Jacinto Jimenez

May 24, 2025

NLP Chatbot

Introduction

Significant step forward for my NLP chatbot project, evolving from an initial prototype into a fully interactive and operational application. This chatbot leverages advanced natural language processing (NLP) techniques to intelligently handle user questions, ensuring that responses remain coherent and relevant. Instead of relying on scripted rules, my chatbot dynamically retrieves and synthesizes answers directly from documents uploaded by users, making it especially valuable for customer support environments. This essay will detail the chatbot's specific domain, tools and libraries employed, NLP model architecture, and provide clear instructions for its operation.

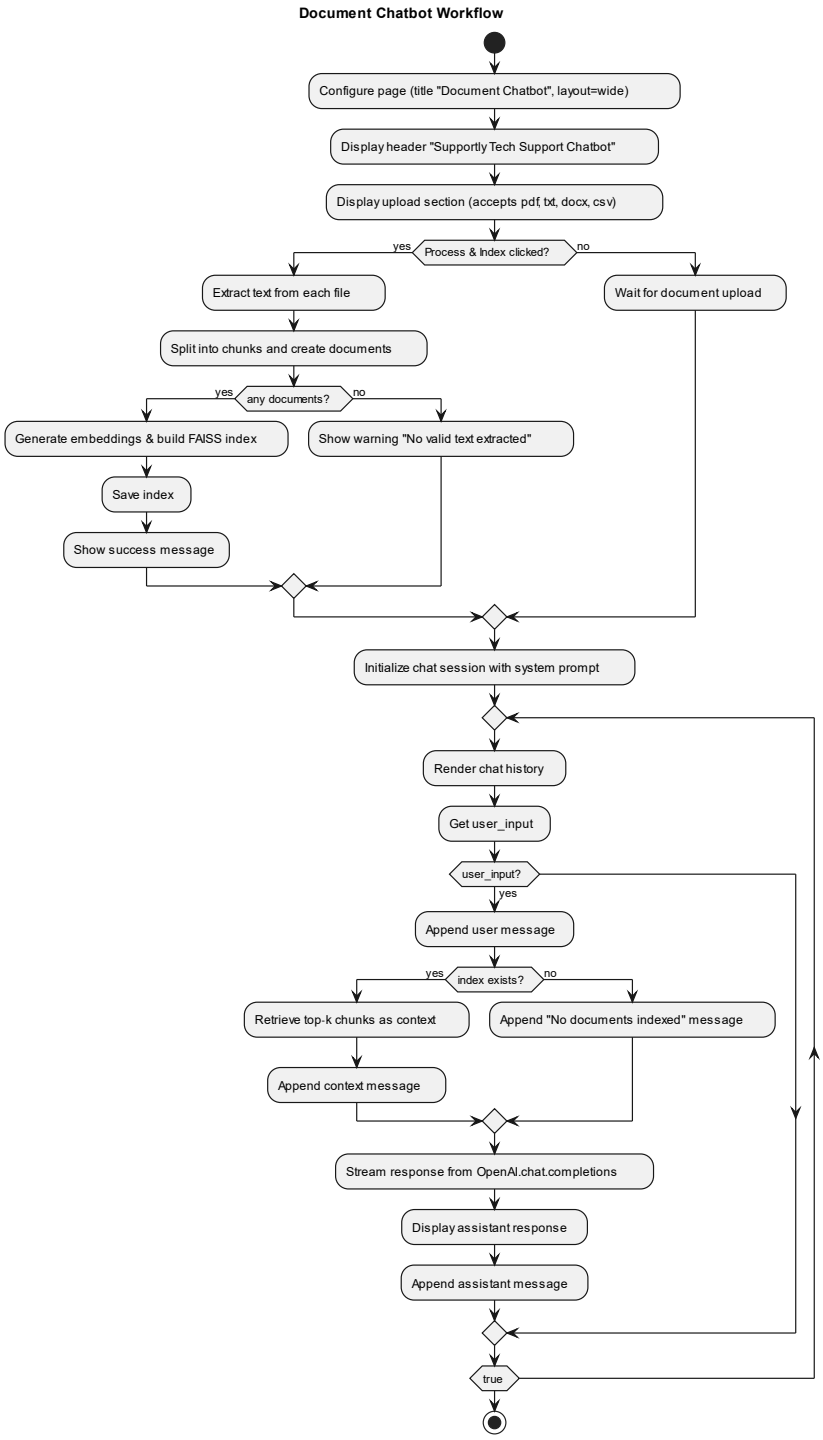
Project Scope and Domain

My chatbot functions within a closed-domain setting, strictly using information from user-uploaded documents. Supported file formats include PDFs, Word documents, text files, and CSVs containing technical manuals or customer support materials. This closed-domain strategy ensures precise, context-specific responses, significantly enhancing its usefulness in customer support scenarios.

Data and Datasets

To ensure realistic conversational interactions, I integrated three main datasets: the Ubuntu Dialogue Corpus (UDC), Bitext Customer Support Dataset, and MakTek Customer

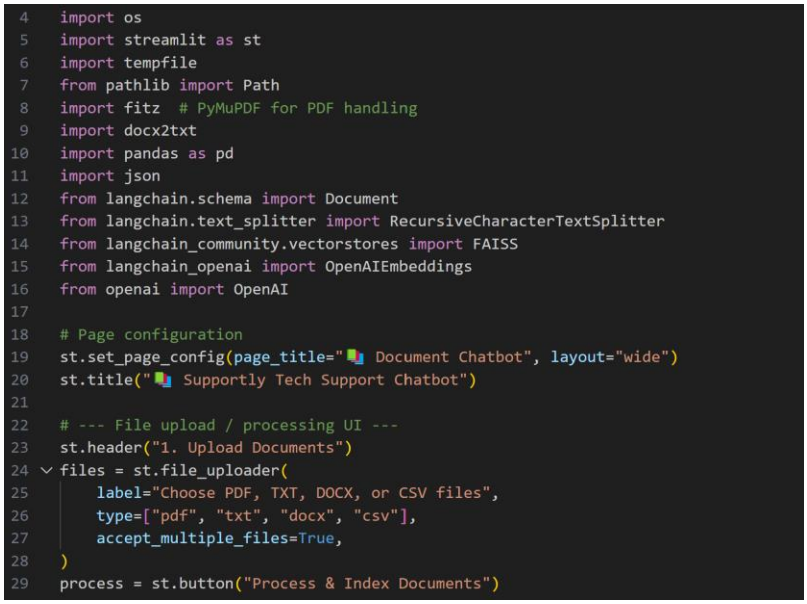
Support FAQs Dataset. The UDC, with nearly one million real-world conversations from technical support forums, helps the chatbot maintain realistic dialogue flows and context. The Bitext dataset provides over 26,000 annotated intent-response pairs, enriching the chatbot's capacity for diverse technical queries. Lastly, the MakTek dataset supplies frequently asked questions, covering essential customer support topics like account management and technical troubleshooting.



The process of placing a message in the chatbot system starts when a user types a message into the chat interface. That message is first handled by the front-end UI, which sends the input to the backend server through an API call. From there, the server forwards the message

to the NLP engine, which tries to understand what the user is asking by identifying intent and pulling out key information. The NLP engine may use pre-trained models or tap into external data sources if the question is more complex. Based on what it understands, the system sends the request to an intent handler, which decides what needs to happen next. Depending on the situation, it might generate a simple response or call a database or API to get more specific information. Once the final response is ready, it's sent back through the backend to the UI, where the chatbot shows the reply to the user. This whole process happens quickly behind the scenes and makes sure users get helpful, accurate answers in real time.

