Adaptive Recommendation Chatbot with RAG and Vector Database

Anmol Valecha

July 2024

1 Introduction

1.1 Overview of the Project and Objectives

The assignment involves developing a domain-specific chatbot application using a Large Language Model (LLM) for natural language understanding and a vector database (Annoy) for data storage and retrieval.

Objectives:

- Implement preprocessing of recipe data for a cooking and kitchen-related chatbot.
- Build a backend using Flask integrating Annoy for similarity search, GPT-2 for text generation, and RAG for enhanced recommendation responses.
- Create a user-friendly frontend with Streamlit for interacting with the recommendation system.

2 Virtual Environment Setup on Mac (M1 Chip)

2.1 Python Environment

1. Install Python 3.9.7 via pyenv:

```
pyenv install 3.9.7
```

2. Create and Activate Virtual Environment (venv):

```
python3 -m venv venv
source venv/bin/activate
```

2.2 Package Installations

Ensure you are in your virtual environment (venv) before running these commands.

2.2.1 Backend Packages (backend.py)

```
pip install flask
pip install sentence-transformers
pip install transformers
pip install annoy
```

2.2.2 Frontend Packages (frontend.py)

```
pip install streamlit
pip install requests
```

2.3 Model Downloads

Download necessary models:

```
python -m transformers.cli.download_pretrained all-MiniLM-L6-v2
```

3 Preprocessing Data

3.1 Data Cleaning and Embedding Generation

Data Cleaning:

- Remove duplicates and handle missing values in the ”Healthy Indian Recipes” dataset.
- Convert columns to appropriate data types (int, float) for analysis.

Embedding Generation:

- Utilize Sentence Transformers to generate embeddings from text data (Dish Name, Ingredients, Instructions).

3.1.1 Code Files: preprocess_recipes.py

```
import os
import pandas as pd
from sentence_transformers import SentenceTransformer
from annoy import AnnoyIndex
import sys
import time  # Import time module for sleep
```

```
def preprocess_data():
    # Load the dataset
    file_path = '/Users/anmolvalecha/Cloud Backups/prompengg/Assignmt5/venv/Indi
    recipes_df = pd.read_csv(file_path)

    # Display the first few rows of the dataset
    print("Initial Data Preview:")
    print(recipes_df.head())

    # Drop duplicates
    recipes_df.drop_duplicates(inplace=True)

    # Handle missing values
    recipes_df.fillna('', inplace=True)

    # Convert columns to appropriate data types and handle specific transformati
    recipes_df['Prep Time'] = recipes_df['Prep Time'].apply(lambda x: int(x.repl
    recipes_df['Cook Time'] = recipes_df['Cook Time'].apply(lambda x: int(x.repl
    recipes_df['Rating'] = pd.to_numeric(recipes_df['Rating'], errors='coerce').
    recipes_df['Number of Votes'] = pd.to_numeric(recipes_df['Number of Votes'],
    recipes_df['Serves'] = pd.to_numeric(recipes_df['Serves'], errors='coerce').
    recipes_df['Views'] = pd.to_numeric(recipes_df['Views'], errors='coerce').fi

    # Display the preprocessed data
    print("Preprocessed Data Preview:")
    print(recipes_df.head())

    # Initialize the sentence transformer model for generating embeddings
    model = SentenceTransformer('all-MiniLM-L6-v2')

    # Combine relevant text fields for embedding generation
    recipes_df['text'] = recipes_df['Dish Name'] + ' ' + recipes_df['Ingredients

    # Generate embeddings
    embeddings = model.encode(recipes_df['text'].tolist(), show_progress_bar=Tru

    # Initialize Annoy index
    dimension = 384  # Adjusted dimension to match SentenceTransformer model out
    annoy_index = AnnoyIndex(dimension, 'euclidean')

    # Add items to the Annoy index
    for i, embedding in enumerate(embeddings):
        annoy_index.add_item(i, embedding)

    # Build the Annoy index
    annoy_index.build(10)  # 10 trees
    # Save the Annoy index to a file
    annoy_index.save('recipes_index.ann')

    # Optionally, you can also save the preprocessed dataframe to use later
    recipes_df.to_csv('preprocessed_recipes.csv', index=False)

    print("Annoy index saved to 'recipes_index.ann'.")
    print("Preprocessed data saved to 'preprocessed_recipes.csv'.")

if __name__ == "__main__":
    preprocess_data()
```

4 Backend Implementation (backend.py)

4.1 Flask Backend Overview

- **Integration with Annoy:** Load and query Annoy index for similarity search based on user input vectors.
- **GPT-2 Text Generation:** Use GPT-2 via Hugging Face Transformers pipeline for generating detailed descriptions of recommended dishes.
- **Retrieval-Augmented Generation (RAG):** Implement RAG to enhance recommendation responses by leveraging pre-existing data stored in the system.

