AID7720:M.Tech Project-II

Project Title

Natural Language to SQL (NL2SQL) using SQLite - Chatbot with GUI Interface

Hem Prakash Dev **Roll-M24DE2009**

Hardik Sharma **Roll-M24DE2008**

Table of Contents

1. Project Objective

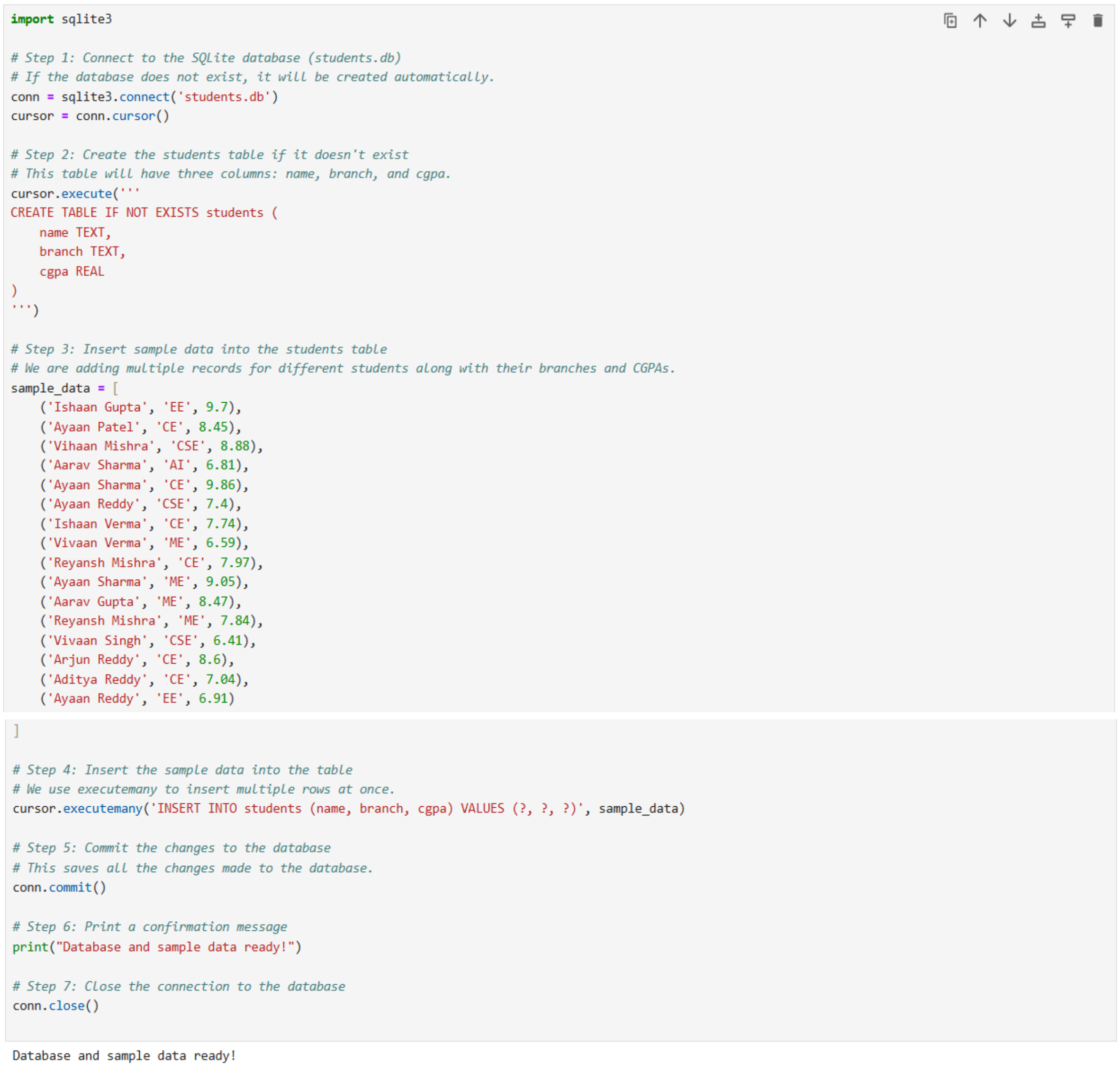
The aim of this project is to demonstrate how Natural Language Processing (NLP) can be used to interpret user queries written in plain English and convert them into SQL queries to retrieve data from a structured database. The goal is to simulate a simple question-answering chatbot that interacts with a student database.

2. Tools and Technologies Used

Tool/Technology	Description
Python	Programming Language used for implementation
SQLite3	Lightweight database used to store student data
Natural Language Processing	For parsing user queries
Random Module	To display varied responses for unmatched queries

