SFBU Customer Support System - text

By Emily Weng 20016

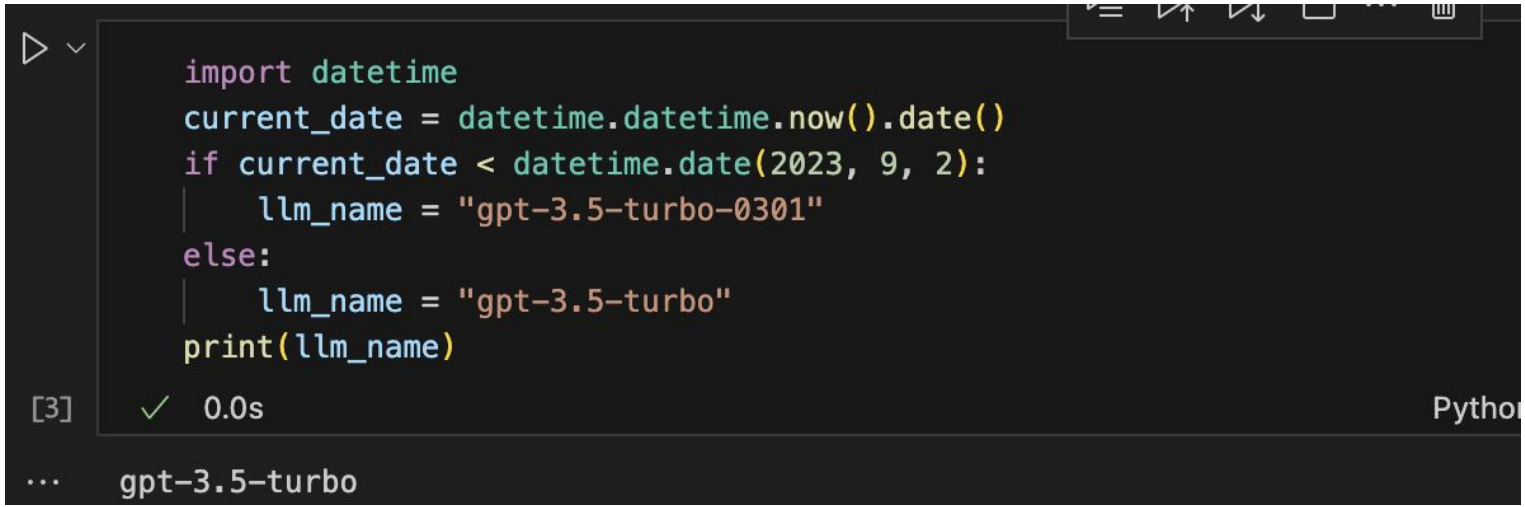
Process for the project implementation of Chat

- Step 1: Overview of the workflow for RAG
  - Document Loading
  - Splitting the content to create chunks (splits)
  - Indexing each chunk (split) by embeddings
  - Storage - Vectorstore
  - Retrieval to select relevant chunks (splits)

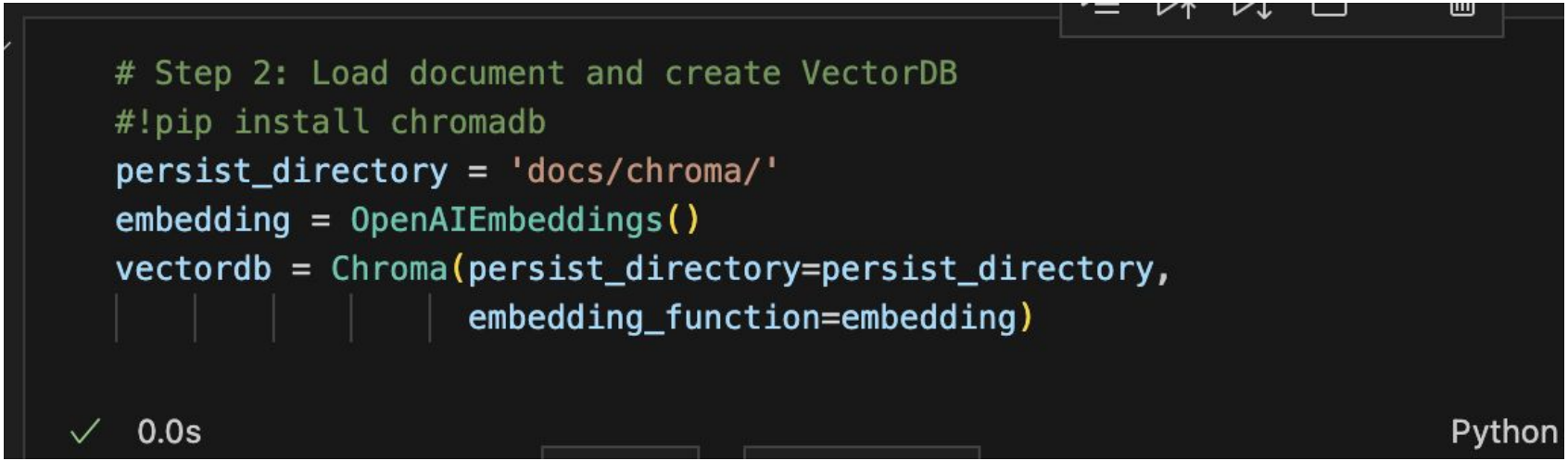
Set up API KEY

```
import os
import openai
import sys
sys.path.append('../..')

import panel as pn  # GUI
pn.extension()


from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

openai.api_key  = os.environ['OPENAI_API_KEY']
```

