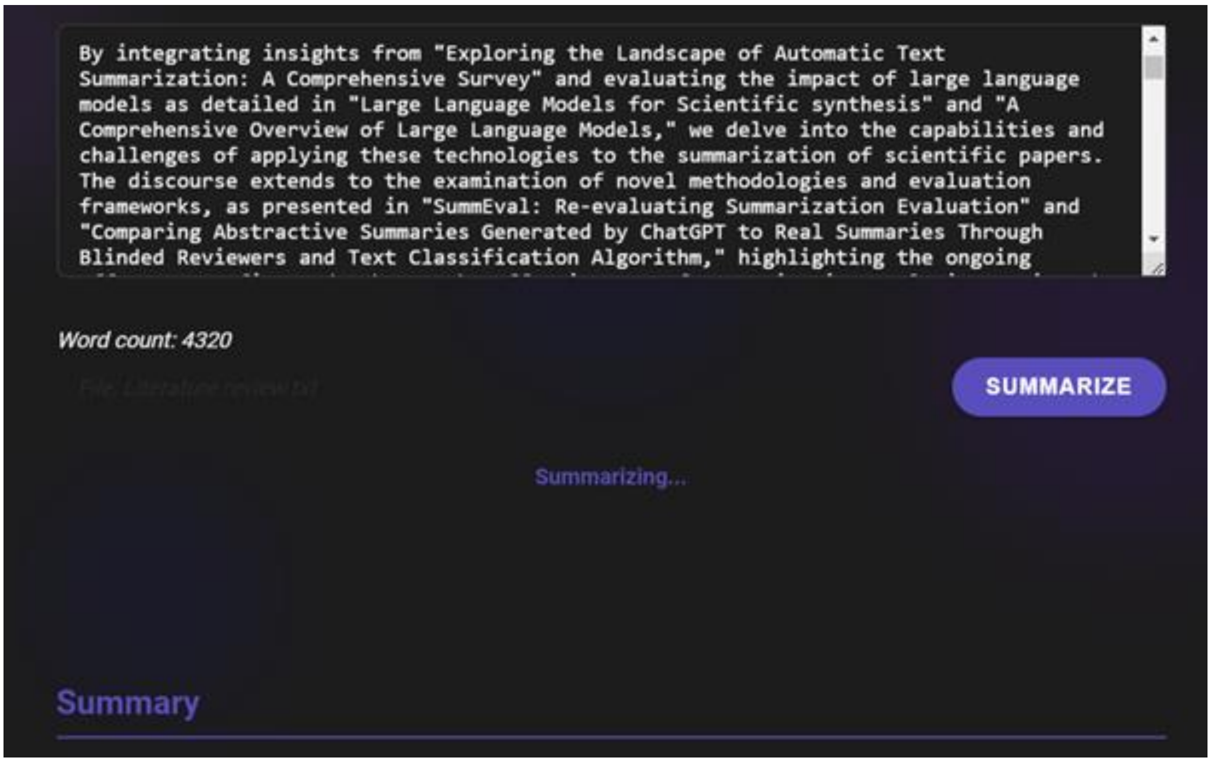
```
By integrating insights from "Exploring the Landscape of Automatic Text
Summarization: A Comprehensive Survey" and evaluating the impact of large language
models as detailed in "Large Language Models for Scientific synthesis" and "A
Comprehensive Overview of Large Language Models," we delve into the capabilities and
challenges of applying these technologies to the summarization of scientific papers.
The discourse extends to the examination of novel methodologies and evaluation
frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and
"Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through
Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing
```

Word count: 4320

File: Literature review.txt

Summarizing...

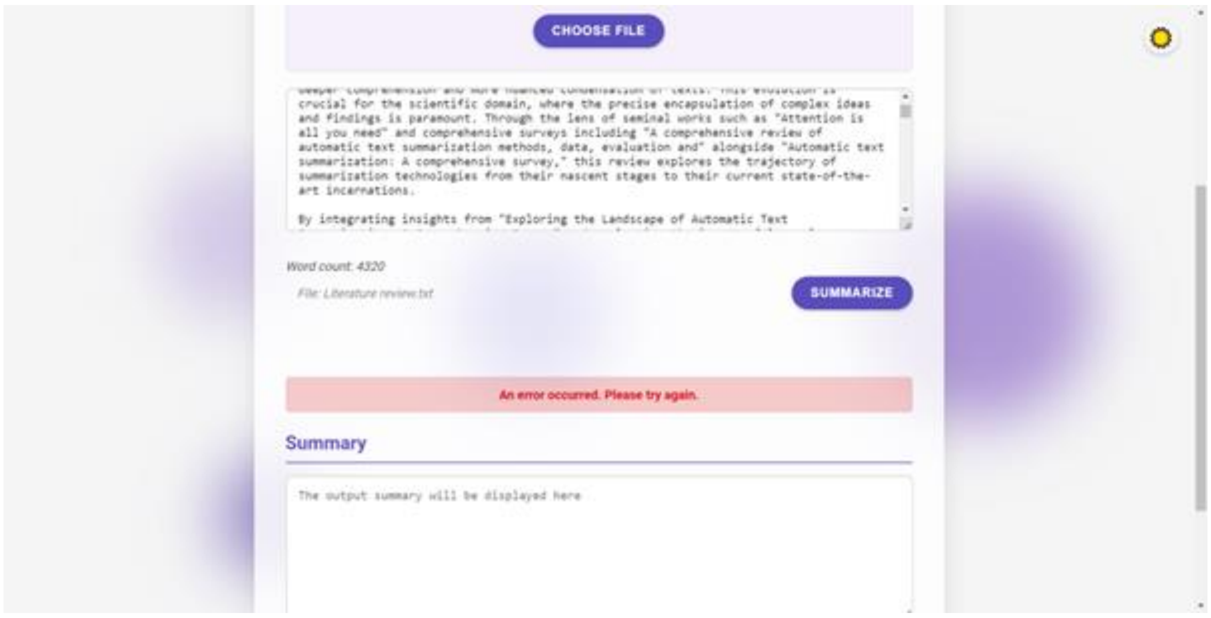
Summary



```
By integrating insights from "Exploring the Landscape of Automatic Text
Summarization: A Comprehensive Survey" and evaluating the impact of large language
models as detailed in "Large Language Models for Scientific synthesis" and "A
Comprehensive Overview of Large Language Models," we delve into the capabilities and
challenges of applying these technologies to the summarization of scientific papers.
The discourse extends to the examination of novel methodologies and evaluation
frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and
"Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through
Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing
```

Word count: 4320

File: Literature review.txt



```
crucial for the scientific domain, where the precise encapsulation of complex ideas
and findings is paramount. Through the lens of seminal works such as "Attention is
all you need" and comprehensive surveys including "A comprehensive review of
automatic text summarization methods, data, evaluation and" alongside "Automatic text
summarization: A comprehensive survey," this review explores the trajectory of
summarization technologies from their nascent stages to their current state-of-the-
art incarnations.

By integrating insights from "Exploring the Landscape of Automatic Text
```

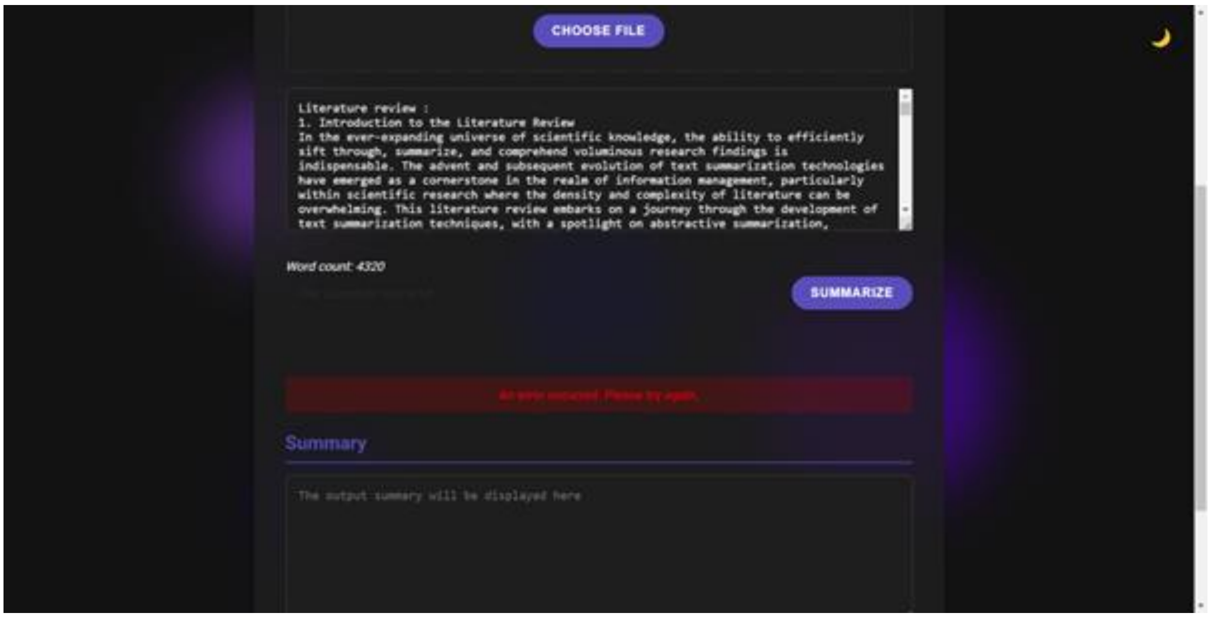
Word count: 4320

File: Literature review.txt

An error occurred. Please try again.