3. Step1: Database Creation Module

- **Tool**: sqlite3
- Creates a database file students.db
- Creates a table students(name, branch, cgpa)
- Inserts 16 sample records



```python
import sqlite3

# Step 1: Connect to the SQLite database (students.db)
# If the database does not exist, it will be created automatically.
conn = sqlite3.connect('students.db')
cursor = conn.cursor()

# Step 2: Create the students table if it doesn't exist
# This table will have three columns: name, branch, and cgpa.
cursor.execute('''
CREATE TABLE IF NOT EXISTS students (
    name TEXT,
    branch TEXT,
    cgpa REAL
)
''')

# Step 3: Insert sample data into the students table
# We are adding multiple records for different students along with their branches and CGPAs.
sample_data = [
    ('Ishaan Gupta', 'EE', 9.7),
    ('Ayaan Patel', 'CE', 8.45),
    ('Vihaan Mishra', 'CSE', 8.88),
    ('Aarav Sharma', 'AI', 6.81),
    ('Ayaan Sharma', 'CE', 9.86),
    ('Ayaan Reddy', 'CSE', 7.4),
    ('Ishaan Verma', 'CE', 7.74),
    ('Vivaan Verma', 'ME', 6.59),
    ('Reyansh Mishra', 'CE', 7.97),
    ('Ayaan Sharma', 'ME', 9.05),
    ('Aarav Gupta', 'ME', 8.47),
    ('Reyansh Mishra', 'ME', 7.84),
    ('Vivaan Singh', 'CSE', 6.41),
    ('Arjun Reddy', 'CE', 8.6),
    ('Aditya Reddy', 'CE', 7.04),
    ('Ayaan Reddy', 'EE', 6.91)
]

# Step 4: Insert the sample data into the table
# We use executemany to insert multiple rows at once.
cursor.executemany('INSERT INTO students (name, branch, cgpa) VALUES (?, ?, ?)', sample_data)

# Step 5: Commit the changes to the database
# This saves all the changes made to the database.
conn.commit()

# Step 6: Print a confirmation message
print("Database and sample data ready!")

# Step 7: Close the connection to the database
conn.close()
```

```
Database and sample data ready!
```

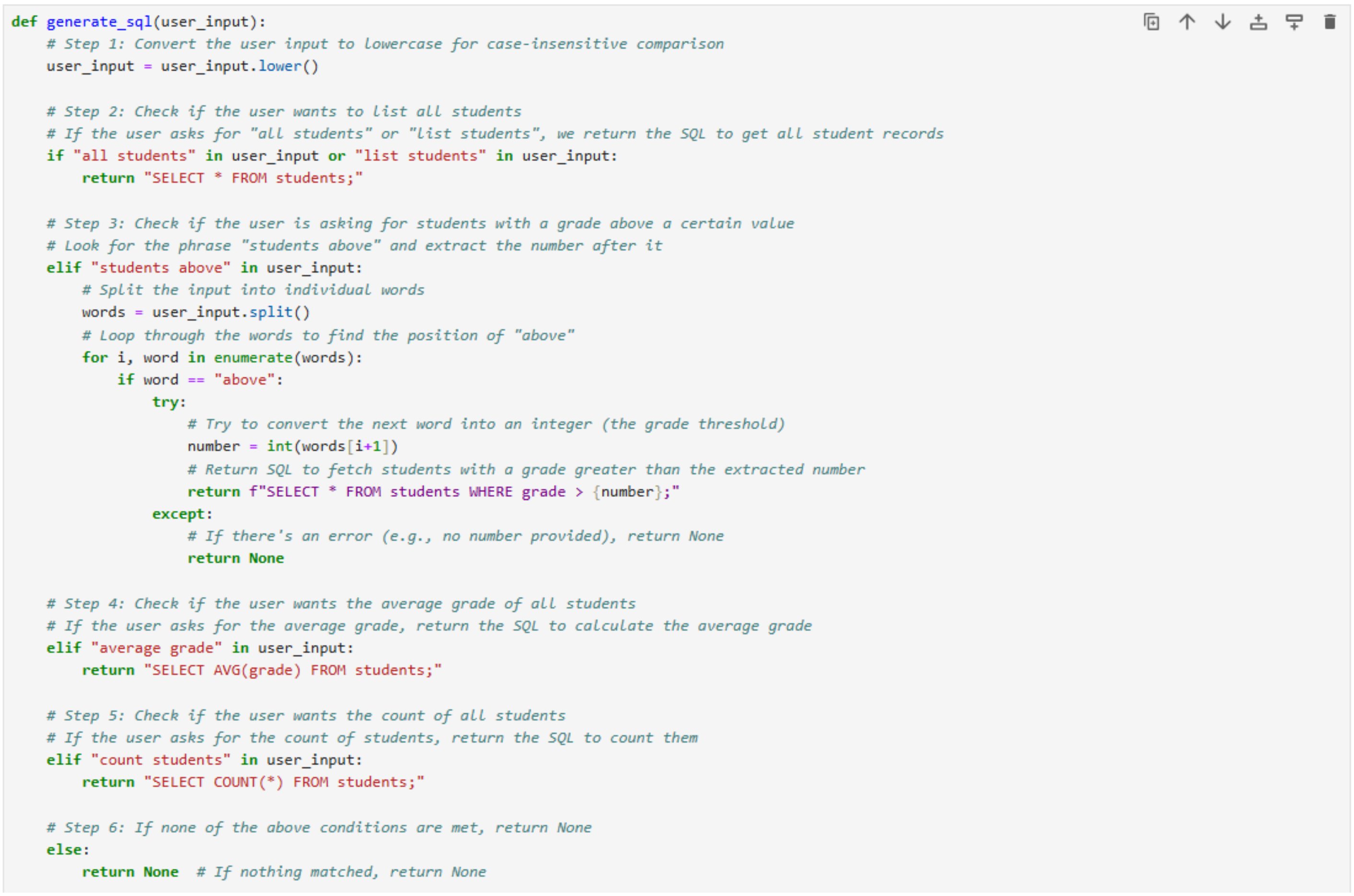
4. Step2: Natural Language to SQL Module

