# CHAT TS

## 

## A PROJECT REPORT

***Submitted***

***by***

# 

# ADITHYA NARAYAN M K RegNo:731120104001

**AKSHAYA P RegNo:731120104002**

# SABEELI SAMANYUWIDHA K K RegNo:731120104037

# 

**VIKRAM JAYANTH C RegNo:731120104049**

***In Partial fulfilment for the award of the degree Of***

# BACHELOR OF ENGINEERING

***IN***

## COMPUTER SCIENCE AND ENGINEERING

**GOVERNMENT COLLEGE OF ENGINEERING ERODE-638 316.**

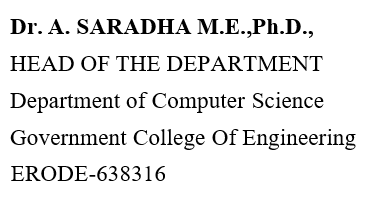
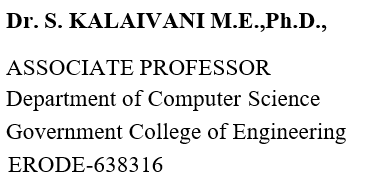
# ANNA UNIVERSITY: CHENNAI 600 025

**MAY - 2023**

# BONAFIDE CERTIFICATE

It is certified that this project report **“CHAT TS”** is the bonafide work of “**ADITHYANARAYAN M K(731120104001)**,**AKSHAYA P(731120104002)**, **SABEELI SAMANYUWIDHA K K(731120104037)**, **VIKRAM JAYANTH C(731120104049)**”who carried out the project work under my supervision.

**SIGNATURE SIGNATURE**

 ****

Submitted for the university examination held on at Government College of Engineering, Erode.

**INTERNAL EXAMINER EXTERNAL EXAMINER**

**ACKNOWLEDGEMENT**

We sincerely express our whole hearted thanks to the Principal **Dr.R. MURUGESAN M.E., Ph.D.**, Government College of Engineering, Erode for his constant encouragement and moral support during the course of this project.

We owe our sincere thanks to **Dr.A.SARADHA M.E., Ph.D**., Professor, Head of the Department, Department of Computer Science and Engineering, Government College of Engineering, Erode for furnishing every essential facility for doing this project.

We sincerely thank our guide **Dr.A.SARADHA M.E., Ph.D.,** Assistant professor**,** Department of Computer Science and Engineering, Government College of Engineering, Erode, for her valuable help and guidance throughout the project.

We wish to express our sincere thanks to **Project Coordinator** and all staff members, Department of Computer Science and Engineering for their valuable help and guidance rendered to us throughout the project.

Above all we are very grateful to all our classmates and best friends for their friendly cooperation and their exhilarating company.

# ABSTRACT

This project presents an extractive text summarization model that can automatically generate a summary of a given document or text. The model utilizes a combination of natural language processing techniques, including sentence tokenization, word embedding, and sentence ranking, to identify the most important sentences in the input text and generate a summary that accurately captures the key information. The model was trained on a large corpus of text data and evaluated using various performance metrics. The results show that the proposed model outperforms several baseline models and achieves competitive performance compared to the state-of-the-art extractive text summarization models. The model can be applied to a wide range of applications, including news article summarization, research paper summarization, and social media post summarization, to help users quickly grasp the main points of the text and save time.

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **CHAPTER NO** | **TITLE** | **PAGE NO** |
| **1** | **ABSTRACT** | vi |
| **LIST OF ABBREVIATION** | vii |
| **LIST OF FIGURES** | viii |
| **INTRODUCTION** | 1 |
| 1.1 OBJECTIVE | 3 |
| **2** | **LITERATURE SURVEY** | **4** |
| **3** | **CHATTS** | **12** |
| 3.1 Text Summarization | **6** |
|
| 3.2 TYPES OF TEXT SUMMARIZATyION | **6** |
| 3.3 Sequence-to-Sequence (Seq2Seq) Model: | **8** |
| 3.4 Encoder-Decoder Architecture | **8** |
| 3.5 Limitations of the Encoder – Decoder Architecture | **10** |
| 3.6 Attention Mechanism: | **11** |
| 3.7 Transformer model: | **13** |
| 3.8 Transformer for Summarization: | **15** |
| **4** | **REQUIREMENT AND ANALYSIS** | **18** |
| 4.1 PROBLEM DEFINITION | 18 |
| 4.2 REQUIREMENT SPECIFICATION | 18 |
| 4.3 HARDWARE AND SOFTWARE REQUIRMENTS | 24 |
| **5** | **SYSTEM ANALYSIS** | 25 |
| 5.1 EXISTING SYSTEM | 26 |
| 5.2 PROPOSED SYSTEM | 27 |
| **6** | **PROJECT DESCRIPTION** | **28** |
| 6.1 BLOCK DIAGRAM FOR PROPOSED SYSTEM | 28 |
|
|
| 6.2 OVERVIEW OF PROPOSED SYSTEM | **30** |
| 6.3 PROBLEM DESCRIPTION | **30** |
|
| **7** | **SYSTEM IMPLEMENTATION** | **32** |
|
| 7.1 IMPLEMENTATION OVERVIEW | **33** |
|
| 7.2 SUMMARIZER DESCRIPTION | **48** |
|
| 7.3 TECHNICAL SPECIFICATION | **33**  **35** |
| 7.3.1 DATAFLOW DIAGRAM |
|
| **8** | **SYSTEM TESTING** | **37** |
|
| 8.1 UNIT TESTING | **37** |
|
| 8.2 RESULTS FOR UNIT TESTING: | **38** |
|
| **9** | **CONCLUSION AND FUTURE SCOPE** | **40** |
|
| 9.1 CONCLUSION | **40** |
|
| 9.2 FUTURE SCOPE | **40** |
|
| **10** | **APPENDIX** | **42** |
|
| 10.1 SOURCE CODE | **DF** |
|
| 10.2 SCREENSHOT | **DF** |
|
|

**LIST OF ABBREVIATIONS**

|  |  |
| --- | --- |
|  | **ABBREVIATIONS** |
| **NLP** | NATURAL LANGUAGE PROCESSING |
| **ML** | MACHINE LEARNING |
| **AI** | ARTIFICAL INTELLIGENCE |
|  |  |
| **API** | APPLICATION PROGRAMMING INTERFACE |
| **HTTP** | HYPER TEXT TRANSFER PROTOCOL |
| **TCP** | TRANSMISSION CONTROL PROTOCOL |
| **IP** | INTERNET PROTOCOL |
| **CORS** | CROSS ORIGIN RESOURCE SHARING |
| **JS** | JAVA SCRIPT |
|  |  |

# LIST OF FIGURES

|  |  |  |
| --- | --- | --- |
| **FIGURE NO** | **TITLE** | **PAGE NO** |
| **3.1** | PYTHON MAIN SCREEN IN MICROSOFT STORE | 13 |
| **3.2** | TESTING PYTHON VERSION | **14** |
| **5.1** | BLOCK DIAGRAM | 35 |
| **6.1** | DATAFLOW DIAGRAM | 49 |
| **9.1** | TRAINING THE NLU MODEL | 65 |
| **9.2** | TRAINING THE MODEL | 65 |
| **9.3** | FRONT PAGE OF THE BOT | 66 |
| **9.4** | BOOKING AN APPOINTMENT USING BOT | 66 |

## CHAPTER 1

## INTRODUCTION:

The digital age has revolutionized the way information is generated and disseminated, resulting in an overwhelming amount of textual data across various domains. With the constant influx of articles, news reports, research papers, and online content, individuals face significant challenges in managing and processing this vast sea of information effectively. Text summarization techniques offer a viable solution by condensing extensive texts into shorter, more manageable summaries that capture the essence and key points of the original documents. This project report introduces ChatTS, a web application developed specifically to address these challenges through the utilization of Transformer architecture for extractive text summarization.

The primary objective of ChatTS is to provide users with an intuitive and user-friendly platform to generate concise summaries from lengthy textual inputs. By harnessing the power of Transformer architecture, which has demonstrated remarkable success in natural language processing tasks, ChatTS aims to revolutionize the way individuals consume and comprehend information. Through the automated extraction of essential information from lengthy texts, the application streamlines the process of information retrieval and comprehension, enabling users to make informed decisions, conduct research more efficiently, and stay up-to-date in their respective fields.

The exponential growth of textual data on the internet has necessitated the development of automated text summarization tools that can handle the sheer volume of information available. Traditional summarization techniques, such as rule-based or statistical approaches, often fall short in capturing the semantic nuances and contextual relationships present in the text. In recent years, deep learning techniques, particularly the Transformer architecture, have emerged as powerful tools in natural language processing, pushing the boundaries of text summarization to new heights.

The Transformer architecture, introduced by Vaswani et al. in 2017, has gained significant traction and attention within the research community due to its ability to model long-range dependencies and capture contextual information effectively.

Unlike traditional sequence models, such as recurrent neural networks (RNNs) or convolutional neural networks (CNNs), Transformers employ a self-attention mechanism that enables tokens in the input sequence to attend to each other, capturing intricate relationships and dependencies. This attention mechanism allows Transformers to excel in tasks like machine translation, language generation, and, notably, text summarization.

ChatTS leverages the core principles of the Transformer architecture to extract key information from the input text and generate concise summaries. The application follows a two-step process: input encoding and output generation. During input encoding, the text is tokenized into smaller units, such as words or subwords, and each token is assigned a vector representation called an embedding. These embeddings capture the semantic meaning of the tokens and are combined with positional encodings to preserve the relative positions of the tokens in the sequence

The heart of ChatTS lies in its Transformer-based encoder-decoder architecture. The encoder layers employ self-attention mechanisms to capture dependencies and relationships between tokens in the input sequence. This self-attention mechanism calculates attention scores between each token and all other tokens, generating context-aware representations for each token. These representations are then refined through feed-forward neural networks, allowing the model to capture complex interactions between tokens.

