User Manual for StudentVLE Data Processing and Analysis Notebook Using Google Colab

This document has all the steps you need to properly navigate the StudentVLE Google Colab Notebook. The steps are in chronological order, so if there arises the need to recreate the Notebook, following these steps in order will prove to be sufficient.

Table of Content

1. Mounting Google Drive

Purpose: To access files stored in Google Drive directly from the Colab environment. This is necessary if your database and data files are stored in Drive.

Code:

```
from google.colab import drive
drive.mount('/content/drive')
```

Usage:

- Run this cell at the beginning of your notebook.
- It will prompt you to authenticate your Google account by providing an authorization code. This step gives Colab permission to access your Drive files.
- Ensure that the target file (in this case, the SQLite database) is located in the specified path within your Google Drive.

Notes:

- If you move your database files to a different folder in Drive, update the path accordingly.

---

2. Connecting to the SQLite Database

Purpose: To connect to the SQLite database containing the student information and VLE data. This is the backbone for querying and managing data for educational analysis.

Code:

```
import sqlite3
conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE', check_same_thread=False)
cur = conn.cursor()
```

Usage:

- This block establishes a connection with the SQLite database located in Google Drive.
- After running this, you can use `cur` to execute SQL queries, and `conn` to commit any changes (if needed).

Key Parameters:

- **check_same_thread=False**: This allows the connection to be shared across multiple threads, which can be useful in web applications (though less relevant for a notebook).

Notes:

- Ensure that the path is correct and that the database file (`StudentVLE`) exists in the specified location.
- Always close the connection (`conn.close()`) once all database operations are complete to avoid leaving unnecessary connections open.

---

3. Importing Files

Purpose: To upload CSV files from your local machine to the Colab environment for use in database operations.

Code:

```
from google.colab import files
uploaded = files.upload()  # This will prompt you to upload your CSV file
```

Usage:

- Run this cell when you need to upload CSV files that will be inserted into your SQLite database.
- After executing this cell, a file upload dialog will appear, allowing you to select and upload your files. For this database context, the files are
    - courses.csv
    - vle.csv
    - studentInfo.csv
    - assessments.csv
    - studentRegistration.csv
    - studentAssessment.csv
    - studentVle.csv

Notes:

- Make sure the file names and formats match those expected in your database tables to avoid errors during data insertion.

---

4. Table Creation and Manipulation

Purpose: To create main and backup tables in the SQLite database, ensuring data integrity and allowing for potential recovery.

4.1 Drop Table If Exist

Code for main tables:

```
# Drop tables in reverse order of dependencies to avoid foreign key
constraint issues
'''
cur.execute('DROP TABLE IF EXISTS studentVle')
cur.execute('DROP TABLE IF EXISTS studentAssessment')
cur.execute('DROP TABLE IF EXISTS studentRegistration')
cur.execute('DROP TABLE IF EXISTS studentInfo')
cur.execute('DROP TABLE IF EXISTS assessments')
cur.execute('DROP TABLE IF EXISTS vle')
cur.execute('DROP TABLE IF EXISTS courses')

# Commit changes and close the connection
conn.commit()

print("Tables dropped successfully!")
'''
```

Code for backup tables:

```
# Drop tables in reverse order of dependencies to avoid foreign key
constraint issues
'''
cur.execute('DROP TABLE IF EXISTS studentVleBU')
cur.execute('DROP TABLE IF EXISTS studentAssessmentBU')
cur.execute('DROP TABLE IF EXISTS studentRegistrationBU')
cur.execute('DROP TABLE IF EXISTS studentInfoBU')
cur.execute('DROP TABLE IF EXISTS assessmentsBU')
cur.execute('DROP TABLE IF EXISTS vleBU')
cur.execute('DROP TABLE IF EXISTS coursesBU')

# Commit changes and close the connection
conn.commit()

print("Back Up Tables dropped successfully!")
'''
```

4.2 Creating Tables

Code for main tables:

```
# Enable foreign key constraints
cur.execute("PRAGMA foreign_keys = ON")

# Create courses table
cur.execute('''
   CREATE TABLE courses (
       code_module VARCHAR(45),
       code_presentation VARCHAR(45),
       module_presentation_length INT,
       PRIMARY KEY (code_module, code_presentation)
   )
''')

# Create studentInfo table
cur.execute('''
   CREATE TABLE studentInfo (
       code_module VARCHAR(45),
       code_presentation VARCHAR(45),
       id_student INT PRIMARY KEY,
       gender VARCHAR(3),
       imd_band VARCHAR(16),
       highest_education VARCHAR(45),
       age_band VARCHAR(16),
       num_of_prev_attempts INT,
       studied_credits INT,
       region VARCHAR(45),
       disability VARCHAR(3),
       final_result VARCHAR(45),
       FOREIGN KEY (code_module, code_presentation) REFERENCES
courses(code_module, code_presentation)
   )
''')

# Create studentRegistration table
cur.execute('''
   CREATE TABLE studentRegistration (
       code_module VARCHAR(45),
       code_presentation VARCHAR(45),
       id_student INT,
```

```
        date_registration INT,
        date_unregistration INT,
        FOREIGN KEY (id_student) REFERENCES studentInfo(id_student),
        FOREIGN KEY (code_module, code_presentation) REFERENCES
courses(code_module, code_presentation)
    )
''')


# Create assessments table
cur.execute('''
    CREATE TABLE assessments(
        code_module VARCHAR(45),
        code_presentation VARCHAR(45),
        id_assessment INT PRIMARY KEY,
        assessment_type VARCHAR(45),
        date INT,
        weight INT,
        FOREIGN KEY (code_module, code_presentation) REFERENCES
courses(code_module, code_presentation)
    )
''')

# Create studentAssessment table
cur.execute('''
    CREATE TABLE studentAssessment (
        id_student INT,
        id_assessment INT,
        date_submitted INT,
        is_banked TINYINT,
        score FLOAT,
        FOREIGN KEY (id_student) REFERENCES studentInfo(id_student),
        FOREIGN KEY (id_assessment) REFERENCES
assessments(id_assessment)
    )
''')

# Create studentVle table
cur.execute('''
    CREATE TABLE studentVle (
        id_site INT,
```

```
        id_student INT,
        code_module VARCHAR(45),
        code_presentation VARCHAR(45),
        date INT,
        sum_click INT,
        FOREIGN KEY (id_site) REFERENCES vle(id_site),
        FOREIGN KEY (id_student) REFERENCES studentInfo(id_student),
        FOREIGN KEY (code_module, code_presentation) REFERENCES
courses(code_module, code_presentation)
    )
''')

# Create vle table
cur.execute('''
    CREATE TABLE vle (
        id_site INT PRIMARY KEY,
        code_module VARCHAR(45),
        code_presentation VARCHAR(45),
        activity_type VARCHAR(45),
        week_from INT,
        week_to INT,
        FOREIGN KEY (code_module, code_presentation) REFERENCES
courses(code_module, code_presentation)
    )
''')

# Commit changes and close the connection
conn.commit()

print("Tables created successfully!")
```

