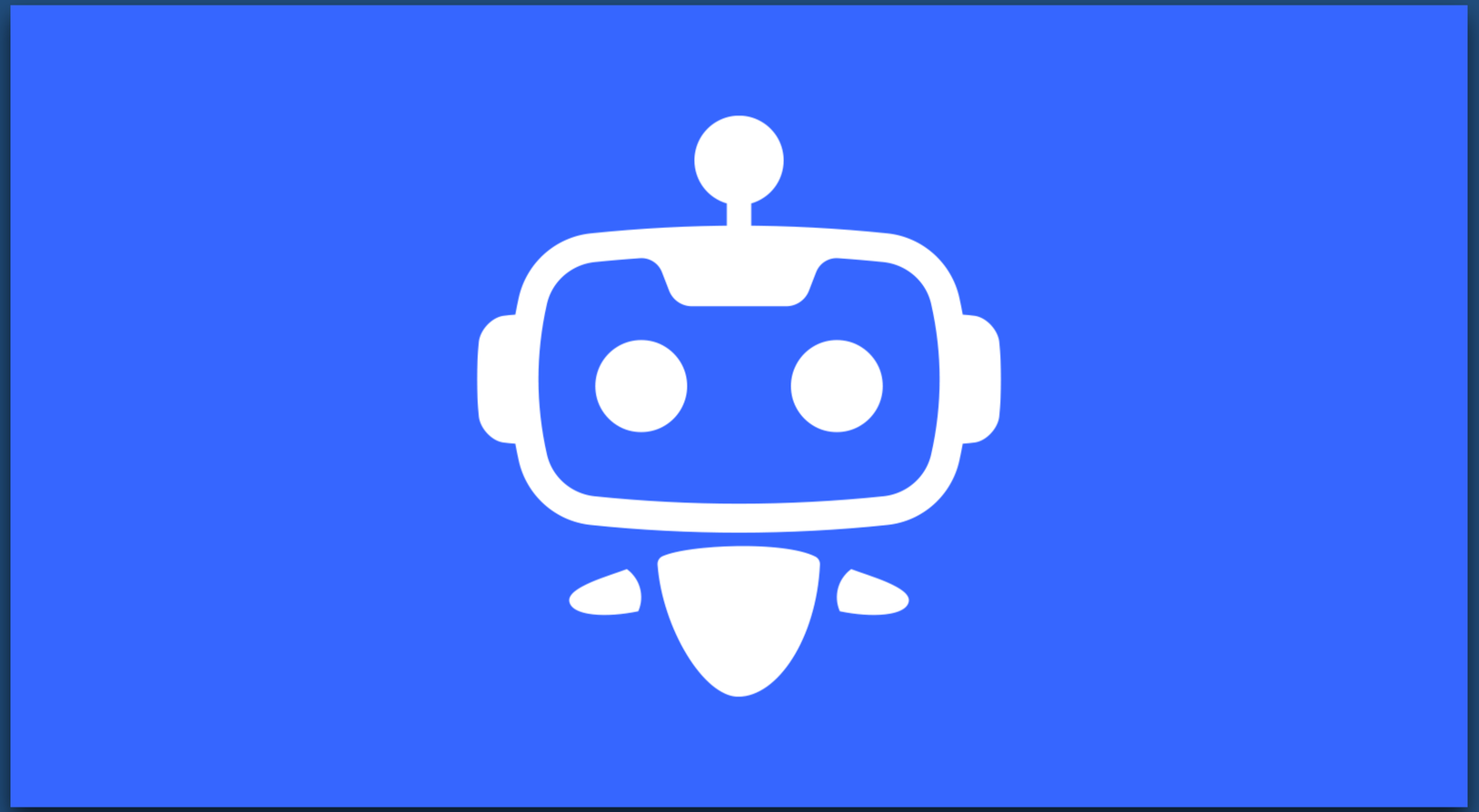


LLM Agent

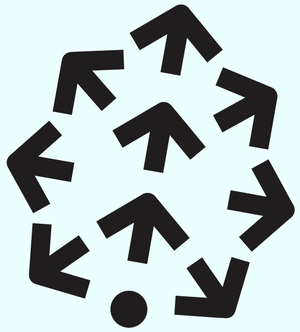
Text Mining Project
Professor Flora Amato
By Yahya Momtaz, Tirlangi Harsha Vardhan

What is Langchain?