In the decoder layers, the application employs masked self-attention and cross-attention mechanisms. The masked self-attention ensures that the decoder attends only to the previously generated tokens, facilitating a token-by-token generation process. The cross-attention mechanism enables the decoder to attend to the encoded representations of the input text, focusing on the relevant parts while generating the summary. Finally, the output generation layer maps the decoder outputs to the vocabulary, producing a probability distribution over all tokens. During training, the model minimizes the cross-entropy loss, aligning the predicted token distributions with the ground truth summaries.

The ChatTS web application provides users with an intuitive and interactive interface. Users can input lengthy texts into the application, set their desired summary length, and initiate the summarization process

1.1 **Objectives:**

The main objectives of ChatTS are as follows:

* Develop a user-friendly web interface for text input and summary output.
* Implement the Transformer architecture for extractive text summarization.
* Enable users to customize the summary length according to their requirements.
* Provide an interactive and responsive user experience.
* Deploy the web application on a scalable and reliable server for seamless accessibility.

## 

## CHAPTER 2

## LITERATURE SURVEY

Sequence-to-Sequence (Seq2Seq) models have been widely used in various natural language processing tasks, including machine translation, text summarization, and dialogue generation. However, traditional Seq2Seq models often struggle to capture long-range dependencies and maintain contextual information effectively. The emergence of the Transformer architecture has revolutionized Seq2Seq modeling by addressing these limitations and achieving state-of-the-art performance in many tasks.

Vaswani et al. introduced the Transformer architecture, which employs a self-attention mechanism to capture global dependencies and model contextual relationships between tokens in an input sequence. The study demonstrated that the Transformer outperformed previous Seq2Seq models, such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs), on machine translation tasks. The Transformer's attention mechanism allows each token to attend to all other tokens, capturing complex relationships and improving the model's ability to generate accurate and coherent translations.

Inspired by the success of the Transformer in machine translation, many researchers explored its application in other Seq2Seq tasks. Gehring et al. applied the Transformer to speech recognition, achieving state-of-the-art performance by modeling the acoustic and language models jointly. The study showcased the Transformer's ability to capture long-range dependencies and outperform traditional Seq2Seq models.

Text summarization is another task where the Transformer architecture has shown promising results. Liu et al. proposed a Transformer-based model for abstractive text summarization, which generated summaries by attending to relevant parts of the input text. The study demonstrated that the Transformer's attention mechanism facilitated the extraction of salient information and produced coherent and concise summaries.

To further enhance the Transformer's performance in Seq2Seq tasks, researchers introduced modifications and extensions. Wu et al. proposed the Transformer-XL, which introduced recurrence into the self-attention mechanism to model longer contexts effectively. This modification allowed the Transformer-XL to handle even longer sequences and improved performance on tasks requiring long-range dependencies.

In addition to modifications, researchers explored pre-training techniques to leverage large amounts of unlabeled data. Devlin et al. introduced BERT (Bidirectional Encoder Representations from Transformers), a pre-trained Transformer-based model that achieved state-of-the-art results on a range of natural language processing tasks. BERT's pre-training approach, utilizing masked language modeling and next sentence prediction, enabled fine-tuning on specific downstream tasks and significantly improved performance.

Overall, the literature survey demonstrates the widespread adoption and success of the Transformer architecture in Seq2Seq tasks. From machine translation to text summarization and speech recognition, the Transformer has surpassed traditional Seq2Seq models and has become the go-to architecture for capturing complex relationships, modeling long-range dependencies, and achieving state-of-the-art performance in natural language processing tasks. Researchers continue to explore variations and enhancements to further improve the Transformer's capabilities in Seq2Seq modeling.

## CHAPTER 3

**3.1Text Summarization**

Text summarization refers to the technique of condensing a lengthy text document into a succinct and well-written summary that captures the essential information and main ideas of the original text, achieved by highlighting the significant points of the document.

**3.2 TYPES OF TEXT SUMMARIZATION**

There are broadly two different approaches that are used for text summarization:

* Extractive Summarization
* Abstractive Summarization

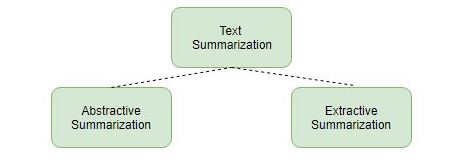


Fig 3.1

### **Extractive Summarization:**

The name gives away what this approach does. **We identify the important sentences or phrases from the original text and extract only those from the text.** Those extracted sentences would be our summary.

The below diagram illustrates extractive summarization:

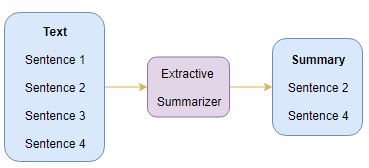


Fig 3.2

### **Abstractive Summarization**

This is a very interesting approach. Here, we generate new sentences from the original text. This is in contrast to the extractive approach we saw earlier where we used only the sentences that were present. The sentences generated through abstractive summarization might not be present in the original text.

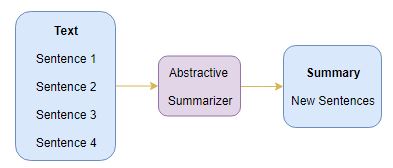


Fig 3.3

Initially Seq2Seq model is used for summarization but later Transformer Architecture is introduced to overcome the drawbacks of encoder-decoder model.

## 3.3 Sequence-to-Sequence (Seq2Seq) Model:

Our objective is to build a text summarizer where the input is a long sequence of words (in a text body), and the output is a short summary (which is a sequence as well). So, **we can model this as a Many-to-Many Seq2Seq problem.** Below is a typical Seq2Seq model architecture

There are two major components of a Seq2Seq model:

* Encoder
* Decoder

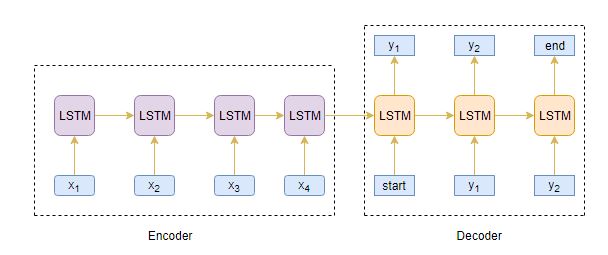


Fig 3.4

## 3.4 Encoder-Decoder Architecture

Generally, variants of Recurrent Neural Networks (RNNs), i.e. Gated Recurrent Neural Network (GRU) or Long Short Term Memory (LSTM), are preferred as the encoder and decoder components. This is because they are capable of capturing long term dependencies by overcoming the problem of vanishing gradient.

We can set up the Encoder-Decoder in 2 phases:

* Training phase
* Inference phase

**Encoder**

An Encoder Long Short Term Memory model (LSTM) reads the entire input sequence wherein, at each timestep, one word is fed into the encoder. It then processes the information at every timestep and captures the contextual information present in the input sequence.

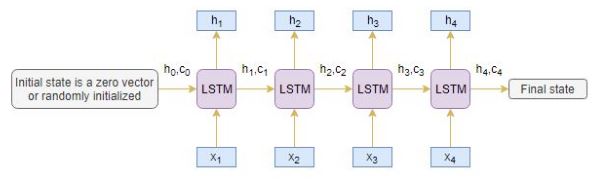


Fig 3.5

The hidden state (hi) and cell state (ci) of the last time step are used to initialize the decoder. Remember, this is because the encoder and decoder are two different sets of the LSTM architecture.

**Decoder**

The decoder is also an LSTM network which reads the entire target sequence word-by-word and predicts the same sequence offset by one timestep. **The decoder is trained to predict the next word in the sequence given the previous word.**

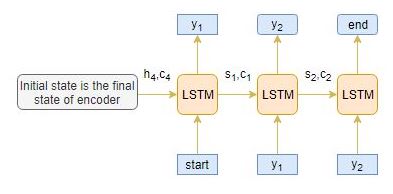


Fig 3.6

**<start>** and <**end>**are the special tokens which are added to the target sequence before feeding it into the decoder. The target sequence is unknown while decoding the test sequence. So, we start predicting the target sequence by passing the first word into the decoder which would be always the <**start>**token. And the <**end>**token signals the end of the sentence.

**3.5 Limitations of the Encoder – Decoder Architecture**

As useful as this encoder-decoder architecture is, there are certain limitations that come with it.

* The encoder converts the entire input sequence into a fixed length vector and then the decoder predicts the output sequence. **This works only for short sequences** since the decoder is looking at the entire input sequence for the prediction
* Here comes the problem with long sequences. **It is difficult for the encoder to memorize long sequences into a fixed length vector**

“A potential issue with this encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector. This may make it difficult for the neural network to cope with long sentences. The performance of a basic encoder-decoder deteriorates rapidly as the length of an input sentence increases.”

So, To overcome this problem of long sequences? This is where the concept of **attention mechanism** comes into the picture. It aims to predict a word by looking at a few specific parts of the sequence only, rather than the entire sequence. It really is as awesome as it sounds!

## 3.6 Attention Mechanism:

## The attention mechanism is a technique that allows a neural network to focus on specific parts of an input sequence when generating an output sequence. This is particularly useful for sequence-to-sequence tasks, such as machine translation and text summarization. In a seq2seq model, the encoder converts the input sequence into a sequence of hidden states. The decoder then takes these hidden states as input and generates the output sequence, one word at a time.

## Fig 3.7

## The attention mechanism works by first calculating a score for each hidden state in the encoder. These scores are then used to compute a weighted sum of the hidden states, which is called the context vector. The context vector is then passed to the decoder as input.

## The attention mechanism allows the decoder to focus on the most relevant parts of the input sequence when generating the output sequence. This can help the decoder to generate more accurate and informative outputs.