Summary

```
The output summary will be displayed here
```



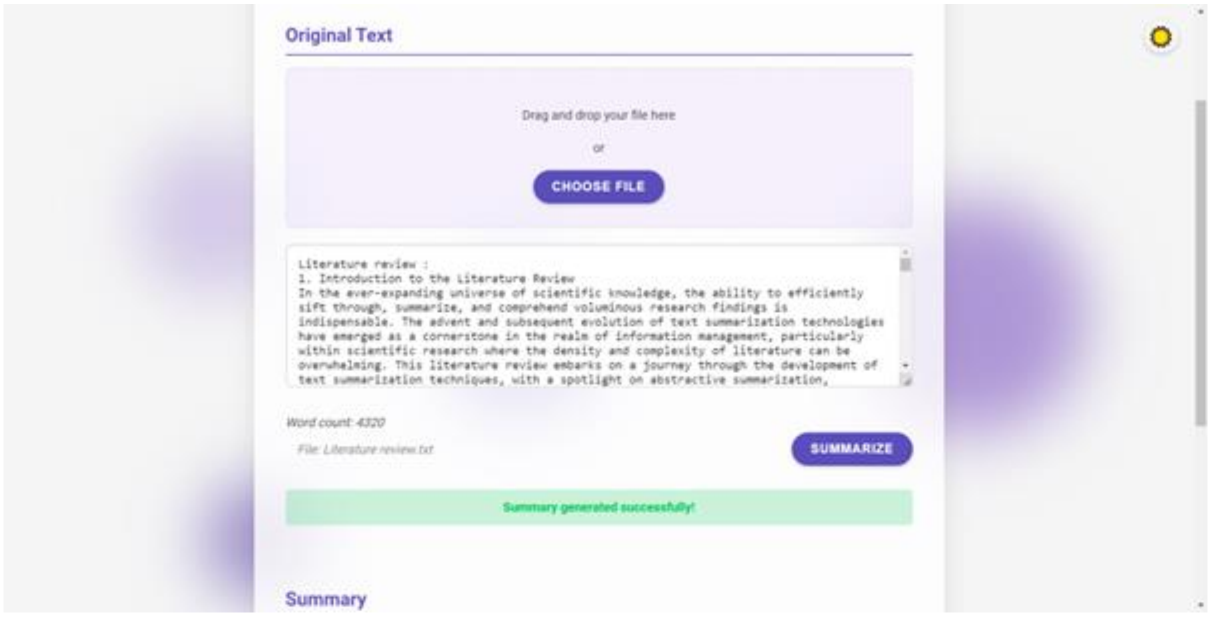
```
Literature review :
1. Introduction to the Literature Review
In the ever-expanding universe of scientific knowledge, the ability to efficiently
sift through, summarize, and comprehend voluminous research findings is
indispensable. The advent and subsequent evolution of text summarization technologies
have emerged as a cornerstone in the realm of information management, particularly
within scientific research where the density and complexity of literature can be
overwhelming. This literature review embarks on a journey through the development of
text summarization techniques, with a spotlight on abstractive summarization,
```

Word count: 4320

An error occurred. Please try again.

Summary

```
The output summary will be displayed here
```


Original Text

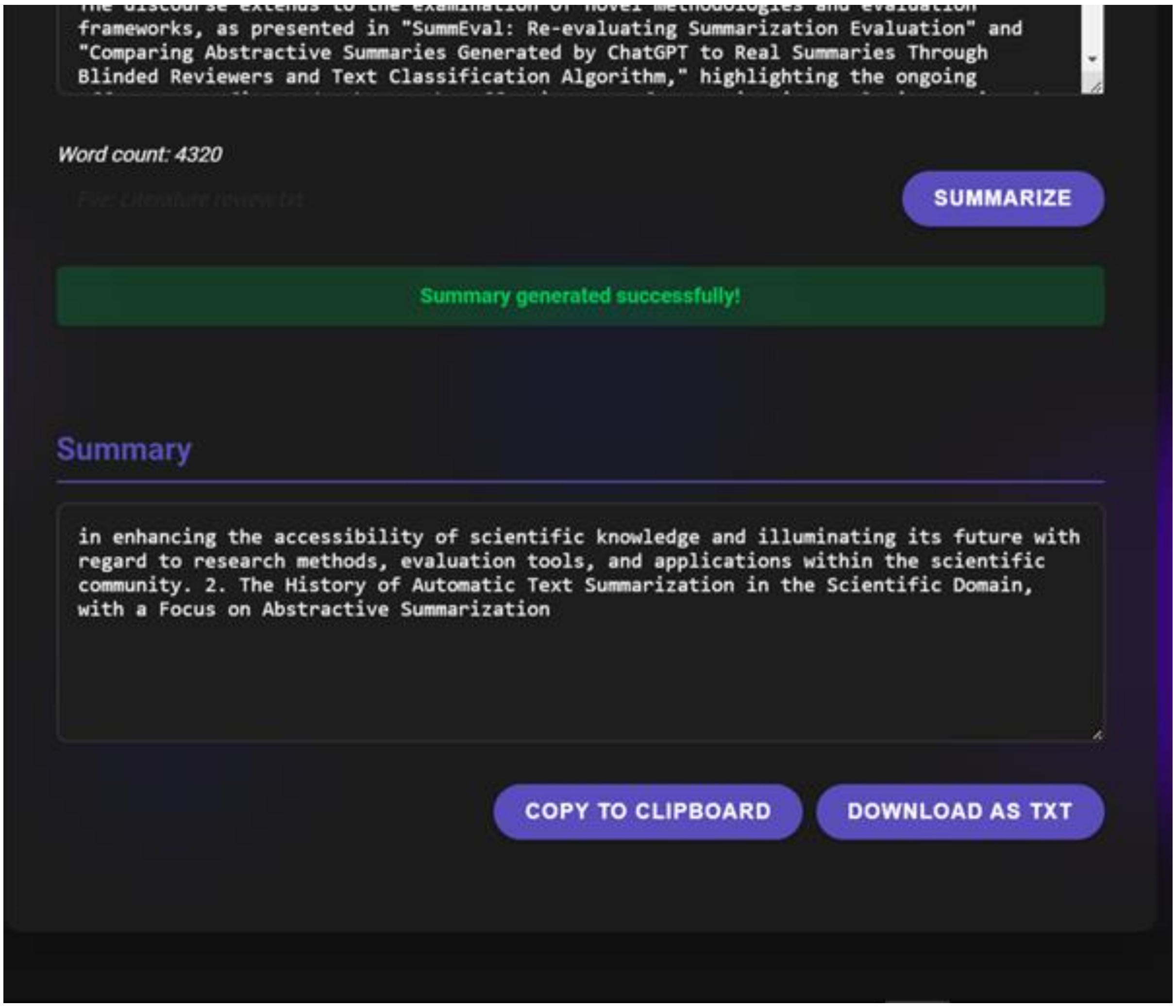
```
Literature review :
1. Introduction to the Literature Review
In the ever-expanding universe of scientific knowledge, the ability to efficiently
sift through, summarize, and comprehend voluminous research findings is
indispensable. The advent and subsequent evolution of text summarization technologies
have emerged as a cornerstone in the realm of information management, particularly
within scientific research where the density and complexity of literature can be
overwhelming. This literature review embarks on a journey through the development of
text summarization techniques, with a spotlight on abstractive summarization,
```

Word count: 4320

File: Literature review.txt

Summary generated successfully!

Summary



```
The discourse extends to the examination of novel methodologies and evaluation
frameworks, as presented in "SummEval: Re-evaluating Summarization Evaluation" and
"Comparing Abstractive Summaries Generated by ChatGPT to Real Summaries Through
Blinded Reviewers and Text Classification Algorithm," highlighting the ongoing
```

Word count: 4320

File: Literature review.txt

Summary generated successfully!

Summary

```
in enhancing the accessibility of scientific knowledge and illuminating its future with
regard to research methods, evaluation tools, and applications within the scientific
community. 2. The History of Automatic Text Summarization in the Scientific Domain,
with a Focus on Abstractive Summarization
```