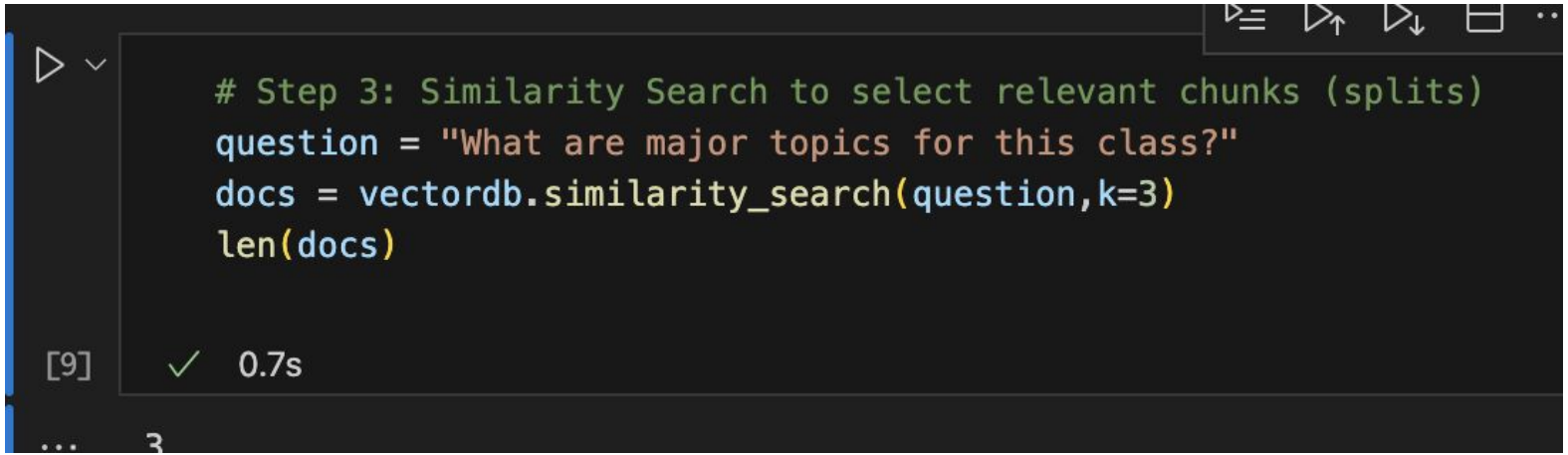
Step 1.1.2: LLM model selection



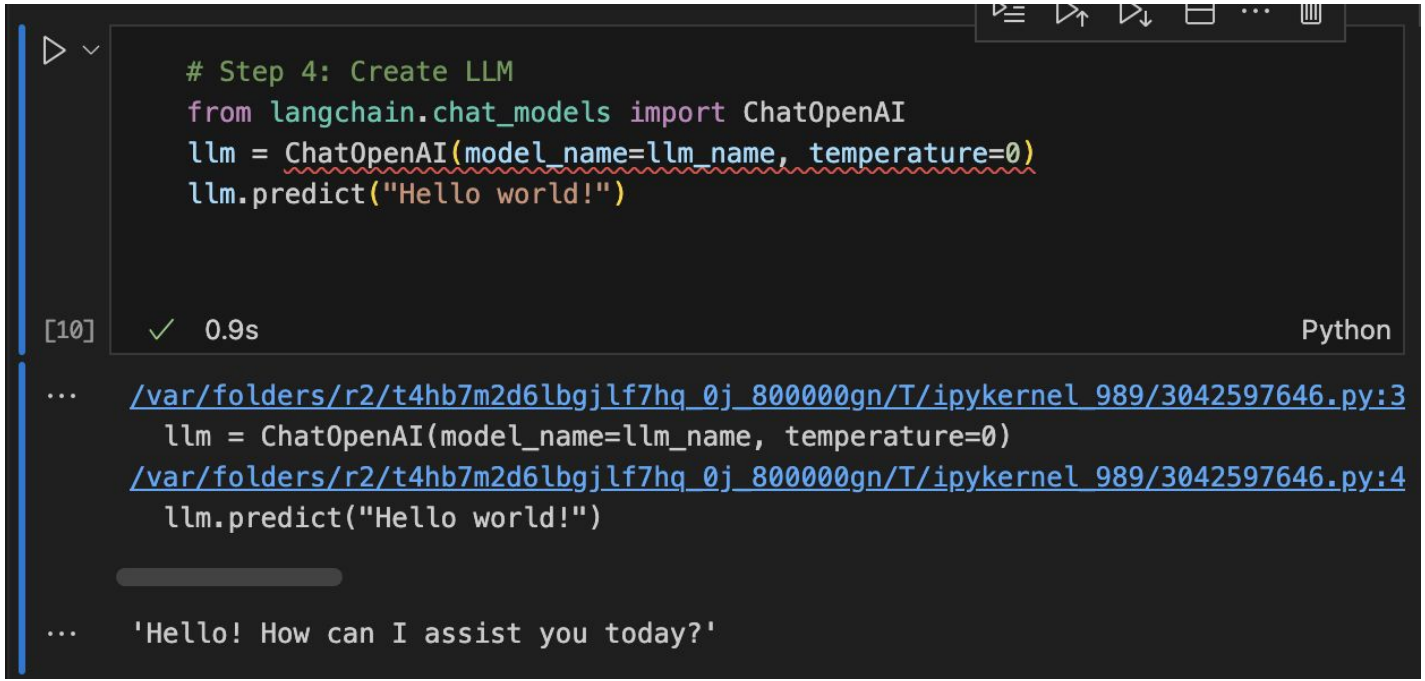
Step 2: Load document and create VectorDB (i.e., Vectorstore)



Step 3: Similarity Search to select relevant chunks (splits)



Step 4: Create LLM



Step 5: Retrieval QA Chain
Step 5.1: Create a prompt template

```
# Step 5: RetrievalQA Chain
# Step 5.1: Craete a prompt template
from langchain.prompts import PromptTemplate

template = """Use the following pieces of \
   context to answer \
   the question at the end. If you don't know \
   the answer, \
   just say that you don't know, don't try \
   to make up an \
   answer. Use three sentences maximum. \
   Keep the answer as \
   concise as possible. Always say \
   "thanks for asking!" \
   at the end of the answer.
{context}
Question: {question}
Helpful Answer:"""
```

[11] ✓ 0.0s

Step 5.2: Create QA Chain Prompt from prompt template

```
# Step 5.2: Create QA Chain Prompt from prompt template
QA_CHAIN_PROMPT = PromptTemplate(
    input_variables=["context", "question"],
    template=template,)
```

✓ 0.0s

Step 5.3: Run QA chain from the "QA Chain Prompt" using "Stuff" chain type

```
# Step 5.3: Run QA chain from the "QA Chain Prompt" using "Stuff" chain type
from langchain.chains import RetrievalQA

question = "Is probability a class topic?"
qa_chain = RetrievalQA.from_chain_type(llm,
    retriever=vectordb.as_retriever(),
    return_source_documents=True,
    chain_type_kwargs={"prompt": QA_CHAIN_PROMPT})


result = qa_chain({"query": question})
result["result"]
```