LangChain is a framework for developing applications powered by language models.

It enables applications that are:

Data-aware: connect a language model to other sources of data
Agentic: allow a language model to interact with its environment



Pinecone

Vector databases store and query embeddings quickly and at scale

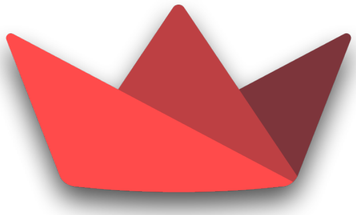
Pinecone makes it easy to provide long-term memory for high-performance AI applications. It's a managed, cloud-native vector database with a simple **API** and no infrastructure hassles.

Pinecone serves fresh, filtered query results with low latency at the scale of billions of **vectors**.

Vector databases like **Pinecone** offer optimized storage and querying capabilities for embeddings.

Traditional **scalar-based** databases can't keep up with the complexity and scale of such data, making it difficult to extract insights and perform real-time analysis.

Vector databases combine the familiar features of traditional databases with the optimized performance of vector indexes.



Streamlit

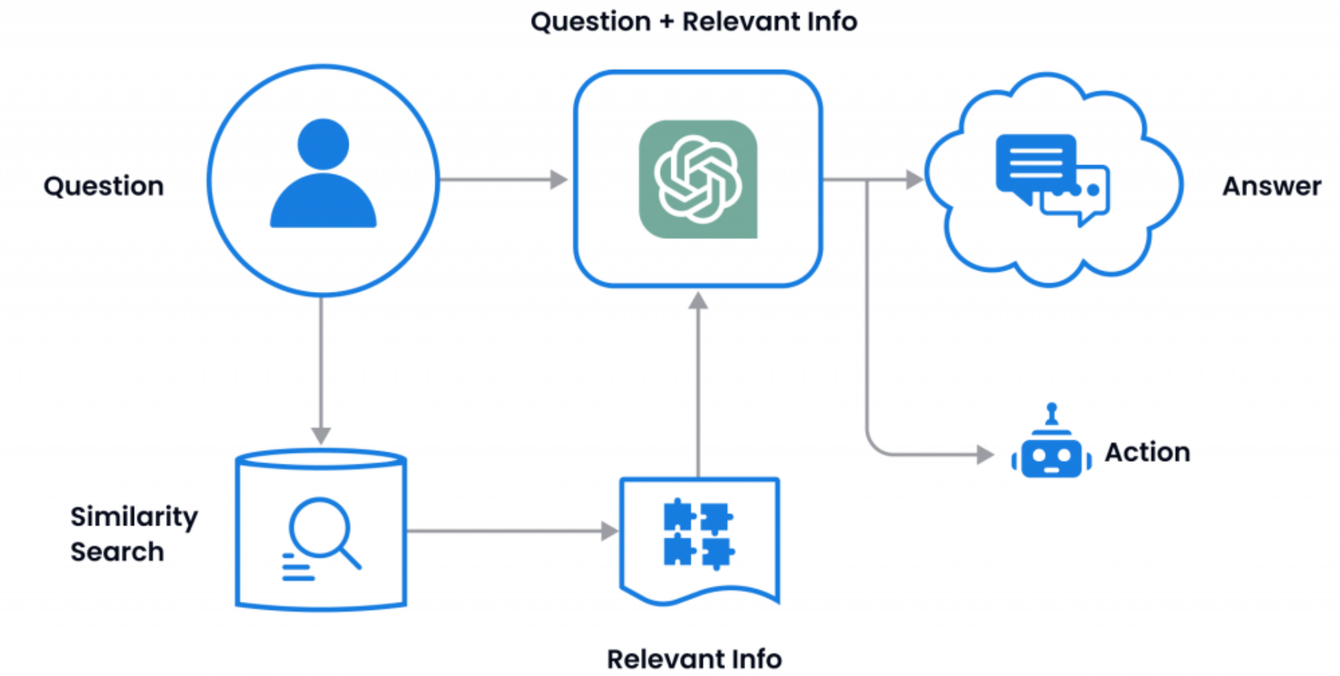
Streamlit is an open-source Python library that makes it easy to create and share beautiful, custom web apps for machine learning and data science. In just a few minutes you can build and deploy powerful data apps.