Source code:

```python
import pandas as pd
import requests
import xml.etree.ElementTree as ET
import re
import os
from pdfminer.high_level import extract_text
import time
from pdfminer.psparser import PSSyntaxError
import fitz

# Load the dataset
cs_papers = pd.read_csv("data/cs_papers_api.csv")

category_mapping = {
    'cs.CV': 'Computer Vision and Pattern Recognition',
    'cs.NE': 'Neural and Evolutionary Computing',
    'cs.MA': 'Multiagent Systems',
    'cs.RO': 'Robotics',
    'cs.CL': 'Computation and Language',
    'cs.LG': 'Machine Learning',
    'cs.AI': 'Artificial Intelligence',
    'cs.CR': 'Cryptography and Security',
    'cs.HC': 'Human-Computer Interaction',
    'cs.IR': 'Information Retrieval',
    'cs.GT': 'Computer Science and Game Theory',
    'cs.SE': 'Software Engineering',
    'cs.CY': 'Computers and Society',
    'cs.ET': 'Emerging Technologies',
    'cs.NI': 'Networking and Internet Architecture',
    'cs.MM': 'Multimedia',
    'cs.SI': 'Social and Information Networks',
    'cs.CC': 'Computational Complexity',
    'cs.DB': 'Databases',
    'cs.IT': 'Information Theory',
    'cs.PL': 'Programming Languages',
    'cs.DS': 'Data Structures and Algorithms',
    'cs.SD': 'Sound',
    'cs.LO': 'Logic in Computer Science',
    'cs.DL': 'Digital Libraries',
    'cs.DC': 'Distributed, Parallel, and Cluster Computing',
    'cs.OH': 'Other Computer Science',
    'cs.CE': 'Computational Engineering, Finance, and Science',
    'cs.AR': 'Hardware Architecture',
    'cs.FL': 'Formal Languages and Automata Theory',
    'cs.GR': 'Graphics',
    'cs.MS': 'Mathematical Software',
    'cs.CG': 'Computational Geometry',
    'cs.SC': 'Symbolic Computation',
    'cs.PF': 'Performance',
    'cs.OS': 'Operating Systems',
    'cs.DM': 'Discrete Mathematics',
    'cs.NA': 'Numerical Analysis',
    'cs.SY': 'Systems and Control',
    'cs.GL': 'General Literature'
}
```

```
# Map the primary_category column to full titles
cs_papers['primary_category_full']                                    =
cs_papers['primary_category'].map(category_mapping)

cs_papers

def fetch_metadata(paper_id, retries=5, delay=10):
                                        base_url                      =
f"http://export.arxiv.org/api/query?id_list={paper_id}"
    for attempt in range(retries):
        try:
            response = requests.get(base_url, timeout=20)
            if response.status_code == 200:
                return response.text
            elif response.status_code == 500:
                   print(f"HTTP  500  error  for  paper_id  {paper_id},
retrying...")
                time.sleep(delay)
            else:
                    print(f"Failed  to  fetch  metadata  for  paper_id
{paper_id}, Status code: {response.status_code}")
                return None
        except requests.exceptions.RequestException as e:
            print(f"Attempt {attempt + 1} failed: {e}")
            if attempt < retries - 1:
                time.sleep(delay)
            else:
                    print(f"Failed  to  fetch  metadata  for  paper_id
{paper_id} after {retries} attempts.")
                return None

def parse_pdf_url(response_text):
    root = ET.fromstring(response_text)
    for entry in root.findall('{http://www.w3.org/2005/Atom}entry'):
                                         for         link            in
entry.findall('{http://www.w3.org/2005/Atom}link'):
            if link.attrib.get('title') == 'pdf':
                return link.attrib['href']
    return None

def clean_text(text):
    text = re.sub(r'\s+', ' ', text)  # Replace multiple whitespace
with a single space
    text = re.sub(r'\[[^]]*\]', '', text)  # Remove text in square
brackets
    text = re.sub(r'\([^)]*\)', '', text)  # Remove text in parentheses
    text = re.sub(r'<[^>]*>', '', text)  # Remove HTML tags
    text = re.sub(r'http[s]?://\S+', '', text)  # Remove URLs
    text = text.strip()
    return text

def remove_abstract_from_full_text(full_text, abstract):
    clean_abstract = clean_text(abstract)
    abstract_pattern = re.escape(clean_abstract)
    full_text_cleaned = re.sub(abstract_pattern, '', full_text,
flags=re.IGNORECASE)

    if full_text_cleaned == full_text:
        keywords = ["introduction", "1 introduction", "background",
"related work", "methodology", "methods",
                    "results", "discussion", " Keywords"]
        abstract_start = re.search(r'\babstract\b', full_text,
re.IGNORECASE)
        if not abstract_start:
            return full_text

        abstract_end = len(full_text)
        for keyword in keywords:
            match = re.search(r'\b' + re.escape(keyword) + r'\b',
full_text[abstract_start.end():], re.IGNORECASE)
            if match:
                abstract_end = abstract_start.end() + match.start()
                break

        full_text_cleaned = full_text[:abstract_start.start()] +
full_text[abstract_end:]

    return full_text_cleaned

def fetch_and_clean_full_text(paper_id, abstract, category_dir,
retries=5, delay=10):
    metadata = fetch_metadata(paper_id, retries, delay)
    if metadata:
        pdf_url = parse_pdf_url(metadata)
        if pdf_url:
            for attempt in range(retries):
                try:
                    response = requests.get(pdf_url, timeout=20)
                    if response.status_code == 200:
                              pdf_path  =  os.path.join(category_dir,
f"pdfs/{paper_id}.pdf")
                        with open(pdf_path, 'wb') as f:
                            f.write(response.content)

                        try:
                            full_text = extract_text(pdf_path)
                           full_text_cleaned = clean_text(full_text)
                                               full_text_cleaned   =
remove_abstract_from_full_text(full_text_cleaned, abstract)

                            if full_text_cleaned:
                                txt_path = os.path.join(category_dir,
f"texts/{paper_id}.txt")
                              with open(txt_path, 'w', encoding='utf-
8') as f:
                                    f.write(full_text_cleaned)
                                return full_text_cleaned
                        except PSSyntaxError as e:
                                print(f"PSSyntaxError  for  paper_id
{paper_id}: {e}")
                            return None
                    elif response.status_code == 500:
                      print(f"HTTP 500 error for paper_id {paper_id},
retrying...")
                        time.sleep(delay)
                    else:
                           print(f"Failed  to  fetch  PDF  for  paper_id
{paper_id}, Status code: {response.status_code}")
                        return None
                except requests.exceptions.RequestException as e:
                   print(f"Attempt {attempt + 1} to fetch PDF failed:
{e}")
                    if attempt < retries - 1:
                        time.sleep(delay)
                    else:
                            print(f"Failed to fetch PDF for paper_id
{paper_id} after {retries} attempts.")
                            return None
    return None

def fetch_full_texts_by_category(df, category, chunk_size=50):
    filtered_df = df[df['primary_category_full'] == category].copy()
    num_chunks = (len(filtered_df) + chunk_size - 1) // chunk_size

    category_dir = os.path.join("full", category)
    os.makedirs(os.path.join(category_dir, "pdfs"), exist_ok=True)
    os.makedirs(os.path.join(category_dir, "texts"), exist_ok=True)
    os.makedirs(os.path.join(category_dir, "chunks"), exist_ok=True)

    for i in range(num_chunks):
        chunk_df = filtered_df.iloc[i * chunk_size:(i + 1) *
chunk_size].copy()
        full_texts = []

        for paper_id, abstract in zip(chunk_df['paper_id'],
chunk_df['abstract']):
            full_text = fetch_and_clean_full_text(paper_id, abstract,
category_dir)
            full_texts.append(full_text)

        chunk_df.loc[:, 'cleaned_full_text'] = full_texts
        chunk_df.loc[:, 'cleaned_abstract'] =
chunk_df['abstract'].apply(clean_text)

        output_csv = os.path.join(category_dir,
f"chunks/cleaned_texts_chunk_{i + 1+16}.csv")
        chunk_df.to_csv(output_csv, index=False, escapechar='\\')

        print(f"Saved chunk {i + 1} to {output_csv}")

categories = ['Artificial Intelligence', 'Computer Vision and Pattern
Recognition']

for category in categories:
    print(f"Processing category: {category}")
    fetch_full_texts_by_category(cs_papers, category, chunk_size=50)
    print(f"Finished processing category: {category}")
```

```python
categories2    =    ['General    Literature',    'Operating    Systems',
'Programming Languages']

for category in categories2 :
    print(f"Processing category: {category}")
    fetch_full_texts_by_category(cs_papers, category, chunk_size=50)
    print(f"Finished processing category: {category}")

categories3  =  ['Computer  Vision  and  Pattern  Recognition',  'Data
Structures and Algorithms']

for category in categories3 :
    print(f"Processing category: {category}")
    fetch_full_texts_by_category(cs_papers, category, chunk_size=50)
    print(f"Finished processing category: {category}")

fetch_full_texts_by_category(cs_papers,      'Machine      Learning',
chunk_size=50)

categories = ['Artificial Intelligence', 'Computer Vision and Pattern
Recognition']

cs_ai  =  cs_papers[cs_papers.primary_category_full  ==  'Artificial
Intelligence']

cs_ai_shape = cs_ai.shape
ai_index = cs_ai_shape[0]- 800
cs_ai_tail = cs_ai.tail(ai_index)
cs_ai_tail

for category in categories:
    print(f"Processing category: {category}")
    fetch_full_texts_by_category(cs_ai_tail, category, chunk_size=50)
    print(f"Finished processing category: {category}")

def   fetch_and_clean_full_text(paper_id,   abstract,   category_dir,
retries=5, delay=10):
    metadata = fetch_metadata(paper_id, retries, delay)
    if metadata:
        pdf_url = parse_pdf_url(metadata)
        if pdf_url:
            for attempt in range(retries):
                try:
                    response = requests.get(pdf_url, timeout=20)
                    if response.status_code == 200:
                              pdf_path  =  os.path.join(category_dir,
f"pdfs/{paper_id}.pdf")
                        with open(pdf_path, 'wb') as f:
                            f.write(response.content)

                        try:
                            # Using PyMuPDF for text extraction
                            doc = fitz.open(pdf_path)
                            full_text = ""
                            for page in doc:
                                full_text += page.get_text()

                           full_text_cleaned = clean_text(full_text)
                                               full_text_cleaned   =
remove_abstract_from_full_text(full_text_cleaned, abstract)

                            if full_text_cleaned:
                               txt_path = os.path.join(category_dir,
