Automated Students' short answers assessment

CS 4624
Multimedia, Hypertext, and Information Access
Virginia Tech, Blacksburg VA 24061

Nov 26th, 2023

Professor: Dr. Mohamed Farag
Client: Dr. Mohamed Farag

Prepared By: Mathew Padath, Wenmiao Wang, Ryan McGovern, Westin Jiang, Yifei Wan
# Table of Contents

Figures and Tables................................................................................................................. 4

1 Executive Summary............................................................................................................... 4

2 Overview.............................................................................................................................. 6
   2.1 Introduction.................................................................................................................... 6
   2.2 Client Introduction........................................................................................................ 7
   2.3 Client Requirements..................................................................................................... 8
   2.4 Project Evolution and Motivation............................................................................... 9
   2.5 Team Introduction.......................................................................................................10

3 Requirements.................................................................................................................... 12
   3.1 Backend.....................................................................................................................12
   3.2 Frontend.....................................................................................................................13

4 Design................................................................................................................................ 14
   4.1 General Approach......................................................................................................14
   4.2 System Architecture.................................................................................................15
   4.3 Backend....................................................................................................................17
   4.4 Frontend....................................................................................................................18
   4.5 Dataset.......................................................................................................................19
   4.6 Data Flow...................................................................................................................20

5 Implementation................................................................................................................ 22
   5.1 Backend....................................................................................................................22
   5.2 Alternative Backend.................................................................................................36
   5.3 Frontend....................................................................................................................44
   5.4 Dataset.......................................................................................................................68

6 Developer Manual............................................................................................................ 70
   6.1 Overview....................................................................................................................70
   6.2 Purpose of the Developer Manual............................................................................70
   6.3 Target Audience.........................................................................................................70
   6.4 Prerequisites.............................................................................................................70
   6.5 Installation..................................................................................................................71

7 User Manual...................................................................................................................... 76
   7.1 Instruction...................................................................................................................76
   7.2 Use Environment.......................................................................................................76
   7.3 User Interface Overview.........................................................................................76
   7.4 Use Cases/ Tasks Supported....................................................................................77
   7.5 Tutorials on Use.......................................................................................................80

8 Lessons learned.................................................................................................................82

9 Acknowledgement..............................................................................................................84

10 Reference.........................................................................................................................85
Figures and Tables

Figure 1: System Architecture Diagram
Figure 2: Data Flow Diagram
Figure 3: Automated Students’ Short Answers Assessment Tool homepage
Figure 4: After Pressing the Grade Button Page
Figure 5: Download Grade Page
Figure 6: Submission Files Page
Figure 7: View Previous Submissions Page
Figure 8: Teacher File Page

1 Executive Summary

The objective of this innovative project was to create an automated web application for the assessment and scoring of computer science-related short answers. This solution directly addresses the often labor-intensive and time-consuming process of manually grading written responses, a challenge that educators across various academic disciplines frequently encounter. The developed web application stands out not just for its efficiency but also for its versatility, being applicable to a wide range of subjects beyond computer science, provided that appropriate teacher answer files are supplied.

At the heart of the application lies a user-friendly interface created using ReactJS. This frontend allows educators to seamlessly upload 'teacher' and 'student' files in .tsv format. Following the upload, the application's backend, developed using Flask, takes over. It processes these submissions by comparing student responses against predefined model answers. The scoring mechanism of the application is particularly noteworthy. It employs an advanced semantic analysis approach, utilizing a pre-existing deep learning model, RoBERTa Large. This model is integral to the AutoGrader class, which is responsible for the semantic evaluation of the text.

The grading logic embedded within the AutoGrader class is both innovative and sophisticated. It assesses student responses by breaking them down into phrases and then computing the semantic similarity between each phrase and the concepts outlined in the model answers. The process employs SentenceTransformer to generate text embeddings, allowing for a nuanced evaluation based on cosine similarity between vector representations. This method ensures a grading system that transcends simple keyword matching, delving into the semantic content and understanding of the student answers.

The application boasts several key features that enhance user experience and provide educators with comprehensive insights into student performance. These include the ability to display scores and grades directly on the web application, download detailed Grade Reports that include each question, student's response, the grade awarded, and the model answer. Additionally, the application allows for the viewing of previous submissions and the downloading of historical documents such as past versions of 'teacher file', 'student file', and grade reports.
In terms of future development, the project team has outlined several ambitious goals. These include implementing a dataset-driven strategy for enhancing the training of deep learning models, thereby significantly advancing the current framework. Another focus will be on allowing for a variety of file types to be uploaded for both teacher and student files, thereby increasing the accessibility and usability of the system. Lastly, there are plans to update the functionality and appearance of the web application, incorporating features such as scrolling, standardized formatting, and improved design elements to enhance the overall user experience.

The project was developed with the invaluable guidance and support of Dr. Mohamed Farag, a research associate at the Center for Sustainable Mobility at Virginia Tech. Dr. Farag's expertise in computer science and his commitment to educational innovation have been instrumental in steering the project towards success.

In conclusion, this project marks a significant advancement in the field of educational technology, particularly in the realm of academic grading. By leveraging the power of artificial intelligence and modern web technologies, it provides an efficient, reliable, and versatile tool for educators, streamlining the grading process and offering a scalable solution adaptable to various academic contexts. The future developments outlined promise to further enhance the capabilities of this already impressive tool, pointing towards a new era in academic assessment.
2 Overview

2.1 Introduction

In today's dynamic educational environment, the pursuit of both efficiency and accuracy in assessment practices is increasingly paramount. Educators face significant challenges in evaluating students' written responses, especially short answers, which are crucial for understanding students' comprehension in depth. Our project, titled "Automated Students' Short Answers Assessment using NLP and LLM", emerges as a confluence of cutting-edge technology and pedagogical need, aiming to revolutionize the way student assessments are conducted.

Building on the foundations laid by initiatives that have successfully utilized artificial intelligence for content summarization and analysis, our project takes a leap forward by employing Natural Language Processing (NLP) and Information Extraction techniques. The objective is to develop a sophisticated tool capable of accurately assessing student responses. This tool is designed not just as a one-way assessment mechanism but as a platform for providing iterative, meaningful feedback. By incorporating expert reviews and continuous feedback loops into the development process, we ensure that the tool remains sensitive to the nuanced requirements of the curriculum and effectively addresses the diverse needs of students.

The envisioned platform is more than a mere automation tool; it's an ecosystem designed to facilitate a more interactive, responsive, and beneficial student-educator dialogue. It synthesizes the advancements in technology with the fundamental principles of education, aiming to transform the assessment process into a more enriching and formative experience for all stakeholders involved.
2.2 Client Introduction

Our project has been fortunate to benefit from the expertise and guidance of Dr. Mohamed Farag, a distinguished research associate at the Center for Sustainable Mobility (CSM). Dr. Farag's eclectic blend of interests spans intelligent transportation systems, automated vehicles, machine learning, and beyond. His multifaceted research portfolio has continually pushed the frontiers of technological innovation, making him an ideal mentor for a project at the intersection of technology and education.

Dr. Farag's academic journey, which includes a Ph.D. in Computer Science from Virginia Tech and an M.Sc. from the Arab Academy for Science, Technology, and Maritime Transport, Egypt, brings a rich perspective to the project. His global exposure, coupled with his profound research experiences, has been instrumental in shaping our project's trajectory, ensuring that it is rooted in both academic rigor and practical relevance.
2.3 Client Requirements

Dr. Farag's vision for the "Automated Students' Short Answers Assessment using NLP and LLM" project is encapsulated in a set of precise requirements, guiding the development to meet the real-world needs of educators and students.

**Objective Overview:**
- **Primary Aim:** To create an automated tool for the formative assessment of students’ short answers, applicable across various academic assessments like surveys, quizzes, and exams.
- **Secondary Aim:** To analyze and evaluate the relevance of individual concepts in student answers, ensuring alignment with the curriculum.

**Modeling Considerations:**
- Implementing a Regression model approach and concept-based classification, evaluating against a spectrum of model answers ranging from very bad to very good.

**Dataset Development:**
- **Training Dataset:** Comprising questions, model answers, and related concepts, forming the foundation for score evaluation.
- **Testing Dataset:** Featuring questions, student answers, and predicted scores, to validate the model's efficacy.

**Data Modeling:**
- Developing a data model that takes into account the complexities of NLP, Information Extraction, and educational sector nuances.

**Web Application for Students:**
- A user-friendly platform enabling students to submit answers and receive scored feedback, with scores out of 10, including decimals for finer feedback granularity.

**Project Deliverables:**
- Source codes, scripts, user and developer manuals, and a comprehensive report detailing the tool’s accuracy.

**Expected Impact:**
- The integration of NLP and Information Extraction is anticipated to streamline the assessment process, reducing educators' manual workload and providing immediate feedback to students, thereby enhancing the overall educational experience.

Dr. Farag’s insights and the benchmarks set in related research have been critical in directing our efforts. We aim to deliver a solution that is not only technologically sophisticated but also deeply rooted in educational value.
2.4 Project Evolution and Motivation

As the project progressed, the team encountered various challenges and learning opportunities that shaped the final outcome. One of the primary motivations behind this project was to alleviate the burdensome task of manual grading for educators, allowing them to focus more on interactive teaching and less on the administrative aspects of education.

The approach involved a blend of technological innovation and a deep understanding of pedagogical principles. By employing advanced NLP techniques and a robust machine learning framework, the project aimed to create a tool that can adapt to various subject matters, understand the intricacies of language, and provide accurate and meaningful assessments of student answers.

Throughout the development process, the team remained committed to creating a user-friendly, accessible, and efficient tool. This commitment was reflected in every aspect of the project, from the intuitive design of the web application to the sophisticated backend algorithms that power the grading system.

As the project nears completion, it stands as a testament to the possibilities of leveraging technology to enhance educational practices. It is a step towards a future where technology and education work hand in hand to foster a more interactive, engaging, and effective learning environment.

This enhanced introduction provides a comprehensive overview of the project, its motivation, the approach taken, and the evolution of the project over time. It also elaborately describes the client's role and expectations, setting the stage for a detailed report on the project's outcomes and impacts.
2.5 Team Introduction

Mathew Padath
Team Lead
Design & Research

Yifei Wan
Frontend Development

Wenmiao Wang
Backend & Frontend Development
Westin Jiang
Frontend Development

Ryan McGovern
Backend Development
3 Requirements

3.1 Backend

The backend of our automated answer evaluation system serves as the central hub for processing and evaluating student answers. Currently, it is optimized to manage datasets in .csv file format, which are essential for housing model answers, associated concepts, and a broad spectrum of student responses with their corresponding scores. The .csv format is specifically chosen for its simplicity and ease of data parsing, which aligns with the system's core objective to facilitate efficient reception, processing, and evaluation of student answers.

Central to the backend's functionality is the integration of an advanced deep learning model. This model is responsible for adeptly processing student answers contained within the .csv files and performing evaluations based on the predefined model answers and related concepts. It has been intricately designed to interpret the complex nuances of human language, including the ability to recognize synonyms, paraphrasing, and a wide range of linguistic variations. The architecture of the backend is strategically developed to be adaptable, allowing for future updates or modifications in the model and the potential inclusion of additional file formats.

