Problem Statement 1

1. API Data Retrieval and Storage

Objective:

To fetch book data from a REST API, store it locally using SQLite, and display the retrieved records.

Approach:

I used a public REST API that returns book-related information in JSON format. The API response contained attributes such as book title, author name, and publication details. Using Python's `requests` library, I sent an HTTP GET request and parsed the JSON response.

For storage, I used SQLite as it is lightweight, easy to integrate with Python, and does not require additional setup. A database file was created programmatically, and a table schema was defined to store book attributes. After inserting the records, I queried the database and displayed the stored data to verify correctness.

Assumptions:

- The API always returns valid JSON data
- The API response contains at least one author per book
- SQLite is sufficient for local storage needs

Outcome:

A local SQLite database containing book records fetched from the API, with successful retrieval and display of stored data.

2. Data Processing and Visualization

Objective:

To fetch student score data from an API, calculate the average score, and visualize the data using a bar chart.

Approach:

I fetched structured user data from a trusted public API. Since the dataset did not directly contain test scores, I made a documented assumption to map a measurable attribute (username length) to a mock test score for demonstration purposes.

The processed data was used to compute the average score using Python's numerical operations. I then used Matplotlib to create a bar chart that visually represents individual student scores.

Assumptions:

- Username length reasonably simulates test score values
- Dataset size is small enough for in-memory processing

Outcome:

Average score calculation and a bar chart visualization representing student performance.

3. CSV Data Import to Database

Objective:

To read user data from a CSV file and insert it into an SQLite database.

Approach:

I created a CSV file containing user names and email addresses. Using Python's `csv` module, I read the file and inserted each record into an SQLite database table. SQLite automatically created the database file during execution. I verified successful insertion by querying the database.

Assumptions:

- CSV data is clean and well-formatted
- Email values are unique per user

Outcome:

A SQLite database populated with user records imported from the CSV file.

4. Most Complex Python Code

GitHub Link: https://www.kaggle.com/code/tanushreekakad/kitchen

https://www.kaggle.com/code/tanushreekakad/kitchen/edit

5. Most Complex Database Code

GitHub Link: *https://github.com/tanu0009/Auction-system*

Problem Statement 2

1. Self-Assessment (A/B/C)

- **AI**: A – Can design and implement solutions independently
- **Machine Learning:** A – Comfortable with model building and evaluation
- **Deep Learning:** B – Can implement models with supervision
- **Large Language Models:** B – Understands concepts and integration, gaining hands-on experience

2. Key Architectural Components of an LLM-Based Chatbot

An LLM-based chatbot consists of several core components. The user interface collects user input, which is passed to a prompt handling layer. This layer formats the input and manages conversational context. The prompt is then sent to the large language model for response generation.

To improve factual accuracy, a vector database is often integrated using a Retrieval-Augmented Generation (RAG) approach. Relevant documents are retrieved based on semantic similarity and provided to the LLM as context. The generated response is then post-processed for safety, formatting, and relevance before being shown to the user.

3. Vector Databases

Vector databases store numerical embeddings that represent semantic meaning. These databases allow efficient similarity search using distance metrics such as cosine similarity.

Hypothetical Use Case:

A chatbot answering questions from thousands of internal security documents.

Chosen Vector Database: FAISS

Reason:

FAISS is open-source, optimized for fast similarity search, scalable for large datasets, and easy to integrate with machine learning workflows.

Conclusion

This assignment demonstrates my practical experience with Python, APIs, data visualization, and databases, along with my conceptual understanding of AI, ML, and LLM-based systems. It reflects my structured approach to problem-solving and ability to document assumptions clearly.