Streamlit is designed to be used by data scientists and machine learning engineers who do not have experience with web development. It provides a simple, intuitive API that allows you to create interactive web apps without having to worry about the underlying HTML, CSS, and JavaScript.

Goal

Build an Interactive Chatbot
with Langchain, ChatGPT, Pinecone, and Streamlit

How it works?


Question + Relevant Info
Question
Answer
Action
Similarity
Search
Relevant Info

Libraries

streamlit: This library helps us to create interactive web apps for machine learning and data science projects.

streamlit_chat: This Streamlit component is used for creating the chatbot user interface.

langchain: This is a framework for developing applications powered by language models. It provides a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications.

sentence_transformers: This library allows us to use transformer models like BERT, RoBERTa, etc., for generating semantic representations of text (i.e., embeddings), which we'll use for our document indexing.

openai: This is the official OpenAI library that allows us to use their language models, like GPT-3.5-turbo, for generating human-like text. unstructured and unstructured[local-inference]: These are used for document processing and managing unstructured data.

pinecone-client: This is the client for Pinecone, a vector database service that enables us to perform similarity search on vector data.

Index Document

It involves loading the documents from a directory.
I used the DirectoryLoader class provided by LangChain to achieve this.
This class accepts a directory as input and loads all the documents present in it.

```
from langchain.document_loaders import DirectoryLoader

directory = '/content/data'

def load_docs(directory):
  loader = DirectoryLoader(directory)
  documents = loader.load()
  return documents

documents = load_docs(directory)
len(documents)
```

Splitting documents

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

def split_docs(documents,chunk_size=500,chunk_overlap=20):
  text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk_overlap=chunk_overlap)
  docs = text_splitter.split_documents(documents)
  return docs

docs = split_docs(documents)
print(len(docs))
```

After loading the documents, the script proceeds to split these documents into smaller chunks.The size of the chunks and the overlap between these chunks can be defined by the user. This is done to ensure that the size of the documents is manageable and that no relevant information is missed out due to the splitting. The RecursiveCharacterTextSplitter class from LangChain is used for this purpose.

Creating embeddings

```
from langchain.embeddings import SentenceTransformerEmbeddings
embeddings = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")
```

Once the documents are split, we need to convert these chunks of text into a format that our AI model can understand.

This is done by creating embeddings of the text using SentenceTransformerEmbeddings class provided by LangChain.

Storing embeddings in Pinecone

```python
import pinecone
from langchain.vectorstores import Pinecone
pinecone.init(
    api_key="",
    environment=""
)
index_name = "langchain-chatbot"
index = Pinecone.from_documents(docs, embeddings, index_name=index_name)
```

After the embeddings are created, they need to be stored in a place from where they can be easily accessed and searched. Pinecone is a vector database service that is perfect for this task. The embeddings are stored in Pinecone using the Pinecone class from LangChain.

Build the App with Streamlit

- **Session State Initialisation:** Firstly, we initialise two lists 'responses' and 'requests' within Streamlit's session state. These lists store the history of bot responses and user requests respectively.

- **ConversationBufferWindowMemory:** This memory structure is instantiated with a size of 3, meaning that our chatbot would remember the last three interactions, keeping a manageable memory size for efficiency.

- **PromptTemplate Construction:** We construct a PromptTemplate for our chatbot. The template contains instructions to the language model (LLM), providing structure and context to the input for the LLM to generate a response. Langchain provides different types of MessagePromptTemplate, which includes AIMessagePromptTemplate, SystemMessagePromptTemplate, and HumanMessagePromptTemplate, for creating different types of messages.

Build the App with Streamlit

