

Hackathon

Sun Bot - Documentation

Version 1.0 - Developed & Documented by **M. Sana Ur Rehman**



Thanks to our amazing instructors

Sir Irfan Malik, Dr. Sheraz Nazeer, Sir Haris

Introduction:

SunBot is an AI-powered chatbot, an innovative solution designed to streamline your interaction with textual data. This intelligent assistant accepts file uploads in PDF, DOCX, or ZIP formats, providing a seamless way to extract and analyze information. Engage in natural language conversations by posing questions to the chatbot, allowing you to effortlessly retrieve insights from your uploaded documents. Experience the future of document interaction with our AI chatbot.

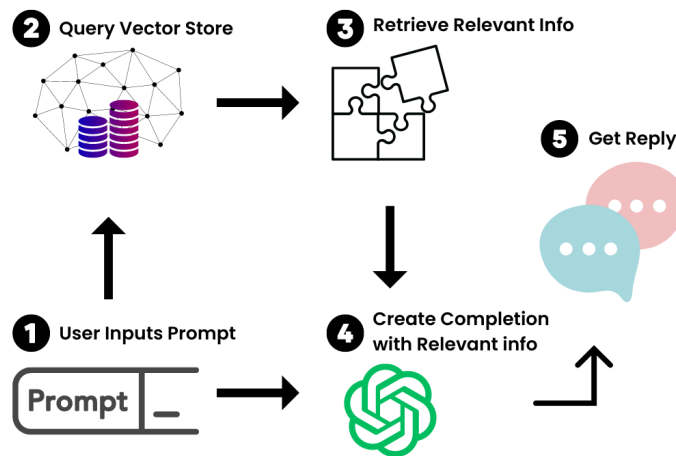
[Video Demo & Explanation](#)



Say Goodbye to Long Boring Files - Talk to Data to Gain Better Insights!

How it Works

In essence, Sun Bot seamlessly integrates these six key components, creating a comprehensive and intelligent system that excels in understanding, processing, and delivering insights from uploaded documents through intuitive conversational interactions. Github Repo: <https://github.com/SunnyRehman/sunbot-xeven-hackathon>



Specifications:

- Platform: LangChain
- Language: Python
- LLM (Large Language Model) : gpt-3.5-turbo-16k
- Additional Tools: Streamlit.io for deployment

Sun Bot Operation Overview:

File Loading:

Sun Bot begins by facilitating effortless file uploads in various formats, such as PDF, DOCX, or ZIP. This foundational step ensures that users can seamlessly integrate their textual data into the AI-powered system.


Custom Loader for Zip Files:

Sun Bot features an exclusive custom loader designed explicitly for ZIP files. This specialized functionality ensures that, even if your documents are compressed within a ZIP archive, Sun Bot adeptly extracts the files and their contents. This seamless integration further simplifies the user experience, allowing effortless engagement with the chatbot regardless of the file packaging, providing a hassle-free approach to document interaction.

PDF Loader using PDF Plumber:

A PDF Loader function handles contents of PDF files and gets contents from the file using `pdfplumber.open` function.

Why I chose to use PDFPlumber over MuPDF for PDF loading

- **Versatility:** PDFPlumber offers broader compatibility with various PDF formats, ensuring Sun Bot can handle diverse documents seamlessly.
 - **Rich Features:** PDFPlumber provides an extensive feature set, including text extraction, table parsing, and annotation retrieval, enhancing Sun Bot's capabilities for extracting valuable information.
 - **Ease of Use:** PDFPlumber's user-friendly design and well-documented functionalities simplify integration, accelerating Sun Bot's development process.
 - **Community Support:** PDFPlumber benefits from an active and engaged community, ensuring ongoing support, updates, and bug fixes for Sun Bot.
- 

-
- **Parsing Complexity:** PDFPlumber excels in parsing intricate PDF structures, making it ideal for documents with complex layouts or multiple layers, contributing to Sun Bot's adaptability.

Docx File Loader:


In our document processing pipeline for Sun Bot, I opted for Docx2txt as our tool of choice for loading DOCX files. The decision to choose Docx2txt over alternatives, such as the Unstructured library, was driven by its simplicity and efficiency in extracting text content from DOCX files. Docx2txt aligns with our emphasis on straightforward implementation, offering a lightweight solution that seamlessly integrates with Sun Bot's document loading process. Its ease of use, reliability, and ability to swiftly retrieve text content from DOCX files made Docx2txt the preferred option for enhancing Sun Bot's overall performance in handling diverse document formats.