Code for backup tables:

```
# Enable foreign key constraints
cur.execute("PRAGMA foreign_keys = ON")

# Create courses table
cur.execute('''
    CREATE TABLE coursesBU (
        code_module VARCHAR(45),
        code_presentation VARCHAR(45),
        module_presentation_length INT,
        PRIMARY KEY (code_module, code_presentation)
```

```
    )
''')

# Create studentInfo table
cur.execute('''
    CREATE TABLE studentInfoBU (
        code_module VARCHAR(45),
        code_presentation VARCHAR(45),
        id_student INT PRIMARY KEY,
        gender VARCHAR(3),
        imd_band VARCHAR(16),
        highest_education VARCHAR(45),
        age_band VARCHAR(16),
        num_of_prev_attempts INT,
        studied_credits INT,
        region VARCHAR(45),
        disability VARCHAR(3),
        final_result VARCHAR(45),
        FOREIGN KEY (code_module, code_presentation) REFERENCES
coursesBU(code_module, code_presentation)
    )
''')

# Create studentRegistration table
cur.execute('''
    CREATE TABLE studentRegistrationBU (
        code_module VARCHAR(45),
        code_presentation VARCHAR(45),
        id_student INT,
        date_registration INT,
        date_unregistration INT,
        FOREIGN KEY (id_student) REFERENCES
studentInfoBU(id_student),
        FOREIGN KEY (code_module, code_presentation) REFERENCES
coursesBU(code_module, code_presentation)
    )
''')


# Create assessments table
cur.execute('''
```

```
   CREATE TABLE assessmentsBU (
       code_module VARCHAR(45),
       code_presentation VARCHAR(45),
       id_assessment INT PRIMARY KEY,
       assessment_type VARCHAR(45),
       date INT,
       weight INT,
       FOREIGN KEY (code_module, code_presentation) REFERENCES
coursesBU(code_module, code_presentation)
   )
''')

# Create studentAssessment table
cur.execute('''
   CREATE TABLE studentAssessmentBU (
       id_student INT,
       id_assessment INT,
       date_submitted INT,
       is_banked TINYINT,
       score FLOAT,
       FOREIGN KEY (id_student) REFERENCES
studentInfoBU(id_student),
       FOREIGN KEY (id_assessment) REFERENCES
assessmentsBU(id_assessment)
   )
''')

# Create studentVle table
cur.execute('''
   CREATE TABLE studentVleBU (
       id_site INT,
       id_student INT,
       code_module VARCHAR(45),
       code_presentation VARCHAR(45),
       date INT,
       sum_click INT,
       FOREIGN KEY (id_site) REFERENCES vleBU(id_site),
       FOREIGN KEY (id_student) REFERENCES
studentInfoBU(id_student),
       FOREIGN KEY (code_module, code_presentation) REFERENCES
coursesBU(code_module, code_presentation)
```

```
   )
''')

# Create vle table
cur.execute('''
   CREATE TABLE vleBU (
       id_site INT PRIMARY KEY,
       code_module VARCHAR(45),
       code_presentation VARCHAR(45),
       activity_type VARCHAR(45),
       week_from INT,
       week_to INT,
       FOREIGN KEY (code_module, code_presentation) REFERENCES
coursesBU(code_module, code_presentation)
   )
''')

# Commit changes and close the connection
conn.commit()

print(" Back Up Tables created successfully!")
```

4.3 Resetting Tables

Code:

```
# ONLY FOR RESET
# List of live tables (the ones you are using for testing)
live_tables = ['studentVLE', 'studentAssessment',
'studentRegistration', 'studentInfo', 'assessments', 'vle',
'courses']

# List of backup tables (the ones used for resetting)
backup_tables = ['studentVleBU', 'studentAssessmentBU',
'studentRegistrationBU', 'studentInfoBU', 'assessmentsBU', 'vleBU',
'coursesBU']

# Step 1: Duplicate each live table using its backup
for i in range(len(live_tables)):
    live_table = live_tables[i]
    backup_table = backup_tables[i]
```

```
    # Step 2: Create a duplicate of the backup table with a new name
    cur.execute(f"CREATE TABLE {live_table}_duplicate AS SELECT *
FROM {backup_table};")
    print(f"Duplicate of '{backup_table}' created as
'{live_table}_duplicate'.")

    # Step 3: Drop the live table if it exists
    cur.execute(f"DROP TABLE IF EXISTS {live_table}")
    print(f"Live table '{live_table}' dropped.")

    # Step 4: Rename the duplicate to the original live table name
    cur.execute(f"ALTER TABLE {live_table}_duplicate RENAME TO
{live_table};")
    print(f"Duplicate renamed to '{live_table}'.")

# Commit changes
conn.commit()

print("All tables have been duplicated and renamed successfully.")
```

Usage:

- This section is meant for resetting and backing up your tables. It creates backups of existing tables, drops them, and renames duplicates.
- It should only be run if absolutely necessary, as it alters the database structure.

Notes:

- The code comments highlight that it is essential to understand the implications of running this cell, especially regarding data integrity.
- Ensure that you have a proper backup of your data before executing this section.

---

5. Inserting CSV Files into Tables

Purpose: To insert data from CSV files into the corresponding tables in the SQLite database.

