CHAPTER- 1

1. INTRODUCTION

1.1 Background and Motivation

History is a treasure trove of knowledge that provides insights into our past, helping us understand the evolution of societies, cultures, and civilizations. However, traditional methods of studying history, such as textbooks and lectures, often fail to engage learners, making the subject appear dry and less interactive.

With the advent of conversational AI and natural language processing (NLP), innovative approaches to learning, such as chatbots, have emerged as powerful tools for delivering information in an engaging and accessible manner.

The integration of conversational AI in education opens new doors to interactive learning experiences. Chatbots have the potential to simplify complex subjects, encourage curiosity, and provide instant access to information. By merging the realms of AI and history, this project aims to create an interactive platform where users can explore historical facts and trivia in an engaging manner. Targeting history enthusiasts, educators, and students, the project strives to foster a deeper connection with historical knowledge through technology

1.2 Problem Statement

The vastness of historical knowledge, spanning across centuries, civilizations, and cultures, poses a significant challenge for individuals seeking to explore and understand it comprehensively. Traditional methods of learning history, such as textbooks and lectures, often fail to engage learners effectively, leading to a lack of interest and limited retention of information. Furthermore, historical information is scattered across numerous resources, making it cumbersome for users to retrieve specific details quickly and efficiently. This gap highlights the need for a solution that consolidates historical knowledge and presents it in an engaging, accessible format.

Modern learners prefer interactive and on-demand access to information. However, existing digital tools like encyclopedias and search engines require users to navigate through multiple pages or refine their search queries repeatedly to obtain desired answers. These platforms often lack interactivity and personalized engagement, which are critical for enhancing understanding and retention. Additionally, while some educational applications provide static historical facts or quizzes, they lack the capability to simulate dynamic, conversational learning experiences. This limitation further underscores the need for a more sophisticated educational tool.

To address these challenges, the development of an AI-powered Historical Facts and Trivia Chatbot emerges as a promising solution. The chatbot leverages conversational AI to provide immediate responses to user queries, share random historical facts, and conduct interactive trivia quizzes. By integrating natural language processing, a comprehensive historical database, and an intuitive user interface, the chatbot bridges the gap between static resources and engaging learning experiences. This project aims to make historical knowledge more accessible, interactive, and enjoyable for history enthusiasts, students, and educators alike.

1.3 Objectives

- **Enhance Accessibility**:

  Offer instant access to a vast repository of historical knowledge, making it easier for users to explore topics of interest.

- **Promote Engagement**:

  Incorporate interactive quizzes and dynamic fact-sharing features to create a fun and engaging learning experience.

- **Enable Intelligent Query Handling**:

  Utilize advanced transformer-based models to understand and respond to user queries in a conversational style.

- **Foster Knowledge Retention**:

  Use trivia quizzes and follow-up questions to reinforce historical knowledge and improve user memory retention.

- **Support Multi-Format Content Delivery**:

  Present historical information in diverse formats, including random facts, detailed answers, and quiz-based interactions.

- **Ensure Scalability**:

  Design the chatbot architecture to allow the seamless addition of new historical topics and facts in the future.

- **Provide an Intuitive User Interface**:

  Use Flask to create a simple, responsive front-end that ensures an easy and enjoyable user experience.

- **Facilitate Efficient Database Management**:

  Use MySQL to store and manage historical data, facts, and quizzes, ensuring fast retrieval and efficient updates.

- **Target Broad User Groups**:

  Cater to history enthusiasts, students, and educators by offering a versatile tool for learning and teaching history.

1.4 Scope and Limitations

Scope

- **Educational Enhancement**:

  The chatbot is designed as a tool for history enthusiasts, educators, and students to make learning history more engaging and interactive.
- **Dynamic Data**

  Access: It retrieves historical data, including facts and trivia, from a MySQL database to provide accurate and concise information.
- **Interactive Features**:

  The chatbot includes features like random fact generation, query handling, and trivia quizzes to cater to diverse user interests.
- **Future Expansion**:

  The project can be extended to cover modern history, integrate multimedia content, and connect with external APIs for a broader knowledge base.

Limitations

- **Limited Coverage**:

  Initially, the chatbot focuses on ancient history, with potential expansion to modern history subject to resource availability.
- **Simple NLP**:

  While the chatbot uses natural language processing to interpret user queries, its understanding is limited to predefined datasets and may not handle highly complex or ambiguous queries.
- **Dependency on Data**:

  The chatbot's effectiveness relies on the quality and comprehensiveness of the stored historical data. Incomplete or outdated data could impact its performance.
- **No Live Updates**:

  The current version does not include real-time updates or integrations with external databases, which limits its ability to provide the latest historical discoveries.
- **Internet Dependency**:

  While most functionalities are offline, users will require an internet connection to access chatbot.

CHAPTER-2

2.Literature Review

2.1 Review of Relevant Research and Projects

2.1.1 Evolution of Conversational AI in Education

- Early conversational agents, such as ELIZA (1966), laid the foundation for natural language interactions. Over time, advancements in AI and machine learning enabled the development of more intelligent systems like IBM’s Watson and Google Assistant.
- Educational chatbots have evolved from simple Q&A systems to interactive learning tools that provide context-aware responses using advanced NLP models.

2.1.2 Historical Education Tools

- Traditional history education relied on textbooks and lectures, but digital tools like Wikipedia and online encyclopedias transformed accessibility.
- The rise of e-learning platforms, such as Coursera and Khan Academy, introduced multimedia and interactive formats for history education.
- Studies show that gamified applications like Duolingo (for languages) inspire similar approaches for other subjects, including history.

2.1.3 Advancements in Natural Language Processing (NLP)

- NLP research evolved from rule-based models in the 1990s to statistical models in the 2000s, and finally, transformer-based architectures in the 2010s.
- Tools like OpenAI’s GPT-3 and Hugging Face Transformers revolutionized chatbots by providing deep contextual understanding, enabling history chatbots to handle diverse queries effectively.

2.1.4 Gamification in Historical Learning

- Gamification in education, especially quizzes and trivia games, has been extensively researched. Tools like Kahoot and Quizizz demonstrate increased user engagement through game-like features.
- Research shows that trivia-based learning improves retention by 25–35%, making it a suitable method for history education.

2.1.5 Structured Historical Knowledge Representation

- The use of structured databases for historical information, such as DBpedia and Wikidata, has allowed for efficient retrieval and linkage of historical facts.
- AI models trained on such databases can generate coherent, factual responses, making them integral to history chatbots.

2.1.6 Integration of AI in Personalized Learning

- Research on adaptive learning systems shows that AI can personalize content delivery based on the learner's preferences and performance.
- Historical chatbots can leverage this research to provide tailored content, quizzes, and fact-sharing for diverse user groups

2.1.7 Chatbots for Historical Knowledge Sharing

- Recent projects, such as ChatGPT and similar AI-driven chatbots, have highlighted the feasibility of using conversational agents for history and education.
- These systems integrate user-friendly interfaces with advanced NLP, enabling real-time query handling and engaging user interactions.

2.1.8 Modern Tools for Historical Education

- Current platforms like Google Arts & Culture and historical data visualization tools have redefined how users explore history.
- These tools integrate multimedia and AI for an immersive experience, setting benchmarks for interactive learning environments.