A significant aspect of the backend is its interaction with the frontend, developed using the MERN stack (MongoDB, Express, React, NodeJS). This interaction ensures swift and secure data transactions between the user interface and the system's core processing unit. The backend also handles user profile management, upholding strict data privacy and security standards. This includes protecting user login credentials and maintaining comprehensive records of user interactions.

The backend system is equipped with advanced reporting and analytics features. These features are designed to provide valuable insights into user behavior, score distributions, and trends in student responses, which are crucial for informed decision-making and continuous system improvement. Scalability is a key requirement, ensuring the system can handle increasing loads, particularly during peak times such as examination periods or assignment deadlines. Regular data backups and robust data recovery mechanisms are implemented to safeguard against potential data loss. Additionally, the development of comprehensive APIs facilitates a wide range of functionalities, from answer submissions to user account management. Lastly, the backend includes sophisticated error handling protocols and a notification system to keep users informed of critical updates or system issues.
3.2 Frontend

The frontend of the automated answer assessment system is a user-centric interface designed for simplicity and efficiency. Developed using the MERN stack, it is tailored to handle .csv file formats for the submission and review of answers. The frontend's design focuses on presenting scores accurately, including decimal values, and allows users to review their past answers and scores, providing a comprehensive understanding of their performance.

Security and data protection are paramount in the frontend design. It includes a robust login mechanism to ensure data privacy and create personalized user experiences. The interface is crafted for ease of navigation, allowing users to move seamlessly between different sections, such as answer submission pages, historical review sections, and analytical parts showcasing metrics like average scores.

Future enhancements for the frontend are planned to enhance user engagement and learning support. These enhancements may include the provision of a dynamic definition tool for vocabulary within questions and a feature to provide the "best possible answer" after a certain number of attempts or post a due date. This will aid in bridging knowledge gaps and guiding students towards the ideal response.
4 Design

4.1 General Approach

The design of our automated grading system, deeply rooted in user-centric principles, is focused on delivering an intuitive and efficient experience for its users, primarily students and educators. The system's design is anchored in practicality and usability, ensuring that it not only incorporates advanced technical functionalities but also remains easily navigable and accessible to its intended audience.

The general approach to the system's design emphasizes simplicity and clarity. The user interface, developed using React, offers a clean and straightforward layout, free from unnecessary complexities. This approach ensures that users, regardless of their technical expertise, can easily interact with the system, upload their files, and understand their grading results. The focus is on creating an environment where technology enhances the educational experience without becoming a barrier.

In terms of system functionality, the design prioritizes core features that directly contribute to the system's educational purpose. This includes handling file uploads for teacher and student answers, processing these submissions, and displaying the grading results in a comprehensible format. The system is designed to handle these tasks efficiently, ensuring that users receive timely and accurate feedback on their submissions.

Additionally, the system is designed with scalability in mind. While currently tailored to specific requirements and functionalities, the design allows for future enhancements and expansions. This includes the potential to handle a wider variety of file formats, incorporate additional educational content, or integrate more advanced grading algorithms.

The backend, developed with Flask, is streamlined to effectively manage the data flow from the frontend to the database and back. It handles crucial operations such as receiving submissions, processing them through the grading logic, and communicating the results to the frontend. The backend design focuses on efficient data processing, ensuring quick response times and minimal delays.

Overall, the general approach to the system's design is one of balance – balancing ease of use with technical efficiency, balancing current capabilities with the potential for future growth, and balancing the needs of students and educators with the possibilities offered by modern web technologies.
4.2 System Architecture

The architecture of our automated grading system seamlessly integrates the frontend, developed using React, and the backend, built with Flask, to provide a coherent and efficient user experience. This thoughtful design ensures the system balances technical sophistication with user accessibility.

In the frontend, React's robustness and flexibility are leveraged to create a dynamic, responsive interface that engages users regardless of their device or platform. This part of the system is responsible for handling user interactions such as file uploads and displaying grading results. It's designed to offer a straightforward and intuitive journey for users, guiding them from the point of data submission to the visualization of results.

The backend, utilizing the Flask framework, acts as the processing core of the system. It manages the reception of data from the frontend, processes this data through the grading logic, and handles communication with the database. Flask's simplicity and efficiency make it an ideal choice for rapid development and smooth integration. This backend setup ensures that all submitted data is accurately processed and stored, and the results are correctly computed and communicated back to the frontend.

While the specifics of database integration aren't detailed in the provided code, typically, a database in such a system would be used to store and retrieve crucial information like user submissions, model answers, and grading results. The design of the system accommodates such database integration, emphasizing secure storage and efficient data access.
Scalability and flexibility are key features of the system's architecture. Designed to accommodate evolving educational needs and technological advancements, the architecture allows for easy updates and modifications. This adaptability ensures the system can be updated with new features, cater to an increasing user base, and adapt to changing educational content and standards.

Integral to the system's operation is the smooth integration and data flow between the frontend and backend. The frontend sends user submissions to the backend, where they are processed and evaluated. The results are then sent back to the frontend for display. This seamless integration ensures cohesive operation, providing a consistent and reliable experience for users.

In essence, the system's architecture is a well-balanced combination of frontend and backend components, each designed to perform its functions effectively while working in unison with the other. This architecture supports the current functionalities and is adaptable for future enhancements, ensuring the system remains relevant and effective for educational purposes.
4.3 Backend

The Flask backend of our automated grading system is a streamlined engine designed for effective data management and grading logic execution. It plays a crucial role in handling the core functionalities of the system, ensuring efficient processing and accurate grading of student submissions.

The Flask application is responsible for managing HTTP requests from the frontend. It processes student submissions by interpreting and validating the incoming data. This involves ensuring the data is in the correct format and ready for the subsequent grading process. The emphasis here is on accurate and efficient data handling, which is key to the reliability of the grading results.

A significant function of the backend is to execute the grading logic. After the student submissions are processed, the backend applies the grading criteria to evaluate the responses. This process is more than just a simple answer check; it involves a comprehensive analysis of the student's input, considering various aspects of their response. The backend then calculates the results, which are relayed back to the frontend for display to the users.

In terms of system performance, the backend is designed to handle a varying number of user requests effectively. While the current implementation may not include advanced scalability features, the Flask framework offers the potential for easy expansion and adaptation to handle larger volumes of data and requests. This aspect is crucial for a system that aims to accommodate a growing user base.

While the current implementation of the backend may not include explicit error handling mechanisms, the Flask framework inherently provides basic error management. This ensures that the system can operate smoothly and that any potential issues in data processing or request handling can be managed effectively.

In summary, the Flask backend in our automated grading system is a testament to functional design, focusing on core operations such as managing data flow, processing submissions, and executing grading logic. It ensures efficient operation and forms a solid foundation for potential future enhancements to increase the system's capabilities.
4.4 Frontend

The frontend of our automated grading system, crafted with React, stands as the interactive face of the project, where users—students and educators alike—engage directly with the application. This component is meticulously designed to prioritize user experience, combining technical efficiency with ease of use.

React, renowned for its dynamic capabilities, is the chosen framework for the frontend. It enables the application to be responsive and interactive, ensuring that the interface reacts seamlessly to user inputs and system updates. This choice is instrumental in creating an application that is not only functional but also engaging and intuitive for its users.

The primary role of the frontend is to facilitate user interactions, such as uploading files and receiving feedback. It is here that students and educators upload the teacher and student files and interact with the system to view the results of the grading process. The interface is designed to be straightforward, guiding users smoothly from the point of data entry to the final display of results. This design strategy ensures that users can navigate the system with ease and clarity, regardless of their technical background.

The React application handles the display of various components such as the details of teacher files, student files, and grading files. It also manages submission details, providing users with a comprehensive view of their interactions with the system. Each component is crafted to present information in a clear and organized manner, enhancing the overall usability of the application.

Interaction with the backend is a critical function of the frontend. It communicates with the Flask backend to send and receive data. This includes sending user-uploaded files to the backend for processing and receiving the evaluated results for display. The frontend ensures that this data exchange is efficient and seamless, maintaining a smooth and responsive user experience.

While the current implementation of the frontend may not include advanced features such as error handling or user authentication, it is designed with the potential for future enhancements. The React framework offers flexibility and scalability, allowing for the integration of additional features and improvements as the system evolves.

In essence, the React-based frontend of our automated grading system is a blend of user-friendly design and technical prowess. It provides a platform where educational processes are not only automated but also made more accessible and engaging. The frontend stands as a testament to our commitment to creating a tool that enhances the educational experience for both students and educators.
4.5 Dataset

In our automated grading system, the dataset and its handling are crucial components, forming the backbone of the grading process. The system's efficacy in evaluating student submissions hinges significantly on the structured and well-managed dataset.

The dataset primarily consists of question-answer pairs and related concepts, a format that is conducive to the system's requirement for clarity and precision. This structure is particularly important as it facilitates the system's ability to match student responses against model answers and evaluate them based on the concepts they cover. The design of the dataset is such that it can efficiently be parsed and processed by the Flask backend, ensuring that the data is ready for use in the grading process.

In terms of data handling, the system is designed to manage the data flow effectively, starting from the point where users upload the teacher and student files. The frontend, developed in React, captures this data and sends it to the Flask backend. The backend then processes this data by extracting relevant information from the uploaded files and preparing it for the grading logic.

The grading logic, a crucial part of the backend, then takes over. It evaluates the student's answers by comparing them against the model answers and concepts stored in the dataset. This process is not merely a direct comparison but involves a nuanced analysis that takes into account the context and content of the student's response. The backend computes the grading results based on this analysis.

While the current implementation of the system may not include complex data processing features like machine learning algorithms, the design allows for future integration of such advanced techniques. This potential for future enhancement is critical, as it means the system can evolve to incorporate more sophisticated grading mechanisms, such as natural language processing algorithms, to improve accuracy and efficiency.

Data storage and retrieval, although not explicitly detailed in the provided code, are essential aspects of the system's design. The efficient storage of data, including the question-answer pairs, model answers, and student submissions, ensures that the system can quickly access and utilize this data whenever needed. This efficient data management is key to the system's performance and reliability.

In summary, the dataset and data handling in our automated grading system are designed with a focus on functionality and future scalability. The structured dataset, effective data flow management, and potential for incorporating advanced data processing techniques ensure that the system is not only efficient in its current state but also poised for future enhancements and improvements.
4.6 Data Flow

The dataflow in our automated grading system is meticulously architected to ensure a smooth and efficient movement of information from the initial point of user interaction to the final output of grading results. This dataflow is crucial in maintaining the system's integrity and accuracy, ensuring that each step in the process adds value and contributes to the overall goal of providing reliable educational assessments.

Dataflow begins at the frontend, where users, typically students and educators, interact with the React-based interface. Here, the users upload the necessary files, such as teacher and student answer files. The React application is designed to facilitate this data entry in a user-friendly manner, ensuring that the process is intuitive and straightforward.

Once the data is entered into the system through the frontend, it is sent to the Flask backend. This marks a critical phase in the dataflow, where the backend takes on the responsibility of processing the uploaded files. The Flask application parses and validates the incoming data, ensuring it adheres to the expected format and standards. This step is vital to the accuracy of the grading process, as it sets the foundation for the subsequent analysis.