Code:

```
import pandas as pd
```

```
# Enable foreign key constraints
cur.execute("PRAGMA foreign_keys = ON")

def insert_data_from_csv(table_name, file_path, connection):
    df = pd.read_csv(file_path)
    print(f"Columns in {table_name} CSV:", df.columns.tolist())

    # Remove duplicates based on primary key if applicable
    if 'id_student' in df.columns:
        df.drop_duplicates(subset=['id_student'], inplace=True)

    try:
        df.to_sql(table_name, connection, if_exists='append',
index=False)
        print(f"Data inserted into {table_name} successfully.")
    except sqlite3.IntegrityError as e:
        print(f"IntegrityError while inserting into {table_name}:
{e}")

# Insert CSV data in the correct order of dependencies
insert_data_from_csv('courses', '/content/courses.csv', conn)
insert_data_from_csv('vle', '/content/vle.csv', conn)
insert_data_from_csv('studentInfo', '/content/studentInfo.csv',
conn)
insert_data_from_csv('assessments', '/content/assessments.csv',
conn)
insert_data_from_csv('studentRegistration',
'/content/studentRegistration.csv', conn)
insert_data_from_csv('studentAssessment',
'/content/studentAssessment.csv', conn)
insert_data_from_csv('studentVle', '/content/studentVle.csv', conn)
insert_data_from_csv('coursesBU', '/content/courses.csv', conn)
insert_data_from_csv('vleBU', '/content/vle.csv', conn)
insert_data_from_csv('studentInfoBU', '/content/studentInfo.csv',
conn)
insert_data_from_csv('assessmentsBU', '/content/assessments.csv',
conn)
insert_data_from_csv('studentRegistrationBU',
'/content/studentRegistration.csv', conn)
insert_data_from_csv('studentAssessmentBU',
'/content/studentAssessment.csv', conn)
```

```
insert_data_from_csv('studentVleBU', '/content/studentVle.csv',
conn)

# Commit changes and close the connection
conn.commit()

print("CSV data inserted successfully!")
```

Usage:

- This block is responsible for reading CSV files and inserting their contents into the appropriate tables in the SQLite database.
- It also ensures that duplicate entries are removed based on the primary key (if applicable).

Notes:

- The order of CSV insertion is crucial to maintain foreign key constraints. Make sure the files are correctly formatted and in the right sequence for insertion.
- After data insertion, always commit the changes to ensure that they are saved in the database.

---

6. Testing Stage Codes

Purpose: To trial and error logic before committing to utilising it. It's existence is solely for reference. Feel free to explore the codes but it is not required of you to run it.

6.1 Getting Tables from the Database

Purpose: To retrieve a list of all tables present in the SQLite database. This helps in understanding the database structure and available data.

Code:

```
cur.execute("SELECT name FROM sqlite_master WHERE type='table'")
tables = cur.fetchall()
print("Tables in the database:")
for table in tables:
    print(table[0])

conn.commit()
```

Usage:
This query fetches the names of all tables in the SQLite database and prints them. Useful for exploratory purposes.

Key Details:

- `sqlite_master`: A system table in SQLite that stores metadata about tables and indexes.
- Use this command to ensure you're familiar with the database schema before executing specific queries.

Notes:
Remember to commit the connection if necessary, and always ensure to close the connection when you're done.

6.2 Create Student with Dropdown Form

Purpose: To create a form with input fields, allowing educators to insert student information into the database using data from existing tables.

Code:

```
import ipywidgets as widgets
from IPython.display import display

# Function to get available options from courses table
def get_course_options():
    cur.execute("SELECT DISTINCT code_module, code_presentation FROM
courses")
    results = cur.fetchall()

    # Convert list of tuples into individual lists for dropdowns
    modules = sorted(set([row[0] for row in results if row[0] is not
None]))
    presentations = sorted(set([row[1] for row in results if row[1]
is not None]))
    return modules, presentations

# Function to get unique values from database for various fields,
excluding None values
def get_unique_options(field_name):
    cur.execute(f"SELECT DISTINCT {field_name} FROM studentInfo")
    results = cur.fetchall()
```

```
    # Filter out None values before sorting
    return sorted([row[0] for row in results if row[0] is not None])

# Get the module and presentation options from the database
module_options, presentation_options = get_course_options()

# Get options for dropdown fields from the database
highest_education_options = get_unique_options('highest_education')
imd_band_options = get_unique_options('imd_band')
region_options = get_unique_options('region')

# Define a custom layout for the input fields
layout = widgets.Layout(width='400px')

# Define custom style to expand the label width
style = {'description_width': '150px'}

# Creating input widgets for form
module_dropdown = widgets.Dropdown(options=module_options,
value=module_options[0], description='Module:', layout=layout,
style=style)
presentation_dropdown =
widgets.Dropdown(options=presentation_options,
value=presentation_options[0], description='Presentation:',
layout=layout, style=style)
id_student_input = widgets.IntText(description="Student ID:",
layout=layout, style=style)
gender_input = widgets.Dropdown(options=['M', 'F', 'Other'],
description='Gender:', layout=layout, style=style)
imd_band_input = widgets.Dropdown(options=imd_band_options,
description="IMD Band:", layout=layout, style=style)
highest_education_input =
widgets.Dropdown(options=highest_education_options,
description="Highest Education:", layout=layout, style=style)
age_band_input = widgets.Dropdown(options=['0-35', '35-55', '55<='],
description="Age Band:", layout=layout, style=style)
num_of_prev_attempts_input = widgets.IntText(description="Previous
Attempts:", layout=layout, style=style)
```

```
studied_credits_input =
widgets.Dropdown(options=studied_credits_options,
description="Studied Credits:", layout=layout, style=style)
region_input = widgets.Dropdown(options=region_options,
description="Region:", layout=layout, style=style)
disability_input = widgets.Dropdown(options=['Y', 'N'],
description='Disability (Yes/Y or No/N):', layout=layout,
style=style)
final_result_input = widgets.Dropdown(options=['Distinction',
'Pass', 'Fail', 'Withdrawn'], description="Final Result:",
layout=layout, style=style)
```

Usage:
This widget-based form collects student information and allows educators to easily input the data into the SQLite database. The dropdown values are dynamically populated from the database.

Key Features:

- **Widgets**: Different input widgets are used (e.g., Dropdown, IntText) to collect data.
- **Database Integration**: Options for fields such as `code_module`, `highest_education`, and `region` are populated dynamically by querying the database.

Notes:
Once all fields are filled, the data can be inserted into the database using SQL `INSERT` queries, ensuring the information is stored correctly.

6.3 Retrieving Final Results by Module as Visualisation

Purpose: To visualize the percentage breakdown of final results (e.g., Distinction, Pass, Fail, Withdrawn) for each module using a bar chart.