2.1.9 Limitations in Existing Solutions

- Many history-focused chatbots and platforms lack conversational depth, dynamic response generation, and personalized interaction.
- The scope is often limited to static facts or predefined question sets, missing the opportunity for real-time, contextual dialogue

2.2 Theoretical Framework

The theoretical framework for the historical facts and trivia chatbot is built upon several concepts related to Artificial Intelligence (AI), Natural Language Processing (NLP), database systems, and human-computer interaction. This framework ensures that the chatbot can efficiently process user input and provide meaningful and relevant responses based on historical data.

2.2.1 Natural Language Processing(NLP):

NLP is the branch of AI that focuses on the interaction between computers and human languages. In the context of the chatbot, NLP is used to understand and process user queries. The chatbot's ability to comprehend various forms of user input (e.g., questions, statements) and return accurate responses relies on key NLP concepts, such as:

- Tokenization: Breaking down the user input into meaningful units (tokens) for analysis.
- Named Entity Recognition (NER): Identifying key historical terms such as the names of civilizations, historical figures, and events.
- Intent Recognition: Understanding the intent behind a user's query, whether they are asking a factual question or requesting a quiz.

2.2.2 Information Retrieval (IR):

The chatbot uses an information retrieval system to fetch relevant historical facts from the database based on user input. The system leverages keyword matching, semantic search, and querying techniques to find the best answers to users' questions.

- Query Parsing: Understanding the user's input and converting it into a structured query to retrieve the correct data from the database.
- Ranking and Relevance: The information retrieval system ranks the possible answers based on relevance to ensure that the most pertinent information is presented to the user.

2.2.3 Human-Computer Interaction (HCI):

The chatbot's design focuses on effective communication with users. In this case, the user interface (UI) is designed to allow easy access to historical facts, trivia questions, and responses. Key concepts of HCI include:

- Usability: Ensuring that the chatbot is intuitive and easy to use, with features like quick replies and buttons for ease of navigation.

- User-Centered Design: Designing the chatbot to prioritize the user's needs and interests, making it engaging and accessible.

2.2.4 Database Management:

The historical chatbot relies on a well-structured relational database (MySQL) to store and retrieve historical facts and quiz data. The theoretical concepts of relational databases are applied to ensure efficient storage, retrieval, and update of data.

- Normalization: Ensuring that the database tables are optimized for minimal redundancy and maximum efficiency.
- Query Optimization: Implementing efficient queries to handle large datasets and return fast results for the user's query.

2.3 Related Technologies and Tools

2.3.1 Frontend Technologies:

- HTML (HyperText Markup Language): HTML is used to structure the content and layout of the chatbot's user interface, including the display of facts, quizzes, and chat interactions.
- CSS (Cascading Style Sheets): CSS is used to style the chatbot interface, ensuring a visually appealing and responsive design. Custom styles for chat windows, buttons, and message display are created to improve user experience.
- JavaScript: JavaScript powers the dynamic behavior of the webpage. It handles the interaction between the user and the chatbot, sending user input to the server and displaying bot responses. JavaScript frameworks like jQuery or vanilla JavaScript might be used for AJAX requests to asynchronously fetch data without reloading the page.

2.3.2 Backend Technologies:

- Flask: Flask is a lightweight web framework for Python used to build the backend server of the chatbot. Flask is responsible for handling incoming HTTP requests, processing user queries, and fetching appropriate responses from the database. It manages the route handling, session management, and integration with the MySQL database.
- Python: Python is the primary programming language for backend development. It is used for implementing the chatbot logic, including handling queries, interacting with

the database, and processing the user's input.

2.3.3 Database:

MySQL: MySQL is the relational database management system used to store historical facts, quiz data, and user interactions. It allows for efficient storage and retrieval of structured data, with tables dedicated to historical topics, facts, quizzes, and more.

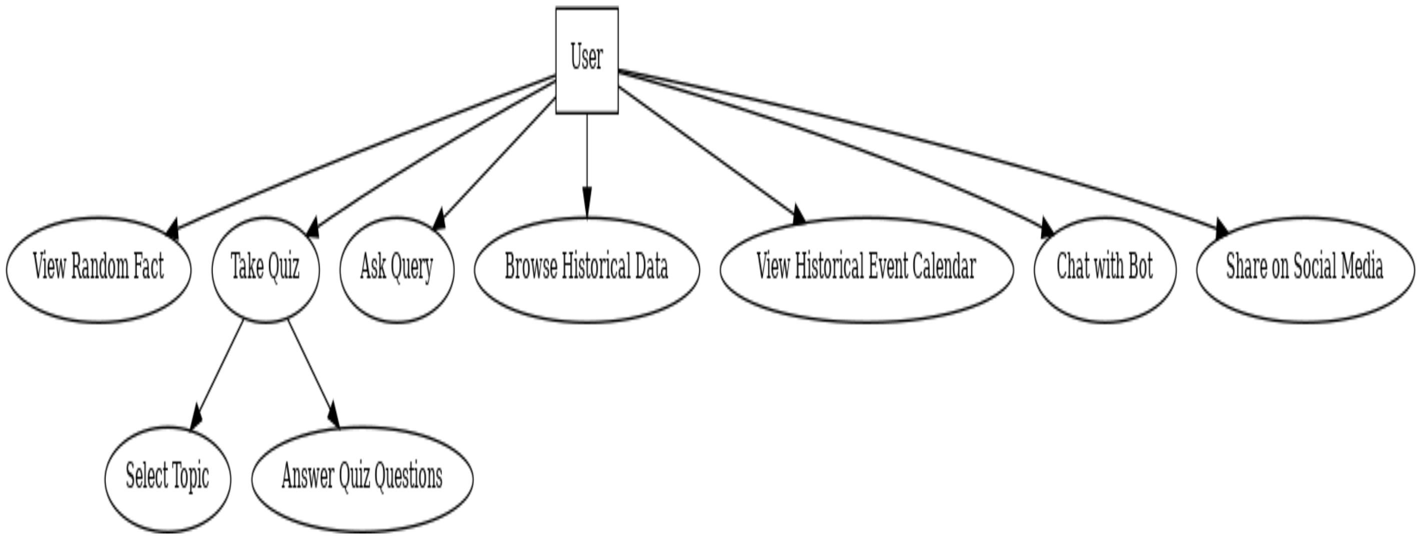
- SQL Queries: SQL is used to interact with the MySQL database. The chatbot sends structured SQL queries to fetch relevant historical data based on user input. The use of SELECT, JOIN, and WHERE clauses allows precise data retrieval.
- MySQL Workbench: This tool is used to design, query, and manage the MySQL database. It helps visualize the schema and troubleshoot database queries.