a.) **generate_sql()** Function

- Handles basic keyword-based mappings
- Simple rule-based parsing for fixed phrases like:
  - ✓ "all students" → SELECT * FROM students;
  - ✓ "students above 8" → SELECT * FROM students WHERE cgpa > 8;
  - ✓ "average grade" → SELECT AVG(cgpa) FROM students;

b.) **fetch_sql_query()** Function

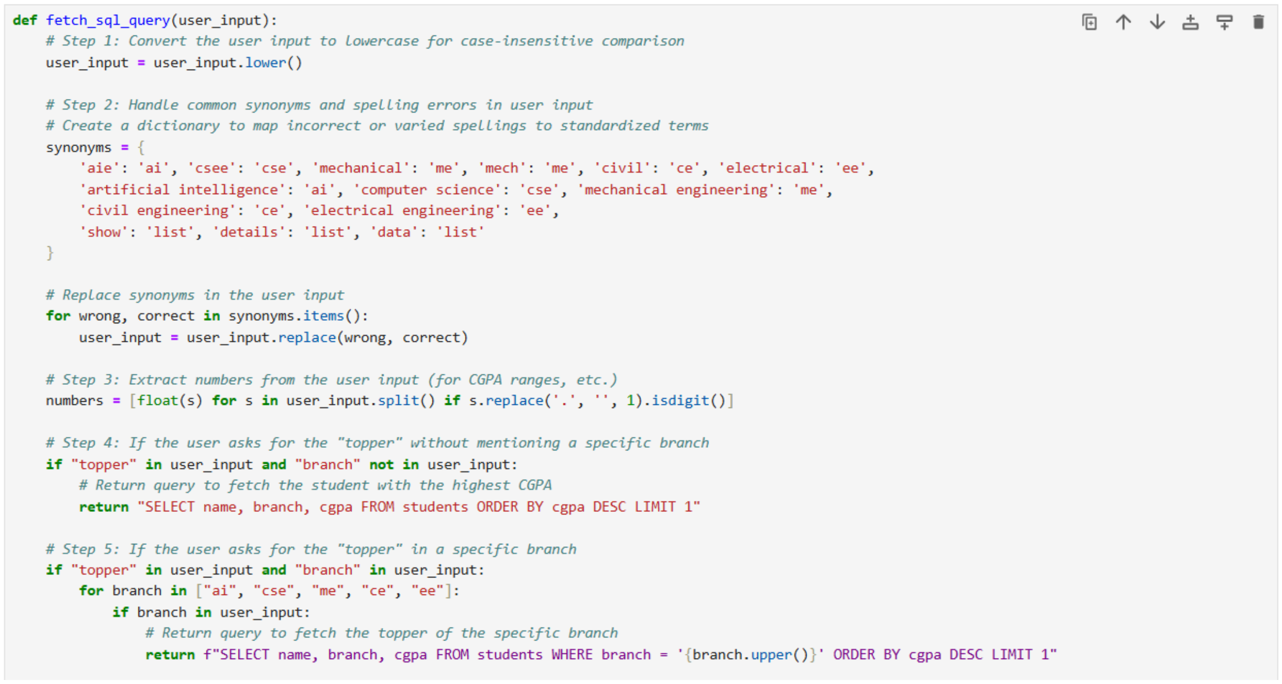
- Handles more advanced queries such as:
  - ✓ Toppers by branch
  - ✓ CGPA ranges
  - ✓ Count per branch
  - ✓ Sorting students
  - ✓ Searching by name
- Also uses synonym replacement for terms like:
  - ✓ "mechanical" → "me"
  - ✓ "computer science" → "cse"