Code:

```
import matplotlib.pyplot as plt
import pandas as pd

# Function to fetch and visualize percentages per code_module
def visualize_final_results_per_module():
    # Step 1: Fetch the counts of final results per code_module
    cur.execute('''
```

```
        SELECT code_module, final_result, COUNT(*) AS count
        FROM studentInfo
        GROUP BY code_module, final_result
    ''')
    rows = cur.fetchall()

    # Step 2: Process the data to calculate the percentages per
code_module
    data = pd.DataFrame(rows, columns=['code_module',
'final_result', 'count'])

    # Step 3: Pivot the table to have final_result as columns and
count as values
    result_counts = data.pivot(index='code_module',
columns='final_result', values='count').fillna(0)

    # Step 4: Calculate the percentage of each final result per
code_module
    result_percentages =
result_counts.div(result_counts.sum(axis=1), axis=0) * 100

    # Step 5: Plot the bar chart
    result_percentages.plot(kind='bar', stacked=True, figsize=(10,
6))

    # Step 6: Customize the chart
    plt.title('Percentage of Final Results per Code Module')
    plt.xlabel('Code Module')
    plt.ylabel('Percentage (%)')
    plt.xticks(rotation=45, ha='right')
    plt.legend(title='Final Result')

    # Step 7: Display the chart
    plt.tight_layout()
    plt.show()

# Call the function to visualize the data
visualize_final_results_per_module()
```

Usage:
Run this function to visualize the percentage distribution of final results for each module. Useful for understanding the performance trends in different modules.

Key Steps:

- The SQL query groups data by `code_module` and `final_result` to count the number of students in each category.
- A pivot table is created to organize the data, followed by calculating percentages.
- A stacked bar chart is plotted using Matplotlib to represent the results.

Notes:
Adjust the chart's size or style as needed for better visualization, especially if the dataset or result categories grow.

6.4 Deleting a Student Record

Purpose: To delete a specific student's record from the `studentInfo` table based on their unique `id_student`.

Code:

```
import ipywidgets as widgets
from IPython.display import display

# Input widget to take Student ID for deletion
id_student_input = widgets.IntText(description="Student ID:",
layout=widgets.Layout(width='400px'))

# Function to delete student from the database
def delete_student(button):
    # Get value from widget
    id_student = id_student_input.value

    # Delete the student from the studentInfo table
    try:
        cur.execute('''
            DELETE FROM studentInfo WHERE id_student = ?
        ''', (id_student,))
        conn.commit()
        print(f"Student {id_student} deleted successfully.")
    except sqlite3.IntegrityError as e:
        print(f"Error deleting student {id_student}: {e}")
```

```
# Button to trigger the deletion
submit_button = widgets.Button(description="Delete Student")
submit_button.on_click(delete_student)

# Display input and button widgets
display(id_student_input, submit_button)
```

Usage:
Enter the student's ID into the widget field, then click the "Delete Student" button to remove their record from the database.

Key Details:

- **Student ID**: The unique identifier `id_student` is used to delete the record.
- **SQL DELETE Query**: The query deletes the record from the `studentInfo` table where the ID matches the input.

Notes:
Handle this operation carefully to avoid accidental deletions.

6.5 Identifying Students' Risk Status

Purpose: To identify students who are at risk of dropping out, at risk of failing, or not at risk based on their assessment scores and interaction with the VLE.

Code:

```
import pandas as pd

def identify_students_risk_status():
    query = '''
    SELECT studentInfo.id_student, studentInfo.final_result,
MAX(studentVle.date) AS last_interaction,
           AVG(studentAssessment.score) AS avg_score,
SUM(studentVle.sum_click) AS total_clicks,
           CASE
               WHEN studentInfo.final_result = 'Fail' AND
AVG(studentAssessment.score) < 30 THEN 'At Risk of Dropping Out'
               WHEN studentInfo.final_result = 'Pass' AND
AVG(studentAssessment.score) < 50 THEN 'At Risk of Failing'
```

```
                WHEN studentInfo.final_result = 'Pass' AND
AVG(studentAssessment.score) >= 50 THEN 'Not at Risk'
                ELSE 'Needs Further Review'
            END AS risk_status
    FROM studentInfo
    JOIN studentVle ON studentInfo.id_student =
studentVle.id_student
    JOIN studentAssessment ON studentInfo.id_student =
studentAssessment.id_student
    GROUP BY studentInfo.id_student, studentInfo.final_result;
    '''
    df = pd.read_sql_query(query, conn)

    # Filter students based on risk status
    students_filtered = df[df['risk_status'].isin(['At Risk of
Failing', 'At Risk of Dropping Out', 'Not at Risk'])]

    # Display results
    print("At-Risk and Not-At-Risk Students:")
    print(students_filtered[['id_student', 'final_result',
'last_interaction', 'avg_score', 'total_clicks', 'risk_status']])

    return students_filtered

# Example usage
identify_students_risk_status()
```

Usage:
This function retrieves students' final results, their last interaction date with the VLE, their average assessment score, and total clicks. Based on these factors, it categorizes students into 'At Risk of Dropping Out', 'At Risk of Failing', 'Not at Risk', or 'Needs Further Review'. The output is a filtered DataFrame showing students' IDs and their risk status.

Key Details:

- **Risk Categories:** Students are categorized based on their final results and average scores.
- **SQL Query:** Joins data from `studentInfo`, `studentVle`, and `studentAssessment` tables to get a complete view of each student's performance and engagement.

Notes:
Adjust the thresholds in the `CASE` statement based on your institution's risk criteria.

6.6 Identifying Students' Risk Status

Purpose: To assess the risk status of students based on their assessment scores, interactions with the virtual learning environment (VLE), and final course outcomes. Students are categorized as "At Risk of Dropping Out", "At Risk of Failing", or "Not at Risk".

Code:

```
import pandas as pd

def identify_students_risk_status():

    query = '''

    SELECT studentInfo.id_student, studentInfo.final_result,
MAX(studentVle.date) AS last_interaction,

           AVG(studentAssessment.score) AS avg_score,
SUM(studentVle.sum_click) AS total_clicks,

           CASE

               WHEN studentInfo.final_result = 'Fail' AND
AVG(studentAssessment.score) < 30 THEN 'At Risk of Dropping Out'

               WHEN studentInfo.final_result = 'Pass' AND
AVG(studentAssessment.score) < 50 THEN 'At Risk of Failing'

               WHEN studentInfo.final_result = 'Pass' AND
AVG(studentAssessment.score) >= 50 THEN 'Not at Risk'

               ELSE 'Needs Further Review'

           END AS risk_status

    FROM studentInfo

    JOIN studentVle ON studentInfo.id_student =
studentVle.id_student

    JOIN studentAssessment ON studentInfo.id_student =
studentAssessment.id_student

    GROUP BY studentInfo.id_student, studentInfo.final_result;
```

```
    '''

    df = pd.read_sql_query(query, conn)


    # Filter students by risk status

    students_filtered = df[df['risk_status'].isin(['At Risk of
Failing', 'At Risk of Dropping Out', 'Not at Risk'])]


    # Display grouped results

    print("At-Risk and Not-At-Risk Students:")

    print(students_filtered[['id_student', 'final_result',
'last_interaction', 'avg_score', 'total_clicks', 'risk_status']])


    return students_filtered


# Example usage

identify_students_risk_status()
```

Usage:
This function retrieves information about students' final results, their last interaction date with the VLE, average assessment score, and total clicks on the VLE. Based on these, the students are categorized into one of the following risk statuses:

- **At Risk of Dropping Out**: Students with a failing grade and an average score below 30.
- **At Risk of Failing**: Students with a passing grade but an average score below 50.
- **Not at Risk**: Students with a passing grade and an average score above 50.