2.3.4 Version Control:

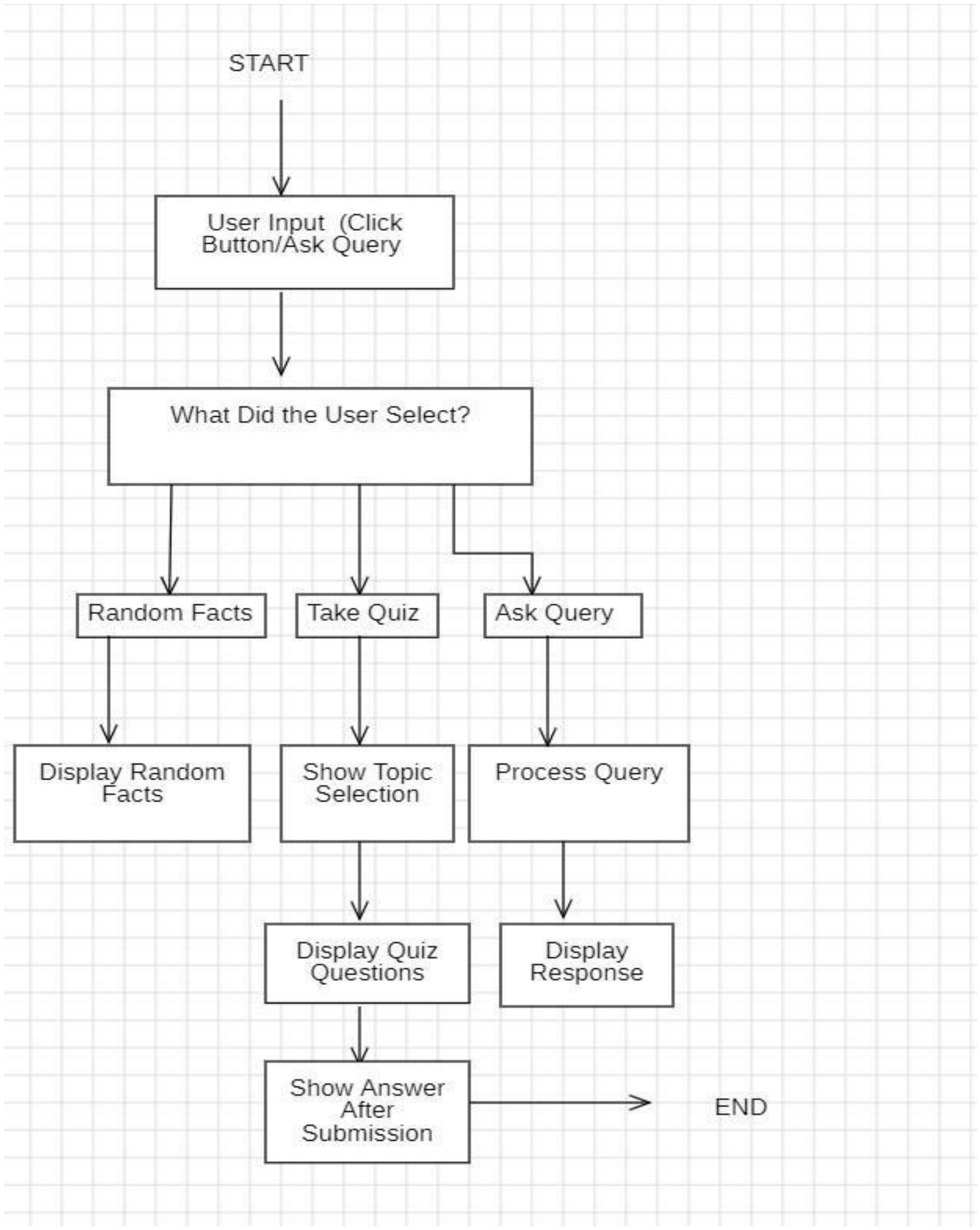
- Git: Git is a version control system used to manage and track changes to the codebase of the project. It ensures that developers can collaborate efficiently and keep track of the project's evolution.
- GitHub: GitHub is a platform used to host and share the project code repository. It allows for team collaboration, issue tracking, and version control of the chatbot’s codebase.

2.4 UML DIAGRAMS:

2.4.1 Use Case Diagram



2.4.2 Activity Diagram



CHAPTER-3

3. METHODOLOGY

3.1 Description of Approach/Method Used

The approach combines Natural Language Processing (NLP) techniques, structured database design, and web development using Flask. The following steps detail the implementation process:

3.1.1 Problem Analysis and Requirement Gathering

- Providing accurate and concise responses to user queries about historical events.
- Sharing random historical facts interactively.
- Delivering trivia quizzes for educational purposes.
- The requirements were categorized into functional (chatbot responses, quiz interface) and non-functional (user-friendly design, quick response time) needs.
- The system architecture was designed to integrate a MySQL database for storing historical data, facts, and quiz questions.

3.1.2 Data Collection and Structuring

- Historical facts, trivia questions, and detailed historical narratives were sourced from:
- Academic textbooks.
- Trusted online encyclopedias such as Wikipedia.
- Open-source datasets tailored for educational purposes.
- The data was categorized into:
- Facts Database: Short, concise, single-line facts categorized by topics.
- Quiz Database: Trivia questions and answers, linked by topic IDs for seamless topic-based navigation.
- Historical Data: Detailed narratives structured with headings and subheadings for easy query resolution.

3.1.3 Database Design and Integration

- A relational database using MySQL was developed, with the following key tables:

- facts: Stores historical facts, linked by topic ID.
- hist_data: Contains comprehensive historical information with associated metadata.
- topics: Acts as a central table linking facts, quizzes, and historical narratives.
- Database normalization was performed to ensure efficient data retrieval and minimal redundancy.

3.1.4 Chatbot Development Using Natural Language Processing (NLP)

- A lightweight NLP pipeline was implemented to:
- Analyze user queries for intent and keyword extraction.
- Match user input with relevant database entries using a similarity algorithm.
- Formulate responses in conversational style to mimic human-like interaction.
- Libraries like NLTK and spaCy were used to enhance the chatbot's language understanding capabilities.
- Rule-based fallback mechanisms were implemented for unrecognized queries, directing users to alternative resources or suggesting available topics.

3.1.5. Trivia Quiz Module Implementation

- A dynamic quiz system was designed to:
- Allow users to select a topic before starting the quiz.
- Fetch questions and options dynamically from the quiz_questions table.
- Display correct answers upon user selection to ensure an interactive learning experience.
- JavaScript was used for front-end interactivity, enabling real-time feedback without page reloads.

3.1.6. Front-End Development Using Flask

- Flask was selected for its simplicity and flexibility, facilitating:
- A responsive, lightweight user interface.
- Integration of chatbot logic with database queries.
- The front-end was designed to:
- Display chatbot responses and historical facts in an interactive format.

3.1.7 Testing and Validation

- Accuracy: Validating responses against the dataset to minimize errors.
- User Experience: Gathering feedback from sample users to improve the interface and chatbot interaction flow.
- Performance metrics such as response time, accuracy, and engagement levels were monitored and optimized.

3.1.8 Deployment and Future Enhancements

- The application was deployed on a local server for initial testing and feedback collection.
- Future scope includes:
- Integrating advanced NLP models (e.g., GPT-based transformers) for deeper query understanding.
- Expanding the database to include modern history and other educational domains.
- Adding multimedia elements (images, videos) to enhance user engagement.

3.2 Software and Hardware:

3.2.1 Software Requirements:

Frontend

- Programming Language:

  HTML/CSS: For structuring and styling the chatbot interface, ensuring an interactive and visually appealing design.

  JavaScript: Used for making the frontend dynamic, enabling real-time interaction with the chatbot, fetching random facts, and handling quizzes.

- Web Framework:

  Flask: Although Flask is typically backend, it also serves as the framework for rendering HTML templates, routing, and managing user input in the frontend.

- Libraries:

  Bootstrap: For responsive and mobile-friendly design, helping structure the layout and style of the webpage.

  jQuery: (Optional) Used for handling AJAX requests to dynamically fetch content from the backend without refreshing the page.

Backend:

- Programming Language:

  Python: The primary backend language, handling the chatbot logic, user queries, and interaction with the database.

- Web Framework:

  Flask: The lightweight Python web framework handles HTTP requests, rendering the frontend pages, and serving as the interface between the user and the backend logic.

