Jessup Cellars Chatbot Development Report

Introduction

The Jessup Cellars chatbot is designed to provide an engaging and informative interaction platform for visitors to the Jessup Cellars website. The chatbot utilizes state-of-the-art Natural Language Processing (NLP) techniques to answer questions from a predefined corpus of information and direct users to contact the business for out-of-corpus queries. This report details the development process, architecture, and methodologies used in creating this chatbot.

Objectives

- Provide instant answers to user queries based on a specific corpus.
- Maintain conversation context for a seamless user experience.
- Redirect users to contact the business for questions beyond the provided corpus.
- Ensure minimal latency for real-time interaction.
- Implement a minimalistic UI for user interaction.

Development Environment

- **Programming Language**: Python, JavaScript
- **Frameworks**: Flask (for backend), HTML/CSS (for frontend)
- **Libraries**:
    - Transformers (for Question Answering)
    - SentenceTransformers (for semantic similarity)
    - sklearn (for cosine similarity)
- **Tools**: Flask-CORS, Fetch API
- **IDE used:** Pycharm

Architecture and Components

The chatbot consists of the following main components:

1. **Frontend**: An HTML/CSS interface for user interaction, supplemented by JavaScript for handling user input and displaying responses.
2. **Backend**: A Flask application that handles user queries, processes them, and returns appropriate responses.
3. **NLP Models**:
    - **Question Answering Model**: A fine-tuned DistilBERT model for answering questions based on the provided context.
    - **Sentence Embedding Model**: A SentenceTransformer model for generating embeddings of the corpus and user queries.

Development Process

Step 1: Setting Up the Flask Application

For setting the flask application, here first all the libraries are loaded.

```python
from flask import Flask, request, jsonify, render_template
from flask_cors import CORS
from transformers import AutoTokenizer, AutoModelForQuestionAnswering, pipeline
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

app = Flask(__name__)
CORS(app)
```

Step 2: Loading the Corpus

The corpus is loaded from a text file, with each paragraph representing a separate piece of information.

```python
@app.route('/')
def home():
    return render_template('index2.html')

# Load the corpus
with open('Corpus.txt', 'r', encoding='utf-8') as f:
    corpus = f.read().split('\n\n')  # Assuming paragraphs are separated by blank lines

print("Corpus loaded. Number of paragraphs:", len(corpus))
```

Step 3: Loading the Models

Two models are loaded: one for question answering and one for generating sentence embeddings.

```python
qa_model_name = "distilbert-base-cased-distilled-squad"
qa_tokenizer = AutoTokenizer.from_pretrained(qa_model_name)
qa_model = AutoModelForQuestionAnswering.from_pretrained(qa_model_name)
qa_pipeline = pipeline( task: "question-answering", model=qa_model, tokenizer=qa_tokenizer)

embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
corpus_embeddings = embedding_model.encode(corpus)

print("Models loaded and corpus embeddings generated.")
```