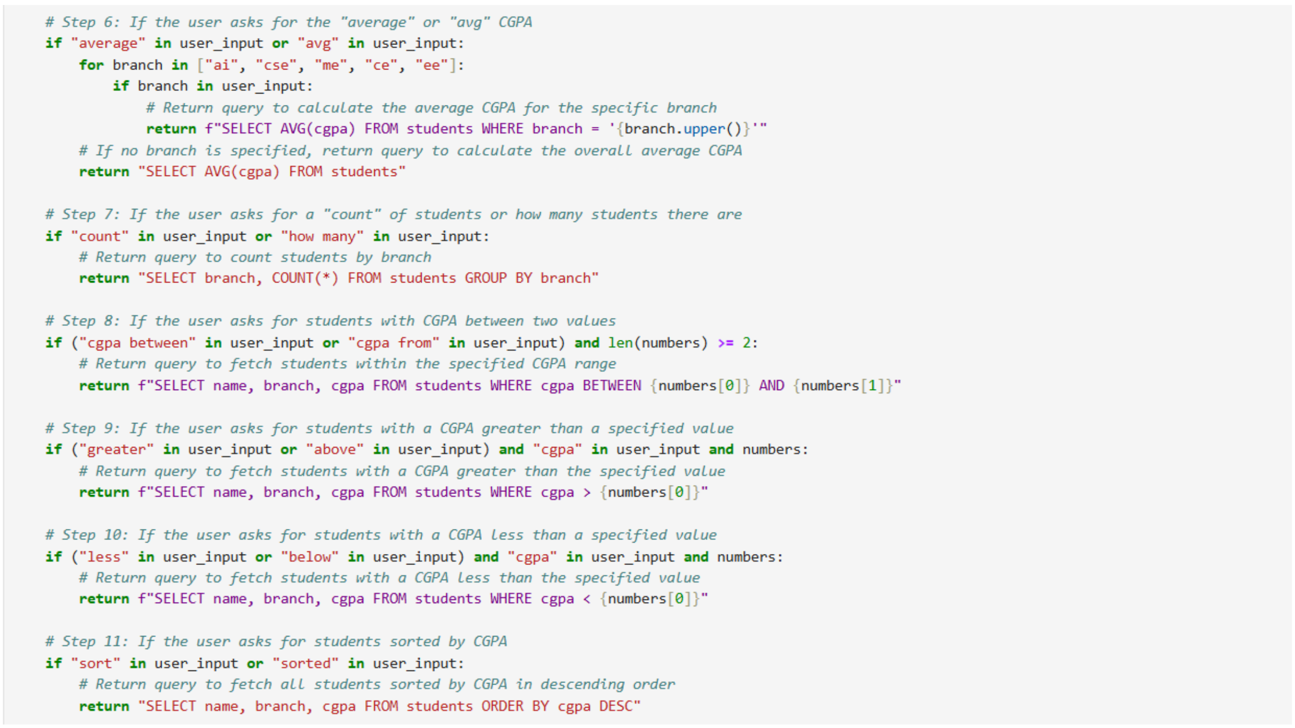
```python
def generate_sql(user_input):
    # Step 1: Convert the user input to lowercase for case-insensitive comparison
    user_input = user_input.lower()

    # Step 2: Check if the user wants to list all students
    # If the user asks for "all students" or "list students", we return the SQL to get all student records
    if "all students" in user_input or "list students" in user_input:
        return "SELECT * FROM students;"

    # Step 3: Check if the user is asking for students with a grade above a certain value
    # Look for the phrase "students above" and extract the number after it
    elif "students above" in user_input:
        # Split the input into individual words
        words = user_input.split()
        # Loop through the words to find the position of "above"
        for i, word in enumerate(words):
            if word == "above":
                try:
                    # Try to convert the next word into an integer (the grade threshold)
                    number = int(words[i+1])
                    # Return SQL to fetch students with a grade greater than the extracted number
                    return f"SELECT * FROM students WHERE grade > {number};"
                except:
                    # If there's an error (e.g., no number provided), return None
                    return None

    # Step 4: Check if the user wants the average grade of all students
    # If the user asks for the average grade, return the SQL to calculate the average grade
    elif "average grade" in user_input:
        return "SELECT AVG(grade) FROM students;"

    # Step 5: Check if the user wants the count of all students
    # If the user asks for the count of students, return the SQL to count them
    elif "count students" in user_input:
        return "SELECT COUNT(*) FROM students;"

    # Step 6: If none of the above conditions are met, return None
    else:
        return None  # If nothing matched, return None
```