The heart of the dataflow lies in the grading logic implemented in the backend. Here, the processed student responses are evaluated against the model answers and concepts stored in the dataset. The backend's grading logic is designed to not only match answers but also to analyze the content and context of the student's response. This nuanced approach to grading is essential for providing a fair and accurate assessment of the student's understanding.
After the evaluation is complete, the results are generated and sent back to the frontend. This return of data completes the dataflow cycle, with the frontend displaying the grading results to the users. The presentation of results is an important aspect of the frontend design, aiming to provide clear and understandable feedback to the users about their performance.

While the current system may not incorporate complex data processing techniques or extensive error handling, the dataflow design ensures that the essential functionalities of file processing and grading are handled efficiently and accurately. The system's design also allows for future enhancements, including the integration of more advanced data processing and analysis techniques.

In essence, the dataflow within our automated grading system is a testament to a well-considered design that prioritizes efficiency, accuracy, and user experience. Each step in the dataflow process, from data entry to result display, is crafted to contribute meaningfully to the system's primary goal of providing automated, reliable educational assessments.
5 Implementation

5.1 Backend

Import from the libraries and Setup

```python
1 from flask import Flask, request, jsonify
2 from flask_cors import CORS
3 from sentence_transformers import SentenceTransformer, util
4 import csv
5 import io
6 from pymongo import MongoClient
7 from datetime import datetime
8 from bson.json_util import dumps
9 from bson import ObjectId
```

1. Flask Web Framework and Utilities
   - Flask: Imports the Flask class, which is necessary to create a new Flask web application.
   - request: Provides access to the request object, allowing you to access incoming request data (e.g., form data, files).
   - jsonify: A helper function in Flask to return a response in JSON format.
2. CORS (Cross-Origin Resource Sharing)
   - CORS: This import is used to handle Cross-Origin Resource Sharing. It allows or restricts resources on a web server to be requested from another domain.
3. Sentence Transformers for NLP
   - SentenceTransformer: Imports the SentenceTransformer class, which is used for generating sentence embeddings.
   - util: This utility module from the Sentence Transformers library provides additional functionalities, such as calculating cosine similarity between embeddings.
4. CSV and IO Handling
   - csv: This module is used to read and write CSV files, commonly used for data storage and transfer.
   - io: The io module provides tools for working with streams (like file streams), which is essential for handling file-like objects in memory.
5. MongoDB Client
   - MongoClient: This is used to connect to a MongoDB database. It enables the application to interact with the database for storing and retrieving data.
6. Date and Time Handling
   - datetime: This module provides classes for manipulating dates and times. It's often used for timestamping in applications.
7. BSON Utilities for MongoDB
- **dumps**: A utility function for converting BSON objects to JSON strings, useful for MongoDB data handling.
- **ObjectId**: This is a special type used by MongoDB as a primary key for documents. It's essential for querying specific documents in a MongoDB collection.

```python
app = Flask(__name__)
CORS(app)
app.config['UPLOAD_FOLDER'] = 'uploads'
client = MongoClient('mongodb://localhost:27017')
db = client['automated_students_short_answers_assessment_db']
submissions = db['submissions']
```

1. **Flask App Setup**
   - `app = Flask(__name__)`
     - Initializes an instance of the Flask class. `__name__` is a Python special variable that gets set to the name of the module in which it is used. Flask uses this to know where to look for resources like templates and static files.
   - `CORS(app)`
     - Applies CORS to the Flask app instance. This is necessary for the app to accept requests from web pages hosted on different domains.
   - `app.config['UPLOAD_FOLDER'] = 'uploads'
     - Sets a configuration value for the Flask app. This line specifies the folder ('uploads') where uploaded files will be stored.

2. **MongoDB Connection Setup**
   - `client = MongoClient('mongodb://localhost:27017')`
     - Creates an instance of MongoClient, connecting to the local MongoDB instance running on the default port (27017).
   - `db = client['automated_students_short_answers_assessment_db']`
     - Selects a specific database (automated_students_short_answers_assessment_db) from the MongoDB server. If this database doesn't exist, it will be created when data is first written to it.
   - `submissions = db['submissions']`
     - Selects (or creates) a collection named 'submissions' in the specified database. Collections in MongoDB are analogous to tables in relational databases and are used to store documents (data).
class AutoGrader

Grading Logic
The grading logic in the AutoGrader class is designed to assess student responses by comparing them to the concepts of the model answer. The process begins by breaking down the student's answer into phrases and then computing the semantic similarity between each phrase and the concepts in the model answer. Semantic Similarity is determined by using a deep learning model to generate embeddings for each phrase and the model concepts. The similarity score is calculated using cosine similarity, which measures the cosine of the angle between the two vectors in the embedding space. If the similarity for a concept exceeds a certain threshold (0.6 in this case), the concept is considered to be successfully detected in the student's answer. The final grade is computed as the proportion of detected concepts to the total number of concepts in the model answer, multiplied by a scaling factor (10 in this case). This grade is then rounded to the nearest whole number to represent the student's performance.

Embeddings
Embeddings are dense vector representations of text that capture the semantic meaning of words, phrases, or sentences. In the AutoGrader class, embeddings are generated by the SentenceTransformer model, which is a modification of the RoBERTa model fine-tuned for sentence-level tasks. When the compute_similarity method is called, it uses this model to convert the input sentences (student's phrase and model concept) into embeddings. These embeddings are high-dimensional vectors that encode the input text in such a way that semantically similar phrases are placed closer together in the vector space. Consequently, by computing the cosine similarity between embeddings, we can obtain a measure of how semantically close two pieces of text are.

Deep Learning Model
The SentenceTransformer model used in the AutoGrader class is a pre-trained deep-learning model based on the RoBERTa architecture. RoBERTa is a variation of BERT (Bidirectional Encoder Representations from Transformers), which is a transformer-based model designed to understand the context and relationships between words in a sentence. The model is pre-trained on a large corpus of text and fine-tuned on sentence similarity and paraphrase identification tasks, which makes it particularly well-suited for semantic similarity comparison. In this application, the model is used to generate embeddings for both the student's answer and the model answer's concepts. The effectiveness of this model lies in its ability to capture the nuances of language and produce embeddings that reflect the underlying meaning of the text, allowing for accurate semantic comparisons.

By utilizing this advanced deep learning model, the AutoGrader can perform nuanced grading that goes beyond keyword matching, evaluating the student's understanding of concepts by considering the semantic content of their answers.

```
class AutoGrader:
    def __init__(self):
        self.model = SentenceTransformer('roberta-large-nli-stsb-mean-tokens')
```

- This is the initialization method of the AutoGrader class. When an instance of the class is created, this method is called.
- self.model is an instance variable that stores the SentenceTransformer model. The model used here is 'roberta-large-nli-stsb-mean-tokens', which is pre-trained on natural language inference (NLI) and semantic textual similarity benchmark (STSB) tasks, making it adept at understanding sentence meanings.

```python
def load_student_answers(self, student_data):
    reader = csv.reader(io.StringIO(student_data), delimiter='\t')
    next(reader)
    student_answers = {row[0]: row[1] for row in reader}
    return student_answers
```

- This method loads student answers from a string student_data that represents tab-separated values (TSV) data. The csv.reader is used to parse this string.
- io.StringIO(student_data) turns the string into a stream, which csv.reader can read from as if it were a file.
- next(reader) is used to skip the header row of the input data, assuming the first row is headers and not actual data.
- student_answers is a dictionary comprehension that creates a dictionary where each key-value pair corresponds to a student ID and their answer.

```python
def compute_similarity(self, sentence1, sentence2):
    embedding1 = self.model.encode(sentence1, convert_to_tensor=True)
    embedding2 = self.model.encode(sentence2, convert_to_tensor=True)
    similarity = util.pytorch_cos_sim(embedding1, embedding2).item()
    return similarity
```

- This method computes the cosine similarity between two sentences.
- self.model.encode generates embeddings for each sentence, with convert_to_tensor=True indicating that the output should be a PyTorch tensor.
- util.pytorch_cos_sim computes the cosine similarity between the two embeddings.
- .item() is called to get the Python scalar value from the resulting tensor.

**Clarification:**

1. **Encoding Sentences into Embeddings**
   - The SentenceTransformer model's encode function takes a piece of text and converts it into a vector (embedding).
   - self.model.encode(sentence1, convert_to_tensor=True) takes sentence1 and turns it into an embedding, represented as a PyTorch tensor. The same happens with sentence2.
   - The embeddings are created such that semantically similar sentences should have embeddings that are close to each other in the vector space.

2. **Computing Cosine Similarity**
   - Cosine similarity is a measure that calculates the cosine of the angle between two vectors in a multi-dimensional space.
   - In NLP, it's used to determine how similar two text embeddings are. A cosine similarity close to 1 means the two vectors are very close in orientation, implying high similarity.
● util.pytorch_cos_sim(embedding1, embedding2) computes this similarity between the two sentence embeddings.
● The result is a tensor containing a single value that represents the similarity score.

3. Extracting the Similarity Score
● .item() extracts the similarity score as a standard Python floating-point number from the PyTorch tensor.
● This number is what the method returns.

```python
def grade_answer(self, student_answer, model_answer, model_concepts):
    max_n = 1
    words_list = student_answer.split()
    student_phrases = [''.join(words_list[i:i+n]) for i in range(1, min(max_n+1, len(words_list))) for i in range(len(words_list)-n+1)]
```

● This method grades a student's answer by comparing it to the model answer and its concepts.
● max_n is set to 1, which indicates that the phrases will be single words (no n-grams of size greater than 1).
● student_answer.split() splits the student's answer into a list of individual words.
● student_phrases is a list comprehension that creates a list of all possible phrases from the student's answer. However, since max_n is 1, this will just be the same list of words.

```python
total_concepts = len(model_concepts)
detected_concepts = 0
```

● total_concepts is the number of concepts in the model answer.
● detected_concepts are initialized to 0 and will be used to count the number of concepts detected in the student's answer.

```python
for model_concept in model_concepts:
    max_similarity = 0
    embeddings_student_phrases = self.model.encode(student_phrases, convert_to_tensor=True)
    embeddings_model_concept = self.model.encode(model_concept, convert_to_tensor=True)
    similarities = util.pytorch_cos_sim(embeddings_model_concept, embeddings_student_phrases)
    max_similarity = max(max_similarity, similarities)
```

● The loop iterates over each concept in model_concepts.
● For each concept, self.model.encode is called to generate embeddings for all student phrases and the current model concept.
● util.pytorch_cos_sim calculates the cosine similarities between the concept's embedding and each of the student's phrase embeddings.
● max_similarity is updated to the maximum similarity score found between the model concept and any of the student phrases.

Clarification:

```python
for model_concept in model_concepts:
```

● This line initiates a loop that will iterate over each "model concept". The model_concepts are likely key concepts or phrases that the student's answer is expected to contain or address.
This line initializes a variable `max_similarity` to zero. This variable will be used to track the highest similarity score between the embeddings of the student phrases and the current model concept.

```python
embeddings_student_phrases = self.model.encode(student_phrases, convert_to_tensor=True)
```

Here, `self.model.encode` is called on the `student_phrases`, which is a list of phrases from the student's answer. This function converts each phrase into an embedding.

These embeddings are vectors in a high-dimensional space that numerically represent the semantic meaning of the phrases.