Step 4: Handling User Queries

A route is created to handle user queries, process them, and return appropriate responses.

```python
def get_relevant_context(query, top_k=3):
    query_embedding = embedding_model.encode([query])
    similarities = cosine_similarity(query_embedding, corpus_embeddings)[0]
    top_indices = np.argsort(similarities)[-top_k:][::-1]
    return ' '.join([corpus[i] for i in top_indices])

new *
@app.route( rule: '/query', methods=['POST'])
def chat():
    data = request.json
    query = data['message']
    history = data.get('history', [])

    print("Received query:", query)

    context = get_relevant_context(query)

    # Check if query is about something in the corpus
    similarity_score = cosine_similarity(embedding_model.encode([query]), corpus_embeddings).max()
    print("Similarity score:", similarity_score)

    if similarity_score < 0.3:
        return jsonify({
            "response": "I'm sorry, I don't have information about that. Please contact the business directly for more details."
        })

    # Use QA model to generate response
    qa_input = {
        'question': query,
        'context': context
    }
    response = qa_pipeline(qa_input)
    print("QA response:", response)

    return jsonify({"response": response['answer']})

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

· **Define Function** get_relevant_context:

- **Parameters**:
  - query: The query text.
  - top_k: The number of top similar contexts to return (default is 3).
- **Process**:
  - Encodes the query using an embedding model.
  - Computes cosine similarities between the query embedding and the corpus embeddings.
  - Selects the top k similar contexts from the corpus.
  - Joins and returns these top contexts as a single string.

· **Flask Route** /query **(POST Method)**:

- **Function**: chat()
- **Process**:
  - Receives JSON data from the POST request.
  - Extracts the message (query) and history (optional).
  - Logs the received query.
  - Calls get_relevant_context to find relevant contexts for the query.

· **Check Query Similarity**:

- Computes the cosine similarity between the query embedding and the corpus embeddings.
- Logs the similarity score.
- If the highest similarity score is less than 0.3, it returns a response indicating insufficient information.

· **Generate QA Model Response**:

- Prepares input for the QA model with the query and retrieved context.
- Calls qa_pipeline to get the response.
- Logs the QA model's response.
- Returns the response in JSON format.

· **Run Flask App**:

- Starts the Flask application in debug mode.

Step 5: Creating the Frontend

The frontend is designed using HTML and CSS, with JavaScript handling user interactions and AJAX requests.Step 6: Debugging and Testing

To ensure the chatbot functions correctly, thorough testing and debugging are conducted. This includes:

- **Flask Console Logs**: Monitoring the Flask console for incoming requests and responses.
- **Browser Developer Tools**: Using the network tab to inspect the requests and responses.
- **Error Handling**: Adding comprehensive error handling in both Flask and JavaScript.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Jessup Cellars Chatbot</title>
   <style>
      body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 0;
      background-color: #f0f0f0;
   }

   .chat-container {
      max-width: 600px;
      margin: 20px auto;
      background-color: #fff;
      border-radius: 8px;
      box-shadow: 0 0 10px rgba(0,0,0,0.1);
      overflow: hidden;
   }

   .chat-header {
      background-color: #4a0e0e;
      color: #fff;
      padding: 15px;
      text-align: center;
   }
```

```
    .chat-header h1 {
        margin: 0;
        font-size: 20px;
    }

    .chat-messages {
        height: 400px;
        overflow-y: auto;
        padding: 15px;
    }

    .message {
        margin-bottom: 15px;
        padding: 10px;
        border-radius: 5px;
        max-width: 80%;
    }

    .user-message {
        background-color: #e1f5fe;
        margin-left: auto;
    }

    .bot-message {
        background-color: #f5f5f5;
    }

    .chat-input {
        display: flex;
        padding: 15px;
    }

    .chat-input input {
        flex-grow: 1;
        padding: 10px;
        border: 1px solid #ddd;
        border-radius: 4px;
        margin-right: 10px;
    }

    .chat-input button {
        padding: 10px 20px;
        background-color: #4a0e0e;
        color: #fff;
        border: none;
        border-radius: 4px;
        cursor: pointer;
    }
    </style>
</head>
<body>
    <div class="chat-container">
        <div class="chat-header">
            <h1>Jessup Cellars Chatbot</h1>
        </div>
```

```
    <div class="chat-messages" id="chatMessages"></div>
    <br>
    <div class="chat-input">
      <input type="text" id="userInput" placeholder="Type your message here..." onkeypress="handleKeyPress(event)">
      <button onclick="sendMessage()">Send</button>
    </div>
  </div>
  <script>
    let chatHistory = [];
    function addMessage(message, isUser) {
      const chatMessages = document.getElementById('chatMessages');
      const messageElement = document.createElement('div');
      messageElement.classList.add('message');
      messageElement.classList.add(isUser ? 'user-message' : 'bot-message');
      messageElement.textContent = message;
      chatMessages.appendChild(messageElement);
      chatMessages.scrollTop = chatMessages.scrollHeight;
    }

    async function sendMessage() {
      const userInput = document.getElementById('userInput');
      const message = userInput.value.trim();

      if (message) {
        addMessage(message, true);
        chatHistory.push(message);
        userInput.value = '';

        try {
          console.log("Sending message:", message);
          const response = await fetch('/query', {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json',
            },
            body: JSON.stringify({ message: message, history: chatHistory }),
          });

          console.log("Response received:", response);

          if (!response.ok) {
            throw new Error('Network response was not ok');
          }

          const data = await response.json();
          console.log("Data received:", data);
          addMessage(data.response, false);
          chatHistory.push(data.response);

        } catch (error) {
          console.error('Error:', error);
          addMessage('Sorry, there was an error processing your request.', false);
        }
      }
```

```
        }

        function handleKeyPress(event) {
            if (event.key === 'Enter') {
                sendMessage();
            }
        }
        document.getElementById('userInput').addEventListener('keypress', handleKeyPress);
    </script>
</body>
</html>
```

Sure, here's a point-wise explanation without the code snippets:

HTML Structure

1. Document Type and Language:

- Specifies the document type and sets the language to English.

2. Head Section:

- Sets character encoding to UTF-8.

- Configures the viewport for responsive design.

- Sets the webpage title to "Jessup Cellars Chatbot".

- Includes inline CSS for styling the chatbot.

3. Body Section:

- Contains the main structure of the chatbot:

  - A container for the chat interface.

  - A header displaying the chatbot title.

  - A section for displaying chat messages.

  - An input area for typing messages and a button to send messages.

- Includes inline JavaScript for handling chat interactions.

CSS Styles

1. Body Styling:

- Sets the font to Arial, removes default margins and padding, and applies a light grey background color.

2. Chat Container Styling:

- Defines a container with a maximum width, centered margin, white background, rounded corners, and shadow.

3. Chat Header Styling:

- Styles the header with a dark background, white text, centered content, and padding.

4. Chat Messages Section Styling:

- Sets the height, enables vertical scrolling, and adds padding.

5. Message Styling:

- Defines general message styling, with different background colors for user and bot messages.

6. Chat Input Section Styling:

- Styles the input area and send button with padding, border, and background color.

JavaScript Functionality

1. Initialize Chat History:

- Initializes an empty array to keep track of chat history.

2. Function to Add Message to Chat:

- Adds a new message to the chat display, differentiating between user and bot messages, and ensures the chat scrolls to the bottom.

3. Function to Send Message:

- Sends the user's message to the server, handles the response, and updates the chat history. It also includes error handling.

4. Function to Handle Enter Key Press:

- Sends the message when the Enter key is pressed by the user.

This setup provides a simple chat interface where users can interact with a chatbot, and messages are dynamically added and sent to the server.

Conclusion

The Jessup Cellars chatbot provides an efficient and user-friendly interface for users to interact with the business. By leveraging advanced NLP techniques and ensuring a robust architecture, the chatbot delivers relevant and timely responses to user queries. Future enhancements could include expanding the corpus, integrating with other business systems, and adding more advanced conversational capabilities.