Text Splitting:

In order to split the text into chunks, I have implemented the following:

1. **Split by Character:** `get_chunks_by_character()` function

The `get_chunks_by_character` function facilitates the division of a given text into manageable chunks using the `CharacterTextSplitter` module. With a specified separator and parameters such as chunk size, overlap, and length function, this function ensures effective segmentation of the input text. In essence, it systematically breaks down the text into smaller, comprehensible units, enhancing the efficiency of subsequent processes, such as text embedding or retrieval. Users can adjust the parameters to tailor the chunking process based on the characteristics of their specific textual data, providing flexibility and adaptability within the document processing pipeline.



2. **Recursively Split by Character:** `recursively_split_by_character()`

The `recursively_split_by_character` function is designed to iteratively break down a given text into smaller chunks, employing a recursive approach for enhanced precision. By utilizing the `CharacterTextSplitter` module with specified parameters such as chunk size, overlap, and minimum chunk size, this function ensures a thorough and detailed segmentation of the input text. The recursive nature allows each initial chunk to be further subdivided, creating a hierarchical structure of text fragments. Subsequently, the function filters out small chunks that fall below the defined minimum size, producing a refined set of text segments. This recursive splitting mechanism provides users with a flexible and dynamic tool for meticulous text processing, particularly useful when dealing with intricate document structures or when seeking fine-grained control over the segmentation process.

Embedding Generation:

The chatbot leverages state-of-the-art embedding models to convert textual information into rich numerical representations. This embedding generation process forms the core of Sun Bot's ability to understand and process user queries with a deep understanding of the uploaded content.

To generate text embeddings, I have used two specified embedding hugging face models i.e.

1. 'sentence-transformers/all-MiniLM-L12-v2'



2. "BAAI/bge-small-en-v1.5"

Comparative Analysis of Results

1. 'sentence-transformers/all-MiniLM-L12-v2'

- Vector Size: 384
- Max Sequence Length: 128
- Case Sensitivity: False
- Transformer Model: BertModel
- Word Embedding Dimension: 384
- Pooling Mode (CLS Token): False
- Pooling Mode (Mean Tokens): True
- Pooling Mode (Max Tokens): False
- Pooling Mode (Mean Sqrt Len Tokens): False
- Normalization: Applied

```
client=SentenceTransformer( (0): Transformer({'max_seq_length': 128, 'do_lower_case': False}) with  
Transformer model: BertModel (1): Pooling({'word_embedding_dimension': 384,  
'pooling_mode_cls_token': False, 'pooling_mode_mean_tokens': True, 'pooling_mode_max_tokens':  
False, 'pooling_mode_mean_sqrt_len_tokens': False}) (2): Normalize() )  
model_name='sentence-transformers/all-MiniLM-L12-v2' cache_folder=None model_kwargs={}  
encode_kwargs={}
```

2. "BAAI/bge-small-en-v1.5"

- Max Sequence Length: 512
- Vector Size: 384
- Case Sensitivity: True
- Transformer Model: BertModel
- Word Embedding Dimension: 384

-
- Pooling Mode (CLS Token): True
 - Pooling Mode (Mean Tokens): False
 - Pooling Mode (Max Tokens): False
 - Pooling Mode (Mean Sqrt Len Tokens): False
 - Normalization: Applied

```
client=SentenceTransformer( (0): Transformer({'max_seq_length': 512,
'do_lower_case': True}) with Transformer model: BertModel (1):
Pooling({'word_embedding_dimension': 384, 'pooling_mode_cls_token': True,
'pooling_mode_mean_tokens': False, 'pooling_mode_max_tokens': False,
'pooling_mode_mean_sqrt_len_tokens': False}) (2): Normalize() )
model_name='BAAI/bge-small-en-v1.5' cache_folder=None model_kwargs={}
encode_kwargs={}
```

Comparative Analysis:

Max Sequence Length:

- 'sentence-transformers/all-MiniLM-L12-v2': 128
- "BAAI/bge-small-en-v1.5": 512
- The second model allows for a longer maximum sequence length, accommodating larger text inputs.

Case Sensitivity:

- 'sentence-transformers/all-MiniLM-L12-v2': False
- "BAAI/bge-small-en-v1.5": True
- The second model considers case sensitivity, potentially capturing nuanced information from the input text.

Pooling Mode (CLS Token, Mean Tokens, Max Tokens, Mean Sqrt Len Tokens):

-
- Both models use BertModel for transformation and offer various pooling modes, allowing users to choose the most suitable option based on their specific requirements.

Word Embedding Dimension:

- Both models share the same word embedding dimension (384), ensuring consistency in the dimensionality of the output embeddings.