f"texts/{paper_id}.txt")
                              with open(txt_path, 'w', encoding='utf-
8') as f:
                                    f.write(full_text_cleaned)
                                return full_text_cleaned
                        except Exception as e:
                              print(f"Error  for  paper_id  {paper_id}:
{e}")
                            return None
                    elif response.status_code == 500:
                      print(f"HTTP 500 error for paper_id {paper_id},
retrying...")
                        time.sleep(delay)
                    else:
                           print(f"Failed  to  fetch  PDF  for  paper_id
{paper_id}, Status code: {response.status_code}")
                        return None
                except requests.exceptions.RequestException as e:
                   print(f"Attempt {attempt + 1} to fetch PDF failed:
{e}")
```

```
                    if attempt < retries - 1:
                        time.sleep(delay)
                    else:
                            print(f"Failed to fetch PDF for paper_id
{paper_id} after {retries} attempts.")
                        return None
    return None

print(f"Processing category: Networking and Internet Architecture")
fetch_full_texts_by_category(cs_papers,  'Networking  and  Internet
Architecture', chunk_size=50)
print(f"Finished  processing  category:  Networking  and  Internet
Architecture")
```

```
#!/usr/bin/env python
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud


def read_all_data(base_dir='full'):
    all_dfs = []

    # Iterate through each category directory
    for category in os.listdir(base_dir):
        category_dir = os.path.join(base_dir, category, 'chunks')
        if os.path.isdir(category_dir):
           # Iterate through each CSV file in the chunks subdirectory
            for file_name in os.listdir(category_dir):
                if file_name.endswith('.csv'):
                    file_path = os.path.join(category_dir, file_name)
                    df = pd.read_csv(file_path)
                    all_dfs.append(df)

    # Concatenate all DataFrames into one
    combined_df = pd.concat(all_dfs, ignore_index=True)

    return combined_df
```

```python
# Use the function to read all data into one DataFrame
combined_df = read_all_data()




combined_df


# Display the combined DataFrame
print(combined_df.head())
print(combined_df.info())




# Check for missing values
print(combined_df.isnull().sum())

# Ensure that the necessary columns are present
expected_columns  =  ['paper_id',  'abstract',  'cleaned_full_text',
'cleaned_abstract', 'category']
missing_columns = set(expected_columns) - set(combined_df.columns)
if missing_columns:
    print(f"Missing columns: {missing_columns}")
else:
    print("All expected columns are present.")

# Sample some rows to ensure data is correctly formatted
print(combined_df.sample(5))


#  Adding  the  category  column  based  on  primary_category_full  or
primary_category
combined_df['category'] = combined_df['primary_category_full']

# Drop rows with missing cleaned_full_text
cleaned_combined_df                                                  =
combined_df.dropna(subset=['cleaned_full_text'])



# Display the updated DataFrame
```

```
print(cleaned_combined_df.head())
print(cleaned_combined_df.info())

# Verify data integrity again
print(cleaned_combined_df.isnull().sum())
print(cleaned_combined_df.sample(5))


# Confirm all expected columns are present
expected_columns = ['paper_id', 'title', 'abstract', 'year',
'primary_category', 'categories', 'primary_category_full',
'cleaned_full_text', 'cleaned_abstract', 'category']
missing_columns = set(expected_columns) -
set(cleaned_combined_df.columns)
if missing_columns:
    print(f"Missing columns: {missing_columns}")
else:
    print("All expected columns are present.")

# Check for any remaining missing values
print(cleaned_combined_df.isnull().sum())

# Display summary information
print(cleaned_combined_df.info())
print(cleaned_combined_df.describe(include='all'))

# Sample some rows to verify data integrity
print(cleaned_combined_df.sample(5))


# Save the final DataFrame to a CSV file
cleaned_combined_df.to_csv('final_combined_dataset.csv',
index=False, escapechar='\\')

# Assuming cleaned_combined_df is your DataFrame

# 1. Distribution of Papers Across Categories
plt.figure(figsize=(12, 6))
sns.countplot(data=cleaned_combined_df, x='category',
order=cleaned_combined_df['category'].value_counts().index)
plt.title('Distribution of Papers Across Categories')
plt.xlabel('Category')
```

```python
plt.ylabel('Number of Papers')
plt.xticks(rotation=45)
plt.show()



# 2. Distribution of Text Lengths
def plot_length_distribution(column, title, xlabel):
    plt.figure(figsize=(12, 6))
    text_lengths = cleaned_combined_df[column].apply(lambda x:
len(x.split()))
    sns.histplot(text_lengths, bins=50, kde=True)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel('Frequency')
    plt.show()


# 3. Word Clouds
def plot_wordcloud(column, title):
    text = ' '.join(cleaned_combined_df[column])
    wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(text)
    plt.figure(figsize=(12, 6))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(title)
    plt.axis('off')
    plt.show()


# 6. Average Length of Full Texts and Abstracts by Category
def plot_average_length_by_category(column, title, ylabel):
    avg_lengths =
cleaned_combined_df.groupby('category')[column].apply(lambda x:
x.str.split().str.len().mean()).reset_index()
    plt.figure(figsize=(12, 6))
    sns.barplot(data=avg_lengths, x='category', y=column,
order=avg_lengths.sort_values(by=column,
ascending=False)['category'])
    plt.title(title)
    plt.xlabel('Category')
    plt.ylabel(ylabel)
    plt.xticks(rotation=45)
    plt.show()


def plot_boxplot_lengths(column, title, ylabel):
                        cleaned_combined_df[f'{column}_length']          =
cleaned_combined_df[column].apply(lambda x: len(x.split()))
    plt.figure(figsize=(12, 6))
                sns.boxplot(data=cleaned_combined_df,      x='category',
y=f'{column}_length')
    plt.title(title)
    plt.xlabel('Category')
    plt.ylabel(ylabel)
    plt.xticks(rotation=45)
    plt.show()


plot_length_distribution('cleaned_abstract',     'Distribution     of
Abstract Lengths', 'Number of Words in Abstract')


plot_length_distribution('cleaned_full_text', 'Distribution of Full
Text Lengths', 'Number of Words in Full Text')


plot_wordcloud('cleaned_full_text', 'Word Cloud for Full Texts')


plot_wordcloud('cleaned_abstract', 'Word Cloud for Abstracts')


plot_average_length_by_category('cleaned_abstract',  'Average  Length
of Abstracts by Category', 'Average Number of Words')


plot_average_length_by_category('cleaned_full_text', 'Average Length
of Full Texts by Category', 'Average Number of Words')


plot_boxplot_lengths('cleaned_full_text',  'Box  Plot  of  Full  Text
Lengths by Category', 'Number of Words in Full Text')


plot_boxplot_lengths('cleaned_abstract',   'Box   Plot   of   Abstract
Lengths by Category', 'Number of Words in Abstract')
```

```
cleaned_combined_df=
cleaned_combined_df[cleaned_combined_df["category"]  ==  "Networking
and Internet Architecture"]


# Select 10 papers for case testing
case_test_df = cleaned_combined_df.sample(n=10)


# Display the selected papers for case testing
case_test_df


# Exclude the case test papers from the main dataset
remaining_df = cleaned_combined_df.drop(case_test_df.index)

# Display the size of the remaining dataset
print(f"Remaining dataset: {len(remaining_df)} samples")


# Save the case test papers to a CSV file for documentation
case_test_df.to_csv('case_test_papers.csv', index=False)

print("Case test papers have been saved to 'case_test_papers.csv'.")


# Save the case test papers to a CSV file for documentation
remaining_df.to_csv('remaining_papers.csv', index=False)

print("remaining papers have been saved to 'remaining_df.csv'.")
```

```
# -*- coding: utf-8 -*-
"""notebook four.ipynb

Automatically generated by Colab.

Original file is located at
```

```
    https://colab.research.google.com/drive/1xUO9rzv8g55da7Y3QlCiGyd
JlsYyJhV2

# setup
"""

# install Hugging Face Libraries
!pip install "peft==0.2.0"
!pip install "transformers==4.27.2" "datasets==2.9.0"
"accelerate==0.17.1" "evaluate==0.4.0" "bitsandbytes==0.37.1" loralib
--upgrade --quiet
# install additional dependencies needed for training
!pip install rouge-score tensorboard py7zr

!pip uninstall -y bitsandbytes
!pip install bitsandbytes

"""# Imports"""

# Google Colab integration
from google.colab import drive

# Data manipulation and processing
import pandas as pd
import numpy as np

# Dataset loading and handling
from datasets import load_dataset, Dataset, DatasetDict,
concatenate_datasets, load_from_disk

# TensorFlow library
import tensorflow as tf

# Hugging Face Transformers for model and tokenizer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from transformers import DataCollatorForSeq2Seq
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments

# Parameter-efficient fine-tuning (PEFT) for transformer models
from peft import LoraConfig, get_peft_model,
prepare_model_for_int8_training, TaskType
```

```
# Bits and Bytes library for efficient model training
import bitsandbytes as bnb


import os
import evaluate
import torch

"""# Set The Enviroment"""

# utlize 100 % of the gpu
!nvidia-smi -L

gpu_devices = tf.config.list_physical_devices('GPU')
for device in gpu_devices:
    tf.config.experimental.set_memory_growth(device, True)

"""Conccet to the drive"""

# conect to the drive
drive.mount('/content/drive')

# Commented out IPython magic to ensure Python compatibility.
# move to the Working directory
# %cd drive/MyDrive/FYP/experiment1

"""# Load dataset"""

df = pd.read_csv('remaining_papers.csv')

df.info()

keep = ['title','abstract', 'cleaned_full_text','category']

df = df[keep]

dataset = Dataset.from_pandas(df)

dataset

# Split the data into training and testing sets (80% train, 20% test)
train_test_split = dataset.train_test_split(test_size=0.15)
```

```
train_test_split

# Combine the splits into a single DatasetDict
dataset = DatasetDict({
    'train': train_test_split['train'],
    'test': train_test_split['test']
})

# Inspect the DatasetDict
print(dataset)

print(f"Train dataset size: {len(dataset['train'])}")
print(f"Test dataset size: {len(dataset['test'])}")

dataset["train"].save_to_disk("data/datset-train/")
dataset["test"].save_to_disk("data/dataset-test/")

"""# modeling"""

model_id="google/flan-t5-xxl"

# Load tokenizer of FLAN-t5-XXL
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Combine train and test datasets and tokenize the 'abstract' column
tokenized_inputs      =      concatenate_datasets([dataset["train"],
dataset["test"]]).map(
    lambda x: tokenizer(x["cleaned_full_text"], truncation=True),