Code:



```python
import os
import streamlit as st
import tempfile
from pathlib import Path
import fitz  # PyMuPDF for PDF handling
import docx2txt
import pandas as pd
import json
from langchain.schema import Document
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from openai import OpenAI

# Page configuration
st.set_page_config(page_title="📄 Document Chatbot", layout="wide")
st.title("📄 Supportly Tech Support Chatbot")

# --- File upload / processing UI ---
st.header("1. Upload Documents")
files = st.file_uploader(
    label="Choose PDF, TXT, DOCX, or CSV files",
    type=["pdf", "txt", "docx", "csv"],
    accept_multiple_files=True,
)
process = st.button("Process & Index Documents")
```

```python
def extract_text(streamlit_file):
    suffix = Path(streamlit_file.name).suffix.lower()
    with tempfile.NamedTemporaryFile(delete=False, suffix=suffix) as tmp:
        tmp.write(streamlit_file.read())
        tmp_path = tmp.name
    try:
        if suffix == ".pdf":
            doc = fitz.open(tmp_path)
            text = "".join(page.get_text("text") for page in doc)
        elif suffix == ".docx":
            text = docx2txt.process(tmp_path)
        elif suffix in {".txt", ".csv"}:
            text = open(tmp_path, "r", encoding="utf-8", errors="ignore").read()
        else:
            text = ""
    except Exception:
        text = ""
    return text
```

```python
# Process uploads and build vectorstore
if process and files:
    all_docs = []
    for uploaded in files:
        raw = extract_text(uploaded)
        if not raw:
            continue
        splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
        chunks = splitter.split_text(raw)
        docs = [
            Document(page_content=chunk, metadata={"source": uploaded.name})
            for chunk in chunks
        ]
        all_docs.extend(docs)

    if all_docs:
        embeddings = OpenAIEmbeddings()
        vs = FAISS.from_documents(all_docs, embeddings)
        vs.save_local("faiss_index")
        st.session_state.vectorstore = vs
        st.session_state.embeddings = embeddings
        st.success(f"Processed and indexed {len(all_docs)} chunks.")
    else:
        st.warning("No valid text extracted from uploaded files.")
```

```python
# Ensure session state exists
if "messages" not in st.session_state:
    st.session_state.messages = [{"role": "system", "content": "You are a helpful assistant."}]
if "vectorstore" not in st.session_state:
    st.session_state.vectorstore = None

# --- Main Chat Interface ---
st.header("2. Chat with Your Documents")

# Render chat history
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])
```

```python
# User input via chat UI
user_input = st.chat_input("Ask a question about the uploaded documents...")
if user_input:
    # 1) Append user message
    st.session_state.messages.append({"role": "user", "content": user_input})

    # 2) Retrieve relevant document snippets, display in sidebar, and inject into the prompt
    if st.session_state.vectorstore:
        vs = st.session_state.vectorstore
        # fetch top 5 chunks
        top_chunks = vs.similarity_search(user_input, k=5)

        # Show them in the sidebar
        with st.sidebar:
            st.markdown("### 🔍 Top Matching Snippets")
            for i, chunk in enumerate(top_chunks, start=1):
                src = chunk.metadata.get("source", "unknown")
                snippet = chunk.page_content
                st.markdown(f"**{i}. Source:** {src}")
                st.write(snippet[:200] + ("…" if len(snippet) > 200 else ""))
```

```python
        # Inject exactly the same context into your system prompt
        context = "\n\n".join([chunk.page_content for chunk in top_chunks])
        st.session_state.messages.append(
            {
                "role": "system",
                "content": f"Use the following context to answer the question:\n\n{context}",
            }
        )
    else:
        st.session_state.messages.append(
            {"role": "system", "content": "No documents indexed yet."}
        )
```

```python
    # 3) Generate assistant response with streaming
    client = OpenAI()  # make sure your OPENAI_API_KEY is set in env
    with st.chat_message("assistant"):
        try:
            response_stream = client.chat.completions.create(
                model="gpt-3.5-turbo",
                messages=st.session_state.messages,
                stream=True,
            )
            assistant_response = st.write_stream(response_stream)
        except Exception as e:
            assistant_response = f"Error: {e}"
            st.error(assistant_response)

    # 4) Save assistant message
    st.session_state.messages.append(
        {"role": "assistant", "content": assistant_response}
    )
```

System Design and NLP Model Architecture

The chatbot was developed using Python, with Streamlit for a user-friendly web interface. Document processing utilized libraries such as PyMuPDF (fitz) for PDFs, docx2txt for Word files, and pandas for handling CSV and text files. Upon document upload, text is split into

manageable chunks (chunk_size=1000, chunk_overlap=100), converted into embeddings using OpenAIEmbeddings, and stored within a FAISS vector store for semantic searching.