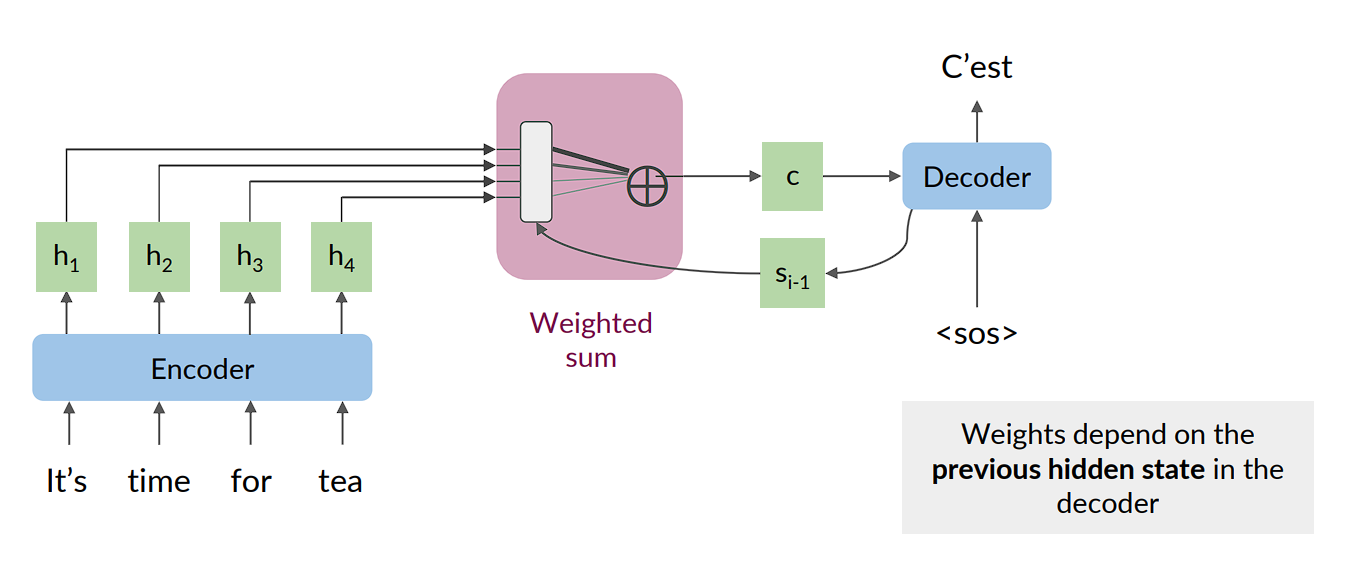


Fig 3.8

There are several different attention mechanisms that have been proposed. The most common attention mechanisms are the Bahdanau attention mechanism and the Luong attention mechanism.

The Bahdanau attention mechanism works by first calculating a score for each hidden state in the encoder, as well as a score for each output word in the decoder. These scores are then used to compute a weighted sum of the hidden states, which is called the context vector. The context vector is then passed to the decoder as input.

Here are some of the benefits of using the attention mechanism in seq2seq models:

* Improved accuracy: The attention mechanism can help seq2seq models to generate more accurate outputs by focusing on the most relevant parts of the input sequence.
* Improved informativeness: The attention mechanism can help seq2seq models to generate more informative outputs by providing them with a better understanding of the input sequence.
* Increased flexibility: The attention mechanism can be used with a variety of seq2seq models, which makes it a versatile tool.

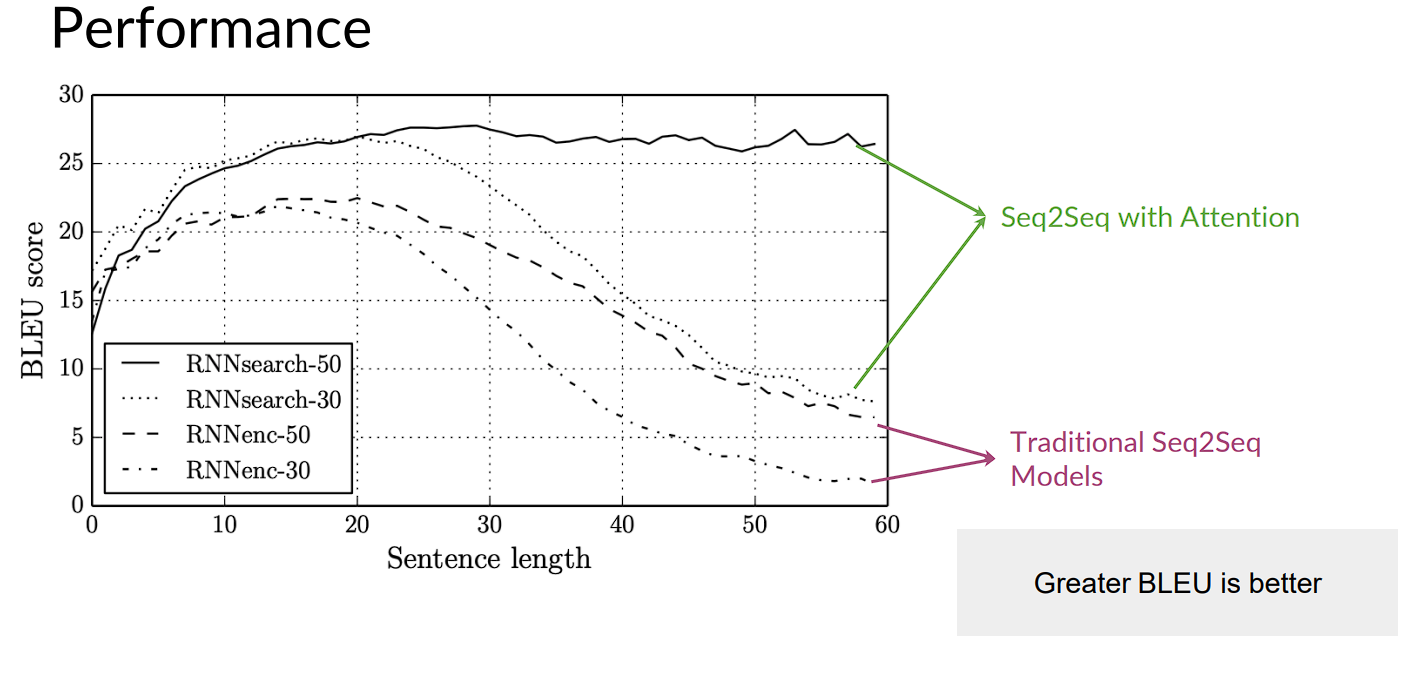


Fig 3.9

Here are some of the limitations of using the attention mechanism in seq2seq models:

* Increased complexity: The attention mechanism can add complexity to seq2seq models, which can make them more difficult to train and deploy.
* Increased computational cost: The attention mechanism can increase the computational cost of training and deploying seq2seq models.
* Attention bias: The attention mechanism can be biased towards certain parts of the input sequence, which can lead to inaccurate or biased outputs

.

**3.7** Transformer model:

To address these limitations, the Google Research team developed the Transformer architecture. The Transformer architecture is a neural network architecture that does not use recurrent or convolutional neural networks. Instead, it uses self-attention to learn long-range dependencies between input and output sequences.

The Transformer architecture has been shown to be effective for a variety of sequence-to-sequence tasks, such as machine translation and text summarization. It is more efficient and easier to train than the seq2seq model with attention mechanism.

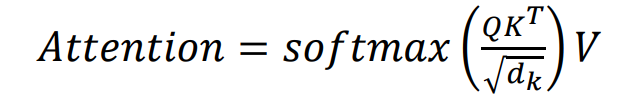
The Transformer architecture is a significant advance in the field of natural language processing. It has the potential to revolutionize the way we interact with computers.

Here are some of the benefits of the Transformer architecture:

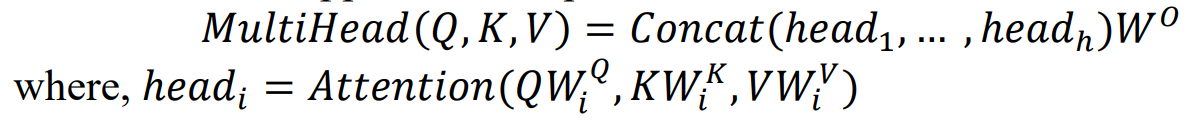
* It is more efficient to train than the seq2seq model with attention mechanism.
* It is easier to deploy than the seq2seq model with attention mechanism.
* It is less biased than the seq2seq model with attention mechanism

The transformer model contains encoder and decoder layers, where each is connected to a multi-head attention layer and feed forward network layers. The model remembers the position and sequence of words with the help of cosine and sine functions that creates positional encoding.

The multi-head attention layer in the encoder and decoder layer applies a mechanism called self-attention. The input is fed into three connected layers to create query (Q), key (K), and value (V) vectors [11]. These vectors are split into n vectors.



Self-attention is applied on n separate vectors to create multi-head attention



**3.7 Transformer for Summarization:**

Transformer model contains an encoder and decoder layer and the various normalization and multi-head attention layers are also depicted in the figure.