4.1.1 Code Files: backend.py

```
from flask import Flask, jsonify, request
from annoy import AnnoyIndex
import pandas as pd
from sentence_transformers import SentenceTransformer
from transformers import pipeline

app = Flask(__name__)

# Load the Annoy index and preprocessed data
annoy_index = AnnoyIndex(384, 'euclidean')
annoy_index.load('recipes_index.ann')

recipes_df = pd.read_csv('preprocessed_recipes.csv')

# Initialize the sentence transformer model for generating embeddings
model = SentenceTransformer('all-MiniLM-L6-v2')
```

```
# Initialize the language model for text generation (GPT-2)
generator = pipeline("text-generation", model="gpt2")

# Keywords to check if the query is food-related
food_related_keywords = ["recipe", "dish", "food", "ingredient", "cooking", "mea

# Accepted cuisines that the chatbot can recommend
accepted_cuisines = ["indian"]

@app.route('/interactive_recommendation', methods=['POST'])
def interactive_recommendation():
    # Get JSON request data
    data = request.get_json()
    user_message = data['message']

    # Simple keyword check to determine if the query is related to food recipes
    if not any(keyword in user_message.lower() for keyword in food_related_keywo
        response = {
            'message': "I␣am␣a␣recipe␣recommendation␣system␣and␣this␣question␣is
        }
        return jsonify(response)

    # Check if the user is asking for cuisines other than Indian
    if any(cuisine in user_message.lower() for cuisine in accepted_cuisines):
        # Encode the user message to get the query vector
        query_vector = model.encode([user_message])

        # Number of nearest neighbors to retrieve
        top_k = 20  # Increase this to get a broader set and filter later

        # Query the Annoy index
        result_indices, distances = annoy_index.get_nns_by_vector(query_vector[0

        # Prepare response with recommended dishes and generated descriptions
        recommended_dishes = []
        for neighbor_index, distance in zip(result_indices, distances):
            dish = recipes_df.iloc[neighbor_index]
            dish_name = dish['Dish␣Name']
            description = dish.get('Description', 'No␣description␣available')
            ingredients = dish.get('Ingredients', 'No␣ingredients␣listed')
            instructions = dish.get('Instructions', 'No␣instructions␣provided')
            spice = dish.get('Spice', 'No␣spice␣information␣available')
            rating = dish.get('Rating', 'No␣rating␣available')
            dietary_info = dish.get('Dietary␣Info', 'No␣dietary␣information␣avai
        # Combine dish information to generate a more detailed description
        prompt = (f"Dish Name: {dish_name}\n"
                  f"Description: {description}\n"
                  f"Ingredients: {ingredients}\n"
                  f"Instructions: {instructions}\n"
                  f"Spice Level: {spice}\n"
                  f"Rating: {rating}\n"
                  f"Dietary Info: {dietary_info}\n\n"
                  "Detailed overview of the dish including its ingredients, 

        # Generate description with increased max tokens
        generated_description = generator(prompt, max_new_tokens=300, num_re

        recommended_dishes.append({
            'dish_name': dish_name,
            'distance': distance,
            'generated_description': generated_description
        })

    # Filter and rank the dishes based on their relevance to the user query
    relevant_dishes = [
        dish for dish in recommended_dishes
        if any(keyword in (dish['dish_name'] + " " + dish['generated_descrip
    ]

    # Sort relevant dishes by distance
    relevant_dishes = sorted(relevant_dishes, key=lambda x: x['distance'])

    # Assign ranks to the dishes
    for rank, dish in enumerate(relevant_dishes, start=1):
        dish['rank'] = rank

    # Limit to top 3 dishes
    relevant_dishes = relevant_dishes[:3]

    # If no relevant dishes found, respond accordingly
    if not relevant_dishes:
        response = {
            'message': f"Sorry, I couldn't find any recipes matching '{user_
        }
    else:
        # Simulate a chat-like response
        response = {
            'message': f"Here are the top {len(relevant_dishes)} recommended
            'recommended_dishes': relevant_dishes
        }
    else:
        # Respond that the chatbot can only recommend Indian dishes
        response = {
            'message': "I␣recommend␣only␣Indian␣dishes.␣Please␣ask␣me␣about␣Indi
        }

    return jsonify(response)

if __name__ == '__main__':
    app.run(debug=True)
```

5 Frontend Implementation (`frontend.py`)

5.1 Streamlit Frontend Overview

- **User Interaction:** Display a text input for users to query recipes.
- **Display Recommendations:** Show recommended dishes ranked by relevance and distance. Provide generated descriptions for each dish.

5.1.1 Code Files: `frontend.py`

```
import streamlit as st
import requests

def fetch_recommendations(query_message):
    url = 'http://127.0.0.1:5000/interactive_recommendation'
    response = requests.post(url, json={'message': query_message})
    if response.status_code == 200:
        return response.json()
    else:
        return None

def main():
    st.title('Recipe␣Recommendation␣System')

    user_input = st.text_input('You:')

    if user_input:
        recommendations = fetch_recommendations(user_input)
        if recommendations:
            if 'recommended_dishes' in recommendations:
                st.write(recommendations['message'])
                for rec in recommendations['recommended_dishes']:
                    st.write(f"{rec['rank']}.␣{rec['dish_name']}␣(Distance:␣{rec
                    st.write(f"    Generated Description: {rec['generated_descri
                else:
                    st.write(recommendations['message'])
        else:
            st.write('Error fetching recommendations.')

if __name__ == '__main__':
    main()
```

6 Examples of User Prompts

6.1 Bot Interface

- **Example 1:** Asking for specific recommendations.
  - **User Query:** "spicy indian food"
- **Example 2:** Inquiring about ingredient-based recipes.
  - **User Query:** "indian potato recipe"
- **Example 3:** Asking for specific cuisine recommendations which is out of scope of dataset.
  - **User Query:** "italian dishes"
- **Example 4:** Testing the system's response to non-food-related queries
  - **User Query:** "why is sky blue"
- **Example 5:** Inquiring about specific recipes
  - **User Query:** "punjabi pyaaz recipe"
- **Example 6:** Inquiring about specific ingredient in recipes
  - **User Query:** "indian chicken recipe"

7 Conclusion

7.1 Summary

In conclusion, the Adaptive Recommendation Chatbot successfully integrates RAG for enhanced recommendation responses using a vector database and demonstrates effective interaction through a Streamlit frontend. The backend, powered by Flask, handles user queries by leveraging Annoy for similarity search and GPT-2 for text generation, fulfilling the objectives of providing personalized recipe recommendations in the cooking domain.

8 References

- GitHub Repository: `https://github.com/your_username/your_chatbot_repo`