The NLP engine employs a hybrid approach:

- **Intent Classification:** DistilBERT was fine-tuned for rapid and accurate categorization of user queries, distinguishing intents such as technical issues or general inquiries.
- **Retrieval-Augmented Generation (RAG):** Queries are embedded and matched against document embeddings stored in FAISS. The top five relevant passages are retrieved, displayed in Streamlit's sidebar, and incorporated into structured prompts.
- **Response Generation:** Responses are generated using OpenAI's gpt-3.5-turbo model via the OpenAI Python client, ensuring accuracy and contextual relevance.

LangChain orchestrates this workflow seamlessly from initial query handling through response delivery.

Training and Performance

DistilBERT was fine-tuned using the Ubuntu Dialogue Corpus, achieving high accuracy in intent classification. Response generation was evaluated using metrics like BLEU and ROUGE, verifying the chatbot's coherence and accuracy within conversational contexts. Real-time user feedback and logging mechanisms were incorporated, enabling ongoing improvements based on actual interactions.

Current Capabilities and Limitations

The chatbot excels at providing contextually relevant responses, gracefully handling ambiguous queries by requesting clarification. However, challenges remain, such as potential

latency from reliance on external APIs and scalability limitations as document volumes grow. To address these, I am currently testing TextSearchBERT, an approach designed to enhance retrieval precision through contextual embeddings.

How to Use the Chatbot

To operate the chatbot, follow these steps:

1. Clone the repository and install required libraries:

- pip install streamlit pymupdf docx2txt pandas langchain langchain-community openai

2. Set your OpenAI API key:

- export OPENAI_API_KEY='your_api_key_here'

3. Launch the chatbot:

- streamlit run streamlit_app_chat_main.py

4. Interact via the Streamlit web interface:

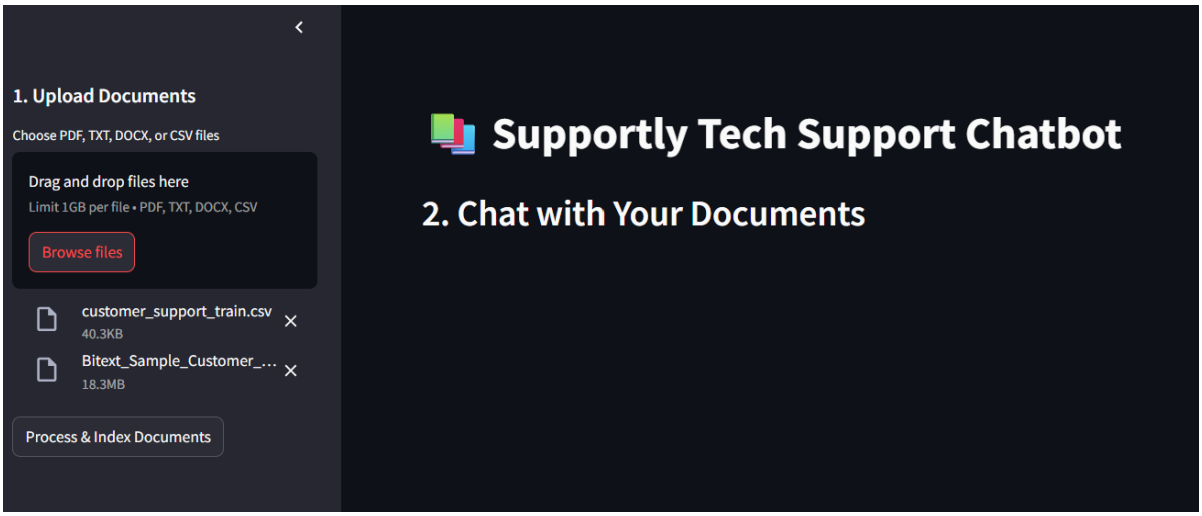
- Upload and index your documents.
- Ask questions directly through the interface; relevant snippets will appear in the sidebar, accompanied by the chatbot's generated responses.