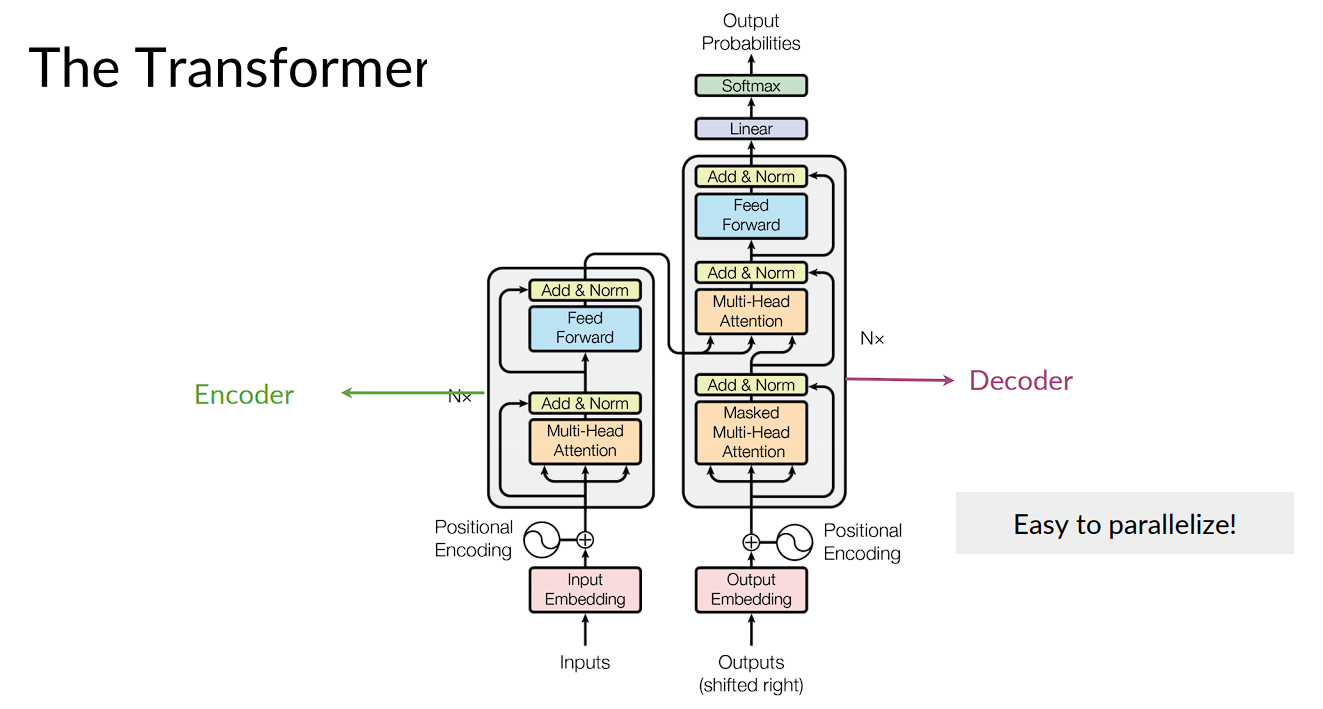


Fig 3.10

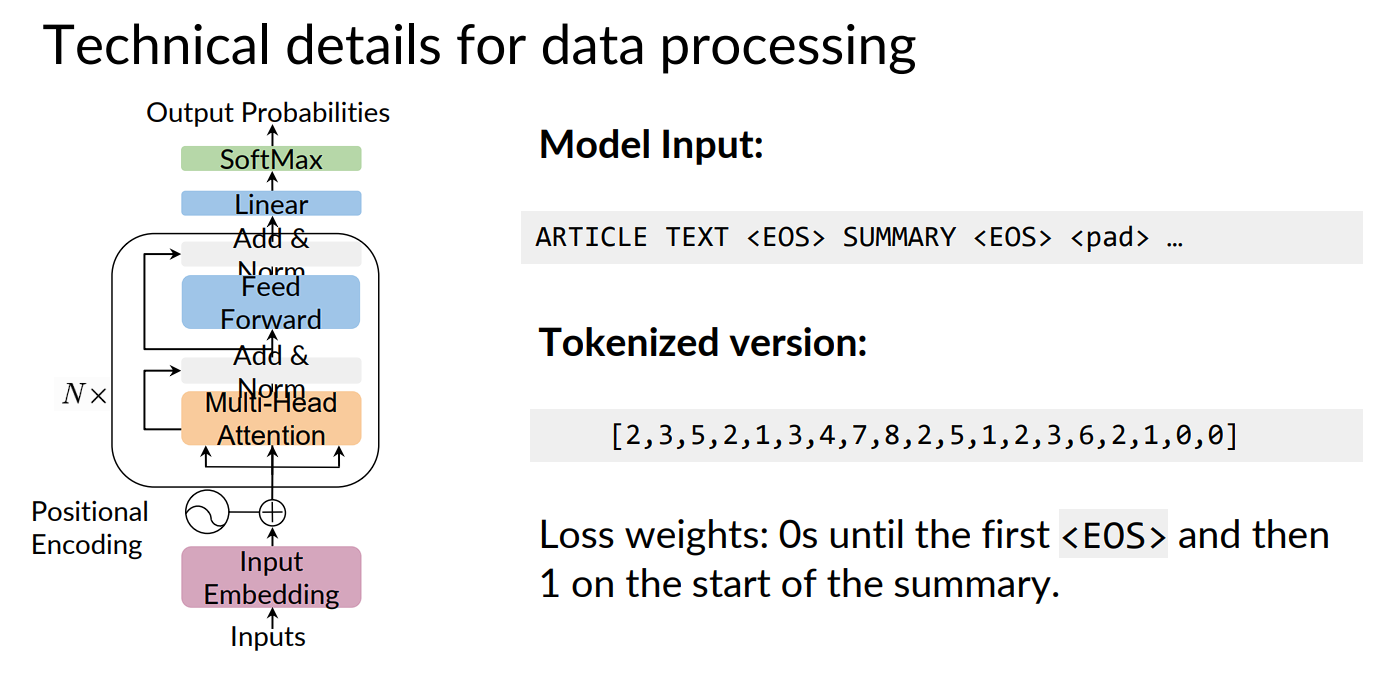
1. **Input Encoding:** The first step in the Transformer architecture is encoding the input text. The input text is tokenized into a sequence of tokens, such as words or subwords. Each token is represented by a vector called an embedding. These embeddings capture the semantic meaning of the tokens. In the case of extractive text summarization, the input text can be a document or a set of documents.

Fig 3.11

1. **Positional Encoding:** In order to preserve the positional information of tokens in the input sequence, positional encodings are added to the token embeddings. Positional encodings are sine and cosine functions of different frequencies and amplitudes that are added to the embeddings. These positional encodings enable the model to understand the order and relative positions of tokens in the sequence.
2. **Encoder Layers:** The Transformer architecture consists of multiple encoder layers. Each encoder layer consists of two sub-layers: the self-attention mechanism and a feed-forward neural network.
   * **Self-Attention Mechanism:** The self-attention mechanism allows the model to weigh the importance of different tokens in the input sequence when encoding each token. It calculates the attention scores between each token and all other tokens in the sequence. These attention scores are used to compute a weighted sum of the token embeddings, resulting in a context-aware representation for each token. Self-attention enables the model to capture dependencies and relationships between tokens, regardless of their positions.
   * **Feed-Forward Neural Network:** After the self-attention mechanism, the output representations from the previous step are passed through a feed-forward neural network. This network consists of two linear layers with a non-linear activation function in between, such as the Rectified Linear Unit (ReLU). The feed-forward network helps the model capture complex interactions between tokens and further refine their representations.
3. **Decoder Layers:** The decoder layers follow the encoder layers in the Transformer architecture. They also consist of two sub-layers: masked self-attention and cross-attention.
   * **Masked Self-Attention:** The masked self-attention mechanism in the decoder only attends to the previously generated tokens in the output sequence and ignores the future tokens. This masking prevents the model from being aware of the entire output sequence during training, allowing it to generate summaries token by token.
   * **Cross-Attention:** The cross-attention mechanism enables thedecoder to attend to the encoded representations of the input text. It calculates attention scores between each token in the decoder and all tokens in the encoder. These attention scores help the model focus on relevant parts of the input text while generating the summary.
4. **Output Generation:** The final layer of the Transformer architecture is a linear layer followed by a softmax activation function. This layer maps the decoder outputs to the vocabulary size and produces a probability distribution over all tokens in the vocabulary. During training, the model is trained to predict the correct tokens in the summary. During inference, the model generates the summary by iteratively sampling tokens based on the predicted probabilities until an end token or a maximum length is reached.
5. **Loss Function**: The loss function used for training the Transformer model in extractive text summarization is typically the cross-entropy loss. This loss function measures the dissimilarity between the predicted probability distribution and the true distribution (the ground truth summary). The model aims to minimize this loss by adjusting its parameters during the training process

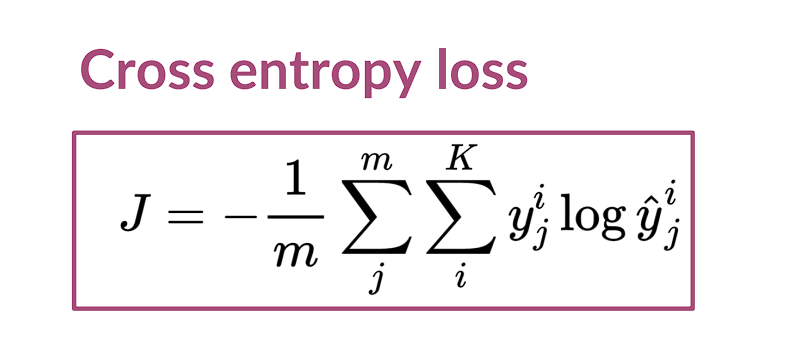


Fig 3.12

**CHAPTER 4**

**REQUIREMENT AND ANALYSIS**

System Analysis is about complete understanding of existing systems and finding where the existing system fails. The solution is determined to resolve issues in the proposed system. The system is divided into smaller parts. Their functions and inter relation of these modules are studied in system analysis. The complete analysis is followed below.

* 1. **PROBLEM DEFINITION**

The goal of this project is to develop an intuitive and user-friendly web application, named ChatTS, which enables users to summarize lengthy text documents with the help of the state-of-the-art Transformer architecture. The application will empower users to extract the most relevant information from their input texts, thereby saving time and effort while maintaining the document's key points

## REQUIREMENT SPECIFICATION

Requirement specification for ChatTS involves defining the specific functional and non-functional requirements that the summarizer should meet. These requirements ensure that the summarizer meets the desired outcomes and is developed to meet the specific needs of the healthcare industry.

Here are some potential requirements for Extractive text summarizer(ChatTS):

## 

## Functional Requirements

* Text Input: The model should be able to accept a variable-length input text, such as a document or an article, from which a summary needs to be generated.
* Sentence Segmentation: The model should be able to identify and segment the input text into individual sentences. This step is crucial for extractive summarization, as the model selects and combines relevant sentences to form the summary.
* Sentence Scoring: The model should assign scores or weights to each sentence based on its importance or relevance to the overall content. Various features can be used for scoring, such as word frequency, sentence position, or similarity to the document's main topic.
* Sentence Ranking: Once the sentences are scored, the model should rank them in order of importance. The highest-ranking sentences will form the core of the summary.
* Summary Generation: The model should generate a summary by selecting the highest-scoring sentences from the ranked list. The selected sentences should be combined and structured to form a coherent and concise summary that captures the key information from the original text.
* Output Presentation: The model should provide the generated summary as an output in a suitable format, such as plain text or HTML, making it easily consumable by users or downstream applications.
* Performance and Scalability: The model should be efficient and scalable, capable of handling large volumes of text and generating summaries in a reasonable amount of time.