[13] ✓ 2.4s Python

```
/var/folders/r2/t4hb7m2d6lbgjlf7hq_0j_800000gn/T/ipykernel_989/301425952.py:11: LangChainDeprecationWarning: The method `Chair
  result = qa_chain({"query": question})
```

```
'Yes, probability is a class topic that is often covered in courses related to n-grams, Hidden Markov Models, text classifiers
```

Step 6: Conversational Retrieval Chain
Step 6.1: Create Memory

```python
# Step 6: ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(
    memory_key="chat_history",
    # Set return messages equal true
    # - Return the chat history as a  list of messages
    #   as opposed to a single string.
    # - This is  the simplest type of memory.
    #   + For a more in-depth look at memory, go back to
    #     the first class that I taught with Andrew.
    return_messages=True
)
```

[14] ✓ 0.0s Python

```
/var/folders/r2/t4hb7m2d6lbgjlf7hq_0j_800000gn/T/ipykernel_989/3234829016.py:4: LangChainDeprecationWarning: Please see the m
  memory = ConversationBufferMemory(
```

Step 6.2: QA with Conversational Retrieval Chain

```
# Step 6.2: QA with ConversationalRetrievalChain
from langchain.chains import ConversationalRetrievalChain

retriever=vectordb.as_retriever()
qa = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    memory=memory
)
```

✓ 0.0s

Step 6.3: Test Conversational Retrieval Chain
Step 6.3.1: First Question

```
# Step 6.3: Test ConversationalRetrievalChain
# Step 6.3.1: First Question
question = "Is probability a class topic?"
result = qa({"question": question})

result['answer']
```

✓ 2.3s Python

```
'Yes, probability is likely a class topic in the context provided, as it is often a fundamental concept in courses covering t
```

Step 6.3.2: Follow-up Question

```
# Step 6.3.2: Follow-up Question
question = "why are those prerequesites needed?"
result = qa({"question": question})

result['answer']
```

[17] ✓ 3.3s Python

```
'A solid understanding of probability theory, n-grams, Hidden Markov Models, text classifiers, and recurrent neural networks i
```

Step 7: Create a chatbot that works on your documents

```
# Step 7: Create a chatbot that works on your documents

from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter, RecursiveCharacterTextSplitter
from langchain.vectorstores import DocArrayInMemorySearch
from langchain.document_loaders import TextLoader
from langchain.chains import RetrievalQA, ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import TextLoader
from langchain.document_loaders import PyPDFLoader
```

[18] ✓ 0.0s

Step 7.1: Create a chatbot that works on your documents

Step 7.1.1: load_db function

```
# Step 7.1: Create a chatbot that works on your documents - Create Business logic
# Step 7.1.1: load_db function

Tabnine | Edit | Test | Explain | Document | Ask
def load_db(file, chain_type, k):
    # load documents
    loader = PyPDFLoader(file)
    documents = loader.load()
    # split documents
    text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=150)
    docs1 = text_splitter.split_documents(documents)
    # define embedding
    embeddings = OpenAIEmbeddings()
    # create vector database from data
    db = DocArrayInMemorySearch.from_documents(docs1,
            embeddings)
    # define retriever
    retriever = db.as_retriever(search_type="similarity",
            search_kwargs={"k": k})
    # create a chatbot chain. Memory is managed externally.
    qa = ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(model_name=llm_name, temperature=0),
        chain_type=chain_type,
        retriever=retriever,
        return_source_documents=True,
        return_generated_question=True,
    )
    return qa
```

Step 7.1.2: cbfs class

```
import panel as pn
import param

# Step 7.1.2: cbfs class

class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])
    # Step 7.1.2.1: init function
    ############################################################
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "/Users/emilyweng/Documents/SFBU/DS565 GenAI Pr
        self.qa = load_db(self.loaded_file,"stuff", 4)
```