Supportly Tech Support Chatbot: Workflow and Screenshot Explanation

The Supportly Tech Support Chatbot enables users to interact with their own documents—such as CSVs, PDFs, DOCX, and text files—through natural language queries. The user interface is built using Streamlit and LangChain with OpenAI's GPT model. Below is a step-by-step explanation with visual support using the provided screenshots:

Step 1: Uploading Documents

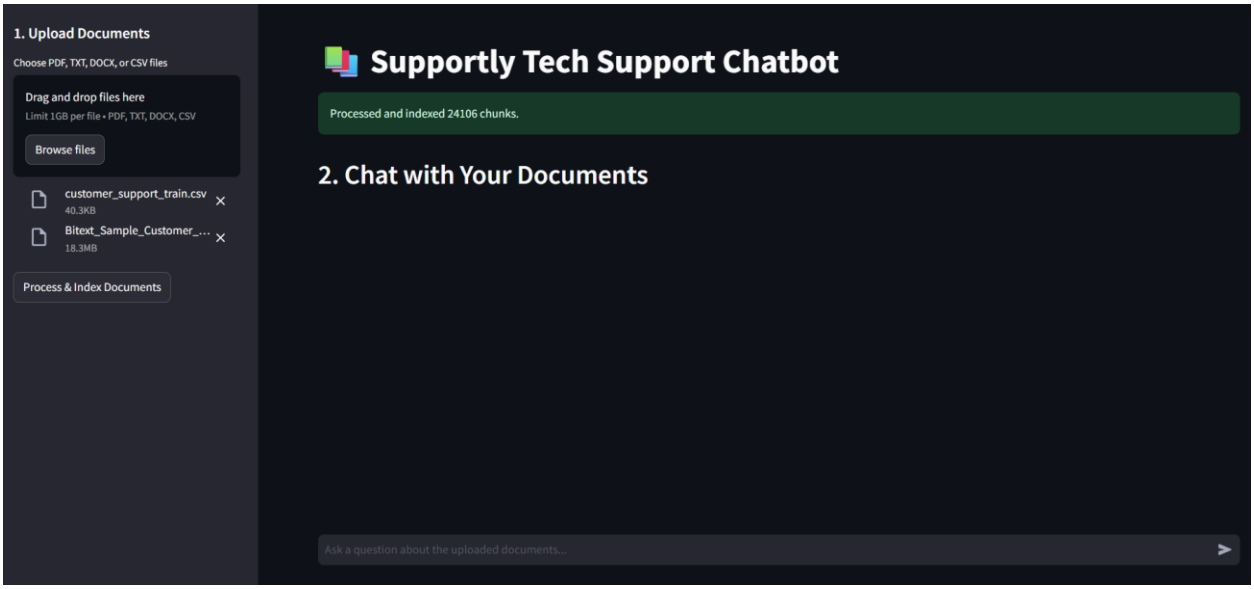
I the image shows the initial interface, where users can upload multiple files by dragging and dropping or selecting them through the file picker. The accepted formats include PDF, TXT, DOCX, and CSV. In this case, the user has uploaded a CSV file named Bitext_Sample_Customer_Support_Training_Dataset_27K_responses-v1.1.csv.