    batched=True,
        remove_columns=["title",   "abstract",   "cleaned_full_text",
"category"]
)

# Calculate the lengths of tokenized input sequences
input_lengths = [len(x) for x in tokenized_inputs["input_ids"]]

# Determine the 85th percentile length for maximum source length
max_source_length = int(np.percentile(input_lengths, 85))
print(f"Max source length: {max_source_length}")

# Combine train and test datasets and tokenize the 'cleaned_full_text'
column
```

```
tokenized_targets      =      concatenate_datasets([dataset["train"],
dataset["test"]]).map(
    lambda x: tokenizer(x["abstract"], truncation=True),
    batched=True,
        remove_columns=["title",   "abstract",   "cleaned_full_text",
"category"]
)

# Calculate the lengths of tokenized target sequences
target_lengths = [len(x) for x in tokenized_targets["input_ids"]]

# Determine the 90th percentile length for maximum target length
max_target_length = int(np.percentile(target_lengths, 90))
print(f"Max target length: {max_target_length}")

"""testing long text"""

def preprocess_function(sample, padding="max_length"):
    # Add prefix to the input for T5
        inputs    =    ["summarize:    "    +    item    for    item    in
sample["cleaned_full_text"]]

    # Tokenize inputs
     model_inputs  =  tokenizer(inputs,  max_length=max_source_length,
padding=padding, truncation=True)

    # Tokenize targets with the `text_target` keyword argument
           labels     =     tokenizer(text_target=sample["abstract"],
max_length=max_target_length, padding=padding, truncation=True)

    # If we are padding here, replace all tokenizer.pad_token_id in
the labels by -100 when we want to ignore
    # padding in the loss.
    if padding == "max_length":
        labels["input_ids"] = [
            [(l if l != tokenizer.pad_token_id else -100) for l in
label] for label in labels["input_ids"]
        ]

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs
```

```
# Assuming you have defined max_source_length and max_target_length
earlier
tokenized_dataset = dataset.map(preprocess_function, batched=True,
remove_columns=["title", "abstract", "cleaned_full_text",
"category"])
print(f"Keys of tokenized dataset:
{list(tokenized_dataset['train'].features)}")

# Save datasets to disk for later easy loading
tokenized_dataset["train"].save_to_disk("data/train")
tokenized_dataset["test"].save_to_disk("data/test")
tokenized_dataset.save_to_disk("data/full")

"""# read tokenized data"""

# Load the tokenized datasets from disk
tokenized_train_dataset = load_from_disk("data/train")
tokenized_test_dataset = load_from_disk("data/test")

# Verify the loaded datasets
print(tokenized_train_dataset)
print(tokenized_test_dataset)

temp = tokenized_train_dataset.train_test_split(test_size=0.15)
tokenized_train_dataset = temp['train']
tokenized_validation_dataset = temp['test']

# Verify the loaded datasets
print(tokenized_train_dataset)
print(tokenized_validation_dataset)
print(tokenized_test_dataset)

"""# Fine tuning"""

# huggingface hub model id
model_id = "philschmid/flan-t5-xxl-sharded-fp16"

# load model from the hub
model = AutoModelForSeq2SeqLM.from_pretrained(model_id,
load_in_8bit=True, device_map="auto")

# Define LoRA Config
```

```
lora_config = LoraConfig(
 r=16,
 lora_alpha=32,
 target_modules=["q", "v"],
 lora_dropout=0.05,
 bias="none",
 task_type=TaskType.SEQ_2_SEQ_LM
)
# prepare int-8 model for training
model = prepare_model_for_int8_training(model)

# add LoRA adaptor
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()

# we want to ignore tokenizer pad token in the loss
label_pad_token_id = -100
# Data collator
data_collator = DataCollatorForSeq2Seq(
    tokenizer,
    model=model,
    label_pad_token_id=label_pad_token_id,
    pad_to_multiple_of=8
)

output_dir = "lora-flan-t5-xxl"

latest_checkpoint = None

if os.path.exists(output_dir):
     checkpoints = [folder for folder in os.listdir(output_dir) if
folder.startswith('checkpoint-')]
    if checkpoints:
             latest_checkpoint   =  max(checkpoints,  key=lambda  x:
int(x.split('-')[1]))
                   latest_checkpoint    =    os.path.join(output_dir,
latest_checkpoint)

latest_checkpoint

# Define training args
```

```python
training_args = Seq2SeqTrainingArguments(
    output_dir=output_dir,
    auto_find_batch_size=True,
    learning_rate=1e-5, # higher learning rate
    num_train_epochs=3, # reduce epochs
    logging_dir=f"{output_dir}/logs",
    logging_strategy="steps",
    logging_steps=500,  # Save checkpoint every 500 steps
    save_strategy="steps",
    save_steps=500,  # Save checkpoint every 500 steps
    save_total_limit=2,  # Keep only the last 2 checkpoints
    report_to="tensorboard",
    fp16=True, # use mixed precision
    optim="adamw_torch",  # Use PyTorch's AdamW
    resume_from_checkpoint=latest_checkpoint,
    overwrite_output_dir=True

)

# Create Trainer instance
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=tokenized_train_dataset,
    eval_dataset=tokenized_validation_dataset,
)
model.config.use_cache = False  # silence the warnings. Please re-
enable for inference!

# Commented out IPython magic to ensure Python compatibility.
# %load_ext tensorboard
# %tensorboard --logdir lora-flan-t5-xxl/logs

# train model
trainer.train()

trainer.train()

# Save the final model
trainer.save_model(os.path.join(output_dir, "final_model"))
```

```
# Save the config file
model.config.save_pretrained(os.path.join(output_dir,
"final_model"))

# Save the LoRA adapter separately
model.save_pretrained(os.path.join(output_dir, "lora_adapter"))

# Save the tokenizer
tokenizer.save_pretrained(os.path.join(output_dir, "tokenizer"))

tokenizer.save_pretrained(os.path.join(output_dir, "tokenizer"))

final_model_dir = os.path.join(output_dir, "final_model")
lora_adapter_dir = os.path.join(output_dir, "lora_adapter")
tokenizer_dir = os.path.join(output_dir, "tokenizer")


# List contents of the directories to verify
print("Contents of final_model directory:")
print(os.listdir(final_model_dir))

print("\nContents of lora_adapter directory:")
print(os.listdir(lora_adapter_dir))

print("\nContents of tokenizer directory:")
print(os.listdir(output_dir))
```

```
# -*- coding: utf-8 -*-
"""notebook five.ipynb

Automatically generated by Colab.

Original file is located at
               https://colab.research.google.com/drive/1Z_CQuzS0j7c-
ef9zgCOfWF2bNT4syWoN
"""

# install Hugging Face Libraries
!pip install "peft==0.2.0"
```

```
!pip      install      "transformers==4.27.2"      "datasets==2.9.0"
"accelerate==0.17.1" "evaluate==0.4.0" "bitsandbytes==0.37.1" loralib
--upgrade --quiet
# install additional dependencies needed for training
!pip install rouge-score tensorboard py7zr

!pip uninstall -y bitsandbytes
!pip install bitsandbytes

# Google Colab integration
from google.colab import drive

# Data manipulation and processing
import pandas as pd
import numpy as np

# Dataset loading and handling
from    datasets    import    load_dataset,    Dataset,    DatasetDict,
concatenate_datasets, load_from_disk

# TensorFlow library
import tensorflow as tf

# Hugging Face Transformers for model and tokenizer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from transformers import DataCollatorForSeq2Seq
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments

# Parameter-efficient fine-tuning (PEFT) for transformer models
from       peft       import       LoraConfig,       get_peft_model,
prepare_model_for_int8_training, TaskType

# Bits and Bytes library for efficient model training
import bitsandbytes as bnb


import os
import evaluate
import torch

import torch
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
from peft import PeftModel, PeftConfig
from google.colab import drive

import evaluate
import numpy as np
from datasets import load_from_disk
from tqdm import tqdm
import matplotlib.pyplot as plt

from datasets import load_dataset
from random import randrange
import pandas as pd
import seaborn as sns

# utlize 100 % of the gpu
!nvidia-smi -L

gpu_devices = tf.config.list_physical_devices('GPU')
for device in gpu_devices:
    tf.config.experimental.set_memory_growth(device, True)

# conect to the drive
drive.mount('/content/drive')

# huggingface hub model id
model_id = "philschmid/flan-t5-xxl-sharded-fp16"

# load model from the hub
model        =        AutoModelForSeq2SeqLM.from_pretrained(model_id,
load_in_8bit=True, device_map="auto")

# Load tokenizer

tokenizer = AutoTokenizer.from_pretrained(model_id)
print("Base model loaded from Hugging Face")

adapter_model_path                                                   =
"/content/drive/MyDrive/FYP/experiment1/results/lora_adapter"

# Load the LoRA adapter
model = PeftModel.from_pretrained(model, adapter_model_path)
```

```
model.eval()

print("LoRA adapter added to the model successfully")

test_dataset                                                            =
load_from_disk("/content/drive/MyDrive/FYP/experiment1/data/dataset-
test")

test_dataset

# Load dataset from the hub and get a sample
sample = test_dataset[randrange(len(test_dataset)+1)]

input_ids          =          tokenizer(sample["cleaned_full_text"],
return_tensors="pt", truncation=True).input_ids.cuda()
# with torch.inference_mode():
outputs   =  model.generate(input_ids=input_ids,  max_new_tokens=150,
min_length=40,  do_sample=True, top_p=0.9)
print(f"input paper: {sample['cleaned_full_text']}\n{'---'* 20}")

print(f"summary:\n{tokenizer.batch_decode(outputs.detach().cpu().num
py(), skip_special_tokens=True)[0]}")



# load test dataset from distk
test_dataset                                                            =
load_from_disk("/content/drive/MyDrive/FYP/experiment1/data/test/").
with_format("torch")

test_dataset

sample_test = test_dataset.train_test_split(test_size=0.01)
test_dataset = sample_test['test']

test_dataset

# Metric
metric = evaluate.load("rouge")

def evaluate_peft_model(sample,max_target_length=50):
    # generate summary
```

```
                                                  outputs              =
model.generate(input_ids=sample["input_ids"].unsqueeze(0).cuda(),
do_sample=True, top_p=0.9, max_new_tokens=max_target_length)
    prediction = tokenizer.decode(outputs[0].detach().cpu().numpy(),
skip_special_tokens=True)
    # decode eval sample
    # Replace -100 in the labels as we can't decode them.
     labels  =  np.where(sample['labels']  !=  -100,  sample['labels'],
tokenizer.pad_token_id)
    labels = tokenizer.decode(labels, skip_special_tokens=True)

    # Some simple post-processing
    return prediction, labels

# run predictions
predictions, references = [] , []
for sample in tqdm(test_dataset):
    p,l = evaluate_peft_model(sample)
    predictions.append(p)
    references.append(l)

# compute metric
rogue            =            metric.compute(predictions=predictions,
references=references, use_stemmer=True)

# print results
print(f"Rogue1: {rogue['rouge1']* 100:2f}%")