- Database:

  MySQL: A relational database management system used to store historical facts, quizzes, and chatbot-related data.

- Libraries and Tools:

  Transformers (Hugging Face): For natural language processing (NLP), allowing the chatbot to process and generate human-like responses based on user input.

  NumPy: For data processing and manipulation, helping with efficient handling of numerical data if needed.

  Pandas: For managing and analyzing structured historical data, including facts and quizzes.

  Matplotlib: (Optional) Used for creating visualizations of historical data if required.

  OpenCV: (Optional) For handling image-related tasks, such as displaying historical images or processing visual content.

3.2.2 Hardware Requirements:

- Processor:

  Multi-core CPU (Intel i5/i7 or equivalent) for smooth performance during AI model inference and web hosting.

- RAM:

  Minimum of 8GB RAM, ideally 16GB or more for handling large datasets and running machine learning models efficiently.

- Storage:

  SSD with at least 100GB of free space for storing the MySQL database, historical data, images, and chatbot logs.

- Internet Connection:

  Stable internet connection for downloading required libraries, hosting the chatbot.

CHAPTER-4

4. IMPLEMENTATION

4.1 Detailed Explanation of How the Project Was Executed

4.1.1 Setting Up Development Environment

Install Python:

- Install the latest stable version of Python (preferably 3.7 or above). You can download it from the official Python website.

Install pip:

- Ensure pip (Python's package manager) is installed. This comes by default with newer Python versions but can be manually installed if needed.

Install Required Libraries:

- Flask for the web framework, allowing for routing and rendering templates.
- NumPy, Pandas, and Matplotlib for data manipulation, processing, and visualization (if needed).
- Transformers (Hugging Face) for leveraging NLP models for the chatbot.
- MySQL Connector for Python to connect Flask to MySQL database.
- Additional libraries like OpenCV (if image-related tasks are involved).

Set Up IDE/Editor:

- VS Code for a lightweight, customizable experience.
- PyCharm for a more feature-rich Python development environment.
- Google Colab for cloud-based development.

Install MySQL:

- Download and install MySQL from the official website or use a local stack like XAMPP (which includes MySQL).
- Ensure MySQL Workbench or phpMyAdmin is installed for easy database management. Use MySQL Workbench for a GUI-based approach to create, query, and manage databases.

Create Database and Tables:

- Create a new database where historical facts, quiz questions, and user data will be stored.
- Define tables to organize data effectively.

4.1.2 Backend Development

Set Up Flask App:

- Start with creating a basic Flask application that handles user requests and serves HTML pages. Ensure the app is modular and clean for scalability.
- Set routes for the homepage, random facts, and quiz interface:
    a. @app.route("/") for the homepage.
    b. @app.route("/random_fact") for displaying facts.
    c. @app.route("/quiz") for the quiz interface.
    d. @app.route("/chatbot") for handling chatbot interactions.

Database Connection:

- Use MySQL Connector to connect Flask to the MySQL database.
- Create a function that connects the app to the database to retrieve or store data.
- Example: Fetch facts, quiz data, or store user responses dynamically based on requests.

Design Backend Logic:

Write backend functions to handle:

a. Fetching random facts from the database when requested by the user.
b. Serving quiz questions from the database based on the selected topic.
c. Storing user responses to track quiz scores or chat history.

- Create a chatbot logic that uses Transformers (Hugging Face) models to process natural language queries and generate relevant responses from historical data.
- Implement any additional features like fetching images or visualizations related to historical events.

Integrate NLP Models:

- Use pre-trained language models from Transformers to power the chatbot. Choose models like GPT-3 or BERT for handling general queries or topic-specific historical data queries.
- Fine-tune or use specialized models for generating responses related to history. Integrate these models into the Flask backend for smooth interaction.

User Interaction Handling:

- Write logic to handle user inputs from the frontend (e.g., quiz answers or chatbot queries).
- Implement follow-up actions based on the user's answers, such as providing the correct answers for wrong quiz responses or retrieving more historical facts upon request.

4.1.3 Frontend Development

Create HTML Templates:

Design basic HTML pages for each section of the chatbot, including:

- Homepage: Introduce the chatbot and provide buttons for taking quizzes, displaying facts, and starting the conversation.
- Quiz Page: Display the selected quiz topic and multiple-choice questions dynamically.
- Chatbot Interface: Design a section where the user can ask questions and receive chatbot responses.

Styling with CSS:

- Style the chatbot interface to be visually appealing and responsive across devices.
- Use CSS frameworks like Bootstrap for a flexible grid system and responsive design.
- Customize the interface to fit the golden theme and interactive buttons as per project requirements.

Dynamic Interaction with JavaScript:

Use JavaScript or jQuery for handling dynamic elements like:

a. Fetching and displaying random facts without refreshing the page.
b. Displaying quiz questions and options and showing correct answers once selected.

Mobile-Friendly Layout:

- Ensure the frontend is responsive, making the chatbot easy to use on both desktop and mobile devices.
- Implement features like a collapsible menu or scrollable sections to improve the UI on smaller screens.

Frontend-Backend Integration:

Link frontend elements (like buttons) to the backend logic using Flask routes. For example, when the "Take Quiz" button is clicked, the frontend will trigger the backend to fetch quiz questions from the database.Ensure the data passed from the backend is correctly displayed.

4.1.4 Integration and Deployment

Frontend and Backend Integration:

- Ensure the smooth flow of data between the frontend and backend. For example, when the user selects a quiz topic, the frontend sends this information to the backend, which then fetches the appropriate questions from the database.
- Ensure proper integration of the chatbot logic to generate responses based on user queries and provide dynamic answers related to history.

Testing Locally:

- Test the application on your local machine to check if:
    a. The Flask server is running without errors.
    b. Data retrieval and insertion from the MySQL database work as expected.
    c. The chatbot responds to user queries accurately.
    d. The quiz interface displays questions and answers correctly.

Debugging:

- Debug any issues related to data flow between the frontend and backend.
- Fix any bugs or errors in handling user inputs or displaying data.

Deployment:

- Deploy the Flask application to a cloud platform (such as Heroku, AWS, or Google Cloud).
- Configure the database connection for the cloud environment, ensuring smooth transition from local development to production.
- Set up necessary environment variables for security, including database credentials and any API keys used in the application.

4.2 Code Structure and Architecture

The project is organized in a modular format to ensure clarity, scalability, and separation of concerns. The architecture consists of a frontend, a backend, and a database. The structure follows a clear division of files and folders, making it easy to maintain and expand.

```
Historical_Chatbot/
│
├── app.py                    # Main Flask application file
├── chatbotlogic.py             # Logic for chatbot responses
├── fact.py                   # Handles random fact retrieval
├── quiz.py                   # Manages quiz functionality
│
├── static/
│   ├── styles.css             # Styling for the user interface
│   ├── facts.js               # JavaScript for facts display
│   ├── chatbot.js              # JavaScript for chatbot interactions
│   └── quiz.js                # JavaScript for quiz logic
│
├── templates/
│   └── index.html              # Frontend HTML template
│
└── database/
    └── history_db        # SQL file for database schema
```