Step 7.1.2.2: call_load_db function

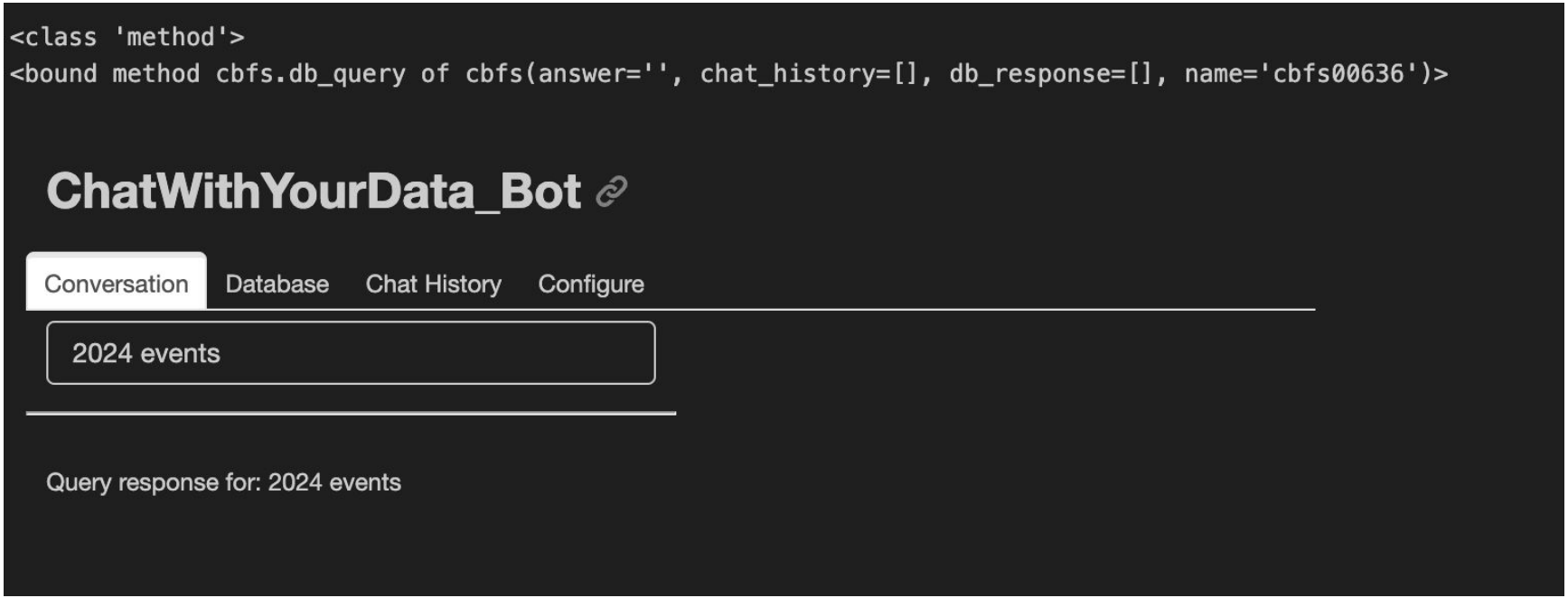
```
############################################################
# Step 7.1.2.2: call_load_db function
############################################################
Tabnine | Edit | Test | Fix | Explain | Document | Ask
def call_load_db(self, count):
    # init or no file specified :
    if count == 0 or file_input.value is None:
        return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
    else:
        file_input.save("temp.pdf")  # local copy
        self.loaded_file = file_input.filename
        button_load.button_style="outline"
        self.qa = load_db("temp.pdf", "stuff", 4)
        button_load.button_style="solid"
    self.clr_history()
    return pn.pane.Markdown(
        f"Loaded File: {self.loaded_file}")
```

Step 7.1.2.3: convchain(self, query) function

```
#########################################################
# Step 7.1.2.3: convchain(self, query) function
#########################################################
def convchain(self, query):
    if not query:
        return pn.WidgetBox(pn.Row('User:',
           pn.pane.Markdown("", width=600)), scroll=True)
    result = self.qa({"question": query,
                      "chat_history": self.chat_history})
    self.chat_history.extend([(query, result["answer"])])
    self.db_query = result["generated_question"]
    self.db_response = result["source_documents"]
    self.answer = result['answer']
    self.panels.extend([
        pn.Row('User:', pn.pane.Markdown(query, width=600)),
        pn.Row('ChatBot:', pn.pane.Markdown(self.answer,
           width=600,
           style={'background-color': '#F6F6F6'}))
    ])
    inp.value = ''  #clears loading indicator when cleared
    return pn.WidgetBox(*self.panels,scroll=True)
```