print(f"rouge2: {rogue['rouge2']* 100:2f}%")
print(f"rougeL: {rogue['rougeL']* 100:2f}%")
print(f"rougeLsum: {rogue['rougeLsum']* 100:2f}%")

import evaluate
import numpy as np
from datasets import load_from_disk
from tqdm import tqdm
import matplotlib.pyplot as plt


# Metric
metric = evaluate.load("rouge")

def evaluate_peft_model(sample, max_target_length, min_length):
    # generate summary
    outputs = model.generate(
        input_ids=sample["input_ids"].unsqueeze(0).cuda(),
        do_sample=True,
        top_p=0.9,
        max_new_tokens=max_target_length,
        min_length=min_length
    )
    prediction = tokenizer.decode(outputs[0].detach().cpu().numpy(),
skip_special_tokens=True)

    # decode eval sample
    # Replace -100 in the labels as we can't decode them.
     labels  =  np.where(sample['labels']  !=  -100,  sample['labels'],
tokenizer.pad_token_id)
    labels = tokenizer.decode(labels, skip_special_tokens=True)

    # Some simple post-processing
    return prediction, labels

def run_evaluation(test_dataset, max_lengths, min_lengths):
    results = []

    for max_len in max_lengths:
        for min_len in min_lengths:
            predictions, references = [], []
            for sample in tqdm(test_dataset):
                p, l = evaluate_peft_model(sample, max_len, min_len)
                predictions.append(p)
                references.append(l)
                   rouge   =   metric.compute(predictions=predictions,
references=references, use_stemmer=True)
            results.append({
                'max_length': max_len,
                'min_length': min_len,
                'rouge1': rouge['rouge1'] * 100,
                'rouge2': rouge['rouge2'] * 100,
                'rougeL': rouge['rougeL'] * 100,
                'rougeLsum': rouge['rougeLsum'] * 100,
            })
            print(f"Max length: {max_len}, Min length: {min_len}")
    return results

# Define the ranges for max_target_length and min_length
max_lengths = [50, 100, 150, 200]
min_lengths = [10, 20, 30, 40]

# Run the evaluation
results = run_evaluation(test_dataset, max_lengths, min_lengths)

# Convert results to a DataFrame
df = pd.DataFrame(results)

# Plotting
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

sns.lineplot(ax=axes[0,  0],  data=df,  x='max_length',  y='rouge1',
hue='min_length', marker='o')
axes[0, 0].set_title('Rouge-1 Score')

sns.lineplot(ax=axes[0,  1],  data=df,  x='max_length',  y='rouge2',
hue='min_length', marker='o')
axes[0, 1].set_title('Rouge-2 Score')

sns.lineplot(ax=axes[1,  0],  data=df,  x='max_length',  y='rougeL',
hue='min_length', marker='o')
axes[1, 0].set_title('Rouge-L Score')

sns.lineplot(ax=axes[1, 1], data=df, x='max_length', y='rougeLsum',
hue='min_length', marker='o')
axes[1, 1].set_title('Rouge-Lsum Score')

plt.tight_layout()
plt.show()



test_case                                                              =
pd.read_csv("/content/drive/MyDrive/FYP/experiment1/case_test/case_t
est_papers.csv")

keep = ['title','abstract', 'cleaned_full_text','category']
```

```
test_case = test_case[keep]

test_case_dataset = Dataset.from_pandas(test_case)

test_case_dataset

# Assuming you have defined max_source_length and max_target_length
earlier
tokenized_test_case_dataset                                                =
test_case_dataset.map(preprocess_function,              batched=True,
remove_columns=["title",       "abstract",       "cleaned_full_text",
"category"])
print(f"Keys            of            tokenized            dataset:
{list(tokenized_test_case_dataset.features)}")

# Metric
metric = evaluate.load("rouge")

def     test_case_evaluate_peft_model(sample,     max_target_length,
min_length):
    # generate summary
            inputs      =      tokenizer(sample["cleaned_full_text"],
return_tensors="pt",      truncation=True,      padding="max_length",
max_length=512).input_ids.cuda()
    outputs = model.generate(
        input_ids=inputs,
        do_sample=True,
        top_p=0.9,
        max_new_tokens=max_target_length,
        min_length=min_length
    )
    prediction = tokenizer.decode(outputs[0].detach().cpu().numpy(),
skip_special_tokens=True)

    # Reference (label) is the 'abstract' field
      labels  =  tokenizer(sample['abstract'],  return_tensors="pt",
truncation=True,                              padding="max_length",
max_length=512).input_ids.numpy()
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    labels = tokenizer.decode(labels[0], skip_special_tokens=True)

    return prediction, labels
```

```python
def test_case_evaluation(test_dataset, max_lengths, min_lengths):
    results = []

    for max_len in max_lengths:
        for min_len in min_lengths:
            predictions, references = [], []
            for sample in tqdm(test_dataset):
                p, l = evaluate_peft_model(sample, max_len, min_len)
                predictions.append(p)
                references.append(l)
                   rouge   =   metric.compute(predictions=predictions,
references=references, use_stemmer=True)
            results.append({
                'max_length': max_len,
                'min_length': min_len,
                'rouge1': rouge['rouge1'] * 100,
                'rouge2': rouge['rouge2'] * 100,
                'rougeL': rouge['rougeL'] * 100,
                'rougeLsum': rouge['rougeLsum'] * 100,
            })

    return results

# Define the ranges for max_target_length and min_length
max_lengths = [50, 100, 200,300,350]
min_lengths = [10, 30, 50]

# Run the evaluation
test_case_results      =      test_case_evaluation(test_case_dataset,
max_lengths, min_lengths)

# Convert results to a DataFrame
test_case_df = pd.DataFrame(test_case_results)

# Plotting
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

sns.lineplot(ax=axes[0,   0],   data=test_case_df,   x='max_length',
y='rouge1', hue='min_length', marker='o')
axes[0, 0].set_title('Rouge-1 Score')
```

```
sns.lineplot(ax=axes[0,   1],   data=test_case_df,   x='max_length',
y='rouge2', hue='min_length', marker='o')
axes[0, 1].set_title('Rouge-2 Score')

sns.lineplot(ax=axes[1,   0],   data=test_case_df,   x='max_length',
y='rougeL', hue='min_length', marker='o')
axes[1, 0].set_title('Rouge-L Score')

sns.lineplot(ax=axes[1,   1],   data=test_case_df,   x='max_length',
y='rougeLsum', hue='min_length', marker='o')
axes[1, 1].set_title('Rouge-Lsum Score')

plt.tight_layout()
plt.show()

test_case

# prompt: save the each abstracts from test_case  in txt file and name
the file by title

import os

# Create a directory to store the abstracts
if                                                                  not
os.path.exists("/content/drive/MyDrive/FYP/experiment1/case_test/abs
tracts"):
    os.makedirs("/content/drive/MyDrive/FYP/experiment1/case_test/ab
stracts")

# Iterate through the test_case dataset
for i, sample in enumerate(test_case_dataset):
    # Extract the title and abstract
    title = sample["title"]
    abstract = sample["abstract"]

    # Remove any invalid characters from the title for the file name
                                      valid_chars                  =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_-"
    title = "".join([c for c in title if c in valid_chars])

    # Create the file name
```

```
                                        file_name                      =
f"/content/drive/MyDrive/FYP/experiment1/case_test/abstracts/{title}
.txt"

    # Write the abstract to the file
    with open(file_name, "w") as f:
        f.write(abstract)

    # Print a message
    print(f"Saved abstract for '{title}' to '{file_name}'")

# prompt: genrate the summry for these 10 papers and save it as a new
cloumn named genrated abs in df

def        summarize_papers(test_case_dataset,        max_length=300,
min_length=50):
  summaries = []
  for sample in tqdm(test_case_dataset):
             inputs     =     tokenizer(sample["cleaned_full_text"],
return_tensors="pt",     truncation=True,     padding="max_length",
max_length=512).input_ids.cuda()
    outputs = model.generate(
        input_ids=inputs,
        do_sample=True,
        top_p=0.9,
        max_new_tokens=max_length,
        min_length=min_length
    )
     summaries.append(tokenizer.decode(outputs[0].detach().cpu().nump
y(), skip_special_tokens=True))
  return summaries

test_case["genrated abs"] = summarize_papers(test_case_dataset)

print(test_case)

import os
from datasets import Dataset

# Create a directory to store the abstracts
```

```
save_dir                                                                   =
"/content/drive/MyDrive/FYP/experiment1/case_test/genrated_abstracts
"
os.makedirs(save_dir, exist_ok=True)

# Convert test_case to Dataset
saving_data = Dataset.from_pandas(test_case)

# Iterate through the test_case dataset
for i, sample in enumerate(saving_data):
    try:
        # Extract the title and abstract
        title = sample["title"]
        abstract = sample["genrated abs"]

        # Remove any invalid characters from the title for the file
name
                                              valid_chars            =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_-"
        title = "".join([c for c in title if c in valid_chars])

        # Truncate title if too long
        max_title_length = 50  # Adjust this value as needed
        if len(title) > max_title_length:
            title = title[:max_title_length]

        # Create the file name
        file_name = os.path.join(save_dir, f"{title}.txt")

        # Write the abstract to the file with utf-8 encoding
        with open(file_name, "w", encoding="utf-8") as f:
            f.write(abstract)

        # Print a message
             print(f"Saved  generated  abstract  for  '{title}'  to
'{file_name}'")
    except Exception as e:
          print(f"Error  saving  abstract  for  '{sample.get('title',
'unknown')}': {e}")
```

```
# -*- coding: utf-8 -*-
"""notebook six.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1urVG6w2cciw2LJS0Jw1txur
2X-br2-Zz
"""

# Install Ngrok
!pip install pyngrok flask flask-ngrok
# install Hugging Face Libraries
!pip install "peft==0.2.0"
!pip      install      "transformers==4.27.2"      "datasets==2.9.0"
"accelerate==0.17.1" "evaluate==0.4.0" "bitsandbytes==0.37.1" loralib
--upgrade --quiet
# install additional dependencies needed for training
!pip install rouge-score tensorboard py7zr

!pip uninstall -y bitsandbytes
!pip install bitsandbytes

import torch
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from peft import PeftModel, PeftConfig
from flask import Flask, request, jsonify, render_template_string
from flask_ngrok import run_with_ngrok
from google.colab import drive
import os
from pyngrok import ngrok

# Mount Google Drive
drive.mount('/content/drive')

model_id="google/flan-t5-xxl"

# Load tokenizer of FLAN-t5-XXL
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