`convert_to_tensor=True` indicates that the output should be a PyTorch tensor, which is a multi-dimensional matrix that can be efficiently processed using PyTorch functions.

```python
embedding_model_concept = self.model.encode(model_concept, convert_to_tensor=True)
```

This line creates an embedding for the current `model_concept`. Just like the student phrases, this embedding is a vector that represents the semantic meaning of the concept.

```python
similarities = util.pytorch_cos_sim(embedding_model_concept, embeddings_student_phrases)
```

`util.pytorch_cos_sim` computes the cosine similarity between the `embedding_model_concept` and each embedding in `embeddings_student_phrases`.

Cosine similarity is a measure of similarity between two non-zero vectors that calculates the cosine of the angle between them. Its value ranges from -1 (meaning exactly opposite), through 0 (meaning orthogonal or no similarity), to 1 (meaning exactly the same). For embeddings, a higher cosine similarity indicates greater semantic similarity.

```python
max_similarity = max(similarities[0])
```

After computing the similarities, this line finds the highest cosine similarity score from the tensor of similarity scores.

`similarities[0]` is used because `util.pytorch_cos_sim` returns a tensor where each row contains the similarity scores between the model concept and each student phrase. Since we're comparing one model concept with multiple student phrases, we're interested in the first row of this tensor.

The `max` function is applied to find the highest similarity score, which represents the closest semantic match between the model concept and any of the student's phrases.

```python
if max_similarity > 0.6:
    detected_concepts += 1
```

If the `max_similarity` for a concept is greater than 0.6, it is considered as detected, and `detected_concepts` is incremented.

```python
grade = (detected_concepts / total_concepts) * 10
return round(grade)
```

The final grade is calculated as the ratio of `det`
def format_qa_data(teacher_data, student_data)

This function is crucial for transforming raw text data into a structured format that can be easily used for further processing, such as analysis or feeding into a grading algorithm.

Function Definition

- def format_qa_data(teacher_data, student_data): Defines a function named format_qa_data that takes two parameters: teacher_data and student_data. This function is responsible for formatting the question-answer data from teachers and students into a structured format.

Processing Teacher and Student Data

- teacher_data.strip().split('
') and student_data.strip().split('
'): These expressions split the input data into lines. strip() removes any leading and trailing whitespace, including newline characters. split('
') splits the string into a list of lines.
- line.split('	'): For each line, this splits the string by the tab character ('\t'), assuming that the input data is tab-delimited. This is typical for data that might be exported from a spreadsheet or a similar format.
- if line.strip(): This condition filters out any empty lines or lines with only whitespace.

Removing Headers

- if teacher_qa_pairs:
  teacher_qa_pairs.pop(0)
- if student_qa_pairs:
  student_qa_pairs.pop(0)

These lines check if the lists teacher_qa_pairs and student_qa_pairs are not empty and then remove the first element from each list using pop(0). This is done under the assumption that the first line of each input data set is a header that should not be included in the formatted data.

Formatting Teacher Data
formatted_teacher_data = []
for pair in teacher_qa_pairs:
    if len(pair) == 3:
        question, model_answer, concepts = pair
        formatted_teacher_data.append({'question': question.strip(),
                                        'modelAnswer': model_answer.strip(),
                                        'concepts': concepts.split(';')})

formatted_student_data = []
for pair in student_qa_pairs:
    if len(pair) == 2:
        question, student_answer = pair
        formatted_student_data.append({'question': question.strip(),
                                        'studentAnswer': student_answer.strip()})

return formatted_teacher_data, formatted_student_data
def upload_file()
The upload_file function is central to processing the uploaded files for the teacher and student data, grading the student responses, and saving the results to the database. It also returns a detailed response to the client, including the grades and relevant details.

Flask Route Definition

```python
@app.route('/api/process_files', methods=['POST'])
def upload_file():
    if 'teacherFile' not in request.files or 'studentFile' not in request.files:
        return jsonify({'error': 'Missing file(s)'}), 400
```

• @app.route('/api/process_files', methods=['POST']): This decorator defines a route in the Flask application. It specifies that the function upload_file will handle HTTP POST requests to the URL path /api/process_files.

• def upload_file(): Declares the upload_file function that gets invoked when the specified route is accessed.

Request Validation

```python
if 'teacherFile' not in request.files or 'studentFile' not in request.files:
    return jsonify({'error': 'Missing file(s)'}), 400
```

• This code checks whether the incoming request contains files with the keys 'teacherFile' and 'studentFile'. If either is missing, it returns a JSON response with an error message and an HTTP status code of 400 (Bad Request).

File Handling

```python
teacher_file = request.files['teacherFile']
student_file = request.files['studentFile']
```

• Retrieves the files from the request. request.files is a dictionary-like object containing uploaded files.

```python
teacher_data = teacher_file.read().decode('utf-8').strip()
student_data = student_file.read().decode('utf-8').strip()
```

• Reads the file content and decodes it from bytes to a UTF-8 string, ensuring that the file content is treated as text. strip() is used to remove any leading or trailing whitespace.

Filename Check

```python
if teacher_file.filename == '' or student_file.filename == '':
    return jsonify({'error': 'No selected file'}), 400
```

• Checks if filenames for the uploaded files are provided. If either filename is empty, it returns an error message with a 400 status code.

Data Processing
Processes the teacher_data to construct a dictionary (model_data) mapping questions to their model answers and associated concepts. The data is expected to be delimited by newlines (\n) and tabs (\t).

Similarly processes the student_data, creating a dictionary (student_answers) mapping questions to student answers.

Grading

- Instantiates the AutoGrader class.
- Initializes variables for accumulating the total score and storing question details.
- Calls format_qa_data to format the raw teacher and student data.

- Iterates through each student's answer, computes the score using the AutoGrader class, and accumulates the scores and details.

Response Preparation
● Computes the total possible score and the overall grade.
● Prepares the response data including the total score, total possible score, grade, and details of each question.

Database Insertion

● Prepares data for database insertion, including a timestamp, formatted data, and grading details.
● Inserts the prepared data into the submissions collection in MongoDB.

Return JSON Response

● Returns the response_data in JSON format as the response to the client.

Invalid File Type Handler

● If the file type does not meet the expected criteria (handled earlier in the function), returns an error message with a 400 status code.
def get_all_submissions()
This function is essential for retrieving all submissions stored in the database, which can be useful for administrative or analytical purposes. The use of try-except for error handling ensures that the server responds gracefully to any issues that may arise during the database query.

Flask Route Definition

```python
@app.route('/api/submissions', methods=['GET'])
def get_all_submissions():
```

- `@app.route('/api/submissions', methods=['GET'])`: This decorator creates a route in the Flask application for the URL path /api/submissions. It specifies that the get_all_submissions function will handle HTTP GET requests to this URL. The GET method is typically used for retrieving data.
- `def get_all_submissions()`: Defines the function that is executed when the route is accessed.

Function Implementation

```python
try:
    all_submissions = list(db.submissions.find({}))
    return dumps(all_submissions), 200
except Exception as e:
    return jsonify({'error': str(e)}), 500
```

- The function implementation is wrapped in a try-except block to handle potential exceptions that might occur during database operations.

1. Database Query
   - `all_submissions = list(db.submissions.find({}))`: This line queries the MongoDB database to retrieve all documents in the submissions collection.
   - `db.submissions.find({})`: The find method of the MongoDB collection is used to retrieve all documents. The `{}` argument is an empty filter, which means no specific filtering is applied, and all documents in the collection are retrieved.
   - `list(...)`: The result from find is a cursor, and list is used to convert this cursor into a list of documents.

2. Successful Response
   - `return dumps(all_submissions), 200`: If the database query is successful, the function returns the submissions.
   - `dumps(all_submissions)`: The dumps function from bson.json_util converts the BSON objects from MongoDB into a JSON string. This is necessary because Flask's jsonify cannot directly handle BSON objects.
   - `, 200`: The function returns an HTTP status code of 200, indicating a successful response.

3. Error Handling
   - `except Exception as e:` Catches any exceptions that might occur during the execution of the database query.
   - `return jsonify({'error': str(e)}), 500`: If an exception is caught, the function returns a JSON response containing the error message and an HTTP status code of 500. The status code 500 indicates an internal server error, which is appropriate for unexpected exceptions.
def get_submission(submission_id)
This function is crucial for retrieving specific submissions by their ID, which can be useful for viewing details of individual submissions, especially in administrative or analysis interfaces. The error handling ensures robustness, providing clear and informative responses for both successful and unsuccessful queries.

Flask Route Definition

@app.route('/api/submissions/<submission_id>', methods=['GET'])

def get_submission(submission_id):

- @app.route('/api/submissions/<submission_id>', methods=['GET']): This line sets up a route in the Flask application. It tells Flask that the function get_submission will handle HTTP GET requests to the URL path /api/submissions/<submission_id>. The <submission_id> in the path is a variable part, which will be passed to the function as the submission_id argument.
- def get_submission(submission_id): Defines the get_submission function, which takes one parameter, submission_id. This parameter will receive the value specified in the URL.

Function Implementation

try:
submission = submissions.find_one({'_id': ObjectId(submission_id)})
if submission:
    return dumps(submission), 200
else:
    return jsonify({'error': 'Submission not found'}), 404
except Exception as e:
    return jsonify({'error': str(e)}), 500

1. Database Query in a Try Block
   - The code inside the try block attempts to find a specific submission in the MongoDB database.
   - submission = submissions.find_one({'_id': ObjectId(submission_id)}): This line searches for a document in the submissions collection where the _id field matches the provided submission_id.
   - find_one(...): A method that retrieves a single document from the collection that matches the query. If no document is found, it returns None.
   - {'_id': ObjectId(submission_id)}: The query filter. MongoDB uses ObjectId for unique document identifiers. ObjectId(submission_id) converts the string submission_id to an ObjectId.
   - if submission:: Checks if a submission was found.
   - return dumps(submission), 200: If a submission is found, it returns the submission data converted to a JSON string using dumps, along with the HTTP status code 200 (OK).

2. Handling Submission Not Found
   - else: return jsonify({'error': 'Submission not found'}), 404: If no submission is found (i.e., submission is None), it returns a JSON response with an error message and an HTTP status code of 404 (Not Found).

3. Exception Handling
except Exception as e: Catches any exceptions that occur during the database query.
return jsonify({'error': str(e)}), 500: In case of an exception, the function returns a JSON response with the error message and an HTTP status code of 500 (Internal Server Error).

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

This line is typically placed at the end of the Flask script. When you run the script directly (e.g., by executing `python scriptname.py`), it will start the Flask application. If the script is imported into another module, this block will not be executed, preventing the Flask application from running unintentionally.

1. **if __name__ == '__main__':**
   - This is a conditional statement that checks whether the script is being run as the main program.
   - __name__ is a special variable in Python. When a Python file is run, Python sets __name__ to "__main__" in that file.
   - If the file is being imported from another module, __name__ will be set to the file's name.
   - Therefore, if __name__ == '__main__': is checking if the script is being run directly (not being imported) and then executes the code block under it.