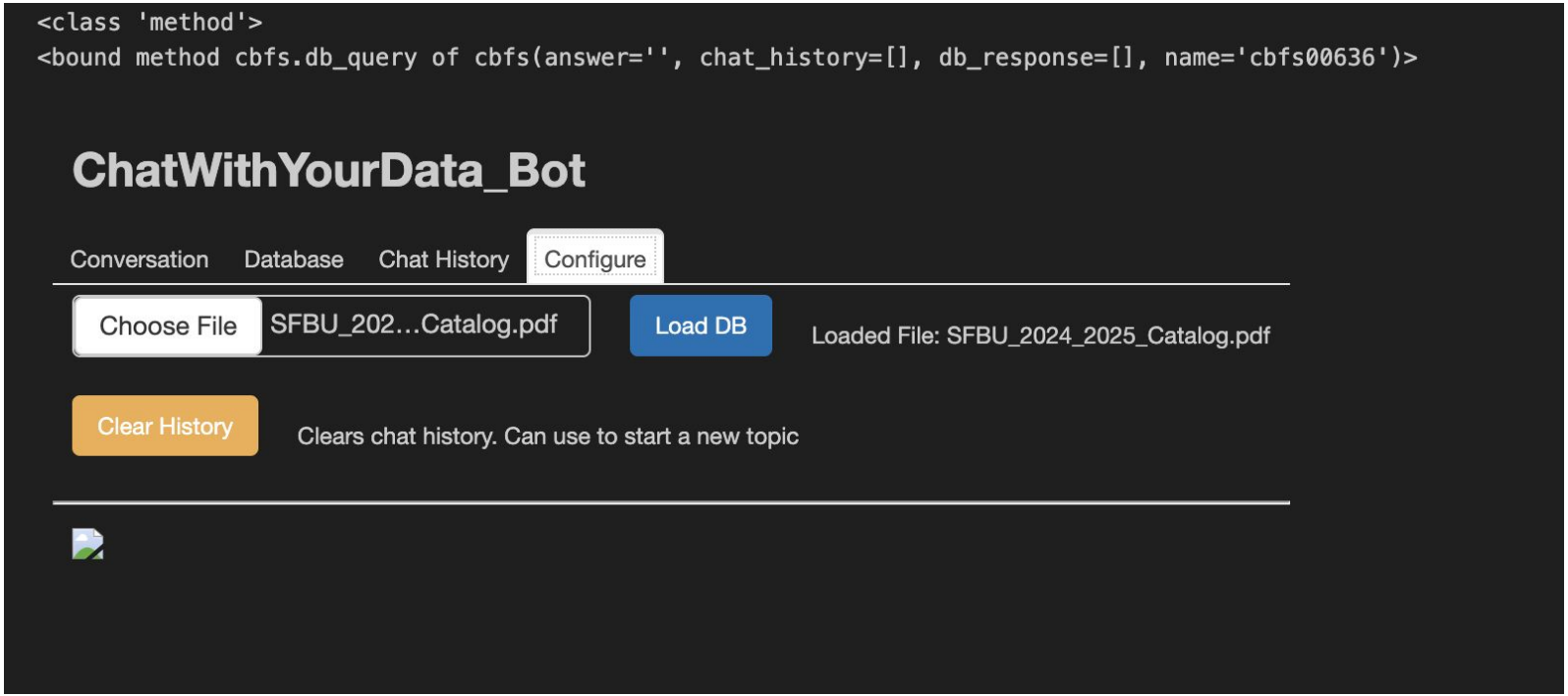
Step 7.2: Create a chatbot GUI

```
<class 'method'>
<bound method cbfs.db_query of cbfs(answer='', chat_history=[], db_response=[], name='cbfs00636')>
```



Step 7.2: Create a chatbot GUI

```
<class 'method'>
<bound method cbfs.db_query of cbfs(answer='', chat_history=[], db_response=[], name='cbfs00636')>
```



Note

Some code were changed in Step 7.2 in order to get the chatbot to work

- Tab1 and tab2 needed to be adjusted
- Step 7.1.24: def convchain(self) function was adjusted as well.
- Step 7.2.3: conversation = pn.bind(cb.db_query, inp)
-

Github

https://github.com/emilywengster/sfbu/tree/d637a270609c945c71030422091fa39da851eaa8/Machine%20Learning/ChatGPT/Customer%20Support%20System/Moderation%2C%20Classification%2C%20Checkout%20and%20Evaluation