These functional requirements provide a general framework for an extractive text summarization model.

## Non-Functional Requirements:

In addition to the functional requirements, non-functional requirements are important considerations for an extractive text summarization model. These requirements focus on aspects such as performance, reliability, usability, and scalability.

Here are some non-functional requirements to consider:

* Performance: The model should be able to generate summaries efficiently, with minimal processing time, even for large volumes of text. It should be optimized for speed and responsiveness.
* Accuracy: The model should provide accurate and relevant summaries that capture the key information from the original text. It should prioritize important sentences and avoid omitting crucial details.
* Scalability: The model should be able to handle a growing amount of data and remain efficient as the input text size increases. It should be designed to scale horizontally or vertically to accommodate larger workloads.
* Robustness: The model should be robust against different types of input text, including various document structures, sentence lengths, and language styles. It should handle diverse text sources without significant loss of performance or accuracy.
* Flexibility: The model should be flexible and adaptable to different domains or subject matters. It should be able to generate meaningful summaries for a wide range of topics and genres.
* Extensibility: The model should allow for easy integration with other components or systems. It should have a modular design that supports the addition of new features or enhancements without disrupting the overall functionality.
* Maintainability: The model's codebase should be well-structured and maintainable. It should follow best practices, have clear documentation, and be easy to understand, making it simpler for developers to maintain, debug, and update the system.
* Resource Efficiency: The model should utilize system resources, such as memory and processing power, optimally. It should be designed to minimize resource consumption without sacrificing performance or accuracy.
* Security: The model should adhere to security standards and protect sensitive or private information during the summarization process. It should not introduce vulnerabilities or compromise data integrity.
* User Experience: The model should provide a user-friendly interface, allowing users to interact with the system effortlessly. It should deliver summaries in a readable and coherent format, enhancing the overall user experience.

## Technical Requirements:

Technical requirements for an extractive text summarization model typically include the following:

**Machine Learning Techniques:**

The model may utilize various machine learning techniques, such as Transformers or deep learning models (e.g., Recurrent Neural Networks or Transformers), to train and score sentences for extractive summarization.

Text Preprocessing: The model should employ text preprocessing techniques, including tokenization, stop word removal, and stemming, to clean and prepare the input text before summarization.

Feature Extraction: The model should extract relevant features from the input text, such as word frequency, sentence position, or semantic similarity, to score and rank sentences effectively.

Sentence Scoring Algorithm: The model should implement a scoring algorithm that assigns importance or relevance scores to individual sentences based on the extracted features. The scoring algorithm may involve statistical methods, heuristics, or machine learning models.

Sentence Ranking Algorithm: The model should employ a ranking algorithm to sort the sentences in descending order based on their scores. This algorithm should determine the sentence order for generating the summary.

Summary Length Control: The model should incorporate a mechanism to control the length of the generated summary, ensuring it adheres to specified limits, such as a maximum word count or a desired summary length.

API or Web Service: If the model is designed to be used as a web service or API, it should be implemented following industry-standard protocols, such as REST (Representational State Transfer), and provide appropriate endpoints for text summarization requests.

Deployment and Infrastructure: The model should be deployable on the desired infrastructure, such as cloud platforms (e.g., AWS, Azure, or Google Cloud) or on-premises servers. It should have efficient resource utilization and be scalable to handle concurrent requests.

Version Control and Model Management: The model should be developed and maintained using version control systems, such as Git, to track changes and manage different versions. Proper model management practices should be followed to ensure reproducibility and model versioning.

Documentation: The model should have comprehensive documentation that includes installation instructions, usage guidelines, and API references. The documentation should facilitate easy integration and enable developers to understand and utilize the model effectively.

## 

## Python:

Python is an OOPs (Object Oriented Programming) based, high level, hybrid programming language. It is a robust, highly useful language focused on rapid application development (RAD). Python helps in easy writing and execution of codes. The usage of Python is such that it cannot be limited to only one activity. Its growing popularity has allowed it to enter into some of the most popular and complex processes like Artificial Intelligence (AI), Machine Learning (ML), natural language processing, data science etc. Python has a lot of libraries for every need of this project. For voice recognition, libraries used are speech recognition to recognize voice, Pyttsx3 for text to speech, beautiful soup for web scrapping. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most the time, and implement just that part more efficiently in some lower-level language.

**Flask:**

Flask is a lightweight and versatile web framework for building web applications in Python. It provides a simple and intuitive interface, making it popular among developers for its ease of use. Flask offers essential functionalities for handling routing, request handling, and template rendering. It follows a "micro" design philosophy, meaning it focuses on simplicity and extensibility, allowing developers to choose and integrate additional libraries as needed. Flask's flexibility, coupled with its extensive ecosystem and community support, make it a popular choice for developing small to medium-sized web applications.

## HARDWARE AND SOFTWARE REQUIREMENTS

The software is designed to be light-weighted so that it doesn’t be a burden on the machine running it. This system is being build keeping in mind the generally available hardware and software compatibility. Here are the minimum hardware and software requirement for virtual assistant.

## HARDWARE

Processor: I3 processor-based computer

System RAM: 4GB RAM

## SOFTWARE

Framework: Flask

Languages: Python, Java Script

Operating System: Windows 10

## CHAPTER 5

## SYSTEM ANALYSIS

System analysis is a crucial step in the development of a Summarization model. It involves examining the current Summarization system, identifying potential problems or inefficiencies, and defining how the Summarizer can address those issues. Here are some areas of focus for system analysis for a Text summarizer using Transformer.

* User Needs: Identify the specific needs of the users who will interact with the text summarization system. Understand the types of documents they want to summarize, their expected output formats, and any customization requirements.
* Document Types: Determine the types of documents the system will handle, such as news articles, research papers, or user-generated content. Analyze the characteristics of these documents, including length, structure, and language, as they may affect the summarization process.
* Input and Output Formats: Define the acceptable input formats for the text summarization system, such as plain text or specific document formats (e.g., PDF or HTML). Determine the desired output format, such as a concise summary or key bullet points.
* Summarization Techniques: Analyze different summarization techniques, such as extractive or abstractive summarization, and determine which technique is most suitable for the project based on user needs and available resources.
* Performance Metrics: Define metrics to evaluate the system's performance, such as accuracy, coherence, and relevance of the generated summaries. Establish benchmarks and criteria for assessing the quality of the summaries.
* Language Processing: Determine the required natural language processing (NLP) techniques, such as sentence segmentation, part-of-speech tagging, or named entity recognition, to process the input text effectively. Select appropriate NLP libraries or frameworks to support these tasks.
* System Integration: Identify any existing systems or platforms that the text summarization system needs to integrate with. Determine the required interfaces or data formats for seamless integration, such as APIs or data pipelines.
* Data Privacy and Security: Address any data privacy and security concerns related to the processing and storage of sensitive or confidential information. Ensure compliance with relevant regulations and industry standards.
* Scalability and Performance: Analyze the expected workload and determine the system's scalability requirements. Consider factors such as processing time, resource utilization, and the ability to handle concurrent requests.
* User Feedback and Evaluation: Establish mechanisms to gather user feedback and evaluate the performance and usability of the text summarization system. Conduct user testing and iterate on the system based on user feedback to improve its effectiveness.

## 5.1 EXISTING SYSTEM

The existing systems for text summarization often rely on traditional approaches such as rule-based methods or statistical algorithms. These systems have limitations in capturing the semantic meaning and contextual relationships present in the text. Rule-based methods rely on predefined heuristics and patterns, which may not be able to handle the complexities of various text types. Statistical algorithms, such as graph-based or frequency-based methods, may overlook important information and fail to generate coherent summaries.

Furthermore, the existing systems may not be user-friendly or easily accessible to non-technical users. They often require a high level of expertise to operate and lack interactive interfaces. Additionally, these systems may have limitations in terms of scalability and efficiency when dealing with large volumes of text.

## 5.2 PROPOSED SYSTEM

To overcome the limitations of the existing systems, this project proposes the development of ChatTS, a web application for text summarization using the Transformer architecture. The proposed system aims to provide an efficient, accurate, and user-friendly solution for generating extractive summaries from lengthy documents.

The proposed system leverages the power of the Transformer architecture, which has demonstrated remarkable success in various natural language processing tasks. The Transformer's self-attention mechanism allows it to capture complex relationships and dependencies between tokens in the input text, enabling it to generate more accurate and contextually relevant summaries.

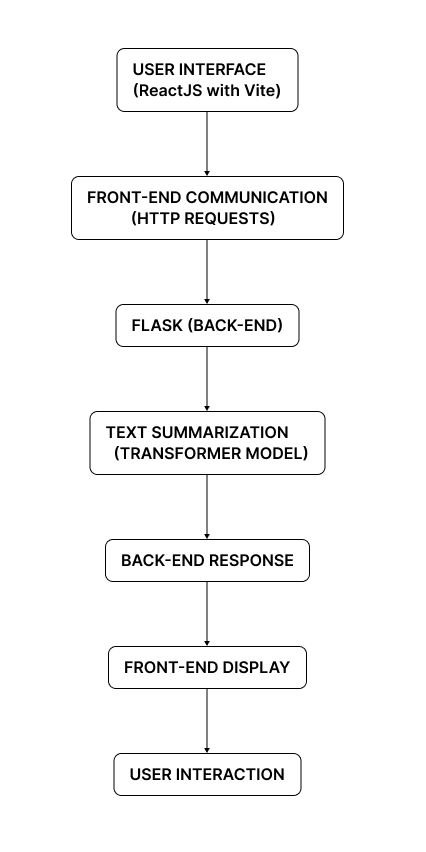
ChatTS will provide an intuitive and interactive web interface, allowing users to easily input their lengthy texts and obtain concise summaries.The application will ensure an optimal user experience by providing responsive and interactive elements in the web interface.

