**DOCUMENTATION**

|\_\_ **data**

| |\_\_ SIP New Script.docx

|

|\_\_ **definitions**

| |\_\_ definition.py

|

|\_\_ **src**

| |\_\_ code

| |\_\_ Preprocess.py

| |\_\_ Vector\_Database.py

| |\_\_ Prompt\_Engineering.py

| |\_\_ Voice\_Agent.py

|

| |\_\_ Vector\_Storage

| |\_\_ vector\_store.index

| |\_\_ vector\_store\_data.pkl

|

|\_\_ **.env**

| |\_\_ {openai api-key}

|

|\_\_ **requirements.txt**

|

|\_\_ **venv** (you have to make a virtual env. for the installation of requirement.txt

1. **Data**: This folder contains the raw **DOCX file**.
2. **Definitions**: This folder contains a **definition.py f**ile, which holds the path to the raw DOCX file.
3. **Src**: it has a **code folder** containing all files and also has a subfolder **vector storage folder**, which holds the vector data.
4. **.env**: This file contains the **OpenAI API key. {"paste your OpenAI api key in the strings"}**
5. **requirements.txt**: This file lists all the **required libraries**.

**Note**: You need to create a **virtual environment**.  
 1.The command is: python -m venv venv

2.In PowerShell or Command Prompt, type: venv\Scripts\activate  
 3.Then, install the required libraries by typing: pip install -r requirements.txt

1. **Preprocess.py**
   * The Preprocess class reads a .docx file specified by the DATA variable and provides two methods:
2. getText: Extracts and returns the text from all paragraphs in the Word document, joining them with newline characters.
3. punctuationRemove: Converts the text to **lowercase** and removes all **punctuation** marks for gaining the accuracy in the NLP process and returning the cleaned text.
4. **Vector\_Database.py**
   * The VectorDatabase class is designed to convert text into embeddings using OpenAI's API, store them as vectors, and save the data for later retrieval.
5. **\_\_init\_\_(self)**: Initializes the class by setting the path for storing vectors and instantiating the Preprocess class (for text preprocessing).
6. **get\_openai\_embeddings(self, text\_list)**: Takes a list of texts and requests their embeddings from OpenAI’s API (text-embedding-ada-002 model). It returns a list of embeddings.
7. **store\_text\_as\_vectors(self)**:
   * Ensures the directory for vector storage exists.
   * Retrieves preprocessed text from the Preprocess class (using punctuationRemove).
   * Calls get\_openai\_embeddings to get embeddings for the text data.
   * Converts the embeddings into a NumPy array and initializes a FAISS index to store the vectors.
   * Saves the FAISS index and the original text data as a pickle file in the specified directory.
8. **Prompt\_Engineering.py**
   * The QueryHandler class is designed to handle queries and generate responses using OpenAI's API and a stored vector database. It performs the following tasks:
9. \_\_init\_\_(self):
   * Initializes the class by setting up the path to the vector storage folder.
   * Loads the FAISS index and extracted text data from the specified paths using faiss.read\_index and pickle.load.
10. generate\_embedding(self, text):
    * Takes a query text and uses OpenAI's text-embedding-ada-002 model to generate an embedding (vector) for the text.
    * Returns the embedding as a NumPy array of type float32.
11. retrieve\_relevant\_data(self, query, k=3):
    * Takes a query and retrieves the top k most relevant text entries from the vector database using FAISS for similarity search.
    * The query is first converted into a vector using generate\_embedding.
    * It performs a similarity search in the FAISS index and returns the corresponding relevant text data based on the top k closest matches.

4. remove\_special\_characters(self, response):

* + Removes unwanted special characters (\*, #, /) from the response by OpenAI model string to clean the generated text.

5. generate\_response(self, prompt, k=3):

* + Retrieves relevant contexts for the given prompt by calling retrieve\_relevant\_data.
  + Combines the relevant contexts and forms a final prompt for generating a response using the OpenAI gpt-4-2024-08-06 model.
  + Sends the prompt to the OpenAI API and generates a response based on the provided context.
  + Calls remove\_special\_characters to clean the response by removing special characters (\*, #, /).

1. **Voice\_Agent.py**
   * The VoiceAgent class is designed to integrate voice recognition and text-to-speech functionality, allowing a user to interact with a system via spoken commands. The class uses the speech\_recognition, pyttsx3, and OpenAI libraries, and interacts with the QueryHandler class to generate context-based responses.
2. **\_\_init\_\_(self)**:
   * Initializes the VoiceAgent by setting up:
     + pyttsx3 for text-to-speech (TTS) functionality.
     + QueryHandler for generating context-aware responses. (prompt\_engineering.py class)
     + speech\_recognition for recognizing speech.
3. **listen\_for\_command(self)**:
   * Listens for a voice command using the microphone.
   * Adjusts for ambient noise and then listens for audio.
   * Uses Google’s speech recognition API (recognize\_google) to transcribe the audio into text and returns the transcript.
   * Handles errors such as unrecognized speech or API request failure.
4. **speak\_response(self, response)**:
   * Converts the provided text response into speech using pyttsx3.
   * The response is spoken aloud using the system's default speech engine.
   * **Highlight:** 
     1. In **if \_\_name\_\_ == "\_\_main\_\_":** 
        1. **Step 1:** listen\_for\_command() listens to the user's voice and converts the speech into text (prompt).
        2. **Step 2:** generate\_response(prompt) processes the prompt using the QueryHandler to generate a relevant response.
        3. **Step 3:** speak\_response(response) converts the generated response back into speech, so the user can hear it.

**Note:**

* **Evaluation and Improvement Suggestions**
  + For evaluation and improvement, we need to **fine-tune** the model on the appropriate dataset with the exact lines required to provide the user with accurate responses. **Fine-tuning** is aimed at improving accuracy, ensuring the model provides precise answers according to the user's needs.
  + For tone and emotion, we can use **NLTK's emotion** and **sentiment analysis**. By analyzing the emotion and sentiment of the pitch, we can modify the response to match the detected emotion or sentiment.