2. **app.run(debug=True)**
   - app.run(...): This method runs the Flask application on the local development server.
   - debug=True: The debug parameter is set to True. This means that the server will automatically reload for code changes and will show a debugger in the browser if an error occurs. It's a useful feature during development because it provides detailed error messages and allows for real-time code changes without needing to restart the server.
   - Running the Flask app with debug=True should only be done in a development environment. It's not recommended for production as it can expose sensitive information and can be a security risk.


5.2 Alternative Backend

Training the deep-learning model
Implementing the script for training a deep learning model to grade student answers involves several crucial steps, each tailored to ensure the model's accuracy and reliability. The process begins with meticulous data preparation, where a synthetic dataset, consisting of question-answer pairs and corresponding grades, is read and preprocessed. This dataset, stored in TSV format, is crucial for providing a diverse range of inputs to the model.

The next step is data splitting, which is fundamental for model training and evaluation. By dividing the dataset into training, validation, and testing subsets, the model can be trained effectively, its performance can be validated, and its generalizability can be tested. This splitting not only aids in model assessment but also plays a vital role in preventing overfitting.

Selecting an appropriate model and tokenizer is a pivotal decision in this process. RoBERTa, renowned for its effectiveness in natural language processing tasks, is chosen for its advanced capabilities in understanding and processing text data. The combination of RobertaForSequenceClassification and RobertaTokenizer facilitates the appropriate formatting and tokenization of the text data, making it suitable for the model.

A critical aspect of the implementation is the custom function tokenize_data, designed to automate the tokenization of the training, validation, and testing datasets. This ensures that the text data is consistently prepared across all subsets, maintaining data integrity and model input standards.

The training phase involves setting up a typical training loop in PyTorch, including the definition of a loss function and an optimizer. Given the regression nature of the grading task, Mean Squared Error is chosen as the loss function, while the Adam optimizer is selected for its efficiency with sparse gradients and adaptive learning rates.

Throughout the training process, close attention is paid to the model's performance on the validation set. This not only guides the training adjustments but also helps in early detection and mitigation of overfitting. Post-training, the model undergoes a thorough evaluation on a separate test set to assess its real-world applicability and performance.

To enhance the practical utility of the model, a function named predict_grade is implemented. This function enables the model to grade any given question-answer pair, thereby extending its use beyond the confines of the training script and into real-world applications where automated grading is required.

In conclusion, the implementation of this script encompasses a series of well-thought-out steps, from data handling to model training and evaluation. The use of advanced NLP tools and methodologies is central to addressing the challenges in understanding and grading natural language content, ultimately achieving a model that is both accurate and applicable in diverse scenarios.

Import Statements
The script imports necessary libraries and modules. torch is the main PyTorch library, torch.nn and torch.optim for neural network layers and optimizers, train_test_split from Scikit-learn for splitting the dataset, and RobertaTokenizer and RobertaForSequenceClassification from the Transformers library.

Reading and Preparing the Dataset

The script reads data from a TSV (tab-separated values) file named 'synthetic_dataset.tsv', skipping the header line. Each row contains a question, an answer, and a numerical grade, which are appended to dataset_from_file.

The dataset is split into separate lists for questions, answers, and grades.

Splitting the Dataset

The dataset is split into a training set and a temporary set using train_test_split.

The temporary set is further split into validation and test sets.

Tokenization

A tokenizer from the RoBERTa model is initialized.
- Defines a function `tokenize_data` to tokenize the questions and answers, with padding and truncation.

```python
def tokenize_data(questions, answers):
    return tokenizer(questions, answers, padding='max_length', truncation=True, max_length=256, return_tensors='pt')
```

- Tokenizes the training, validation, and test datasets.

Creating Tensors for Grades

```python
train_data = tokenize_data(train_questions, train_answers)
valid_data = tokenize_data(valid_questions, valid_answers)
test_data = tokenize_data(test_questions, test_answers)
```

- Converts the grades into PyTorch tensors.

Model Initialization

```python
model = RobertaForSequenceClassification.from_pretrained('roberta-large', num_labels=1)
```

- Initializes the RoBERTa model for sequence classification with a single output label (grade).

Training Setup

```python
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- Sets Mean Squared Error as the loss function and Adam as the optimizer.

Training Loop

```python
num_epochs = 10

for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(input_ids=train_data['input_ids'], attention_mask=train_data['attention_mask'])[0].squeeze()
    loss = criterion(outputs, train_grades_tensor)
    loss.backward()
    optimizer.step()

model.eval()

    with torch.no_grad():
        val_outputs = model(input_ids=valid_data['input_ids'], attention_mask=valid_data['attention_mask'])[0].squeeze()
        val_loss = criterion(val_outputs, valid_grades_tensor)

print(f"Epoch {epoch}/{num_epochs}, Training Loss: {loss.item()}, Validation Loss: {val_loss.item()}")
```

- Sets the number of epochs and runs the training and validation loop.
Testing the Model

```python
62 model.eval()
63 with torch.no_grad():
64    test_outputs = model(input_ids=test_data['input_ids'], attention_mask=test_data['attention_mask'])[0]
65    test_loss = criterion(test_outputs.squeeze(), test_grades_tensor)
66
67 print(f"Test Loss: {test_loss.item()}")
```

- Evaluates the model on the test dataset.

Saving the Model and Tokenizer

```python
75 model.save_pretrained("./trained_model_directory/")
76 tokenizer.save_pretrained("./trained_model_directory/")
```

- Saving the Model and Tokenizer

Function for Making Predictions

```python
69 def predict_grade(question, answer):
70    tokenized_data = tokenize_data([question], [answer])
71    with torch.no_grad():
72        output = model(input_ids=tokenized_data['input_ids'], attention_mask=tokenized_data['attention_mask'])[0]
73        return output.item()
```

- Defines a function to predict grades using the trained model.
Grade provided by deep-learning
Implementing the script for an automated grading system involves integrating advanced natural language processing techniques with practical data management. The process starts with importing necessary libraries, including PyTorch for the deep learning model, CSV for data handling, and I/O operations for output generation. The script's functionality centers around several pivotal functions designed to handle and structure the data efficiently.

The function `load_student_answers` is crafted to read student responses from a TSV file, creating a dictionary that maps each question to its corresponding student answer. This structured approach is essential for the subsequent processing stages. Similarly, `load_questions_data` extracts questions, model answers, and key concepts from a separate dataset, facilitating a comprehensive comparison between student responses and the expected answers.

A critical aspect of the implementation is the scoring of student answers, for which a pre-trained RoBERTa model is employed. The function `predict_score` tokenizes each answer and uses the model to predict a grade. This automated process assesses the alignment of the student's response with the model answer, effectively grading each submission based on established criteria.

The heart of the script lies in iterating through each student answer, retrieving corresponding model answers and concepts, and applying the `predict_score` function. This not only automates the grading process but also enables a detailed examination of each response by including the question, student answer, model answer, and the assigned grade in the output.

The culmination of the script is the calculation of the cumulative score, aggregating individual grades for an overall assessment. The results are displayed in the console and written to a text file, ensuring both immediate accessibility and long-term record-keeping.

In summary, the script exemplifies a comprehensive approach to integrating deep learning in educational technology. By automating grading, it offers scalability and consistency in assessments, demonstrating a significant application of AI in enhancing educational processes and evaluation methods.

Import Statements

```python
1 import csv
2 import io
3 import torch
```

- import csv: Imports Python's built-in CSV module, used here for reading and writing CSV files.
- import io: Imports Python's I/O module for handling various types of I/O operations. In this script, it's likely used for string-based I/O.
- import torch: Imports PyTorch, a popular machine learning library, which is used for working with deep learning models.

Function: load_student_answers
● Defines a function `load_student_answers` that takes a filename as input.
● Initializes an empty dictionary `student_answers`.
● Opens the file in read mode and reads its content using `csv.reader`, expecting tab-separated values (delimiter='\t').
  next(reader): Skips the header row.
● Iterates over each row in the file, assuming each row contains a question and its corresponding answer.
● Maps each question to its answer in the `student_answers` dictionary.
● Returns the `student_answers` dictionary.

Imports from Transformers

```python
from transformers import RobertaTokenizer, RobertaForSequenceClassification
```

● Imports `RobertaTokenizer` and `RobertaForSequenceClassification` from the transformers library. These are used for tokenizing the text and for the sequence classification model, respectively.

Loading the Tokenizer and Model

```python
model_path = './trained_model_directory/
tokenizer = RobertaTokenizer.from_pretrained('./trained_model_directory/
model = RobertaForSequenceClassification.from_pretrained('./trained_model_directory/
```

● Sets the `model_path` to the directory containing the trained model.
● Initializes the `RobertaTokenizer` and `RobertaForSequenceClassification` model using the pre-trained model from the specified directory.

Function: `predict_score`
- Defines a function `predict_score` that takes an answer as input.
- `with torch.no_grad()` disables gradient calculations, used here to reduce memory usage and speed up computations since gradients are not needed for inference.
- Tokenizes the answer and prepares the input tensor.
- Passes the tokenized input to the model and gets the output.
- Extracts the predicted score from the model's output logits.
- Returns the predicted score.

**Function: load_questions_data**

- Defines a function `load_questions_data` for loading questions data from a file.
- Reads the file and parses each row into question, model_answer, and concepts.
- Splits concepts by ';' to create a list of concepts.
- Maps each question to a tuple of its model_answer and concepts_list.
- Returns the questions_data dictionary.

**Script Execution**
The rest of the script sets up file names, loads data, and iterates through student answers to grade them using the predict_score function. It prints the grading details, calculates the cumulative score, and saves the output to a file.

The script demonstrates a comprehensive approach to automated grading, leveraging a deep learning model to assess student responses against model answers, a crucial advancement in educational technology applications.
5.3 Frontend

Import from the libraries

This setup is typical for a React application that uses routing (to navigate between different views or components) and needs to make API calls (with axios) and manage state and lifecycle (with hooks like useState and useEffect). The CSS import is for styling the components of the app.

```javascript
1 import React, { useEffect, useState } from 'react';

- import React: This imports the React library, which is fundamental for using React components.
- { useEffect, useState }: These are React hooks. useState is used for state management within a component. useEffect is used for performing side effects in function components (e.g., fetching data, setting up a subscription, manually changing the DOM in React components, etc.).

2 import axios from 'axios';

- axios: This is a promise-based HTTP client for making HTTP requests to fetch or save data. You'll often see it used in React applications for API calls.

3 import './App.css';

- This imports CSS for styling the React application. App.css will contain the CSS styles specific to this application.

4 import { BrowserRouter as Router, Routes, Route, Link, useParams } from 'react-router-dom';

These are components and hooks imported from react-router-dom, a third-party library used to handle routing in React applications.
- BrowserRouter as Router: This is a router component using the HTML5 history API to keep your UI in sync with the URL. It's being renamed to Router here for convenience.
- Routes and Route: Used for defining various routes (paths) in the application. Routes is a wrapper for multiple Route components.
- Link: A component used to create links to different routes. It's equivalent to an anchor (<a>) tag but works without refreshing the entire page.
- useParams: A hook that lets you access parameters from the URL.
function TeacherFileDetails()
TeacherFileDetails is a React component designed for handling and displaying specific submission details based on a submission ID from the URL. It fetches submission data from a backend service, handles the state, and provides a user interface for viewing these details and downloading them in a TSV (Tab-Separated Values) format. This component demonstrates typical React patterns like state management, side effects, conditional rendering, and event handling.