Key Details:

- **SQL Query:** Joins three tables (`studentInfo`, `studentVle`, and `studentAssessment`) to obtain a complete view of students' performance and engagement.

- **Risk Status Calculation:** The `CASE` statement in the SQL query defines the risk thresholds based on assessment scores and final results.
- **Data Filtering:** Only students who fall under the defined risk categories ('At Risk of Failing', 'At Risk of Dropping Out', 'Not at Risk') are displayed.

Notes:
Adjust the thresholds in the `CASE` statement depending on the institution's specific criteria for identifying students at risk. Also, ensure that the database connection (`conn`) is properly established before running the function.

6.7 Identifying Difficult Materials

Purpose: To identify the top 10 most difficult materials for each module based on students' average assessment scores.

Code:

```
import pandas as pd

def identify_difficult_materials():
    query = '''
    WITH ranked_materials AS (
        SELECT vle.id_site, vle.code_module,
AVG(studentAssessment.score) AS avg_score,
               ROW_NUMBER() OVER (PARTITION BY vle.code_module ORDER
BY AVG(studentAssessment.score)) AS rank
        FROM vle
        JOIN studentVle ON vle.id_site = studentVle.id_site
        JOIN studentAssessment ON studentVle.id_student =
studentAssessment.id_student
        GROUP BY vle.id_site, vle.code_module
    )
    SELECT id_site, code_module, avg_score
    FROM ranked_materials
    WHERE rank <= 10
    ORDER BY code_module, avg_score;
    '''

    df = pd.read_sql_query(query, conn)
    print("Top 10 Most Difficult Materials per Code Module:")
    print(df[['id_site', 'code_module', 'avg_score']])
```

```
# Example usage
identify_difficult_materials()
```

Usage:
This function identifies the top 10 most difficult learning materials per module based on students' average scores. The results are presented by ranking each material's difficulty within the module.

Key Details:

- **Ranked Results:** Materials are ranked based on the average scores, with the lowest scores indicating more difficult materials.
- **SQL Query:** Uses a window function (`ROW_NUMBER()`) to rank materials within each module based on their average score.

Notes:
You can adjust the `rank <= 10` condition if you want to retrieve more or fewer materials per module.

---

7. Collating Functions

The following functions facilitate interactions with the SQLite database to manage student data. These functions include creating, reading, updating, and deleting (CRUD) student records, along with advanced insights.

Code:
```
%%writefile /content/functions8.py

import sqlite3
import pandas as pd
```

Usage:

- This function creates a new file called functions8.py which will be referenced in the flask app code in 8. Flask App Setup

---

7.1 Database Setup

Purpose: Establish a connection to the SQLite database.

Code:
```
def setup_database():
```

```
    conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
    cur = conn.cursor()
    conn.commit()
```

7.2 CRUD Operations (Create, Read, Update, Delete)

Purpose:

These functions handle basic database operations such as inserting, reading, updating, and deleting student data. This helps in managing the student records efficiently.

7.2.1 Insert Student Data

Purpose: Insert a new student record into the database.

Code:

```
def insert_student_to_db(student_data):
    try:
        conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
        cur = conn.cursor()
        cur.execute('''
            INSERT INTO studentInfo (code_module, code_presentation,
id_student, gender, imd_band, highest_education, age_band,
num_of_prev_attempts, studied_credits, region, disability,
final_result)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', student_data)
        conn.commit()
        conn.close()
        return "Student inserted successfully!"
    except sqlite3.IntegrityError as e:
        return f"Error inserting student: {e}"
```

Usage:

- This function inserts a new student record into the `studentInfo` table.
- Call it with the `id_student`, `name`, and `age` parameters.

Notes:

- Ensure the `id_student` is unique to avoid primary key conflicts.

- Always call `conn.commit()` after executing `INSERT`, `UPDATE`, or `DELETE` statements to save the changes to the database.

7.2.2 Read Student Data

Purpose: Retrieve a student's data based on their ID.

Code:

```
def read_student_from_db(id_student):
    conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE', check_same_thread=False)
    cur = conn.cursor()
    cur.execute("SELECT * FROM studentInfo WHERE id_student = ?",
(id_student,))
    student_data = cur.fetchone()
    conn.close()
    return student_data
```

Usage:

- Retrieves a single student's data based on their `id_student`.
- Returns all the columns from the `studentInfo` table for that particular student.

Notes:

- This function returns `None` if no record matches the given `id_student`. Always handle this in your calling code to avoid errors.

7.2.3 Update Student Data

Purpose: Update a student's information based on their ID.

Code:

```
def update_student_in_db(id_student, updated_data):
    try:
        conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
        cur = conn.cursor()
        cur.execute('''
            UPDATE studentInfo
            SET code_module = ?, code_presentation = ?, gender = ?,
imd_band = ?, highest_education = ?, age_band = ?,
```

```
                num_of_prev_attempts = ?, studied_credits = ?,
region = ?, disability = ?, final_result = ?
            WHERE id_student = ?
        ''', (*updated_data, id_student))
        conn.commit()
        conn.close()
        return f"Student {id_student} updated successfully!"
    except sqlite3.IntegrityError as e:
        return f"Error updating student {id_student}: {e}"
```

Usage:

- Updates an existing student's `name` and/or `age` in the `studentInfo` table.
- Only updates the fields that are not `None`.

Notes:

- Ensure that the `id_student` exists before attempting an update.
- You can pass either `name`, `age`, or both to this function.

7.2.4 Delete Student Data

Purpose: Remove a student's record from the database using their ID.

Code:

```
def delete_student_from_db(id_student):
    try:
        conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
        cur = conn.cursor()
        cur.execute('''
            DELETE FROM studentInfo WHERE id_student = ?
        ''', (id_student,))
        conn.commit()
        conn.close()
        return f"Student {id_student} deleted successfully!"
    except sqlite3.IntegrityError as e:
        return f"Error deleting student {id_student}: {e}"
```

Usage:

- Deletes the student record corresponding to `id_student` from the `studentInfo` table.