- **app.py**: This is the main entry point of the Flask application. It handles routing and integrates other backend modules like fact.py, quiz.py, and chatbotlogic.py.
- **fact.py**: Contains logic to fetch random historical facts from the database.
- **quiz.py**: Handles quiz-related operations, including retrieving questions and answers.
- **chatbotlogic.py**: Processes user queries and retrieves relevant data from the database or other sources.
- **templates/index.html**: This file is the main user interface that connects all frontend functionalities. It includes input fields, buttons, and displays the chatbot, facts, and quizzes.
- **static/styles.css**: Provides styling for the chatbot, facts display, and quiz UI. It ensures a clean and visually appealing design.
- **static/facts.js**: Handles fetching and displaying random facts from the backend.
- **static/chatbot.js**: Handles user queries and responses in the chatbot interface.
- **static/quiz.js**: Fetches quiz questions and interacts with the user for answers

CODE:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/js/all.min.js"></script>
  <title>Quiz Example</title>
</head>
<body>

<div class="container">
  <div class="left-side" id="left-side">
    <h2>Facts & Quiz</h2>
    <div class="left-icons">
      <button onclick="loadQuizTopics()"><i class="fas fa-list-alt"></i><span>Select
Topic</span></button>
      <button onclick="loadRandomFact()"><i class="fas fa-lightbulb"></i><span>Random
Facts</span></button>
    </div>

    <div id="content-area"></div>
    <div id="fact-container" class="fact-container"></div> <!-- Container for random fact -->
  </div>

  <div class="right-side" id="right-side">
    <div id="chatbot-container">
      <!-- Chatbot Header -->
      <div id="chatbot-header">
        <img src="{{ url_for('static', filename='chat-bot.gif') }}" alt="Chatbot Icon" id="chatbot-icon">
        <span id="chatbot-name">Vyasa</span>
      </div>
      <div id="chat-display">
        <!-- Chat messages will appear here -->
      </div>

      <div id="chat-input">
        <input type="text" id="user-input" placeholder="Type your message..." />
        <button id="send-btn">Send</button>
```

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/js/all.min.js"></script>
  <title>Quiz Example</title>
</head>
<body>

<div class="container">
  <div class="left-side" id="left-side">
    <h2>Facts & Quiz</h2>
    <div class="left-icons">
      <button onclick="loadQuizTopics()"><i class="fas fa-list-alt"></i><span>Select
Topic</span></button>
      <button onclick="loadRandomFact()"><i class="fas fa-lightbulb"></i><span>Random
Facts</span></button>
    </div>

    <div id="content-area"></div>
    <div id="fact-container" class="fact-container"></div> <!-- Container for random fact -->
  </div>

  <div class="right-side" id="right-side">
    <div id="chatbot-container">
      <!-- Chatbot Header -->
      <div id="chatbot-header">
        <img src="{{ url_for('static', filename='chat-bot.gif') }}" alt="Chatbot Icon" id="chatbot-icon">
        <span id="chatbot-name">Vyasa</span>
      </div>
      <div id="chat-display">
        <!-- Chat messages will appear here -->
      </div>

      <div id="chat-input">
        <input type="text" id="user-input" placeholder="Type your message..." />
        <button id="send-btn">Send</button>
      </div>
    </div>
  </div>
</div>

<script>

  // Assign the bot icon URL to a JavaScript variable
  const botIconUrl = "{{ url_for('static', filename='chat.png') }}";

  document.getElementById("send-btn").addEventListener("click", getChatResponse);

  function getChatResponse() {
    const userInput = document.getElementById("user-input").value;
    if (!userInput.trim()) {
      alert("Please ask a valid question.");
      return;
    }

    const responseDiv = document.getElementById("chat-display");
    responseDiv.innerHTML += `<p><strong>You:</strong> ${userInput}</p>`; // Display user input

    // Fetch bot response from backend
    fetch('/chat', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ user_input: userInput }),
    })
    .then(response => response.json())
    .then(data => {
      const formattedResponse = formatResponse(data.response);
      displayTypingEffect(formattedResponse); // Directly display word-by-word response
    })
    .catch(error => {
      console.error('Error:', error);
      responseDiv.innerHTML += `<p><strong>Bot:</strong> Sorry, something went wrong!</p>`;
    });
  }

  function formatResponse(text) {
    // Clean up response text, remove extra spaces and ensure it's formatted well
    return text.replace(/([.,!?])/g, "$1 ").replace(/\s+/g, " ").trim();
  }

  // Simulate typing effect, word by word
  function displayTypingEffect(text) {
    const responseDiv = document.getElementById("chat-display");
    const words = text.split(' '); // Split response into words
    let wordIndex = 0;

    // Display "Bot:" first, and then append words one by one beside it
    responseDiv.innerHTML += `<p><strong>Bot:</strong>`;

    const intervalId = setInterval(function() {
      if (wordIndex < words.length) {
        responseDiv.innerHTML += ` ${words[wordIndex]}`; // Append words directly after "Bot:"
        wordIndex++;
      } else {
        clearInterval(intervalId);
      }
    }, 100); // Adjust speed by changing this delay (100ms per word)
  }
</script>

<script src="{{ url_for('static', filename='quiz.js') }}"></script>
<script src="{{ url_for('static', filename='facts.js') }}"></script>

</body>
</html>

from flask import Flask, render_template, request, jsonify
import mysql.connector
import faiss
from sentence_transformers import SentenceTransformer
import numpy as np
import re
from fact import fact_blueprint
from quiz import quiz_blueprint

# Initialize the Flask app
app = Flask(_name_)

# Load the embedding model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Global FAISS Index and data
faiss_index = None
indexed_data = None

# Connect to database
def connect_db():
    try:
        return mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            database="history_db"
        )
    except mysql.connector.Error as err:
        print(f"Error: {err}")
        return None

# Build FAISS Index
def build_faiss_index():
    global faiss_index, indexed_data
    conn = connect_db()
    if not conn:
        print("Failed to connect to the database.")
        return
    cursor = conn.cursor(dictionary=True)
    try:
        cursor.execute("SELECT sub_heading, content FROM hist_data")
        data = cursor.fetchall()
        indexed_data = data
        embeddings = [model.encode(row['content']) for row in data]
        faiss_index = faiss.IndexFlatL2(embeddings[0].shape[0])
        faiss_index.add(np.array(embeddings))
    except mysql.connector.Error as err:
        print(f"Database query error: {err}")
    finally:
        cursor.close()
        conn.close()

# Query FAISS Index
def query_faiss_index(query, k=5):
    if not faiss_index:
        return []
    query_vector = model.encode(query)
    distances, indices = faiss_index.search(np.array([query_vector]), k)
    return [indexed_data[idx] for idx in indices[0]]

# Summarize Response by Complete Sentences
def summarize_response(results, max_length=200):
    combined_content = " ".join([res['content'] for res in results])
    sentences = re.split(r'(?<=[.!?]) +', combined_content)
    summary = ""
    for sentence in sentences:
        if len(summary) + len(sentence) <= max_length:
            summary += sentence + " "
        else:
            break
    return summary.strip()

# Route for the homepage
@app.route('/')
def home():
    return render_template('index.html')

# Flask Route for chat functionality
@app.route('/chat', methods=['POST'])
def chat():
    user_query = request.json.get('user_input', '')
    if not user_query:
        return jsonify({"response": "Please provide a valid query."})

    results = query_faiss_index(user_query)
    if not results:
        return jsonify({"response": "No relevant data found for your query."})

    response = summarize_response(results)
    return jsonify({"response": response})

# Register Blueprints
app.register_blueprint(fact_blueprint)
app.register_blueprint(quiz_blueprint)

if _name___== '_main_':
    build_faiss_index()
    if faiss_index:
        print("FAISS index successfully built!")
    else:
        print("Failed to build FAISS index.")
    app.run(debug=True, port=5002)

from flask import Blueprint, jsonify
import mysql.connector

fact_blueprint = Blueprint('fact', __name__)

# MySQL Connection
db = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="history_db"
)

@fact_blueprint.route('/api/random_fact', methods=['GET'])
def get_random_fact():
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT fact FROM facts ORDER BY RAND() LIMIT 1")
    fact = cursor.fetchone()
    return jsonify(fact)

from flask import Blueprint, jsonify
import mysql.connector

quiz_blueprint = Blueprint('quiz', __name__)

# MySQL Connection
db = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="history_db"
)

@quiz_blueprint.route('/get_quiz_topics', methods=['GET'])
def get_quiz_topics():
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT topic_id, topic_name FROM quiz_topics")
    topics = cursor.fetchall()
    return jsonify(topics)

@quiz_blueprint.route('/get_quiz_questions/<topic_name>', methods=['GET'])
def get_quiz_questions(topic_name):
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT * FROM quiz_questions WHERE topic_name = %s", (topic_name,))
    questions = cursor.fetchall()
    return jsonify(questions)

// Function to fetch and display a random fact
function loadRandomFact() {
    // Hide the quiz container when switching to facts
    document.getElementById('content-area').innerHTML = '';

    // Show the fact container when the 'Random Facts' button is clicked
    document.getElementById('fact-container').style.display = 'block';

    fetch('/api/random_fact')
        .then(response => response.json())
        .then(data => {
            // Display the fetched fact
            document.getElementById('fact-container').innerText = data.fact;
        })
        .catch(error => console.error('Error fetching fact:', error));
}

// Function to fetch and load quiz topics
function loadQuizTopics() {
    // Hide the facts container when selecting quiz
    document.getElementById('fact-container').style.display = 'none';

    // Fetch quiz topics and display them
    fetch('/get_quiz_topics')
      .then(response => response.json())
      .then(topics => {
        let topicContent = "";
        topics.forEach(topic => {
        topicContent += `
            <button onclick="loadQuizQuestions('${topic.topic_name}')">${topic.topic_name}</button>
          `;
        });

        document.getElementById('content-area').innerHTML = topicContent;
      });
  }

  // Function to fetch and load quiz questions for a selected topic
  function loadQuizQuestions(topicName) {
    // Hide the fact container when selecting quiz
    document.getElementById('fact-container').style.display = 'none';

    // Clear previous quiz content
    document.getElementById('content-area').innerHTML = '';
    fetch(`/get_quiz_questions/${topicName}`)
      .then(response => response.json())
      .then(questions => {
        let quizContent = '<h3>Quiz Questions</h3><p>Answer the following questions:</p>';

        questions.forEach((question, index) => {
          quizContent += `
            <div class="quiz-container">
              <div class="quiz-question" id="quiz-question-${index}">
                <p><strong>${index + 1}. ${question.question_text}</strong></p>
                <div class="quiz-options">
                  <label class="quiz-option">
                    <input type="radio" name="question-${index}" value="A"> A) ${question.option_a}
                  </label>
                  <label class="quiz-option">
                    <input type="radio" name="question-${index}" value="B"> B) ${question.option_b}
                  </label>
                  <label class="quiz-option">
                    <input type="radio" name="question-${index}" value="C"> C) ${question.option_c}
                  </label>
                  <label class="quiz-option">
                    <input type="radio" name="question-${index}" value="D"> D) ${question.option_d}
                  </label>
                </div>
                <button onclick="checkAnswer(${index}, '${question.correct_answer}')">Submit
Answer</button>
                <p id="answer-${index}"></p>
              </div>
            </div>
          `;
        });

        document.getElementById('content-area').innerHTML = quizContent;
      });
  }

  // Function to check if the selected answer is correct
  function checkAnswer(questionIndex, correctAnswer) {
    const selectedOption = document.querySelector(`input[name="question-
${questionIndex}"]:checked`);
    const answerElement = document.getElementById(`answer-${questionIndex}`);

    if (selectedOption) {
      if (selectedOption.value === correctAnswer) {
        answerElement.textContent = "Correct!";
        answerElement.className = 'correct';
      } else {
        answerElement.textContent = `Incorrect! The correct answer was ${correctAnswer}.`;
        answerElement.className = 'incorrect';
      }
    } else {
      answerElement.textContent = "Please select an answer.";
      answerElement.style.color = "orange";
    }
  }
```