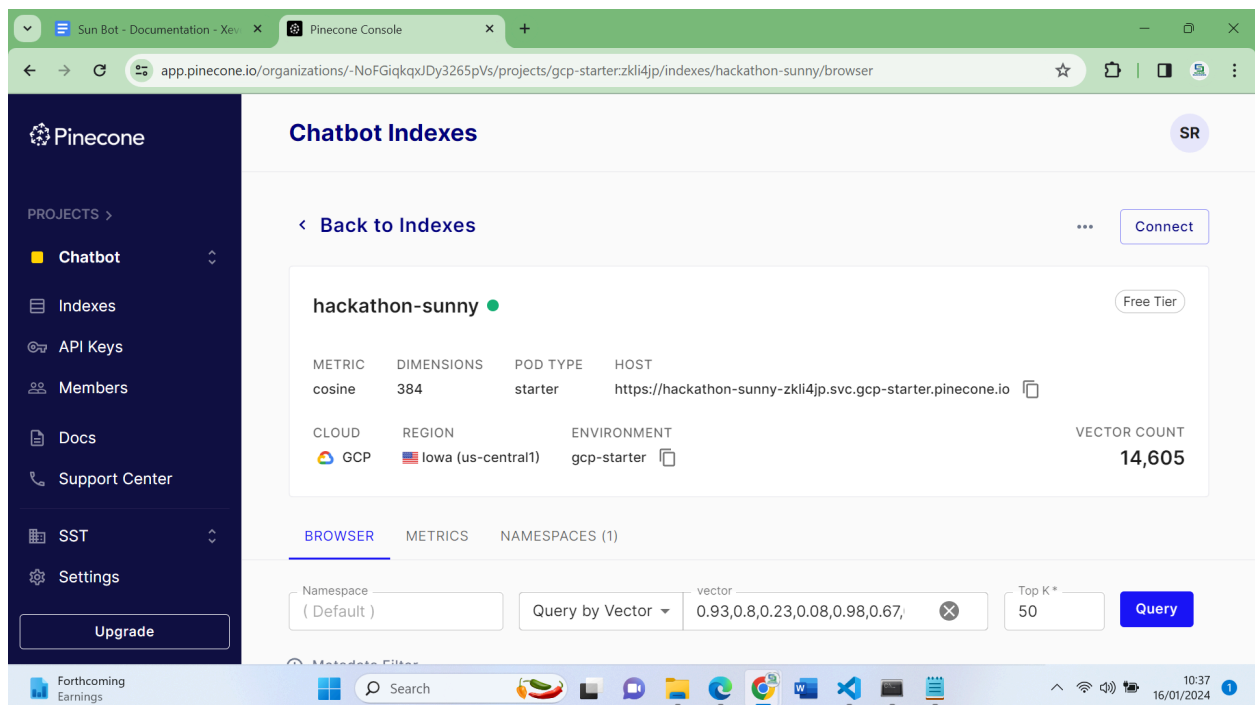
Normalization:

- Both models apply normalization to the embeddings, contributing to a standardized and comparable representation.

Vector Database:

Sun Bot utilizes a sophisticated vector database to store and organize the generated embeddings. This strategic approach allows for efficient retrieval of relevant information during user interactions, ensuring quick and accurate responses. I have integrated it with the Pine Cone database.





Retrievers:

To enhance the conversational experience, Sun Bot incorporates retrievers that intelligently search the vector database. These retrievers play a crucial role in understanding user queries and retrieving pertinent information from the stored embeddings.

1. Vector store-backed retriever
2. Parent Document Retriever

Output using LLM (Language Model):

The final step involves presenting the user with coherent and contextually relevant responses using a Language Model (LLM). Sun Bot's interaction output is powered by advanced language models, ensuring a natural and informative dialogue between the user and the AI chatbot. We're using gpt-3.5-turbo-16k. The model is set to 0 temperature.

SunBot Local Setup Guide

SunBot Local Setup Guide

This guide provides instructions on how to set up and run the Sunbot locally on your machine.

Prerequisites

Ensure you have the following prerequisites installed on your machine:

- Python (version 3.9)
- Pip (Python package installer)
- Git

Installation

1. Clone the repository to your local machine:
2. `git clone https://github.com/SunnyRehman/sunbot-xeven-hackathon`
3. Navigate to the project directory:
4. `cd your-chatbot-repo`
5. Create an Anaconda Environment
`conda create -n name_of_environment python=3.9`

-
6. Activate the Conda Environment
`conda activate name_of_environment`
 7. Install the required Python packages:
`pip install -r requirements.txt`

Configuration

1. Set up environment variables:
Create a `.streamlit` directory and create a `secrets.toml` file inside it and add the following:

`OPENAI_API_KEY=your_openai_api_key`

`PINECONE_API_KEY=your_pinecone_api_key`

2. `PINECONE_ENV=your_pinecone_environment`
3. Replace `your_openai_api_key`, `your_pinecone_api_key`, and `your_pinecone_environment` with your actual API keys.

Running the Chatbot

1. Execute the following command to start the chatbot:
`streamlit run app.py`
2. Open your web browser and navigate to <http://localhost:8501>
3. Interact with the chatbot by following the on-screen instructions

Thank You!