Notes:

- This operation is permanent. Use with caution, especially when working with live data.
- Always confirm that the student exists in the database before deleting.

7.3 Advanced; Summarizing Module Performance

Purpose:

This function summarizes how students are performing in each module, based on their engagement with the Virtual Learning Environment (VLE) and their assessment scores.

Code:

```
# Function to summarize module performance (ADVANCED)
def summarize_modules():
   query = '''
   SELECT studentInfo.id_student, studentInfo.code_module,
          SUM(studentVle.sum_click) AS total_clicks,
          AVG(studentAssessment.score) AS avg_score,
          COUNT(CASE WHEN studentInfo.final_result = 'Pass' THEN 1
END) AS total_pass,
          COUNT(CASE WHEN studentInfo.final_result = 'Distinction'
THEN 1 END) AS total_distinction,
          COUNT(CASE WHEN studentInfo.final_result = 'Fail' THEN 1
END) AS total_fail,
          COUNT(CASE WHEN studentInfo.final_result = 'Withdrawn'
THEN 1 END) AS total_withdrawn
   FROM studentInfo
   JOIN studentVle ON studentInfo.id_student = studentVle.id_student
   JOIN studentAssessment ON studentInfo.id_student =
studentAssessment.id_student
   GROUP BY studentInfo.code_module;
   '''
   conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
   df = pd.read_sql_query(query, conn)
   conn.close()
   return df[['code_module', 'avg_score', 'total_clicks',
'total_pass', 'total_distinction', 'total_fail', 'total_withdrawn']]
```

Usage:

- This query calculates:
  - Total clicks per module (`total_clicks`): Indicator of student engagement.
  - Average score per module (`avg_score`): Reflects overall student performance.
  - Conditional counts to see how many students passed, achieved distinctions, failed, or withdrew.

Notes:

- Ensure the tables `studentInfo`, `studentVle`, and `studentAssessment` are correctly populated before running this summary query.

7.4 Advanced; Identifying Students at Risk

Purpose: This function helps identify students who are at risk of failing or dropping out based on their VLE interaction data and assessment scores.

Code:

```
# Function to identify students at risk (ADVANCED)
def identify_at_risk_students():
   query = '''
   SELECT studentInfo.id_student, studentInfo.final_result,
MAX(studentVle.date) AS last_interaction,
          AVG(studentAssessment.score) AS avg_score,
SUM(studentVle.sum_click) AS total_clicks,
          CASE
              WHEN studentInfo.final_result = 'Fail' AND
AVG(studentAssessment.score) < 30 THEN 'At Risk of Dropping Out'
              WHEN studentInfo.final_result = 'Pass' AND
AVG(studentAssessment.score) < 50 THEN 'At Risk of Failing'
              WHEN studentInfo.final_result = 'Pass' AND
AVG(studentAssessment.score) >= 50 THEN 'Not at Risk'
              ELSE 'Needs Further Review'
          END AS risk_status
   FROM studentInfo
   JOIN studentVle ON studentInfo.id_student = studentVle.id_student
   JOIN studentAssessment ON studentInfo.id_student =
studentAssessment.id_student
   GROUP BY studentInfo.id_student, studentInfo.final_result;
   '''
   conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
```

```
    df = pd.read_sql_query(query, conn)
    conn.close()
    at_risk_students = df[df['risk_status'].isin(['At Risk of
Failing', 'At Risk of Dropping Out', 'Not at Risk'])]
    return at_risk_students[['id_student', 'final_result',
'last_interaction', 'avg_score', 'total_clicks', 'risk_status']]
```

Usage:

- This query calculates:
    - Last interaction with VLE (`last_interaction`): When a student last engaged with the system.
    - Average assessment score (`avg_score`): Indicates overall performance.
    - Total clicks on VLE (`total_clicks`): Gauges engagement.
    - Risk status (`risk_status`): Flags students as "At Risk of Dropping Out", "At Risk of Failing", or "Not at Risk".

7.5 Advanced; Identifying Difficult Materials

Purpose:

This function identifies the most challenging VLE materials by analyzing student performance and ranking each material based on the average assessment scores.

Code:

```
# Function to identify difficult materials (ADVANCED)
def identify_difficult_materials():
    query = '''
    WITH ranked_materials AS (
        SELECT vle.id_site, vle.code_module,
AVG(studentAssessment.score) AS avg_score,
               ROW_NUMBER() OVER (PARTITION BY vle.code_module ORDER
BY AVG(studentAssessment.score)) AS rank
        FROM vle
        JOIN studentVle ON vle.id_site = studentVle.id_site
        JOIN studentAssessment ON studentVle.id_student =
studentAssessment.id_student
        GROUP BY vle.id_site, vle.code_module
    )
    SELECT id_site, code_module, avg_score
    FROM ranked_materials
    WHERE rank <= 10
```

```
    ORDER BY code_module, avg_score;
    '''

    conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE')
    df = pd.read_sql_query(query, conn)
    conn.close()
    return df[['id_site', 'code_module', 'avg_score']]
```

Usage:

- This query ranks the top 10 most difficult VLE materials per module by calculating average scores.
- Lower-ranked materials indicate higher difficulty based on student performance.

---

8. GUI Installations

Install Required Packages:

1. Install Flask for web framework capabilities.
2. Install Pyngrok for tunneling and exposing local servers to the internet.
3. Upgrade Pyngrok to the latest version.

Code:

```
!pip install flask
!pip install pyngrok
!pip install --upgrade pyngrok
```

4. Set Up Ngrok for Tunneling:
    - Import `ngrok` from `pyngrok`.
    - Replace `'YOUR_AUTHTOKEN'` with the authtoken copied from your ngrok dashboard. This allows you to securely tunnel your local Flask app.

Code:

```
from pyngrok import ngrok
# Replace 'YOUR_AUTHTOKEN' with the token you copied from ngrok
dashboard
!ngrok authtoken 'YOUR_AUTHTOKEN'
```

Usage:

These installations are essential for setting up your Flask web application with a tunnel to expose it to the internet using ngrok. This setup allows local development on your machine while making the Flask app accessible remotely for testing and demonstration.

Notes:

- **Flask**: A lightweight web framework for building web applications in Python.
- **Pyngrok**: Provides tunneling services, allowing your local Flask application to be publicly accessible over the internet.
- Make sure you replace `'YOUR_AUTHTOKEN'` with the actual authtoken you get from your ngrok dashboard.