```
# huggingface hub model id
model_id = "philschmid/flan-t5-xxl-sharded-fp16"

# load model from the hub
model          =          AutoModelForSeq2SeqLM.from_pretrained(model_id,
load_in_8bit=True, device_map="auto")

adapter_model_path                                                       =
"/content/drive/MyDrive/FYP/experiment1/results/lora_adapter"

# Load the LoRA adapter
#model = PeftModel.from_pretrained(model, adapter_model_path)

# Set your Ngrok auth token
NGROK_AUTH_TOKEN                                                         =
"2iVxc43VumlKuTemHnb1sNj0Kii_6bYh5tZpmuADmvWefgaBK"   #  Update  this
with your Ngrok auth token
ngrok.set_auth_token(NGROK_AUTH_TOKEN)

app = Flask(__name__)
run_with_ngrok(app)

@app.route('/')
def home():
                                                                    return
render_template_string(open('/content/drive/MyDrive/FYP/experiment1/
host/index_dark.html').read())

@app.route('/summarize', methods=['POST'])
def summarize():
  text = request.json['text']
   input_ids  =  tokenizer("summrize:  "  +  text,  return_tensors="pt",
truncation=True).input_ids.cuda()
   outputs  =  model.generate(input_ids=input_ids,  max_new_tokens=300,
min_length=50, do_sample=True, top_p=0.9)
    summary  =  tokenizer.batch_decode(outputs.detach().cpu().numpy(),
skip_special_tokens=True)[0]
  return jsonify({'summary': summary})



@app.route('/upload', methods=['POST'])
```

```python
def upload_file():
    file = request.files['file']
    if file and file.filename.endswith('.txt'):
        text = file.read().decode('utf-8')
        return jsonify({'text': text})
    else:
        return jsonify({'error': 'Unsupported file type. Please upload a .txt file.'}), 400

# Run the Flask app
if __name__ == "__main__":
    # Setup ngrok

    # Open a ngrok tunnel to the public internet
    public_url = ngrok.connect(5000)
            print(" * ngrok tunnel \"{}\" -> \"http://127.0.0.1:5000\"".format(public_url))

    app.run()
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">


    <!-- Include pdfjs-dist for PDF extraction -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/pdf.js/2.9.359/pdf.min.js"></script>

    <!-- Include mammoth for Word document extraction -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/mammoth/1.4.21/mammoth.browser.min.js"></script>

    <title>Scientific AI Summarizer</title>
    <style>
                                                                        @import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;70
0&display=swap');

        :root {
            --primary-color: rgb(90, 78, 188);
            --primary-light: rgb(90, 78, 188);
            --primary-dark: #0a00b6;
            --bg-color: #f5f5f5;
            --text-color: #333;
            --input-bg: #ffffff;
            --border-color: #e0e0e0;
            --success-color: #00c853;
            --error-color: #d50000;
            --bg-color-dark: #121212;
            --text-color-dark: #f5f5f5;
            --input-bg-dark: #1e1e1e;
            --border-color-dark: #333333;
        }

        body {
            font-family: 'Roboto', sans-serif;
            background-color: var(--bg-color);
            color: var(--text-color);
            margin: 0;
            padding: 20px;
            line-height: 1.6;
            transition: background-color 0.3s, color 0.3s;
        }

        body.dark-mode {
            background-color: var(--bg-color-dark);
            color: var(--text-color-dark);
        }

        .container {
            max-width: 800px;
            margin: 40px auto;
            background-color: var(--input-bg);
            border-radius: 12px;
            padding: 40px;
            box-shadow: 0 10px 30px rgba(0, 0, 0, 0.1);
            transition: all 0.3s ease;
            transform: translateY(0);
        }

        .container:hover {
            transform: translateY(-5px);
            box-shadow: 0 15px 35px rgba(0, 0, 0, 0.15);
        }

        .container.dark-mode {
            background-color: var(--input-bg-dark);
            border-color: var(--border-color-dark);
        }

        h1 {
            text-align: center;
            font-size: 2.5em;
            margin-bottom: 30px;
            color: var(--primary-color);
            text-transform: uppercase;
            letter-spacing: 2px;
            animation: fadeInDown 1s ease-out;
        }

        @keyframes fadeInDown {
            from {
                opacity: 0;
                transform: translateY(-20px);
            }
            to {
                opacity: 1;
                transform: translateY(0);
            }
        }

        .input-area, .output-area {
            margin-bottom: 30px;
            animation: fadeIn 1s ease-out;
        }

        @keyframes fadeIn {
            from { opacity: 0; }
            to { opacity: 1; }
        }

        h2 {
            font-size: 1.5em;
            margin-bottom: 15px;
            color: var(--primary-color);
            border-bottom: 2px solid var(--primary-light);
            padding-bottom: 5px;
        }

        textarea {
            width: 96%;
            height: 150px;
            background-color: var(--input-bg);
            color: var(--text-color);
            border: 2px solid var(--border-color);
            border-radius: 8px;
            padding: 15px;
            font-size: 16px;
            margin-bottom: 15px;
            transition: all 0.3s ease;
            box-shadow: inset 0 1px 3px rgba(0, 0, 0, 0.1);
        }

        .dark-mode textarea {
            background-color: var(--input-bg-dark);
            color: var(--text-color-dark);
            border-color: var(--border-color-dark);
        }

        textarea, .drag-drop-area, .button {
             transition: background-color 0.3s, color 0.3s, border-
color 0.3s;
        }
         .dark-mode textarea, .dark-mode .drag-drop-area, .dark-mode
.button {
            background-color: var(--input-bg-dark);
            color: var(--text-color-dark);
            border-color: var(--border-color-dark);
        }
```

```
textarea:focus {
    outline: none;
    border-color: var(--primary-color);
    box-shadow: 0 0 0 2px var(--primary-light);
}

button {
    background-color: var(--primary-color);
    color: white;
    padding: 12px 24px;
    border: none;
    border-radius: 25px;
    cursor: pointer;
    font-size: 16px;
    font-weight: bold;
    text-transform: uppercase;
    letter-spacing: 1px;
    transition: all 0.3s ease;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.2);
}

button:hover {
    background-color: var(--primary-light);
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    transform: translateY(-2px);
}

button:active {
    transform: translateY(0);
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);
}

.button-group {
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin-bottom: 20px;
}

#file-info {
```

```css
        color: var(--text-color);
        font-style: italic;
        margin-left: 15px;
        opacity: 0.7;
    }

    .drag-drop-area {
        border: 2px dashed var(--border-color);
        border-radius: 8px;
        padding: 30px;
        text-align: center;
        margin-bottom: 20px;
        transition: all 0.3s ease;
        background-color: rgba(98, 0, 234, 0.05);
        cursor: pointer;
    }

    .drag-drop-area:hover {
        border-color: var(--primary-color);
        background-color: rgba(98, 0, 234, 0.1);
        transform: scale(1.02);
    }

    .drag-drop-area.dragover {
        border-color: var(--primary-color);
        background-color: rgba(98, 0, 234, 0.15);
        transform: scale(1.05);
    }

    .loading {
        display: none;
        text-align: center;
        margin-top: 20px;
        font-weight: bold;
        color: var(--primary-color);
    }

    .loading::after {
        content: "";
        animation: loading 1.5s infinite;
    }
```

```css
@keyframes loading {
    0% { content: "."; }
    33% { content: ".."; }
    66% { content: "..."; }
}

.success-message, .error-message {
    padding: 10px;
    border-radius: 5px;
    margin-top: 10px;
    font-weight: bold;
    text-align: center;
    opacity: 0;
    transition: opacity 0.3s ease;
}

.success-message {
    background-color: rgba(0, 200, 83, 0.2);
    color: var(--success-color);
}

.error-message {
    background-color: rgba(213, 0, 0, 0.2);
    color: var(--error-color);
}

.show-message {
    opacity: 1;
    animation: fadeInUp 0.5s ease-out;
}

@keyframes fadeInUp {
    from {
        opacity: 0;
        transform: translateY(20px);
    }
    to {
        opacity: 1;
        transform: translateY(0);
    }
}
```

```css
        #word-count {
            margin-top: 10px;
            font-style: italic;
            animation: fadeIn 1s ease-out;
        }

        #theme-toggle {
            position: fixed;
            top: 20px;
            right: 20px;
            z-index: 1000;
            background: none;
            border: none;
            cursor: pointer;
            font-size: 24px;
            color: var(--text-color);
            padding: 5px;
            border-radius: 50%;
            transition: all 0.3s ease;
        }

        #theme-toggle:hover {
            background-color: rgba(0, 0, 0, 0.1);
            transform: scale(1.1) rotate(15deg);
        }

        .dark-mode #theme-toggle {
            color: var(--text-color-dark);
        }

        .export-buttons {
            display: flex;
            justify-content: flex-end;
            margin-top: 10px;
        }

        .export-buttons button {
            margin-left: 10px;
        }

        .background-animation {
            position: fixed;
            top: 0;
            left: 0;
            width: 100%;
            height: 100%;
            z-index: -1;
            overflow: hidden;
        }

        .shape {
            position: absolute;
            border-radius: 50%;
            filter: blur(40px);
            opacity: 0.5;
            animation: float 15s infinite ease-in-out;
        }

        .shape:nth-child(1) {
            width: 200px;
            height: 200px;
            background-color: var(--primary-light);
            left: 10%;
            top: 20%;
        }

        .shape:nth-child(2) {
            width: 300px;
            height: 300px;
            background-color: var(--primary-color);
            right: 15%;
            bottom: 25%;
            animation-delay: -5s;
        }

        .shape:nth-child(3) {
            width: 150px;
            height: 150px;
            background-color: var(--primary-dark);
            left: 20%;
            bottom: 15%;
            animation-delay: -10s;
        }
```

```
        .shape:nth-child(4) {
            width: 200px;
            height: 200px;
            background-color: var(--primary-dark);
            left: 40%;
            bottom: 45%;
            animation-delay: -10s;
        }

        @keyframes float {
            0%, 100% { transform: translate(0, 0) rotate(0deg); }
            25% { transform: translate(50px, 50px) rotate(90deg); }
            50% { transform: translate(100px, -50px) rotate(180deg);
}
            75% { transform: translate(-50px, 100px) rotate(270deg);
}
        }

        .container {
            background-color: rgba(255, 255, 255, 0.8);
            backdrop-filter: blur(10px);
        }

        .dark-mode .container {
            background-color: rgba(30, 30, 30, 0.8);
        }
    </style>