5. Step3: Execute SQL and Fetch Result

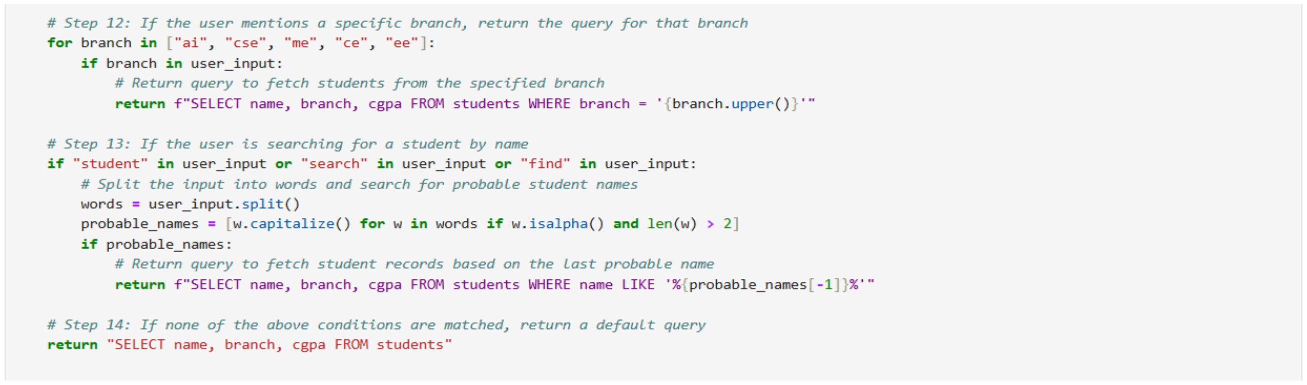
The core of this project lies in translating the user's natural language input into a valid SQL query. This is achieved through the fetch_sql_query() function.



```python
def fetch_sql_query(user_input):
    # Step 1: Convert the user input to lowercase for case-insensitive comparison
    user_input = user_input.lower()

    # Step 2: Handle common synonyms and spelling errors in user input
    # Create a dictionary to map incorrect or varied spellings to standardized terms
    synonyms = {
        'aie': 'ai', 'csee': 'cse', 'mechanical': 'me', 'mech': 'me', 'civil': 'ce', 'electrical': 'ee',
        'artificial intelligence': 'ai', 'computer science': 'cse', 'mechanical engineering': 'me',
        'civil engineering': 'ce', 'electrical engineering': 'ee',
        'show': 'list', 'details': 'list', 'data': 'list'
    }

    # Replace synonyms in the user input
    for wrong, correct in synonyms.items():
        user_input = user_input.replace(wrong, correct)

    # Step 3: Extract numbers from the user input (for CGPA ranges, etc.)
    numbers = [float(s) for s in user_input.split() if s.replace('.', '', 1).isdigit()]

    # Step 4: If the user asks for the "topper" without mentioning a specific branch
    if "topper" in user_input and "branch" not in user_input:
        # Return query to fetch the student with the highest CGPA
        return "SELECT name, branch, cgpa FROM students ORDER BY cgpa DESC LIMIT 1"

    # Step 5: If the user asks for the "topper" in a specific branch
    if "topper" in user_input and "branch" in user_input:
        for branch in ["ai", "cse", "me", "ce", "ee"]:
            if branch in user_input:
                # Return query to fetch the topper of the specific branch
                return f"SELECT name, branch, cgpa FROM students WHERE branch = '{branch.upper()}' ORDER BY cgpa DESC LIMIT 1"
```



```python
    # Step 6: If the user asks for the "average" or "avg" CGPA
    if "average" in user_input or "avg" in user_input:
        for branch in ["ai", "cse", "me", "ce", "ee"]:
            if branch in user_input:
                # Return query to calculate the average CGPA for the specific branch
                return f"SELECT AVG(cgpa) FROM students WHERE branch = '{branch.upper()}'"
        # If no branch is specified, return query to calculate the overall average CGPA
        return "SELECT AVG(cgpa) FROM students"

    # Step 7: If the user asks for a "count" of students or how many students there are
    if "count" in user_input or "how many" in user_input:
        # Return query to count students by branch
        return "SELECT branch, COUNT(*) FROM students GROUP BY branch"

    # Step 8: If the user asks for students with CGPA between two values
    if ("cgpa between" in user_input or "cgpa from" in user_input) and len(numbers) >= 2:
        # Return query to fetch students within the specified CGPA range
        return f"SELECT name, branch, cgpa FROM students WHERE cgpa BETWEEN {numbers[0]} AND {numbers[1]}"

    # Step 9: If the user asks for students with a CGPA greater than a specified value
    if ("greater" in user_input or "above" in user_input) and "cgpa" in user_input and numbers:
        # Return query to fetch students with a CGPA greater than the specified value
        return f"SELECT name, branch, cgpa FROM students WHERE cgpa > {numbers[0]}"

    # Step 10: If the user asks for students with a CGPA less than a specified value
    if ("less" in user_input or "below" in user_input) and "cgpa" in user_input and numbers:
        # Return query to fetch students with a CGPA less than the specified value
        return f"SELECT name, branch, cgpa FROM students WHERE cgpa < {numbers[0]}"

    # Step 11: If the user asks for students sorted by CGPA
    if "sort" in user_input or "sorted" in user_input:
        # Return query to fetch all students sorted by CGPA in descending order
        return "SELECT name, branch, cgpa FROM students ORDER BY cgpa DESC"
```



```python
    # Step 12: If the user mentions a specific branch, return the query for that branch
    for branch in ["ai", "cse", "me", "ce", "ee"]:
        if branch in user_input:
            # Return query to fetch students from the specified branch
            return f"SELECT name, branch, cgpa FROM students WHERE branch = '{branch.upper()}'"

    # Step 13: If the user is searching for a student by name
    if "student" in user_input or "search" in user_input or "find" in user_input:
        # Split the input into words and search for probable student names
        words = user_input.split()
        probable_names = [w.capitalize() for w in words if w.isalpha() and len(w) > 2]
        if probable_names:
            # Return query to fetch student records based on the last probable name
            return f"SELECT name, branch, cgpa FROM students WHERE name LIKE '%{probable_names[-1]}%'"

    # Step 14: If none of the above conditions are matched, return a default query
    return "SELECT name, branch, cgpa FROM students"
```