4.3 Technical Challenges and Solutions

4.3.1 Database Integration and Query Processing

Challenge: Integrating a MySQL database to handle a large volume of historical facts, quiz questions, and chatbot data while ensuring fast and efficient query processing. Queries sometimes failed when user inputs were not an exact match with database entries.

Solution: Implemented SQL LIKE queries with wildcard operators to handle flexible user input. Additionally, the queries were optimized using proper indexing and validation checks to ensure accuracy in data retrieval.

4.3.2 Real-Time Chatbot Interaction

Challenge: Implementing a chatbot that could process user queries in real time without causing delays or impacting the performance of the application.

Solution: Leveraged asynchronous JavaScript (async/await) in the frontend chatbot.js file to fetch responses from the Flask backend seamlessly. The Flask API (chatbotlogic.py) was designed to process requests quickly and return precise answers.

4.3.3 Dynamic Content Rendering

Challenge: Displaying dynamic content such as facts, quizzes, and chatbot replies without requiring a page reload.

Solution: Utilized JavaScript to update the DOM dynamically. For instance, facts.js and quiz.js handle the display of random facts and quiz content, while chatbot.js manages chatbot interactions with minimal latency.

4.3.4 User Input Handling

Challenge: Handling user input variations, including typos, irrelevant queries, or incomplete sentences, which could break the system's logic.

Solution: Added input validation and sanitization in chatbotlogic.py. Used natural language matching techniques to improve query understanding and match user input with database content.

4.3.5 Frontend and Backend Synchronization

Challenge: Ensuring smooth communication between the frontend (index.html) and backend (app.py), especially when managing multiple functionalities like facts, quizzes, and chatbot interactions.

Solution: Created separate routes in app.py to handle different features, such as /chatbot, /facts, and /quiz. AJAX requests were implemented in JavaScript to communicate with these endpoints.