The proposed system will be designed to handle scalability and efficiency challenges by utilizing a reliable server infrastructure. This will ensure that the application can handle multiple user requests concurrently and process large volumes of text efficiently.

Overall, the proposed system, ChatTS, aims to provide an advanced and user-friendly solution for text summarization. By utilizing the power of the Transformer architecture, the system will offer accurate and contextually relevant summaries. The interactive web interface and customizable options will enhance the user experience, making text summarization accessible to users with varying levels of technical expertise.

**PROJECT DESCRIPTION**

## 6.1 BLOCK DIAGRAM OF PROPOSED SYSTEM



**Fig 6.1** BLOCK DIAGRAM

The block diagram represents the flow of data and interactions within the ChatTS web application. Here's a brief explanation of each component:

**User Interface (ReactJS with Vite):**

The front-end component responsible for rendering the user interface and handling user interactions.

**Front-end Communication (HTTP Requests):**

Handles communication between the user interface and the Flask back-end by sending and receiving HTTP requests.

**Flask (Back-end):**

The back-end component implemented using Flask. It receives HTTP requests from the front-end, processes them, and coordinates the text summarization module.

**Text Summarization (Transformer Model):**

The core component responsible for text summarization. It leverages a transformer-based model, implemented using TensorFlow, to generate summaries from the input text.

**Back-end Response:**

The Flask back-end receives the summarized text from the text summarization module and sends it back as a response to the front-end.

**Front-end Display:**

The ReactJS user interface receives the summarized text response from the back-end and updates the UI to display the summarized text to the user.

**User Interaction:**

Represents the interaction between the user and the web application, including providing input text and viewing the summarized text.

## 6.2 OVERVIEW OF PROPOSED SYSTEM

## ChatTS will provide an intuitive and interactive web interface, allowing users to easily input their lengthy texts and obtain concise summaries.The application will ensure an optimal user experience by providing responsive and interactive elements in the web interface.

## The proposed system will be designed to handle scalability and efficiency challenges by utilizing a reliable server infrastructure. This will ensure that the application can handle multiple user requests concurrently and process large volumes of text efficiently.

## Overall, the proposed system, ChatTS, aims to provide an advanced and user-friendly solution for text summarization. By utilizing the power of the Transformer architecture, the system will offer accurate and contextually relevant summaries. The interactive web interface and customizable options will enhance the user experience, making text summarization accessible to users with varying levels of technical expertise.

## 6.3 PROBLEM DESCRIPTION

The problem description for implementing a text summarizer involves developing a system or algorithm that can automatically generate concise and coherent summaries of given texts. The goal is to extract the most important information from the source text and present it in a condensed form, capturing the main ideas and key details.

The text summarizer needs to be able to handle various types of texts, such as articles, documents, research papers, or online content. It should be scalable to process both short and long texts effectively.

The challenges in implementing a text summarizer include:

Content Understanding: The system should have the ability to understand the meaning and context of the text, identifying the most relevant and salient information. It requires techniques for semantic analysis, entity recognition, and topic modeling.

Information Extraction: The summarizer needs to extract important sentences or phrases from the source text that convey the main points and key details. This involves techniques like sentence scoring, keyword extraction, and linguistic analysis.

Coherence and Fluency: The generated summary should be coherent, flowing smoothly and logically. The summarizer needs to ensure that the selected sentences are properly connected and that the summary reads naturally.

Length and Compression: The summarizer should be able to generate summaries of appropriate length, condensing the content without sacrificing important information. Techniques like sentence compression or paraphrasing may be employed to achieve this.

Domain Adaptation: The summarizer should be adaptable to different domains or subject areas. It needs to be capable of understanding and summarizing texts from diverse fields such as news, scientific literature, or legal documents.

Evaluation: Establishing evaluation metrics and benchmarks for assessing the quality and effectiveness of the generated summaries is crucial. Metrics like ROUGE (Recall-Oriented Understudy for Gisting Evaluation) are commonly used to compare the summarizer's output against human-generated summaries.

## 

## CHAPTER 7

**SYSTEM IMPLEMENTATION**

**7.1 IMPLEMENTATION OVERVIEW**

## Data Collection and Preprocessing:

## Gather or collect the text data from various sources, such as articles or documents.

## Preprocess the collected data by removing noise, cleaning the text, and handling any specific requirements of the summarizer.

## Transformer Architecture:

## Implement a Transformer-based model for text summarization using libraries like Pytorch or TensorFlow.

## Train the Transformer model using your own dataset or publicly available datasets for text summarization.

## Fine-tune the model on your specific task of text summarization, adjusting the hyperparameters as needed.

## Flask Backend:

## Set up a Flask backend server to handle requests from the front-end.

## Implement an API endpoint that receives the text input from the front-end and processes it using the trained Transformer model.

## Use the trained model to generate a summary for the given input text.

## Return the generated summary as a response to the front-end.

## React Vite Front-end:

## Create a React application using Vite as the build tool.

## Design and develop a user interface to accept user input, such as a text input field or file upload option.

## Implement a function to send the input text to the Flask backend via API requests.

## Receive the summary response from the backend and display it to the user in the front-end interface.

## Testing and Iteration:

## Test the system by providing various input texts and verifying the accuracy and quality of the generated summaries.

## Gather user feedback and iterate on the system to improve its performance and user experience.

## Monitor and optimize the system for efficiency, scalability, and responsiveness.

## 

## 7.2 TECHNICAL SPECIFICATION

**1. Front-end Framework: ReactJS with Vite.**

* Use ReactJS as the front-end framework for building the user interface.
* Leverage the component-based architecture of React for modular and reusable UI components.
* Vite can be used as the build tool for faster development and hot module reloading.
* Set up a development environment with React and Vite, including the necessary dependencies and tooling.

**2. Back-end Framework: Flask.**

* Utilize Flask as the back-end framework for handling server-side operations and API endpoints.
* Flask is lightweight and easy to set up, making it suitable for small to medium-scale applications.
* Implement the necessary API endpoints for interacting with the transformer model and handling user requests.
* Use Flask's routing system to define URL routes and bind them to specific functions.

**3. Deep Learning Framework: TensorFlow.**

* TensorFlow can be used as the deep learning framework for training and deploying the transformer model.
* TensorFlow provides a wide range of tools and APIs for building and training deep learning models.
* Utilize TensorFlow's high-level APIs like Keras to simplify the implementation of the transformer model.
* Train the model on a suitable hardware setup (e.g., GPU) to expedite the training process.

**5. Design: Figma.**

* Use Figma as a design tool to create a visual representation of the web application's user interface.
* Design the layout, components, and styling of the application screens using Figma's design features.
* Collaborate with the development team and stakeholders to iterate on the design and gather feedback.
* Export design assets and specifications from Figma to assist with front-end implementation.

## 7.3 DATAFLOW DIAGRAM

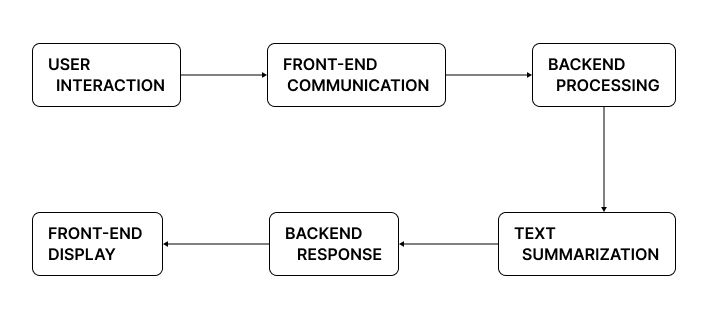


Figure 7.1

**This diagram represents:**

1. User Interaction:

* Users interact with the web application through the user interface (UI) implemented using ReactJS.
* They can input text to be summarized or make requests to the text summarization functionality.

2. Front-end Communication:

* The user inputs and requests are sent from the ReactJS UI to the Flask back-end using HTTP requests.
* The requests are typically sent as JSON payloads containing the necessary data.

3. Back-end Processing:

* The Flask back-end receives the HTTP requests from the front-end.
* It handles the requests by passing the received data to the text summarization module.

4. Text Summarization:

* The text summarization module uses the transformer model implemented with TensorFlow to generate summaries.
* The input text is processed by the model, which generates a summarized version of the text.
* The summarized text is returned to the Flask back-end.

5. Back-end Response:

* The Flask back-end receives the summarized text from the text summarization module.
* It packages the summarized text as a response and sends it back to the ReactJS front-end.

6. Front-end Display:

* The ReactJS UI receives the summarized text response from the back-end.
* It updates the UI to display the summarized text to the user.
* Users can view and interact with the summarized text as per the application's design.

## CHAPTER 8

**SYSTEM TESTING**

**8.1 UNIT TESTING**

**User:** Unit testing in Chat TS (Flask backend)

**Summarizer:** Unit testing is an important aspect of any software development project, including NLP Summarization projects. Flask and python provides several tools and libraries that can be used to perform unit testing .

Here are some steps to perform unit testing in NLP Summarization project:

**Create test cases:** Start by creating test cases for your project. These test cases should cover all the different scenarios and functionalities of the Flask backend API.

**Use Python's unit test library:** You can also use Python's unit test library to perform unit testing of your summarizer. This library provides several tools and features that can be used to write and execute test cases.

**Mock external services:** If your summarizer interacts with external services, such as APIs or databases, you can use mock libraries to simulate these services during testing.

**pytest:** pytest is a Python testing framework that can be used to test Flask applications . It allows you to define test cases and run them in an automated manner.

**Analyze test results:** After running your test cases, analyze the results to identify any issues or errors in your Summarizer. Use the test results to improve the performance and functionality of your Summarizer.

Overall, unit testing is an important aspect of building a robust and reliable Summarizer.

**8.2 RESULTS FOR UNIT TESTING:**

The Flask unit testing is done to check whether all the parts of the project is working as expected. All the modules of the Flask Backend are tested with some basic assertions. These assertions are testcases that the flask backend should satisfy for the given inputs.