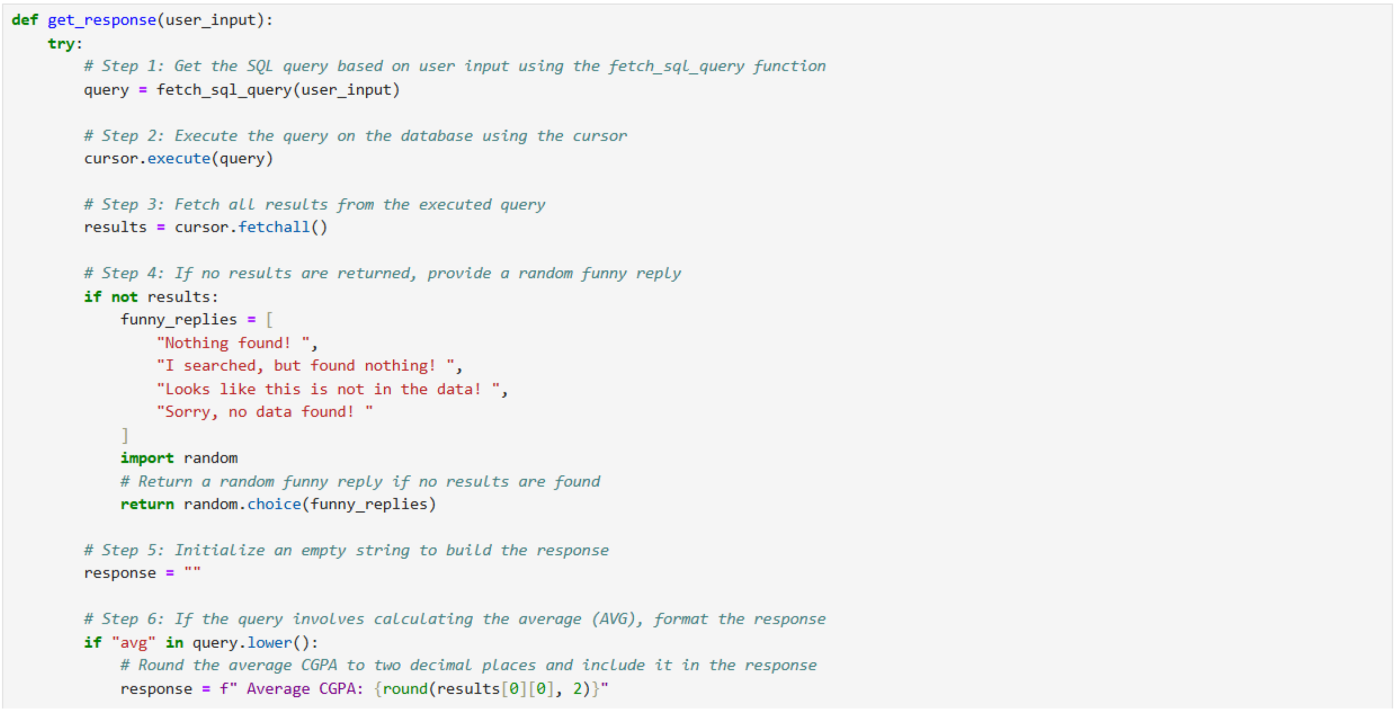
6. Step4: Response Generation Module

get_response() Function:

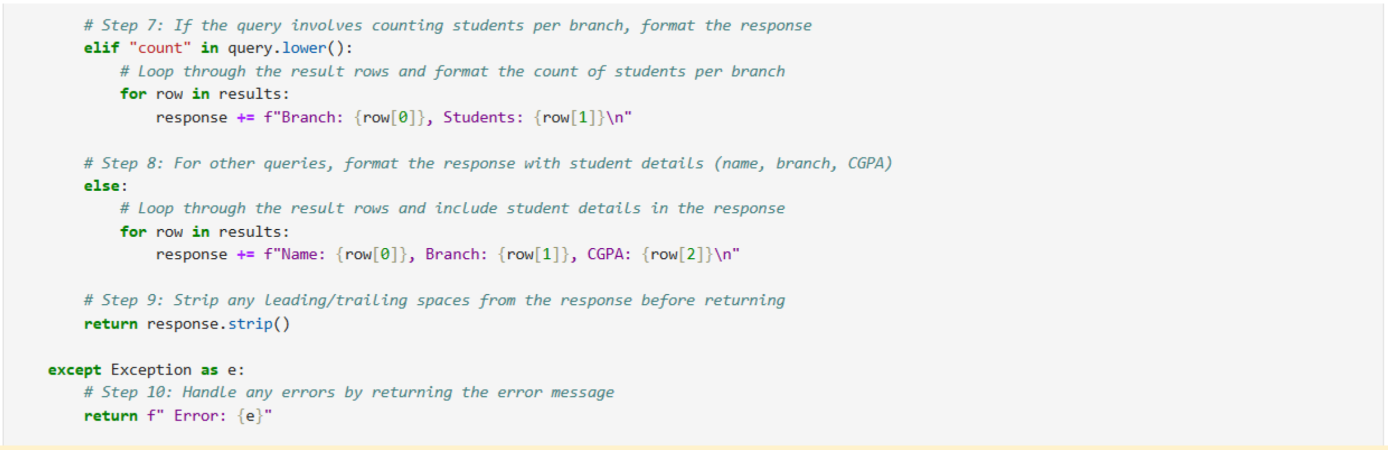
- Executes generated SQL queries using the SQLite database
- Parses results and returns them in readable format
- Includes humorous fallback messages when no data is found

fallback replies:

- "Nothing found!"
- "I searched, but found nothing!"
- "Sorry, no data found!"



```python
def get_response(user_input):
    try:
        # Step 1: Get the SQL query based on user input using the fetch_sql_query function
        query = fetch_sql_query(user_input)

        # Step 2: Execute the query on the database using the cursor
        cursor.execute(query)

        # Step 3: Fetch all results from the executed query
        results = cursor.fetchall()

        # Step 4: If no results are returned, provide a random funny reply
        if not results:
            funny_replies = [
                "Nothing found! ",
                "I searched, but found nothing! ",
                "Looks like this is not in the data! ",
                "Sorry, no data found! "
            ]
            import random
            # Return a random funny reply if no results are found
            return random.choice(funny_replies)

        # Step 5: Initialize an empty string to build the response
        response = ""

        # Step 6: If the query involves calculating the average (AVG), format the response
        if "avg" in query.lower():
            # Round the average CGPA to two decimal places and include it in the response
            response = f" Average CGPA: {round(results[0][0], 2)}"
```



```python
        # Step 7: If the query involves counting students per branch, format the response
        elif "count" in query.lower():
            # Loop through the result rows and format the count of students per branch
            for row in results:
                response += f"Branch: {row[0]}, Students: {row[1]}\n"

        # Step 8: For other queries, format the response with student details (name, branch, CGPA)
        else:
            # Loop through the result rows and include student details in the response
            for row in results:
                response += f"Name: {row[0]}, Branch: {row[1]}, CGPA: {row[2]}\n"

        # Step 9: Strip any leading/trailing spaces from the response before returning
        return response.strip()

    except Exception as e:
        # Step 10: Handle any errors by returning the error message
        return f" Error: {e}"
```