</head>
<body class="dark-mode">
    <div class="background-animation">
        <div class="shape"></div>
        <div class="shape"></div>
        <div class="shape"></div>
        <div class="shape"></div>
    </div>
    <button id="theme-toggle" aria-label="Toggle dark mode">
        <span id="theme-toggle-icon">⚙️</span>
    </button>
    <div class="container dark-mode">
        <h1>Scientific AI Summarizer</h1>
        <div class="input-area">
            <h2>Original Text</h2>
            <div class="drag-drop-area" id="drag-drop-area">
                <p>Drag and drop your file here</p>
                <p>or</p>
                <button id="upload-btn">Choose File</button>
            </div>
                <textarea id="original-text" placeholder="Paste the
original text here or upload a file"></textarea>
            <div id="word-count">Word count: 0</div>
            <div class="button-group">
                <span id="file-info"></span>
                <button id="summarize-btn">Summarize</button>
            </div>
        </div>
        <div class="loading" id="loading">Summarizing...</div>
          <div class="success-message" id="success-message">Summary
generated successfully!</div>
           <div class="error-message" id="error-message">An error
occurred. Please try again.</div>
        <div class="output-area">
            <h2>Summary</h2>
               <textarea id="summary-output" placeholder="The output
summary will be displayed here" readonly></textarea>
            <div class="export-buttons">
                <button id="copy-btn">Copy to Clipboard</button>
                <button id="download-btn">Download as TXT</button>
            </div>
        </div>
    </div>



    <script>
        const uploadBtn = document.getElementById('upload-btn');
        const fileInfoSpan = document.getElementById('file-info');
         const originalTextArea = document.getElementById('original-
text');
       const summarizeBtn = document.getElementById('summarize-btn');
           const summaryOutput = document.getElementById('summary-
output');
```

```
          const  dragDropArea  =  document.getElementById('drag-drop-
area');
        const loadingIndicator = document.getElementById('loading');
          const  successMessage  =  document.getElementById('success-
message');
       const errorMessage = document.getElementById('error-message');
          const  wordCountDisplay  =  document.getElementById('word-
count');
        const copyBtn = document.getElementById('copy-btn');
        const downloadBtn = document.getElementById('download-btn');

        function handleFile(file) {
            fileInfoSpan.textContent = `File: ${file.name}`;
            const reader = new FileReader();
            reader.onload = function(e) {
                const fileType = file.type;
                console.log(`Handling file type: ${fileType}`);
                if (fileType === 'text/plain') {
                    originalTextArea.value = e.target.result;
                    updateWordCount();
                } else if (fileType === 'application/pdf') {
                    extractTextFromPDF(e.target.result);
                                }    else    if    (fileType    ===
'application/vnd.openxmlformats-
officedocument.wordprocessingml.document') {
                    extractTextFromDocx(e.target.result);
                } else {
                    originalTextArea.value = `Unsupported file type:
${file.name}`;
                    updateWordCount();
                }
            };
            reader.readAsText(file);
        }

        async function extractTextFromPDF(arrayBuffer) {
            try {
                   const  loadingTask  =  pdfjsLib.getDocument({  data:
arrayBuffer });
                const pdfDoc = await loadingTask.promise;
                let text = '';
                const numPages = pdfDoc.numPages;
        for (let i = 1; i <= numPages; i++) {
          const page = await pdfDoc.getPage(i);
          const textContent = await page.getTextContent();
           const pageText = textContent.items.map(item =>
item.str).join(' ');
          text += `${pageText}\n`;
        }
        console.log(`Extracted text from PDF: ${text}`);
        originalTextArea.value = text;
        updateWordCount();
      } catch (error) {
         console.error('Error extracting text from PDF:',
error);
        showMessage(errorMessage);
      }
    }

    function extractTextFromDocx(arrayBuffer) {
      mammoth.extractRawText({ arrayBuffer: arrayBuffer })
        .then(result => {
          const text = result.value;
          console.log(`Extracted text from DOCX: ${text}`);
          originalTextArea.value = text;
          updateWordCount();
        })
        .catch(error => {
          console.error('Error extracting text from DOCX:',
error);
          showMessage(errorMessage);
        });
    }

    uploadBtn.addEventListener('click', function(e) {
      e.stopPropagation();
      const input = document.createElement('input');
      input.type = 'file';
      input.accept = '.txt,.doc,.docx,.pdf,.rtf,.odt';
      input.onchange = e => {
        const file = e.target.files[0];
        handleFile(file);
      };
      input.click();
    });

    dragDropArea.addEventListener('dragover', (e) => {
        e.preventDefault();
        dragDropArea.classList.add('dragover');
    });

    dragDropArea.addEventListener('dragleave', () => {
        dragDropArea.classList.remove('dragover');
    });

    dragDropArea.addEventListener('drop', (e) => {
        e.preventDefault();
        dragDropArea.classList.remove('dragover');
        const file = e.dataTransfer.files[0];
        handleFile(file);
    });

    dragDropArea.addEventListener('click', () => {
        uploadBtn.click();
    });

    function showMessage(element) {
        element.classList.add('show-message');
        setTimeout(() => {
            element.classList.remove('show-message');
        }, 3000);
    }

    function updateWordCount() {
        const text = originalTextArea.value;
        const wordCount = text.trim().split(/\s+/).length;
       wordCountDisplay.textContent = `Word count: ${wordCount}`;
    }

    originalTextArea.addEventListener('input', updateWordCount);

    summarizeBtn.addEventListener('click', async function() {
        const originalText = originalTextArea.value;
        if (originalText.trim() === '') {
            showMessage(errorMessage);
            return;
        }

        loadingIndicator.style.display = 'block';
        summarizeBtn.disabled = true;

        try {
            const response = await fetch('/summarize', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ text: originalText }),
            });

            const data = await response.json();
            if (data.error) {
                showMessage(errorMessage);
            } else {
                summaryOutput.value = data.summary;
                showMessage(successMessage);
            }
        } catch (error) {
            console.error('Error:', error);
            showMessage(errorMessage);
        } finally {
            loadingIndicator.style.display = 'none';
            summarizeBtn.disabled = false;
        }
    });

    copyBtn.addEventListener('click', function() {
        summaryOutput.select();
        document.execCommand('copy');
        showMessage(successMessage);
    });

    downloadBtn.addEventListener('click', function() {
        const text = summaryOutput.value;
        const blob = new Blob([text], { type: 'text/plain' });
        const a = document.createElement('a');
        a.href = URL.createObjectURL(blob);
        a.download = 'summary.txt';
```

```
            document.body.appendChild(a);
            a.click();
            document.body.removeChild(a);
        });

        const themeToggle = document.getElementById('theme-toggle');
        const themeToggleIcon = document.getElementById('theme-toggle-icon');

        function toggleDarkMode() {
            document.body.classList.toggle('dark-mode');
            document.querySelector('.container').classList.toggle('dark-mode');
            updateThemeIcon();
            updateShapeColors();
        }

        function updateThemeIcon() {
            if (document.body.classList.contains('dark-mode')) {
                themeToggleIcon.textContent = '🌙'; // Moon icon for dark mode
            } else {
                themeToggleIcon.textContent = '☀️'; // Sun icon for light mode
            }
        }

        themeToggle.addEventListener('click', toggleDarkMode);

        // Initialize the icon based on the initial theme
        updateThemeIcon();

        // Add a simple animation to the drag and drop area
        function animateDragDropArea() {
            dragDropArea.style.animation = 'pulse 1s';
            setTimeout(() => {
                dragDropArea.style.animation = '';
            }, 1000);
        }

        setInterval(animateDragDropArea, 5000);
```

```
         // Add keyframe animation for text highlight and drag-drop
area pulse
        const style = document.createElement('style');
        style.textContent = `
            @keyframes highlight {
                from { background-color: var(--primary-light); }
                to { background-color: transparent; }
            }
            @keyframes pulse {
                0% { transform: scale(1); }
                50% { transform: scale(1.05); }
                100% { transform: scale(1); }
            }
        `;
        document.head.appendChild(style);

        function animateBackgroundShapes() {
            const shapes = document.querySelectorAll('.shape');
            shapes.forEach(shape => {
                shape.style.transform = `translate(${Math.random() *
100 - 50}px, ${Math.random() * 100 - 50}px) rotate(${Math.random() *
360}deg)`;
            });
        }

        // Animate shapes every 5 seconds
        setInterval(animateBackgroundShapes, 5000);

        // Update shape colors when toggling dark mode
        function updateShapeColors() {
            const shapes = document.querySelectorAll('.shape');
           const isDarkMode = document.body.classList.contains('dark-
mode');
            shapes[0].style.backgroundColor = isDarkMode ? '#9d46ff'
: '#b39ddb';
            shapes[1].style.backgroundColor = isDarkMode ? '#6200ea'
: '#673ab7';
            shapes[2].style.backgroundColor = isDarkMode ? '#0a00b6'
: '#4527a0';
        }
        function handleFile(file) {
            fileInfoSpan.textContent = `File: ${file.name}`;
            const reader = new FileReader();
            reader.onload = function(e) {
                const fileType = file.type;
                console.log(`Handling file type: ${fileType}`);
                if (fileType === 'text/plain') {
                    originalTextArea.value = e.target.result;
                    updateWordCount();
                } else if (fileType === 'application/pdf') {
                    extractTextFromPDF(e.target.result);
                                } else if (fileType ===
'application/vnd.openxmlformats-
officedocument.wordprocessingml.document') {
                    extractTextFromDocx(file);
                } else {
                    originalTextArea.value = `Unsupported file type:
${file.name}`;
                    updateWordCount();
                }
            };
              if (file.type === 'application/pdf' || file.type ===
'application/vnd.openxmlformats-
officedocument.wordprocessingml.document') {
                reader.readAsArrayBuffer(file);
            } else {
                reader.readAsText(file);
            }
        }

        async function extractTextFromPDF(arrayBuffer) {
            try {
                    const pdf = await pdfjsLib.getDocument({ data:
arrayBuffer }).promise;
                let text = '';
                for (let i = 1; i <= pdf.numPages; i++) {
                    const page = await pdf.getPage(i);
                    const content = await page.getTextContent();
                   text += content.items.map(item => item.str).join('
') + '\n';
                }
                originalTextArea.value = text;
                updateWordCount();
            } catch (error) {
                console.error('Error extracting text from PDF:',
error);
                showMessage(errorMessage);
            }
        }

        function extractTextFromDocx(file) {
            mammoth.extractRawText({ arrayBuffer: file })
                .then(result => {
                    const text = result.value;
                    console.log(`Extracted text from DOCX: ${text}`);
                    originalTextArea.value = text;
                    updateWordCount();
                })
                .catch(error => {
                    console.error('Error extracting text from DOCX:',
error);
                    showMessage(errorMessage);
                });
        }



        // Call updateShapeColors on page load
        updateShapeColors();

    </script>
</body>
</html>
```