---

9. Flask App Setup

These sections combine to create a functional Flask application for managing student data using a web-based CRUD interface, with live insights available through ngrok.

9.1 Create a Flask Application

Purpose: Initialize the Flask application and set up the database.

Code:

```
from flask import Flask, request, render_template_string, redirect,
url_for
import sqlite3
import pandas as pd
from functions8 import (
    setup_database, insert_student_to_db, read_student_from_db,
    update_student_in_db, delete_student_from_db, summarize_modules,
    identify_at_risk_students, identify_difficult_materials
)

app = Flask(__name__)

# Initialize the database (run once)
setup_database()
```

Usage:

The purpose here is to initialize your Flask app and set up your database using `sqlite3` for managing student data.

Notes:

- `Flask`: Handles web routes and application logic.
- `setup_database()`: This function initializes the database, creating tables if they don't already exist.

9.2 Fetch Dropdown Options

Purpose: Retrieve distinct options for dropdown menus in the web forms.

Code:

```
# Fetch dropdown options from the database
def get_course_options():
   conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE', check_same_thread=False)
   cur = conn.cursor()
   cur.execute("SELECT DISTINCT code_module, code_presentation FROM
studentInfo")
   results = cur.fetchall()
   modules = sorted(set([row[0] for row in results if row[0] is not
None]))
   presentations = sorted(set([row[1] for row in results if row[1]
is not None]))
   return modules, presentations

def get_unique_options(field_name):
   conn = sqlite3.connect('/content/drive/My
Drive/Database/StudentVLE', check_same_thread=False)
   cur = conn.cursor()
   cur.execute(f"SELECT DISTINCT {field_name} FROM studentInfo")
   results = cur.fetchall()
   return sorted([row[0] for row in results if row[0] is not None])

# Fetch options for dropdowns
module_options, presentation_options = get_course_options()
highest_education_options = get_unique_options('highest_education')
imd_band_options = get_unique_options('imd_band')
region_options = get_unique_options('region')
studied_credits_options = get_unique_options('studied_credits')
```

Usage:

This code is used to fetch distinct options from the database, such as courses, presentations, and student information, to populate dropdowns in the HTML forms for the web interface.

Notes:

- `sqlite3`: Retrieves data from your database to populate form options.
- Dynamic dropdowns improve usability by fetching available data directly from the database, ensuring the lists are always current.

9.3 HTML Template

Code:

```
# HTML templates
home_page = '''
<!DOCTYPE html>
<html>
<head>
   <title>Student VLE CRUD Operations</title>
</head>
<body>
   <h1>Student VLE CRUD Operations</h1>
   <ul>
       <li><a href="/insert">Insert Student</a></li>
       <li><a href="/read">Read Student</a></li>
       <li><a href="/update">Update Student</a></li>
       <li><a href="/delete">Delete Student</a></li>
       <li><a href="/insights">Insights</a></li>
   </ul>
</body>
</html>
'''

insights_page = '''
<!DOCTYPE html>
<html>
<head>
   <title>Insights</title>
</head>
<body>
   <h1>Insights</h1>
   <ul>
       <li><a href="/module_summary">Module Summary</a></li>
       <li><a href="/at_risk_students">Identify At-Risk
Students</a></li>
```

```
        <li><a href="/difficult_materials">Identify Difficult
Materials</a></li>
    </ul>
    <br>
    <a href="/">Back to Home</a>
</body>
</html>
'''

insert_page = '''
<!DOCTYPE html>
<html>
<head>
    <title>Insert Student</title>
</head>
<body>
    <h1>Insert Student Information</h1>
    <form method="POST">
        <label>Module:</label>
        <select name="code_module">
            {% for option in module_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Presentation:</label>
        <select name="code_presentation">
            {% for option in presentation_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Student ID:</label><input type="text"
name="id_student"><br><br>
        <label>Gender:</label>
        <select name="gender">
            <option value="M">M</option>
            <option value="F">F</option>
            <option value="Other">Other</option>
        </select><br><br>
        <label>IMD Band:</label>
        <select name="imd_band">
            {% for option in imd_band_options %}
```

```
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Highest Education:</label>
        <select name="highest_education">
            {% for option in highest_education_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Age Band:</label>
        <select name="age_band">
            <option value="0-35">0-35</option>
            <option value="35-55">35-55</option>
            <option value="55<=">55<=</option>
        </select><br><br>
        <label>Previous Attempts:</label><input type="text"
name="num_of_prev_attempts"><br><br>
        <label>Studied Credits:</label>
        <select name="studied_credits">
            {% for option in studied_credits_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Region:</label>
        <select name="region">
            {% for option in region_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Disability:</label>
        <select name="disability">
            <option value="Y">Yes</option>
            <option value="N">No</option>
        </select><br><br>
        <label>Final Result:</label>
        <select name="final_result">
            <option value="Distinction">Distinction</option>
            <option value="Pass">Pass</option>
            <option value="Fail">Fail</option>
            <option value="Withdrawn">Withdrawn</option>
        </select><br><br>
```

```
        <input type="submit" value="Insert Student">
    </form>
    <br><br>
    <a href="/">Back to Home</a>
</body>
</html>
'''

update_page = '''
<!DOCTYPE html>
<html>
<head>
    <title>Update Student</title>
</head>
<body>
    <h1>Update Student Information</h1>
    <form method="POST">
        <label>Student ID:</label><input type="text"
name="id_student"><br><br>
        <label>Module:</label>
        <select name="code_module">
            {% for option in module_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Presentation:</label>
        <select name="code_presentation">
            {% for option in presentation_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Gender:</label>
        <select name="gender">
            <option value="M">M</option>
            <option value="F">F</option>
            <option value="Other">Other</option>
        </select><br><br>
        <label>IMD Band:</label>
        <select name="imd_band">
            {% for option in imd_band_options %}
            <option value="{{ option }}">{{ option }}</option>
```

```
            {% endfor %}
        </select><br><br>
        <label>Highest Education:</label>
        <select name="highest_education">
            {% for option in highest_education_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Age Band:</label>
        <select name="age_band">
            <option value="0-35">0-35</option>
            <option value="35-55">35-55</option>
            <option value="55<=">55<=</option>
        </select><br><br>
        <label>Previous Attempts:</label><input type="text"
name="num_of_prev_attempts"><br><br>
        <label>Studied Credits:</label>
        <select name="studied_credits">
            {% for option in studied_credits_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Region:</label>
        <select name="region">
            {% for option in region_options %}
            <option value="{{ option }}">{{ option }}</option>
            {% endfor %}
        </select><br><br>
        <label>Disability:</label>
        <select name="disability">
            <option value="Y">Yes</option>
            <option value="N">No</option>
        </select><br><br>
        <label>Final Result:</label>
        <select name="final_result">
            <option value="Distinction">Distinction</option>
            <option value="Pass">Pass</option>
            <option value="Fail">Fail</option>
            <option value="Withdrawn">Withdrawn</option>
        </select><br><br>
        <input type="submit" value="Update Student">
```

```
    </form>
    <br><br>
    <a href="/">Back to Home</a>
</body>
</html>
'''

result_page = '''
<!DOCTYPE html>
<html>
<head>
    <title>Result</title>
    <style>
        table, th, td {
            border: 1px solid black;
            border-collapse: collapse;
        }
        th, td {
            padding: 8px;
        }
    </style>
</head>
<body>
    <h1>Result</h1>
    <p>{{ result | safe }}</p>
    <br><br>
    <a href="/">Back to Home</a>
</body>
</html>
'''
```