Sample Inputs & Outputs

Natural Query	SQL Generated	Output
Who is the topper?	SELECT name, branch, cgpa FROM students ORDER BY cgpa DESC LIMIT 1	Ishaan Gupta, EE, 9.7
List all students	SELECT * FROM students;	Displays all rows
Count students	SELECT branch, COUNT(*) FROM students GROUP BY branch	CE - 5, CSE - 3, ME - 3, EE - 2, AI - 1

Natural Query	SQL Generated	Output
Students with CGPA above 9	SELECT name, branch, cgpa FROM students WHERE cgpa > 9	Ayaan Sharma (CE), Ayaan Sharma (ME), Ishaan Gupta (EE)

7. Step5: chatbot_gui.py

This step implements the **Graphical User Interface (GUI)** for the chatbot application using **Tkinter**, allowing users to enter queries in natural language and view chatbot responses that fetch student data from an SQLite database.

1. Database Connection

```python
def connect_db():
    conn = sqlite3.connect('students.db')
    return conn
```

Establishes a connection to the SQLite database (students.db).

2. Query Generation from User Input

```python
def fetch_sql_query(user_input):
    user_input = user_input.lower()

    if "top 5" in user_input or "top five" in user_input:
        return "SELECT name, branch, cgpa FROM students ORDER BY cgpa DESC LIMIT 5"

    elif "topper" in user_input:
        return "SELECT name, branch, cgpa FROM students ORDER BY cgpa DESC LIMIT 1"

    elif "ai branch" in user_input:
        return "SELECT name, branch, cgpa FROM students WHERE branch = 'AI'"

    elif "cse branch" in user_input:
        return "SELECT name, branch, cgpa FROM students WHERE branch = 'CSE'"

    elif "me branch" in user_input:
        return "SELECT name, branch, cgpa FROM students WHERE branch = 'ME'"

    elif "ce branch" in user_input:
        return "SELECT name, branch, cgpa FROM students WHERE branch = 'CE'"

    elif "highest cgpa" in user_input or "top cgpa" in user_input:
        return "SELECT name, branch, cgpa FROM students ORDER BY cgpa DESC LIMIT 1"

    elif "lowest cgpa" in user_input or "bottom cgpa" in user_input:
        return "SELECT name, branch, cgpa FROM students ORDER BY cgpa ASC LIMIT 1"

    elif "cgpa more than" in user_input or "cgpa greater than" in user_input:
        numbers = [float(s) for s in user_input.split() if s.replace('.', '', 1).isdigit()]
        if numbers:
            return f"SELECT name, branch, cgpa FROM students WHERE cgpa > {numbers[0]}"

    elif "cgpa less than" in user_input or "cgpa below" in user_input:
        numbers = [float(s) for s in user_input.split() if s.replace('.', '', 1).isdigit()]
        if numbers:
            return f"SELECT name, branch, cgpa FROM students WHERE cgpa < {numbers[0]}"

    elif "cgpa equal to" in user_input or "cgpa is" in user_input:
        numbers = [float(s) for s in user_input.split() if s.replace('.', '', 1).isdigit()]
        if numbers:
            return f"SELECT name, branch, cgpa FROM students WHERE cgpa = {numbers[0]}"

    else:
        # Default case: fetch all students
        return "SELECT name, branch, cgpa FROM students"
```

- Parses natural language queries like:
  - ✓ "Topper"
  - ✓ "CGPA greater than 8"
  - ✓ "Students from AI branch"
- Maps them to respective SQL queries dynamically using basic string matching and filtering logic.

3. Executing SQL Queries

```python
def get_response(user_input):
    try:
        query = fetch_sql_query(user_input)
        conn = connect_db()
        cursor = conn.cursor()
        cursor.execute(query)
        rows = cursor.fetchall()

        response = ""
        for row in rows:
            response += f"Name: {row[0]}, Branch: {row[1]}, CGPA: {row[2]}\n"

        if not response:
            response = "No records found."

        conn.close()
        return response.strip()

    except Exception as e:
        return f"Error: {e}"
```

- ➢ Calls fetch_sql_query() to get SQL.
- ➢ Executes the SQL query using SQLite.
- ➢ Formats the results into a chatbot-friendly response string.