CHAPTER-5

5. RESULTS

5.1 Presentation of Result

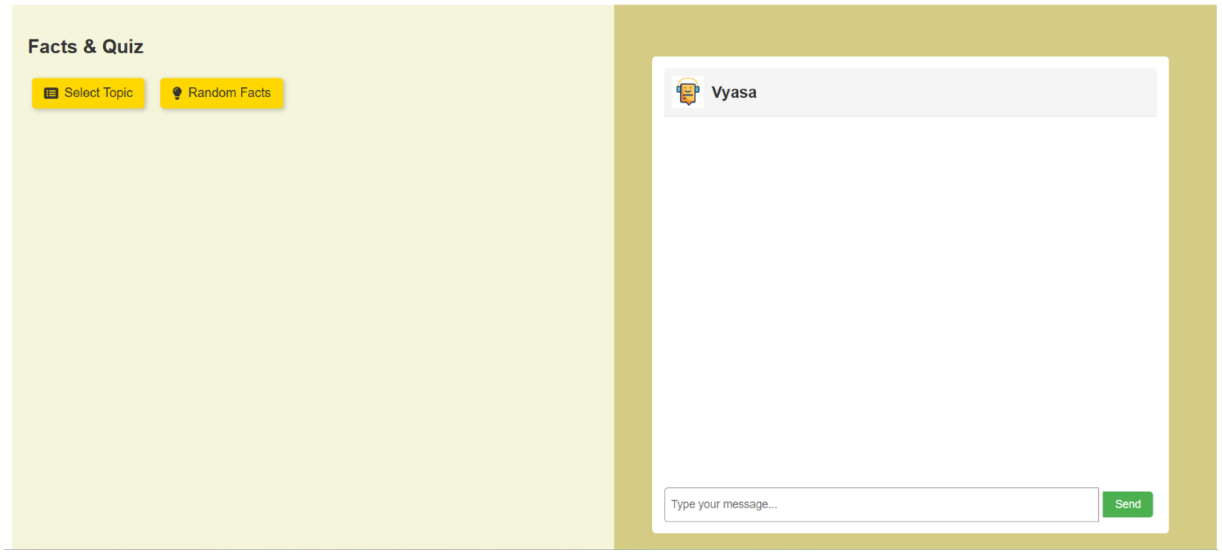
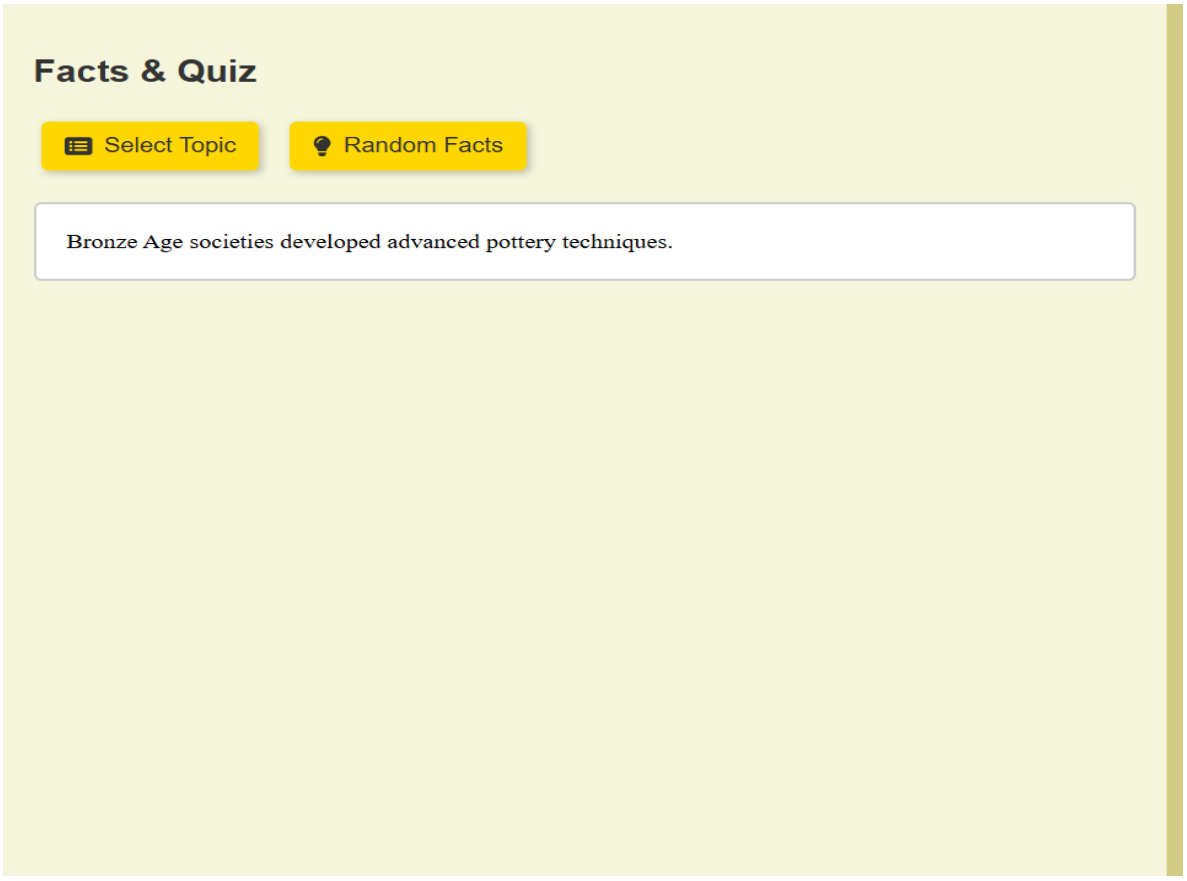
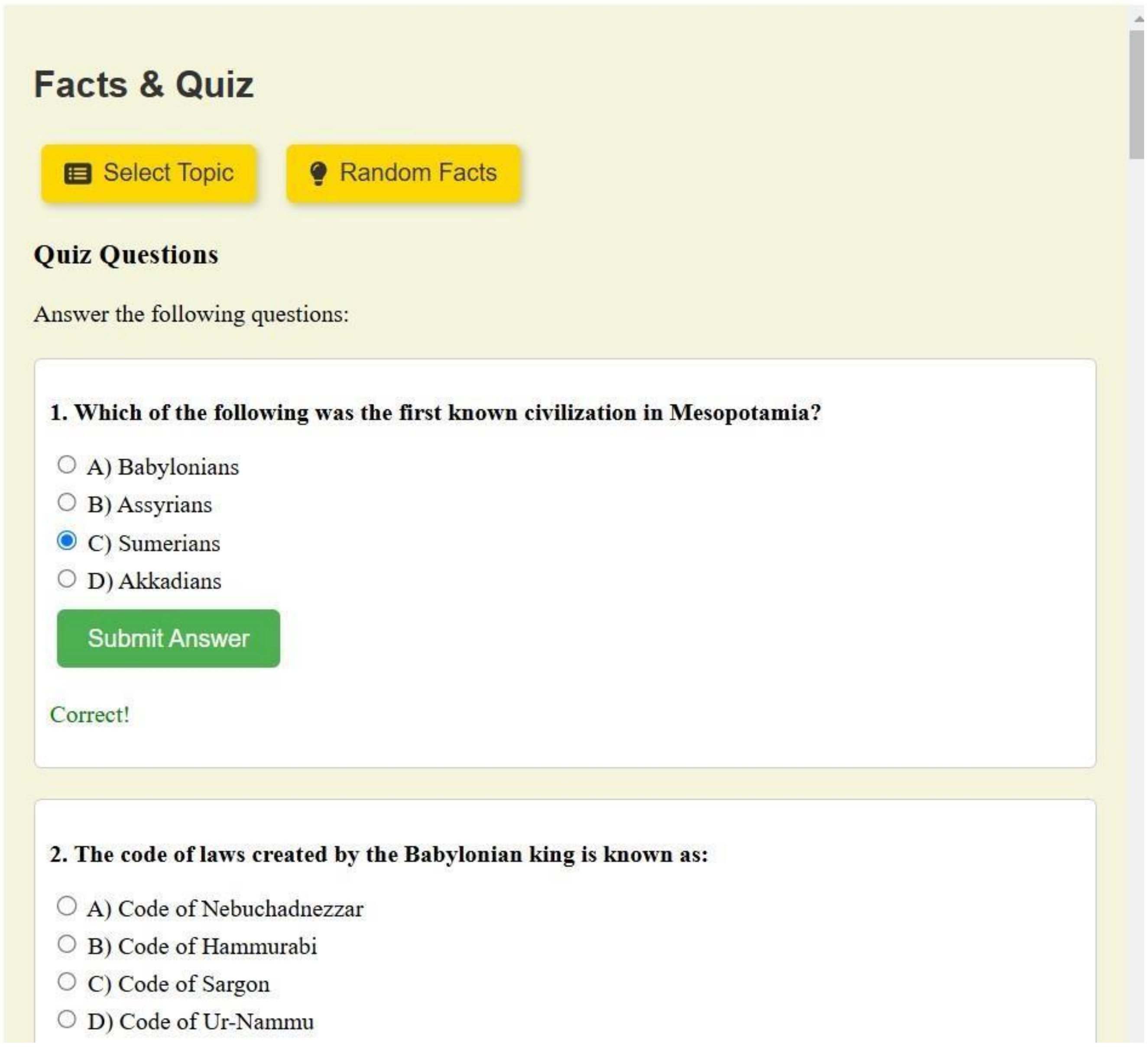


Fig 1. Home Page



Bronze Age societies developed advanced pottery techniques.