Component Structure

```javascript
function TeacherFileDetails() {
  ...
}
```

- function TeacherFileDetails() {...}: Defines a React functional component named TeacherFileDetails.

State and URL Parameters

```javascript
const [submission, setSubmission] = useState(null);
const { id } = useParams();
```

- [submission, setSubmission] = useState(null): Initializes a state variable submission with the default value null. The setSubmission function is used to update this state.
- { id } = useParams(): Retrieves the id parameter from the URL, which is expected to be part of the route where this component is used.

Data Fetching with useEffect

```javascript
useEffect(() => {
  axios.get(`http://127.0.0.1:5000/api/submissions/${id}`)
    .then(response => {
      setSubmission(response.data);
    })
    .catch(error => {
      console.error('Error fetching submission details:', error);
    });
}, [id]);
```

- useEffect(...): This hook runs the contained function when the component mounts and whenever the id changes. It's used here to fetch data from an API.
- axios.get(...): Makes a GET request to the backend (assumed to be running on localhost port 5000) to fetch submission details based on the id.
- .then(response => {...}): Handles the response from the API. The submission state is updated with the data received.
- .catch(error => {...}): Catches and logs any errors that occur during the API call.

File Download Function
1. Initialize File Content String
   - let fileContent = "Question\tModel Answer\tConcepts\n": Starts the file content with a header row containing column titles, separated by tabs (\t).

2. Iterate Over Teacher File Data
   - submission.teacher_file.forEach((item, index, array) => {...}): Iterates over each item in the teacher_file array from the submission state. The item represents each entry, index is the position of the entry in the array, and array is the entire array.

3. Construct File Content
   - let concepts = item.concepts ? item.concepts.join('\t') : '';: Joins the concepts array into a string separated by semicolons. If concepts is empty or undefined, it defaults to an empty string.
   - fileContent += `${item.question}\t${item.modelAnswer}\t${concepts};": Adds a line to fileContent for each item, consisting of the question, model answer, and concepts, separated by tabs.
   - if (index < array.length - 1) { fileContent += "\n"; }: Adds a newline character after each line except the last one.

4. Create and Download Blob
   - const blob = new Blob([fileContent], { type: "text/tab-separated-values;charset=utf-8" });: Creates a new Blob object containing the file content. The Blob object represents file-like data. The MIME type is set to "text/tab-separated-values;charset=utf-8".
   - const url = URL.createObjectURL(blob);: Creates a URL for the Blob object, which can be used as a hyperlink.

5. Create Link and Trigger Download
   - const link = document.createElement("a");: Creates an <a> (anchor) element.
   - link.href = url;: Sets the href attribute of the link to the Blob URL.
   - link.download = "teacher_file.tsv";: Sets the download attribute to suggest a filename for the downloaded file.
• document.body.appendChild(link);: Adds the link to the document body. This is necessary for the click event to work.
• link.click();: Programmatically clicks the link to start the download.
• document.body.removeChild(link);: Removes the link from the document body.
• URL.revokeObjectURL(url);: Releases the created Blob URL to free up memory.

Conditional Rendering for Loading State

```
if (!submission) return <div>Loading...</div>;
```

• if (!submission): This is a conditional check to see if the submission state is null (or falsified). Initially, submission is set to null, and it only gets updated once the data fetch is successful.
• return <div>>Loading...</div>: If submission is null, the function returns a JSX div element displaying the text "Loading...". This serves as a loading indicator to the user, informing them that the data is still being fetched.

Rendering Submission Details

```
return (
  <div className="teacher-file-details">
    <h2>Teacher File</h2>
    {submission.teacher_file && submission.teacher_file.map((item, index) => {
      <div key={index}>
        <p><strong>Question</strong>: {item.question}</p>
        <p><strong>Model Answer</strong>: {item.modelAnswer}</p>
        {item.concepts && (
          <p><strong>Concepts:</strong> {item.concepts.join(' , ')}<br /></p>
        )}
      </div>
    })}
    <button onClick={downloadFile}>Download Teacher File</button>
  </div>
);
```

• <div className="teacher-file-details">: A div element with a class name teacher-file-details. This class can be used to apply specific CSS styles to this div.
• <h2>Teacher File</h2>: A header element that displays the text "Teacher File".
• {submission.teacher_file && submission.teacher_file.map((item, index) => {...})}: This line is a combination of a conditional rendering and the map function.
• submission.teacher_file &&: This checks if submission.teacher_file exists and is truthy. It's a safeguard to ensure that submission.teacher_file is not null or undefined before trying to map over it.
• submission.teacher_file.map((item, index) => {...}): The map function iterates over each element in the submission.teacher_file array. For each element (item), it returns a JSX fragment. index is the current index of the array element in the iteration, used as a key for each child in the list.
• <div key={index}>: Each mapped item is wrapped in a div. The key prop is essential in a list for React to handle the re-rendering of elements efficiently.
- `<p><strong>Question</strong>: {item.question}</p>`: Displays the question. `item.question` accesses the question property of the current item (element of submission.teacher_file).
- `<p><strong>Model Answer</strong>: {item.modelAnswer}</p>`: Similarly, this line displays the model answer.
- `{item.concepts && (<p><strong>Concepts:</strong> {item.concepts.join(', ')}<p>))}: Another conditional rendering. It checks if `item.concepts` exists. If it does, it displays the concepts, joined by commas.
- `<button onClick={downloadFile}>Download Teacher File</button>`: A button element. The `onClick` attribute sets up a click event listener, which calls the `downloadFile` function when the button is clicked. This button allows the user to download the teacher's file.
function StudentFileDetails()

```javascript
function StudentFileDetails() {
    const [submission, setSubmission] = useState(null);
    const { id } = useParams();

    useEffect(() => {
        axios.get(`http://127.0.0.1:5000/api/submissions/${id}`)
            .then(response => {
                setSubmission(response.data);
            })
            .catch(error => {
                console.error('Error fetching submission details:', error);
            });
    }, [id]);

    const downloadFile = () => {
        let fileContent = "Question\tStudent Answer\n";

        submission.student_file.forEach((item, index) => {
            if (index < array.length - 1) {
                fileContent += "\n";
            }
            fileContent += `${item.question}\t${item.studentAnswer}`;
        });

        const blob = new Blob([fileContent], { type: "text/tab-separated-values; charset=utf-8" });
        const url = URL.createObjectURL(blob);

        const link = document.createElement("a");
        link.href = url;
        link.download = "student_file.tsv";
        document.body.appendChild(link);
        link.click();
        document.body.removeChild(link);
        URL.revokeObjectURL(url);
    };

    if (!submission) return <div>Loading...</div>;

    return (
        <div className="student-file-details">
            <h2>Student File</h2>
            {submission.student_file && submission.student_file.map((item, index) => {
                <div key={index}>
                    <p><strong>Question</strong>: {item.question}</p>
                    <p><strong>Student Answer</strong>: {item.studentAnswer}</p>
                </div>
            })
        </div>
        <button onClick={downloadFile}>Download Student File</button>
    );
}
```

Component Setup and State Management
The beginning of the StudentFileDetails function is almost identical to TeacherFileDetails. It initializes the state for storing submission data, retrieves the submission ID from the URL, and fetches submission details from a backend API using axios. The structure is as follows:

- `useState` to initialize the submission state.
- `useParams` to get the id from the URL.
- `useEffect` to fetch submission data from the backend when the component mounts or when the id changes.

**Data Download Function**
The downloadFile function in StudentFileDetails creates a downloadable file for the student's submission:

- It formats the content of `submission.student_file` into a tab-separated values (TSV) string.
- A Blob object is created from this string, and a URL is created for this blob.
- A temporary link element `<a>` is created and programmatically clicked to trigger the download.

**Conditional Rendering and JSX Return**
The JSX returned by the StudentFileDetails function renders the student file details:

- It checks if the submission data is available. If not, it shows a "Loading..." message.
- Once the data is available, it maps over `submission.student_file` to display each question and corresponding student answer.
- A download button is provided to download the student file in TSV format.

**Key Differences from TeacherFileDetails**

- The main difference is in the content it handles and displays: StudentFileDetails deals with student submissions, showing student answers instead of model answers and concepts.
- The download function and button label are adjusted to reflect that it's downloading the student file.
- The CSS class for styling might differ, as indicated by `className="student-file-details"`.

In summary, StudentFileDetails follows a similar pattern to TeacherFileDetails but is tailored to handle and present data related to student submissions. It effectively demonstrates how React components can be structured for similar yet distinct purposes, maintaining consistency in design and functionality.
function GradingFileDetails()

State Initialization

- function GradingFileDetails() { ... }: Defines a React functional component named GradingFileDetails.
- const [submission, setSubmission] = useState(null);: Initializes the submission state variable with a default value of null. setSubmission is the function used to update this state later.
- const { id } = useParams();: Utilizes the useParams hook from react-router-dom to extract the id parameter from the URL.

Fetching Data with useEffect

- useEffect(() => {
  axios.get('http://127.0.0.1:5000/api/submissions/${id}')
    .then((response) => {
      setSubmission(response.data);
    })
    .catch((error) => {
      console.error('Error fetching submission details:', error);
    });
  }, [id]);: This is a React hook that runs the contained function when the component mounts and whenever the id changes. The function makes an API call to fetch submission details.
- axios.get(http://127.0.0.1:5000/api/submissions/${id}'): Makes a GET request using axios to fetch submission data from the backend server. It uses the id obtained from the URL.
- .then(response => { setSubmission(response.data); }): On a successful API response, the submission state is updated with the data received.
- .catch(error => { console.error('Error fetching submission details:', error); }): Logs any error that occurs during the API call.

Calculate Scores Function

- const calculateScores = () => {
  if (submission && submission.details) {
    const totalScore = submission.details.reduce((acc, item) => acc + item.grade, 0);
    const totalPossibleScore = submission.details.length * 10;
    const finalGrade = totalPossibleScore > 0 ? (totalScore / totalPossibleScore * 100).toFixed(2) : "0.00";
    return { totalScore, totalPossibleScore, finalGrade }
  }
  return { totalScore: 0, totalPossibleScore: 0, finalGrade: "0.00" }
};

1. Function Definition
const calculateScores = () => {...}: This line defines a function named calculateScores. It's a constant arrow function with no parameters.

2. **Conditional Check**
   - if (submission && submission.details) {...}: The function first checks if the submission state is not null and if submission.details exists. This ensures that the data required for the calculation is available.

3. **Calculating Total Score**
   - const totalScore = submission.details.reduce((acc, item) => acc + item.grade, 0): This line calculates the total score by summing up the grades of each item in submission.details.
   - reduce is a JavaScript array method that applies a reducer function to each element of the array.
   - (acc, item) => acc + item.grade is the reducer function, where acc is the accumulator that holds the running total, and item is the current element being processed. item.grade is the grade for the current item.
   - 0 is the initial value for the accumulator.

4. **Calculating Total Possible Score**
   - const totalPossibleScore = submission.details.length * 10: This line calculates the total possible score based on the number of items in submission.details.
   - Each item is assumed to have a maximum score of 10, so the total possible score is the length of the submission.details array multiplied by 10.