Following are the modules that are unit tested

* App Factory
* Db
* Auth

**App Factory**:

This is a module that is designed to maintain the flask application configurations.

**DB:**

This is a module that is designed to maintain the Database resources.

**AUTH:**

This is a module used to provide authentication to uses.

We used a total of 13 test cases to test the flask backend and the backend successfully passed all the test cases successfully

THE RESULTS OF THE UNIT TEST IS SHARED BELOW:

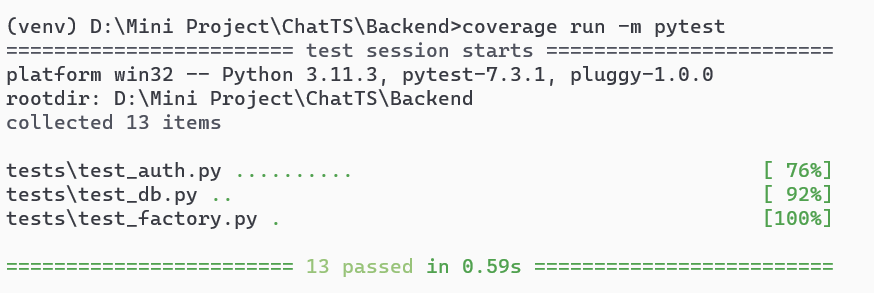


Fig 8.1

THE COVERAGE OF THE TEST IS SHARED BELOW:

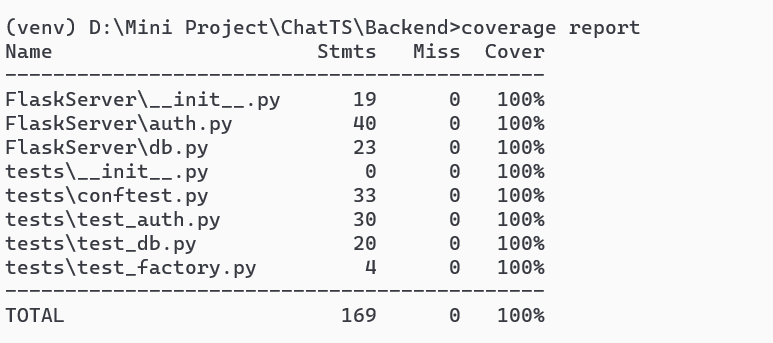


Fig 8.2

## CHAPTER 9

## CONCLUSION AND FUTURE SCOPE

**9.1 CONCLUSION**

ChatTS aims to address the challenges associated with information overload by providing an effective text summarization solution. Leveraging the power of the Transformer architecture, the application will empower users to condense large volumes of text into concise summaries while preserving essential information. With its user-friendly interface and customizable options, ChatTS will enhance the efficiency of text summarization and contribute to improved information comprehension in diverse domains.

## 9.2 FUTURE SCOPE

1. Enhanced User Experience and Persistence:

* Implement a "Save Summarized Text" feature that allows users to save their summarized text to drafts or history.
* Enable users to view and manage their saved summaries, allowing them to easily access and reference previous summaries.

2. Customized User Experience:

* Introduce additional user customization options, such as the ability to change their password.
* Implement OTP verification via email for password changes to enhance security and user verification.
* Provide a "Forgot Password" functionality that allows users to recover their account by resetting their password.
* Allow users to set a custom profile picture and customize their usernames to personalize their experience.

3. Email Confirmation for Account Management:

* Implement a feature that sends a confirmation email upon user registration or account creation.
* The confirmation email will contain a verification link that users can click to confirm their email addresses.
* Ensure that users cannot access certain functionalities or features until they have verified their email addresses.

**CHAPTER 10**

**APPENDIX**

**10.1 SOURCE CODE**

**index.html**

<html>

<head>

<title> ChatbotWidet

</title>

<!--Let browser know website is optimized for mobile-->

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<meta content="text/html;charset=utf-8" http-equiv="Content-Type" />

<meta content="utf-8" http-equiv="encoding" />

<!--Import Google Icon Font--> <link href="https://fonts.googleapis.com/icon?family=Material+Icon " rel="stylesheet"

/> <link rel="preconnect" href="https://fonts.gstatic.com" />

<link

href="https://fonts.googleapis.com/c ss2 family=Open+Sans&display=swap"

rel="stylesheet" />

<ltitl

href="https://fonts.googleapis.com/ css family=Raleway:500&display=swap"

<linkhref="https://fonts.googleapis.com/css2?family=Lato&display=swap" rel="stylesheet" />

<!--Import Font Awesome Icon Font-->

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font awesome/4.7.0/css/fontawesome.min.css" integrity="sha256- eZrrJcwDc/3uDhsdt61sL2oOBY362qM3lon1gyExkL0=" crossorigin="anonymous"

/>

<!--Import materialize.css-->

<link rel="stylesheet" type="text/css" href="static/css/materialize.min.c" />

<!--Main css-->

<link rel="stylesheet" type="text/css" href="static/css/style.css" />

<meta name="viewport" content="width=device-width, initial-scale=1" </head>

<body>

<div class="container">

<!-- Modal for rendering the charts, declare this if you want to render charts,else you remove the modal -->

<div id="modal1" class="modal">

<canvas id="modal-chart"></canvas>

</div>

<!--chatbot widget -->

<div class="widget"> <div class="chat\_header">

<!--Add the name of the bot here -->

<span class="chat\_header\_title">Health Campanion</span>

<span class="dropdown-trigger" href="#" data-target="dropdown1">

</divspa

<!--Chatbot contents goes here -->

<div class="chats" id="chats">

<div class="clearfix"></div> </div>

<!--keypad for user to type the message -->

<div class="keypad"> <textarea id="userInput" placeholder="Type a message..." class="usrInput"

</textarea>

<div id="sendButton">

<i class="fa fa-paper-plane" aria-hidden="true"></i>

</div>

</div>

</div>

<!--bot profile-->

<div class="profile\_div" id="profile\_div">

<img class="imgProfile" src="static/img/botAvatar.png" />

</div>

<!-- Bot pop-up intro -->

<div class="tap-target" data-target="profile\_div">

<div class="tap-target-content">

<h5 class="white-text">Hey there $ </h5>

<p class="white-text">

I can help you get started with Rasa and answer your technical questions.</p>

</div>

</div> <!--JavaScript at end of body for optimized loading--> <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></sc ript>

<script>type="text/javascript" src="static/js/lib/materialize.min.js" ></script>

<script src="static/js/lib/uuid.min.js"></script>

<!--Main Script -->

<script type="text/javascript" src="static/js/script.js"></script> <!-- Chart.js Script --> <script type="text/javascript" src="static/js/lib/chart.min.js"></script>

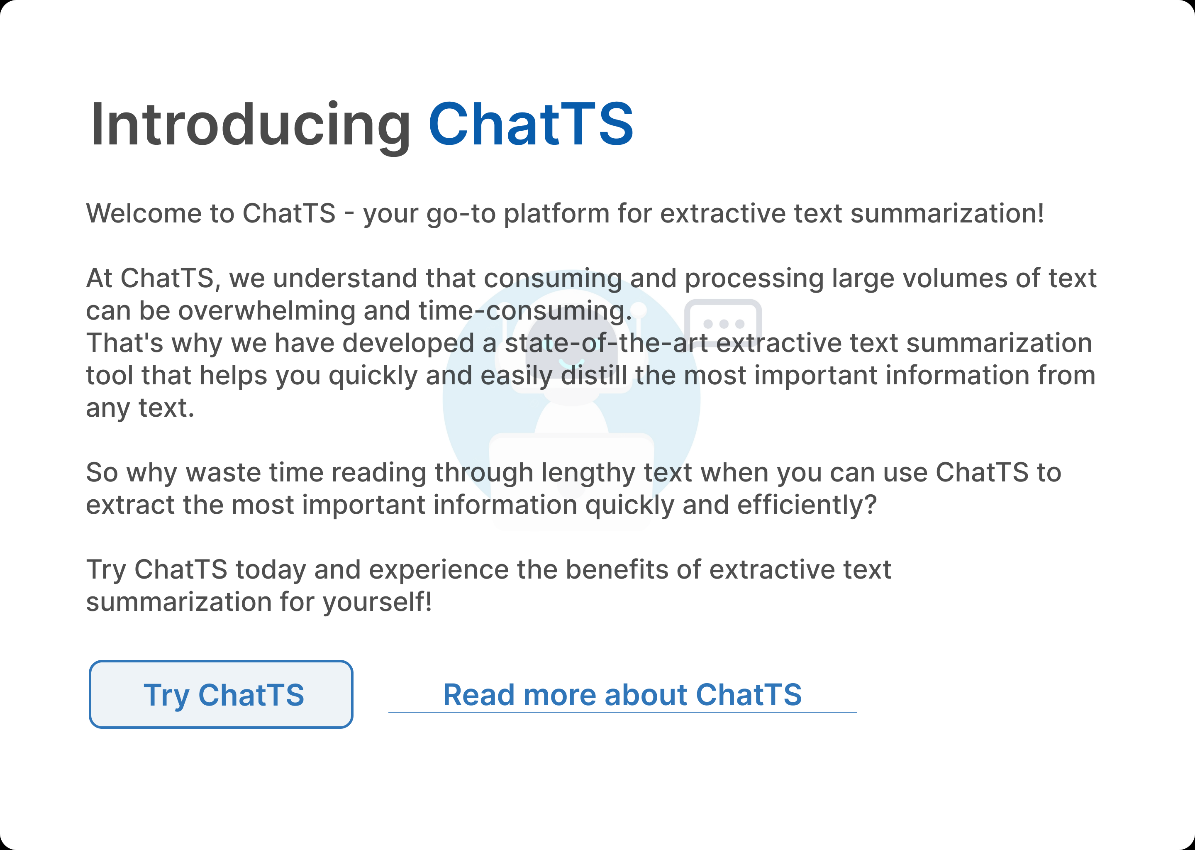
<script type="text/javascript" src="static/js/lib/showdown.min.js"></script>

</body> </html> **stories.yml** version: "2.0"