Fig 2.Facts Area



Facts & Quiz

Quiz Questions

Answer the following questions:

1. Which of the following was the first known civilization in Mesopotamia?

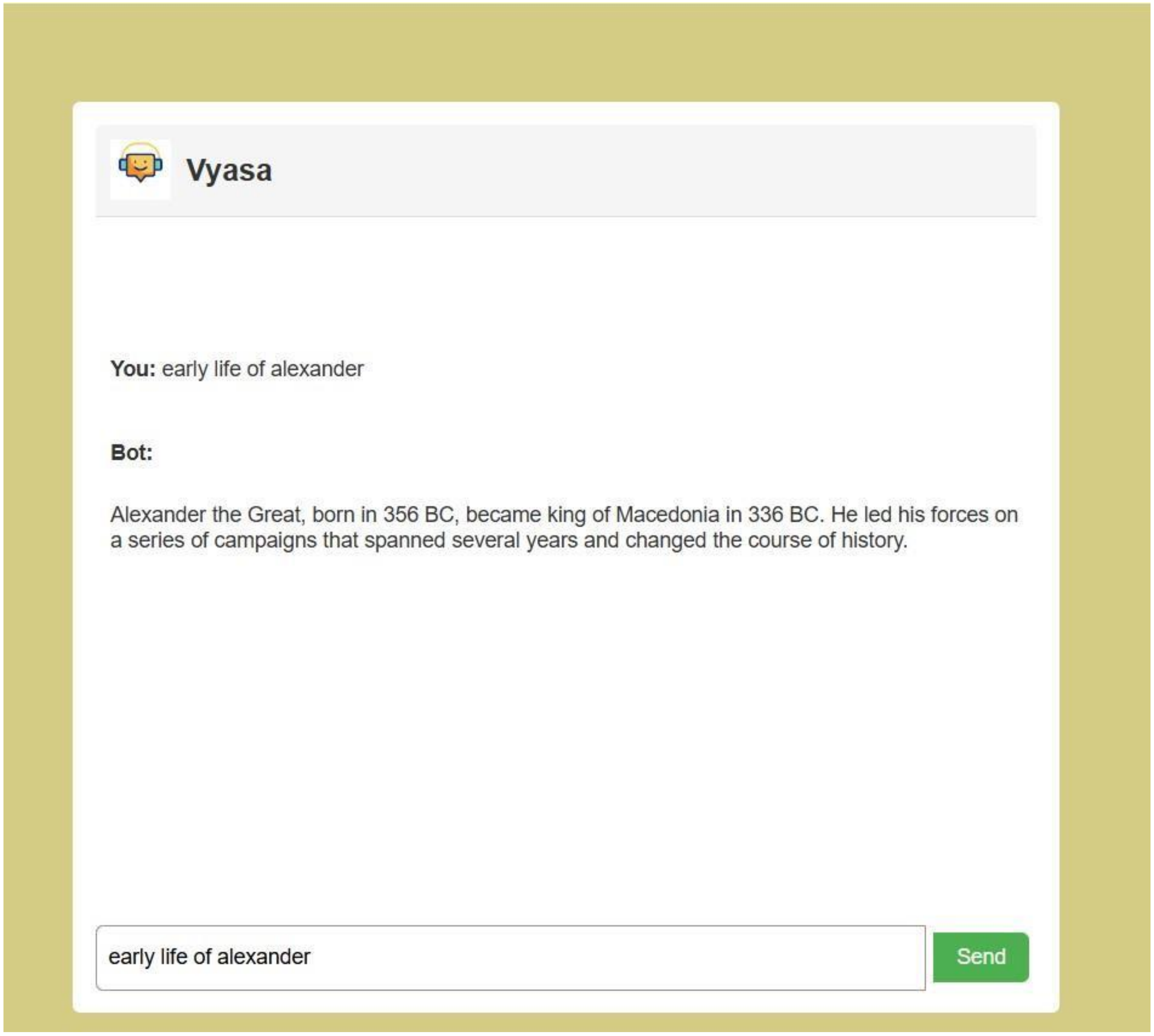
- A) Babylonians
- B) Assyrians
- [✓ C) Sumerians]
- D) Akkadians

Correct!

2. The code of laws created by the Babylonian king is known as:

- A) Code of Nebuchadnezzar
- B) Code of Hammurabi
- C) Code of Sargon
- D) Code of Ur-Nammu

Fig 3. Quiz Area



You: early life of alexander

Bot:

Alexander the Great, born in 356 BC, became king of Macedonia in 336 BC. He led his forces on a series of campaigns that spanned several years and changed the course of history.

early life of alexander

Fig 4. Chatbot Area

CONCLUSION

The Historical Facts and Trivia Chatbot project successfully integrates modern technologies to create an interactive learning tool. By using Flask, MySQL, and NLP models from Hugging Face, the chatbot provides an engaging platform for users to explore historical events and test their knowledge. The system delivers accurate, context-aware responses, ensuring users receive reliable information while enjoying an educational experience.

The chatbot offers a dynamic and personalized approach to history, encouraging user interaction through random facts and quizzes. By structuring historical data within a database, the chatbot can easily retrieve relevant information and respond to queries efficiently. The inclusion of quizzes further enhances user engagement, promoting active learning in a fun and accessible way.

Looking ahead, the chatbot has the potential for further expansion. Future developments could include the addition of modern history, multimedia features like images and videos, and even more advanced AI models. This project demonstrates how AI can transform educational tools and offers a promising foundation for future advancements in interactive learning.

REFERENCES

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. A., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need*. In Advances in Neural Information Processing Systems (NeurIPS 2017).
2. Hugging Face. (2024). *Transformers: State-of-the-art Natural Language Processing for Pytorch and TensorFlow 2.0*. Retrieved from https://huggingface.co/transformers
3. Liu, L., & Liu, Y. (2020). *A survey of chatbot design techniques and applications in education*. *Educational Technology & Society*, *23*(4), 49-59.
4. Finkelstein, L. (2017). *Developing educational chatbots for the digital era*. *Journal of Educational Technology Systems*, *46*(2), 223-239.
5. Zhang, X., Zhao, J., & LeCun, Y. (2015). *Character-level Convolutional Networks for Text Classification*.
6. Python Software Foundation. (2024). *Flask Web Framework*. Retrieved from https://flask.palletsprojects.com
7. MySQL AB. (2024). *MySQL 8.0 Reference Manual*. Retrieved from https://dev.mysql.com/doc/