```
if 'responses' not in st.session_state:
    st.session_state['responses'] = ["How can I assist you?"]

if 'requests' not in st.session_state:
    st.session_state['requests'] = []

llm = ChatOpenAI(model_name="gpt-3.5-turbo", openai_api_key="")

if 'buffer_memory' not in st.session_state:
            st.session_state.buffer_memory=ConversationBufferWindowMemory(k=3,return_messages=True)


system_msg_template = SystemMessagePromptTemplate.from_template(template="""Answer the question as truthfully as possible using the provided context,
and if the answer is not contained within the text below, say 'The information provided to me does not address the query you posed.'""")


human_msg_template = HumanMessagePromptTemplate.from_template(template="{input}")

prompt_template = ChatPromptTemplate.from_messages([system_msg_template, MessagesPlaceholder(variable_name="history"), human_msg_template])

conversation = ConversationChain(memory=st.session_state.buffer_memory, prompt=prompt_template, llm=llm, verbose=True)
```

Make UI

```python
import streamlit as st
from streamlit_chat import message
from utils import *

st.set_page_config(page_title="Law Agent")

st.title(':blue[Law Agent] is here :bulb:')

with st.sidebar:
    st.title('Law Agent')
    #st.sidebar.get_option("theme.primaryColor")

    st.markdown('''
    Made by **Yahya Momtaz**
    - [GitHub](https://github.com/yahyamomtaz)
    - [Kaggle](https://www.kaggle.com/yahyamomtaz)
    - [Linkedin](https://www.linkedin.com/in/yahya-momtaz-601b34108)
    - This agent is based on information regarding the split of marital property.
    ''')

    st.image("data/unina2.png")
```

```python
with textcontainer:
    query = st.text_input("Query: ", key="input")
    if query:
        with st.spinner("typing..."):
            conversation_string = get_conversation_string()
            # st.code(conversation_string)
            refined_query = query_refiner(conversation_string, query)
            st.subheader("Refined Query:")
            st.write(refined_query)
            context = find_match(refined_query)
            # print(context)
            response = conversation.predict(input=f"Context:\n {context} \n\n Query:\n{query}")
        st.session_state.requests.append(query)
        st.session_state.responses.append(response)
with response_container:
    if st.session_state['responses']:

        for i in range(len(st.session_state['responses'])):
            message(st.session_state['responses'][i],key=str(i))
            if i < len(st.session_state['requests']):
                message(st.session_state["requests"][i], is_user=True,key=str(i)+ '_user')
```

Make UI

The Streamlit library allows us to quickly build a user-friendly interface for our chatbot application. The **st.title** function is used to display the chatbot's title at the top of the interface.

The user's queries and the chatbot's responses are displayed in a conversation format using the **st.container** and **st.text_input** functions.

Initialize the Language Model

- **ChatOpenAI Initialisation:** We then create an instance of ChatOpenAI, which utilizes the powerful gpt-3.5-turbo model from OpenAI for language understanding and generation.

```
llm = ChatOpenAI(model_name="gpt-3.5-turbo", openai_api_key="")
```

- **ConversationChain Setup:** Finally, we set up the ConversationChain using the memory, prompt template, and LLM we've prepared. The ConversationChain is essentially a workflow of how our chatbot would operate: it leverages user input, prompt template formatting, and the LLM to conduct an interactive chat.

```
conversation = ConversationChain(memory=st.session_state.buffer_memory, prompt=prompt_template, llm=llm, verbose=True)
```

Regenerate Queries with OpenAI

The **query_refiner** function is used to take the user's query and refine it to ensure it's optimal for providing a relevant answer. It uses OpenAI's DaVinci model to refine the query based on the current conversation log.

With these utility functions, the chatbot can not only generate responses but also refine the user's queries and find the most relevant answers. It ensures a more effective and user-friendly chatbot experience.