5. **Calculating Final Grade**
   - const finalGrade = totalPossibleScore > 0 ? (totalScore / totalPossibleScore * 100).toFixed(2) : "0.00": This line calculates the final grade as a percentage.
   - It first checks if totalPossibleScore is greater than 0 to avoid division by zero.
   - If it is, the final grade is calculated by dividing totalScore by totalPossibleScore, multiplying by 100 (to convert to a percentage), and then formatting it to two decimal places using toFixed(2).
   - If totalPossibleScore is 0, it defaults the final grade to "0.00".

6. **Returning Scores**
   - return { totalScore, totalPossibleScore, finalGrade }: If the conditions are met, the function returns an object containing the totalScore, totalPossibleScore, and finalGrade.
   - return { totalScore: 0, totalPossibleScore: 0, finalGrade: "0.00" }: If the submission or submission.details is not available, the function returns default values for the scores and grade.
Download File Function

```javascript
const downloadFile = () => {
  let fileContent = ""
  if (submission && submission.details) {
    submission.details.forEach(item => {
      fileContent += Question: ${item.question}
      Student Answer: ${item.studentAnswer}
      Grade: ${item.grade}/10
      Model Answer: ${item.modelAnswer}
    });
    fileContent += "Total Score: ${totalScore}/${totalPossibleScore}\n"
    fileContent += "Final Grade: ${finalGrade}\n"
    const blob = new Blob([fileContent], { type: "text/plain; charset=utf-8" });
    const url = URL.createObjectURL(blob);
    const link = document.createElement("a");
    link.href = url;
    link.download = "grade.txt";
    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);
  }
};
```

The `downloadFile` function in the GradingFileDetails component handles the creation of a text file containing detailed grading information and triggers its download. This function showcases various JavaScript techniques including string manipulation, working with Blobs and URLs, DOM manipulation, and event handling, all of which are typical in complex React applications for handling file operations.

1. **Function Definition**
   - `const downloadFile = () => {...}`: Defines an arrow function named `downloadFile`. It is assigned to a constant and takes no parameters.

2. **Initializing File Content**
   - `let fileContent = "";`: Initializes a variable `fileContent` with an empty string. This variable will be used to accumulate the content for the file to be downloaded.

3. **Calculating Scores**
   - `const { totalScore, totalPossibleScore, finalGrade } = calculateScores();`: Calls the `calculateScores` function defined earlier to get the total score, total possible score, and final grade. These values are destructured from the returned object.

4. **Building File Content**
   - `if (submission && submission.details) {...}`: This conditional checks if submission and its details property are available.
   - `submission.details.forEach(item => {...})`: Iterates over each item in the submission.details array.
   - `fileContent += Question: ${item.question}
      Student Answer: ${item.studentAnswer}
      Grade: ${item.grade}/10
      Model Answer: ${item.modelAnswer}
   ;`: For each item, appends a formatted string containing the question, student answer, grade, and model answer to `fileContent`. 

5. **Appending Total Scores and Final Grade**
   - `fileContent += Total Score: ${totalScore}/${totalPossibleScore}\n;`: Adds the total score information to the `fileContent`. 

53
6. Creating a Blob and Download URL
   - const blob = new Blob([fileContent], { type: "text/plain; charset=utf-8" });: Creates a new Blob object containing the fileContent. The Blob represents data that doesn't reside in the JavaScript environment. Here, it's set to plain text format with UTF-8 encoding.
   - const url = URL.createObjectURL(blob);: Creates a URL for the created Blob. This URL is used to trigger the file download.

7. Creating a Download Link and Triggering Download
   - const link = document.createElement("a");: Creates a new anchor (<a>) element in the DOM. This element is used to trigger the file download.
   - link.href = url;: Sets the href attribute of the anchor element to the Blob URL.
   - link.download = "grade.txt";: Sets the download attribute to suggest a filename (grade.txt) for the downloaded file.
   - document.body.appendChild(link);: Adds the anchor element to the document body. This is necessary to make the link part of the document so it can be clicked.
   - link.click();: Programmatically clicks the link, triggering the file download.
   - document.body.removeChild(link);: Removes the anchor element from the document after triggering the download.
   - URL.revokeObjectURL(url);: Releases the created object URL to free up resources. Object URLs should be released when they are no longer needed.

Conditional Rendering for Loading State

```
if (!submission) return <div>Loading...</div>;
```

- if (!submission): This checks if the submission state is null. The submission state is initially set to null and is updated with the submission data once the API call is successful.
- return <div>Loading...</div>: If submission is null, indicating that the data is still being fetched or not available, the function renders and returns a div element displaying the text "Loading...". This serves as a placeholder until the data is loaded.

Score Calculation

```
const { totalScore, totalPossibleScore, finalGrade } = calculateScores();
```

- This line calls the calculateScores function defined earlier in the component.
- The function calculateScores calculates the total score, total possible score, and the final grade based on the details in the submission.
- The returned values are destructured into totalScore, totalPossibleScore, and finalGrade.

JSX Return for Rendering the Component
This return statement renders the JSX for the component. JSX is a syntax extension for JavaScript, used in React to describe the UI.

- `<div className="grading-file-details">`: Creates a div element with a class name grading-file-details. This class can be targeted with CSS for styling.
- `<h2>Grading File</h2>`: Renders an h2 heading element displaying the text "Grading File".
- This block checks if submission.details exists and maps over each item in it.
- submission.details.map((item, index) => {...}): The map function is used to iterate over each item in submission.details. For each item, it returns JSX elements displaying the item's details. index is used as a key for each mapped element, which is a React requirement for list items.
- `<div key={index}>`: Each item in the map is wrapped in a div with a key attribute. The key helps React identify which items have changed, been added, or been removed.
- It renders paragraphs `<p>` for each piece of information: the question, student's answer, grade, and model answer.
- These lines render paragraphs showing the total score and the final grade, calculated earlier by calculateScores.
- This creates a button element with an onClick event listener.
- When the button is clicked, it triggers the downloadFile function, allowing the user to download the grading details.
function SubmissionDetails()
The SubmissionDetails component acts as a hub for accessing different parts of a submission. It fetches the submission data, displays a loading message until the data is available, and then provides navigational links to view various files associated with the submission. This component is an example of combining data fetching, conditional rendering, and navigation within a React application.

State and URL Parameters

- [submission, setSubmission] = useState(null): Initializes a state variable submission with a default value of null. The setSubmission function will be used to update this state.
- { id } = useParams(): Retrieves the id parameter from the URL. This ID is used to fetch the specific submission details.

Data Fetching with useEffect

- useEffect: This hook is used for performing side effects in the component. In this case, it's fetching data from an API.
- axios.get(...): Makes a GET request to the backend to fetch details of the submission identified by id.
- .then(response => { setSubmission(response.data); }): Upon a successful response, the submission state is updated with the data received from the backend.
- .catch(error => { ... }): Catches and logs any errors that occur during the API call.

Conditional Rendering for Loading State

- This line checks if the submission state is null, indicating that the data is still being fetched or not available. If so, it renders and returns a div element displaying the text "Loading...".

JSX Return for Rendering the Component
The return statement contains the JSX to be rendered by this component.

This div wraps the entire content of the component and has a class submission-details for potential CSS styling.

- An h2 heading element displaying the text "Submission Files".
- These lines create three h3 elements, each containing a Link component from react-router-dom.
- Link: This component is used to navigate programmatically in a React Router application. It renders an anchor (<a>) tag under the hood.
- to: The to prop in each Link defines the path to navigate to when the link is clicked. These paths are constructed to navigate to the respective teacher file, student file, and grading file views for the current submission.
- className: Applies CSS classes for styling the links.
function App()

State Initialization in React

```javascript
function App() {
    const [teacherFile, setTeacherFile] = useState(null);
    const [studentFile, setStudentFile] = useState(null);
    const [score, setScore] = useState(null);
    const [totalScore, setTotalScore] = useState(null);
    const [totalPossibleScore, setTotalPossibleScore] = useState(null);
    const [details, setDetails] = useState(null);
    const [submissions, setSubmissions] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState(null);
}
```

1. `const [teacherFile, setTeacherFile] = useState(null);`
   - Initializes a state variable teacherFile with a default value of null.
   - setTeacherFile is a function to update teacherFile. This state likely holds the file data for a teacher's input file.

2. `const [studentFile, setStudentFile] = useState(null);`
   - Similar to teacherFile, this line initializes a state variable studentFile with a default value of null.
   - setStudentFile is the function to update studentFile, which probably holds the student's input file data.

3. `const [score, setScore] = useState(null);`
   - Initializes a state variable score with null. This state is likely used to store the final calculated score or grade of the assessment.

4. `const [totalScore, setTotalScore] = useState(null);`
   - Initializes a state variable totalScore with null. This could be used to store the total score achieved in an assessment.

5. `const [totalPossibleScore, setTotalPossibleScore] = useState(null);`
   - Initializes totalPossibleScore with null. This state might represent the total possible or maximum score for the assessment.

6. `const [details, setDetails] = useState(null);`
   - Initializes a state variable details with null. This state is probably meant to store detailed information or results of the assessment.

7. `const [submissions, setSubmissions] = useState([]);`
   - Initializes submissions with an empty array. This state likely holds a list of all submissions, possibly fetched from a backend server.

8. `const [loading, setLoading] = useState(false);`
   - Initializes a loading state with the boolean value false. This is typically used in React applications to handle asynchronous operations like API calls, indicating whether the application is in the process of loading data.

9. `const [error, setError] = useState(null);`
   - Initializes an error state with null. This state is used to store error messages or flags, useful for error handling and displaying error messages to the user.
useEffect Hook for Fetching Submissions

```javascript
useEffect(() => {
  axios.get('/api/submissions')
    .then(response => {
      setSubmissions(response.data);
    })
    .catch(error => {
      console.error('There was an error fetching the submissions', error);
    });
}, []);
```

1. **useEffect Hook**
   - `useEffect(() => {...}, []);`: This hook is used for performing side effects in the component. The empty dependency array ([]) means this effect runs once when the component mounts, similar to `componentDidMount` in class components.

2. **Fetching Data with Axios**
   - `axios.get('/api/submissions')`: Makes a GET request to the /api/submissions endpoint. Axios is a promise-based HTTP client used to make HTTP requests.

3. **Handling the Response**
   - `.then(response => {...})`: Handles the response from the API call.
   - `setSubmissions(response.data);`: Updates the submissions state with the data received from the backend.

4. **Error Handling**
   - `.catch(error => {...})`: Catches any errors that occur during the API call.
   - `console.error(...):` Logs the error to the console.

Event Handlers for File Inputs

```javascript
const onTeacherFileChange = (e) => {
  setTeacherFile(e.target.files[0]);
};

const onStudentFileChange = (e) => {
  setStudentFile(e.target.files[0]);
};
```

1. **onTeacherFileChange Function**
   - This is an event handler for changes to the teacher file input.
   - `e.target.files[0]`: Accesses the first file from the file input event `e`. In file inputs, `e.target.files` is a `FileList` object representing the files selected by the user.