Usage:

HTML templates define the structure of the web pages where users interact with the application for CRUD operations. These templates use Jinja2 syntax to dynamically inject data like dropdown options.

Notes:

- Templates are rendered using the `render_template_string()` function.
- Variables like `module_options`, `presentation_options` are dynamically populated using the Python code in Flask.

9.4 Define Routes and Templates

Purpose: Define the main route for the web application and render HTML templates for user interaction.

Code:

```
# Routes for CRUD operations
@app.route('/')
def home():
   return render_template_string(home_page)

@app.route('/insights')
def insights():
   return render_template_string(insights_page)

@app.route('/insert', methods=['GET', 'POST'])
def insert_student():
   if request.method == 'POST':
       student_data = (
           request.form['code_module'],
           request.form['code_presentation'],
           request.form['id_student'],
           request.form['gender'],
           request.form['imd_band'],
           request.form['highest_education'],
           request.form['age_band'],
           request.form['num_of_prev_attempts'],
           request.form['studied_credits'],
           request.form['region'],
           request.form['disability'],
           request.form['final_result']
       )
       result = insert_student_to_db(student_data)
       return render_template_string(result_page, result=result)
   return render_template_string(insert_page,
                                 module_options=module_options,

presentation_options=presentation_options,

highest_education_options=highest_education_options,
                                 imd_band_options=imd_band_options,
```

```
                                region_options=region_options,
studied_credits_options=studied_credits_options)

@app.route('/read', methods=['GET', 'POST'])
def read_student():
   if request.method == 'POST':
       id_student = request.form['id_student']

       # Read student data from the database
       student_data = read_student_from_db(id_student)

       if student_data:
           # Assuming student_data is a tuple, unpack it correctly
           module, presentation, id_student, gender, imd_band,
highest_education, age_band, num_of_prev_attempts, studied_credits,
region, disability, final_result = student_data

           # Format student data for display with line breaks
           result = f'Student Data:<br>Module:
{module}<br>Presentation: {presentation}<br>Student ID:
{id_student}<br>Gender: {gender}<br>IMD Band: {imd_band}<br>Highest
Education: {highest_education}<br>Age Band: {age_band}<br>Previous
Attempts: {num_of_prev_attempts}<br>Studied Credits:
{studied_credits}<br>Region: {region}<br>Disability:
{disability}<br>Final Result: {final_result}'
       else:
           # If student is not found, show a "not found" message
           result = f"Student with ID {id_student} not found."

       return render_template_string(result_page, result=result)

   return '''
   <form method="POST">
       <label>Student ID:</label><input type="text"
name="id_student"><br><br>
       <input type="submit" value="Read Student">
   </form>
   <br><br>
   <a href="/">Back to Home</a>
   '''
```

```
@app.route('/update', methods=['GET', 'POST'])
def update_student():
   if request.method == 'POST':
       id_student = request.form['id_student']
       updated_data = (
           request.form['code_module'],
           request.form['code_presentation'],
           request.form['gender'],
           request.form['imd_band'],
           request.form['highest_education'],
           request.form['age_band'],
           request.form['num_of_prev_attempts'],
           request.form['studied_credits'],
           request.form['region'],
           request.form['disability'],
           request.form['final_result']
       )
       result = update_student_in_db(id_student, updated_data)
       return render_template_string(result_page, result=result)
   return render_template_string(update_page,
                                 module_options=module_options,

presentation_options=presentation_options,

highest_education_options=highest_education_options,
                                 imd_band_options=imd_band_options,
                                 region_options=region_options,

studied_credits_options=studied_credits_options)

@app.route('/delete', methods=['GET', 'POST'])
def delete_student():
   if request.method == 'POST':
       id_student = request.form['id_student']
       result = delete_student_from_db(id_student)
       return render_template_string(result_page, result=result)
   return '''
   <form method="POST">
       <label>Student ID:</label><input type="text"
name="id_student"><br><br>
```

```
        <input type="submit" value="Delete Student">
    </form>
    <br><br>
    <a href="/">Back to Home</a>
    '''

# Routes for Advanced Functions
@app.route('/module_summary')
def module_summary():
    df = summarize_modules()
    result = df.to_html(index=False)
    return render_template_string(result_page, result=result)

@app.route('/at_risk_students')
def at_risk_students():
    df = identify_at_risk_students()
    result = df.to_html(index=False)
    return render_template_string(result_page, result=result)

@app.route('/difficult_materials')
def difficult_materials():
    df = identify_difficult_materials()
    result = df.to_html(index=False)
    return render_template_string(result_page, result=result)
```

Usage:

This section defines various routes (`/`, `/insert`, `/update`, `/delete`, etc.) for CRUD operations, each tied to a specific functionality in your web app. It also renders the appropriate HTML templates for user interaction.

Notes:

- **Flask routes** handle the logic for each user action (insert, update, delete).
- `POST` requests handle form submissions, while `GET` requests load the forms initially.
- Database CRUD functions (`insert_student_to_db`, `update_student_in_db`, etc.) are used within the route to perform actions on the database.

9.5 Run the Application

Purpose: Start the Flask application.

Code:

```
# Run Flask app
if __name__ == "__main__":
```

Expose the Local Server with Ngrok: This allows others to access your application from the internet.

Code:

```
from pyngrok import ngrok
   public_url = ngrok.connect(5000)
   print(f" * ngrok tunnel: {public_url}")
   app.run(port=5000)
```

---

10. Final Steps and Improvements

- Test all routes in your Flask app to ensure functionality.
- Modify the HTML templates for better presentation and user experience.
- Consider implementing user authentication for better security, especially if storing sensitive student data.