4. GUI Construction with Tkinter

A clean and functional GUI is created using the Tkinter library:

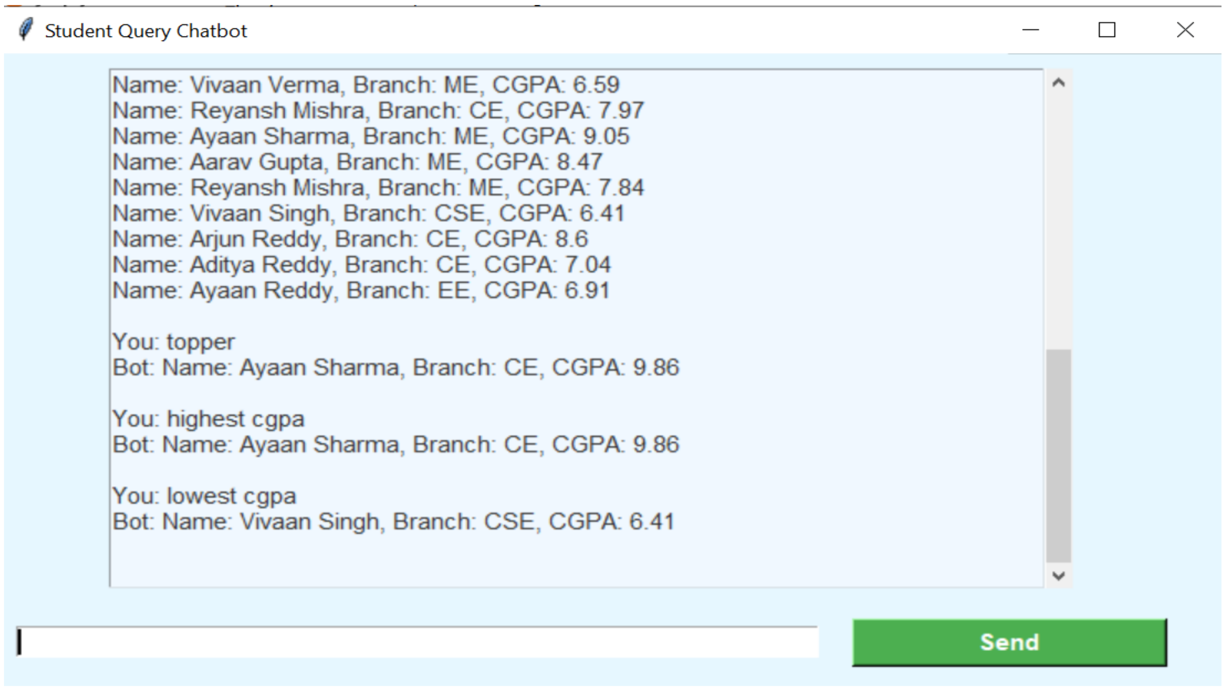
- ➢ ScrolledText widget for chat logs
- ➢ Entry widget for user input
- ➢ Button to trigger query execution
- ➢ bind('<Return>', ...) for pressing Enter to send

```python
root = tk.Tk()
root.title("Student Query Chatbot")
root.geometry("600x450")
root.configure(bg="#E6F7FF")

# Chat log where conversation will be shown
chatlog = scrolledtext.ScrolledText(
    root,
    wrap=tk.WORD,
    width=70,
    height=20,
    state=tk.DISABLED,
    bg="#F0F8FF",
    fg="#333333",
    font=("Arial", 11)
)
chatlog.grid(row=0, column=0, columnspan=2, padx=10, pady=10)

# Entry field for user input
entry = tk.Entry(root, width=60, font=("Arial", 11))
entry.grid(row=1, column=0, padx=10, pady=10)

# Send button to send the message
send_button = tk.Button(
    root,
    text="Send",
    width=20,
    command=send_message,
    bg="#4CAF50",
    fg="white",
    font=("Arial", 11, "bold")
)
send_button.grid(row=1, column=1, padx=10, pady=10)
```

```python
# Binding the Enter key to send message
root.bind('<Return>', send_message)

# Running the GUI Loop
root.mainloop()
```

GUI Overview:

- Title: **Student Query Chatbot**
- Dimensions: **600x450 pixels**
- Color Theme: Light blue background with white text area
- Components:
  - ✓ Scrollable conversation window
  - ✓ Input field for queries
  - ✓ “Send” button for submitting queries



```
Name: Vivaan Verma, Branch: ME, CGPA: 6.59
Name: Reyansh Mishra, Branch: CE, CGPA: 7.97
Name: Ayaan Sharma, Branch: ME, CGPA: 9.05
Name: Aarav Gupta, Branch: ME, CGPA: 8.47
Name: Reyansh Mishra, Branch: ME, CGPA: 7.84
Name: Vivaan Singh, Branch: CSE, CGPA: 6.41
Name: Arjun Reddy, Branch: CE, CGPA: 8.6
Name: Aditya Reddy, Branch: CE, CGPA: 7.04
Name: Ayaan Reddy, Branch: EE, CGPA: 6.91

You: topper
Bot: Name: Ayaan Sharma, Branch: CE, CGPA: 9.86

You: highest cgpa
Bot: Name: Ayaan Sharma, Branch: CE, CGPA: 9.86

You: lowest cgpa
Bot: Name: Vivaan Singh, Branch: CSE, CGPA: 6.41
```

8. Features

- **Dynamic Query Handling**: Understands a wide range of user inputs.
- **Synonym Normalization**: Handles spelling mistakes and varied terminology.
- **Humorous Bot Replies**: Returns funny messages when data is not found.
- **Branch-specific Analytics**: Supports queries like topper/average per branch.

9. Future Enhancements

- Use NLP Models like BERT or T5 for semantic understanding of queries.
- Web Interface using Flask or Streamlit.
- Speech to Text Integration for voice-based query input.
- User Authentication to personalize queries.

10. Conclusion

This project demonstrates a practical implementation of combining NLP with SQL using a local database. It is a minimal yet powerful example of how users can interact with structured data using natural language, and how chatbots can be trained to understand and execute such tasks.