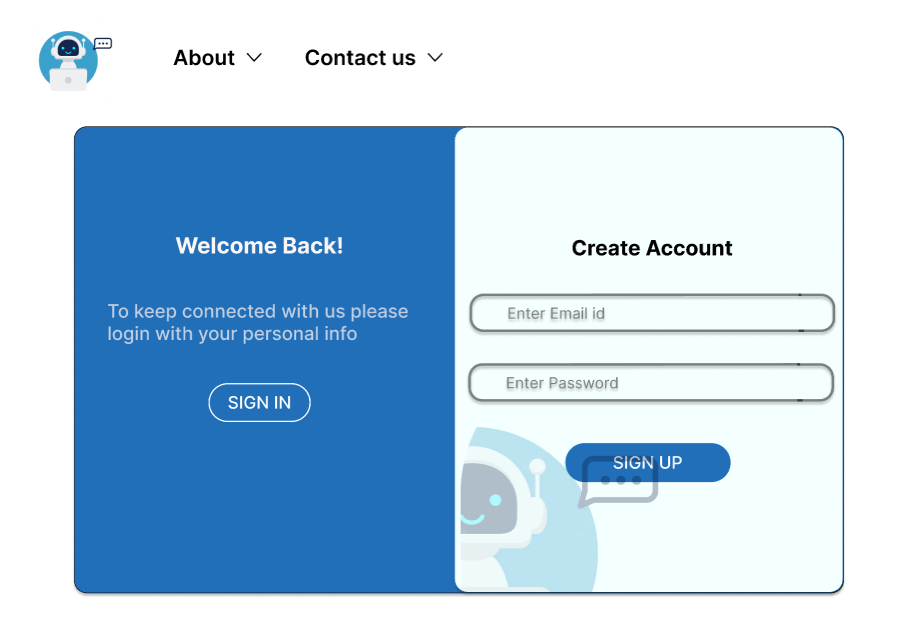
stories:

story: welcome pathsteps:

* intent: greet
* action: action\_get\_started
* intent: OPD - action: utter\_OPD - action:
* action\_submit\_OPD\_details
* action: user\_name\_form
* active\_loop: user\_name\_form
* active\_loop: null
* action: user\_number\_form
* active\_loop: user\_number\_form
* active\_loop: null
* action: user\_email\_form
* active\_loop: user\_email\_form
* active\_loop: null
* action: action\_submit\_OPD
* intent: affirm
* action: utter\_greet\_without\_OPD - story: OPD + close steps:
* active\_loop: user\_number\_form
* active\_loop: null
* action: user\_email\_form
* active\_loop: user\_email\_form
* active\_loop: nu



**Figure 10.3** Intro page of the Summarizer



**Figure 10.4** Creating account page

Backend Coding:

App Factory

from **flask** import **Flask**

import **os**

def **create\_app**(test\_config = None):

    app = **Flask**(\_\_name\_\_, instance\_relative\_config = True)

    app.config.**from\_mapping**(

        DATABASE = **os**.path.**join**(app.instance\_path, "Database.sqlite"),

        SECRET\_KEY = "dev",

        PERMANENT\_SESSION\_LIFETIME = 5000,

    )

    if test\_config is not None:

        app.config.**from\_mapping**(test\_config)

    try:

**os**.**makedirs**(app.instance\_path)

    except **OSError**:

        pass

    from . import **db**

**db**.**init\_app**(app)

    from .**auth** import auth\_bp

    app.**register\_blueprint**(auth\_bp)

**@app.route**("/")

    def **home**():

        return "", 200

    return app

Data Base

from **flask** import g, current\_app

import **click**

import **sqlite3**

def **get\_db**():

    if "db" not in g:

        g.db = **sqlite3**.**connect**(

            current\_app.config["DATABASE"],

            detect\_types= **sqlite3**.PARSE\_DECLTYPES

        )

        g.db.row\_factory = **sqlite3**.**Row**

    return g.db

def **close\_db**(e= None):

    db = g.**pop**("db", None)

    if db is not None:

        db.close()

def **init\_db**():

    db = **get\_db**()

    with current\_app.**open\_resource**("Schema.sql") as f:

        db.**executescript**(f.**read**().decode("utf-8"))

**@click.command**("init-db")

def **init\_db\_command**():

**init\_db**()

**click**.**echo**("The Database has been initialized")

def **init\_app**(app):

    app.teardown\_appcontext(**close\_db**)

    app.cli.add\_command(**init\_db\_command**)

Authentication

from **flask** import session, request, **Blueprint**, **redirect**

from **werkzeug**.**security** import **check\_password\_hash**, **generate\_password\_hash**

from **functools** import **wraps**

from **FlaskServer**.**db** import **get\_db**

auth\_bp = **Blueprint**("auth", \_\_name\_\_, url\_prefix = "/auth")

**@auth\_bp.route**("/register", methods = ["POST"])

def **register**():

    email = request.form["email"]

    password = request.form["password"]

    if email == "":

        return {"info": "email is required."}, 409

    elif password =="":

        return {"info": "Password is required."}, 409

    db = **get\_db**()

    ifExist = db.**execute**("select user\_email from user where user\_email = ?",(email,)).**fetchone**()

    if ifExist:

        return {"info": "User already exist in the database"}, 409

    db.**execute**("insert into user (user\_email, password) values (?, ?)",(email, **generate\_password\_hash**(password)))

    db.**commit**()

    data = db.**execute**("select id from user where user\_email = ?",(email,)).**fetchone**()

    return {"user\_email": email, "id" : data["id"], "info" : "registered Successfully"}, 200

**@auth\_bp.route**("/login", methods = ["POST"])

def **login**():

    email = request.form["email"]

    password = request.form["password"]

    db = **get\_db**()

    data = db.**execute**("select \* from user where user\_email = ?",(email,)).**fetchone**()

    if not data:

        return {"info": "Unauthorized"}, 401

    if not **check\_password\_hash**(data["password"],password):

        return {"info": "Unauthorized"}, 401

    session["user\_id"] = data["id"]

    return {"user\_email" : email, "id" : data["id"],"info" : "login Successful"}, 200

**@auth\_bp.route**("/logout", methods = ["GET"])

def **logout**():

    ses = session.**get**("user\_id", None)

    if ses is None:

        return {"info":"User not logged in"}, 401

    session.**clear**()

    return {"info": "logged out successfully"}, 200

def **login\_required**(func):

**@wraps**(func)

    def **wrapped**(\*args, \*\*kargs):

        if session.**get**("user\_id",None) is None:

            return {"error": "Unauthorized access"}, 401

        return func(\*kargs, \*\*kargs)

    return **wrapped**

Testing:

Testing App Factory

from **FlaskServer** import **create\_app**

def **test\_config**():

    assert not **create\_app**().testing

    assert **create\_app**({'TESTING': True}).testing

Testing Data Base

import **sqlite3**

import **pytest**

from **FlaskServer**.**db** import **get\_db**

def **test\_get\_close\_db**(app):

    with app.app\_context():

        db = **get\_db**()

        assert db is **get\_db**()

    with **pytest**.**raises**(**sqlite3**.**ProgrammingError**) as e:

        db.**execute**('SELECT 1')

    assert 'closed' in **str**(e.value)

def **test\_init\_db\_command**(runner, monkeypatch):

    class **Recorder**(**object**):

        called = False

    def **fake\_init\_db**():

**Recorder**.called = True

    monkeypatch.setattr('FlaskServer.db.init\_db', **fake\_init\_db**)

    result = runner.invoke(args=['init-db'])

**print**(result.output)

    assert 'initialized' in result.output

    assert **Recorder**.called

Testing Authentication

import **pytest**

from **flask** import g, session

from **FlaskServer**.**db** import **get\_db**

import **json**

def **test\_register**(client, app):

    username = "a"

    password = "a"

    response = client.post(

        '/auth/register',

        data={'email': username, 'password': password}

    )

    with app.app\_context():

        assert **get\_db**().**execute**(

            "SELECT \* FROM user WHERE user\_email = 'a'",

        ).**fetchone**() is not None

**@pytest.mark.parametrize**(('username', 'password', 'message'), (

    ('', '', 'email is required.'),

    ('a', '', 'Password is required.'),

    ('test', 'test', 'User already exist in the database'),

))

def **test\_register\_validate\_input**(client, username, password, message):

    response = client.post(

        '/auth/register',

        data={'email': username, 'password': password}

    )

    assert message in **json**.**loads**(response.data.decode())["info"]

def **test\_login**(client, auth):

    response = auth.login()

    with client:

        client.get("/")

        assert session["user\_id"] == 1

**@pytest.mark.parametrize**(('username', 'password', 'message'), (

    ('a', 'test', 'Unauthorized'),

    ('test', 'a', 'Unauthorized'),

    ('test', 'test', 'login Successful'),

    ("", "", "Unauthorized")

))

def **test\_login\_validate\_input**(auth, username, password, message):

    response = auth.login(username, password)

    assert message in **json**.**loads**(response.data.decode())["info"]

def **test\_logout**(client, auth):

    auth.login()

    with client:

        auth.logout()

        assert 'user\_id' not in session

        response = auth.logout()

        assert "User not logged in" in **json**.**loads**(response.data.decode())["info"]

# 

# REFERENCES

1. RASA Documentation: **https://rasa.com/docs/**

2. RASA Tutorial Videos:

**https://**[**www.youtube.com/playlist?list=PL75e0qA87dlHQny7z**](http://www.youtube.com/playlist?list=PL75e0qA87dlHQny7z) **43 Ndu ZHPo6qdcRc**

3]. GEEKS FOR GEEKS

**https://**[**www.geeksforgeeks.org/chatbots-using-python-and-rasa/**](http://www.geeksforgeeks.org/chatbots-using-python-and-rasa/)

4]. Research paper on implementing a chatbot using RASA:

Ujala P, Deepa Gopal, Athira S Babu, Anju S Pillai(2015)’Building Chatbots Using the RASA Stack: A Comprehensive Study’ Proceedings of the 2nd International Conference on Intelligent Computing Link: <https://ieeexplore.ieee.org/document/>