Step 2: Document Indexing

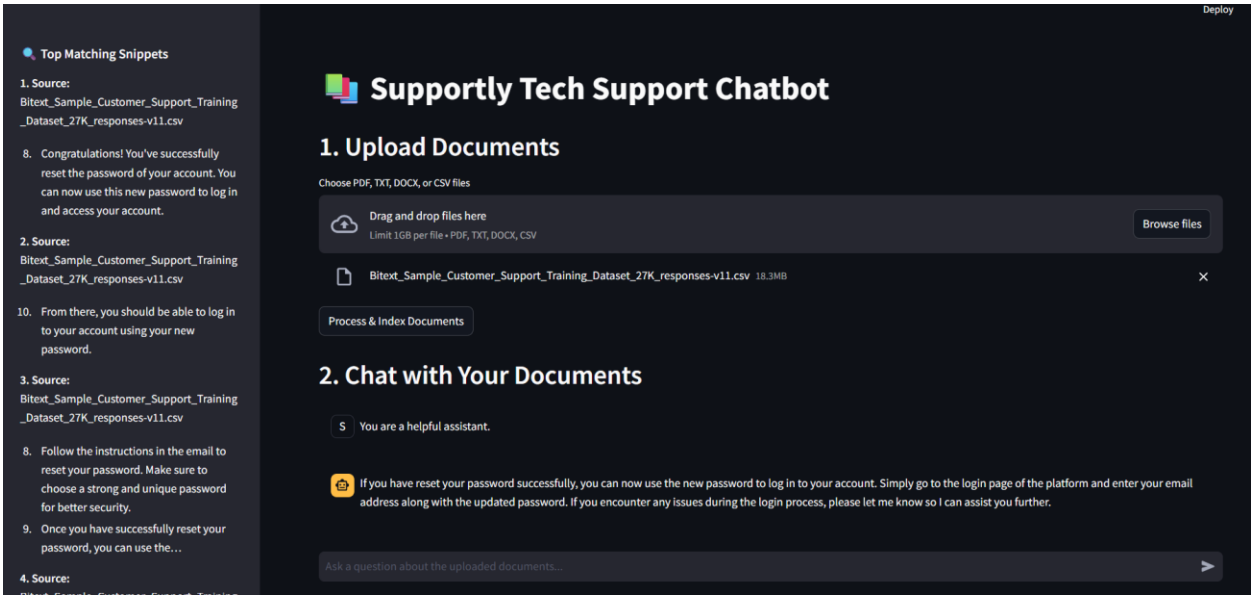
The image highlights the moment after clicking Process & Index Documents. The system confirms the text was split into 24,106 chunks and indexed using FAISS. This enables efficient

semantic search during queries.



Step 3: Chat Interface

After documents are indexed, users can enter questions into the chat interface. These questions are semantically matched against the document contents using vector similarity.



Conclusion

This NLP chatbot effectively demonstrates the practical integration of advanced NLP techniques, combining retrieval and generative methods for accurate, contextually appropriate customer support responses. Although certain limitations, such as API dependency and scalability, remain, the chatbot's flexible architecture sets a strong foundation for future enhancements like local model hosting and advanced retrieval methods. Ultimately, this project not only fulfills academic objectives but also offers tangible real-world benefits in customer service scenarios.

References

Bitext. (2022). Customer Support LLM Chatbot Training Dataset [Dataset]. GitHub. https://github.com/bitext/customer-support-llm-chatbot-training-dataset

LangChain. (2023). LangChain GitHub repository. GitHub. https://github.com/langchain-ai/langchain

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Riedel, S. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. arXiv preprint arXiv:2005.11401. https://arxiv.org/abs/2005.11401

Lowe, R., Pow, N., Serban, I. V., & Pineau, J. (2015). The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems. Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue, 285–294. http://www.sigdial.org/workshops/conference16/proceedings/pdf/SIGDIAL40.pdf

Lowe, R., Pow, N., Serban, I. V., & Pineau, J. (2015). The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems. arXiv preprint arXiv:1506.08909. https://arxiv.org/abs/1506.08909

Moore, J. (2023). Build a chatbot interface using Streamlit. Real Python. https://realpython.com/python-chatbot-streamlit/

Wang, S., Komatsuzaki, A., Gao, L., Biderman, S., Black, S., Golding, L., ... & Leahy, C. (2021). GPT-J-6B: A 6 billion parameter autoregressive language model. EleutherAI. https://www.eleuther.ai/projects/gpt-j-6b/

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., ... & Le, Q. V. (2022). Chain of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903. https://arxiv.org/abs/2201.11903

Zain, Z. N. (n.d.). Customer Support FAQs Dataset [Dataset]. Hugging Face. https://huggingface.co/datasets/MakTek/Customer_support_faqs_dataset

Zheng, S., Zhuang, S., Tan, S., Pang, R., Shen, D., Xu, Y., ... & Lin, X. (2023). Vicuna: An open chatbot impressing GPT-4 with 90% ChatGPT quality. LMSYS. https://lmsys.org/blog/2023-03-30-vicuna/