2. **onStudentFileChange Function**
   - Similarly, this is an event handler for the student file input.
   - `setStudentFile(...):` Updates the studentFile state with the file chosen by the user.
fetchData Function

```javascript
const fetchData = async () => {
  try {
    setLoading(true);
    setScore(null);
    const formData = new FormData();
    formData.append('teacherFile', teacherFile);
    formData.append('studentFile', studentFile);
    const response = await fetch('http://127.0.0.1:5000/api/process_files', {
      method: 'POST',
      body: formData,
    });
    const data = await response.json();
    setScore(data.grade);
    setTotalScore(data.totalScore);
    setTotalPossibleScore(data.totalPossibleScore);
    setDetails(data.details);
    setError(null);
  } catch (error) {
    setError('Failed to fetch data');
  } finally {
    setLoading(false);
  }
};
```

- const fetchData = async () => {...}: Declares an asynchronous function named fetchData. The async keyword allows the use of await within the function to handle promises in a more synchronous manner.

Try-Catch-Finally Structure

1. **Try Block**
   - Initiates a try block to execute code that might throw an exception.

2. **Setting Initial States**
   - setLoading(true): Sets the loading state to true, indicating that an asynchronous operation is in progress.
   - setScore(null): Resets the score state to null to prepare for new data.

3. **Preparing FormData**
   - const formData = new FormData(): Creates a new FormData object, which is used to construct a set of key/value pairs representing form fields and their values, suitable for submission via the fetch API.
   - formData.append('teacherFile', teacherFile): Appends the teacher file to the formData.
   - formData.append('studentFile', studentFile): Appends the student file to the formData.

4. **Making an API Request**
   - const response = await fetch('http://127.0.0.1:5000/api/process_files', {...}): Makes an asynchronous POST request to the backend server with the URL 'http://127.0.0.1:5000/api/process_files'.
   - The body of the request is set to the formData containing the teacher and student files.
5. Handling the Response
   - const data = await response.json();: Parses the JSON response from the server.
   - setScore(data.grade): Updates the score state with the grade from the response.
   - setTotalScore(data.totalScore): Updates the totalScore state.
   - setTotalPossibleScore(data.totalPossibleScore): Updates the totalPossibleScore state.
   - setDetails(data.details): Updates the details state with detailed assessment information.
   - setError(null): Resets any errors by setting error to null.

6. Catch Block for Error Handling
   - catch (error) {...}: Catches any exceptions thrown in the try block.
   - setError('Failed to fetch data'): Sets the error state to an error message, indicating that the data fetch failed.

7. Finally Block
   - finally {...}: This block executes regardless of the result of the try-catch blocks.
   - setLoading(false): Sets the loading state to false, indicating that the asynchronous operation is complete.

fetchSubmissions Function

```javascript
const fetchSubmissions = async () => {
    try {
        const response = await fetch('http://127.0.0.1:5000/api/submissions');
        const data = await response.json();
        setSubmissions(data);
    } catch (error) {
        console.error('Error fetching submissions:', error);
    }
};
```

1. Function Definition
   - const fetchSubmissions = async () => {...}: This defines an asynchronous function called fetchSubmissions. The async keyword allows the use of await inside the function.

2. Fetching Data
   - const response = await fetch('http://127.0.0.1:5000/api/submissions');: Makes an asynchronous call to the backend server to fetch a list of submissions. The await keyword pauses execution until the promise is resolved.

3. Processing the Response
   - const data = await response.json(): Converts the response from the server into a JSON object. Again, await is used to wait for the promise resolution.
   - setSubmissions(data): Updates the submissions state with the fetched data. This will trigger a re-render of any UI components that depend on this state.

4. Error Handling
   - The try-catch block is used for error handling.
   - If an error occurs during the fetch operation, it is caught and logged to the console with console.error.

generateDownloadLink Function
1. **Function Definition**
   - `const generateDownloadLink = () => {`: Defines a function for generating a downloadable link.

2. **Building the Download Content**
   - `let content = '';`: Initializes a string variable to accumulate the content for download.
   - `if (details) {`: The if (details) {...} block checks if the details state is available.
   - `details.forEach(detail => {`: Iterates over each detail item and appends formatted strings to content, including each question, student answer, grade, and model answer.

3. **Appending Total Scores and Final Grade**
   - `content += `Total Score: ${totalScore}/${totalPossibleScore}\n`; Adds the overall total score and final grade to the content.

4. **Creating Blob and Download URL**
   - `const blob = new Blob([content], { type: 'text/plain' });`: Creates a new Blob object from the content string. Blobs represent immutable raw binary data.
   - `const url = URL.createObjectURL(blob);`: Creates an object URL for the Blob. This URL can be used in a link to download the file.

5. **Return Download URL**
   - `return url;`: Returns the URL that points to the created Blob, which can be used to trigger a file download in the UI.
The JSX structure in the App function creates an interactive web interface for an automated short answers assessment application. It allows users to upload files for grading, view the results, download grading details, and navigate to previous submissions. The use of React Router enhances the navigational capabilities of the application, while conditional rendering and event handling make the interface dynamic and responsive to user interactions and data changes.

This structure effectively creates a single-page application (SPA) where different components (SubmissionDetails, TeacherFileDetails, StudentFileDetails, GradingFileDetails) are rendered based on the current URL. React Router handles the navigation and rendering logic, providing a seamless user experience where the page does not reload when navigating between different views. This approach is typical in modern web applications to create an app-like experience in the browser.

Clarification

return (  
<Router>  
  // ... JSX code ...  
</Router>  
);
1. **Router Component**
   - `<Router>`: This component from react-router-dom is used to wrap the application's routing logic. It enables the use of routing-related components and hooks within the app.

2. **Main App Container**
   - `<div className="App">`: A div element that serves as the main container for the application. It has a class App for potential CSS styling.

3. **Application Heading**
   - `<h1>`Automated Short Answers Assessment</h1>: An h1 heading displaying the title of the application.

4. **Routes for Navigation**
   - `<Routes>`: A component that wraps the route definitions. It decides which component should be rendered based on the current URL.

**Route for Home Path**

```
<Route path="/" exact element={...} />
```

- This Route defines the UI for the root path ("/"). The exact prop ensures that this component is rendered only when the URL matches the path exactly.

**Home Path Content**
The element prop of the Route contains JSX that defines the content for the home path. This content includes:

- File input fields for uploading the teacher and student files.
- Buttons for triggering grading and fetching previous submissions.
- Display areas for errors, scores, download links, and a list of previous submissions.

**File Upload Inputs**

```
<div>
  <label className="file-label">
    <strong>Teacher File</strong>:
    <input type="file"OnChange={onTeacherFileChange} className="input-file" />
  </label>
</div>
```

```
<div>
  <label className="file-label">
    <strong>Student File</strong>:
    <input type="file" onChange={onStudentFileChange} className="input-file" />
  </label>
</div>
```

- Two div elements each contain a label and an input of type file.
- `onChange={onTeacherFileChange} and onChange={onStudentFileChange}`: These are event handlers that update the component's state when a file is selected.
Grading and Submission Fetch Buttons
<button onClick={fetchData} disabled={loading || !teacherFile || !studentFile}>...</button>
<button onClick={fetchSubmissions} disabled={loading} className="small-button">...</button>

- These buttons trigger the fetchData and fetchSubmissions functions, respectively.
- The disabled attribute is used to disable the buttons based on certain conditions (like if files are not selected or if an operation is in progress).

Error, Score, and Loading Indicators
{error && <div className="error-message">{error}</div>}
{score !== null && (... )}
{loading && <div className="loading-spinner"></div>}

- Conditional rendering is used to display error messages, scores, and a loading indicator based on the component's state.

Previous Submissions List
<div>
{submissions.map((submission, index) => {
  <div key={index}>
    <Link to={`/submissions/${submission._id.$oid}`}>...</Link>
  </div>
})}
</div>

- This section maps over the submissions state and renders a list of links to individual submission details.

Route Definitions

<table>
<thead>
<tr>
<th>Line</th>
<th>Route Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>339</td>
<td>&lt;Route path=&quot;/submissions/:id&quot; element={&lt;SubmissionDetails /&gt;} /&gt;</td>
</tr>
<tr>
<td>340</td>
<td>&lt;Route path=&quot;/submissions/:id/teacher_file&quot; element={&lt;TeacherFileDetails /&gt;} /&gt;</td>
</tr>
<tr>
<td>341</td>
<td>&lt;Route path=&quot;/submissions/:id/student_file&quot; element={&lt;StudentFileDetails /&gt;} /&gt;</td>
</tr>
<tr>
<td>342</td>
<td>&lt;Route path=&quot;/submissions/:id/grading_file&quot; element={&lt;GradingFileDetails /&gt;} /&gt;</td>
</tr>
</tbody>
</table>

1. Submission Details Route
   - This route matches the path /submissions/:id, where :id is a path parameter representing the ID of a submission.
   - element={<SubmissionDetails />}: Renders the SubmissionDetails component when the URL matches this route. The component is responsible for displaying details of a specific submission.

2. Teacher File Route
   - Matches the path /submissions/:id/teacher_file.
   - Renders the TeacherFileDetails component, which is likely designed to show details of the teacher's file associated with the submission identified by :id.
3. **Student File Route**
   - Matches the path /submissions/:id/student_file.
   - Renders the StudentFileDetails component to display details of the student's file for the specified submission.

4. **Grading File Route**
   - Matches the path /submissions/:id/grading_file.
   - Renders the GradingFileDetails component, which likely provides a detailed breakdown of the grading for the submission.

Closing Tags and Summary
- ```<Routes>`: Closes the Routes wrapper.
- ```</div>`: Closes the div with the class App.
- ```<Router>`: Closes the Router component.

Export Statement
```javascript
export default App;
```

1. **export Keyword:**
   - This is used to export functions, objects, or primitive values from a module so that they can be used by other programs with the import statement.

2. **default Keyword:**
   - The default keyword is used to specify that the value being exported will be the default export from this module. Each module can have only one default export.

3. **App:**
   - This refers to the App function/component you defined in your file. It's the primary React component that represents the entire UI of your application.

Significance of export default App:
- By exporting App as the default export of the file, you make this component available for import in other JavaScript files. For instance, in a typical React application, this App component would be imported in the index.js file to be rendered into the DOM, usually inside a root DOM node.
- The default export is particularly useful when there's only one primary export from a module, as is common with React components. When importing a default export, you don't need to use curly braces and can name the import whatever you like. For example, you can import App in another file like so:

  ```javascript
  import App from './App';
  ```

  This statement imports the App component from the App.js file, assuming the file is named App.js.
- In React applications, the App component often acts as the root component that contains the overall layout and routing information of the application.
In summary, export default App; is an ES6 module syntax for exporting the main React component of your application, enabling it to be imported and used in other parts of your application, such as rendering it to the DOM in your entry point file (commonly index.js).
5.4 Dataset

In creating the dataset for our automated grading system, we took a purposeful step towards simplicity and precision. We began with a dataset structure that includes only two fields: the first containing the question, its model answer and concepts, and the second having the same questions, but also answers and a score related to the answer. We’ve been meticulous in this design; the model answers encompass all related concepts, ensuring a perfect score and serving as a standard for what a comprehensive answer should contain.

The concept field is more than just a list; it's the blueprint of knowledge that students are expected to demonstrate. We separated each concept with a semicolon, a choice that allows our processing model to recognize and retain complex terms as coherent wholes. This method is crucial—it's the difference between recognizing the multifaceted concept of "cellular respiration" as a single entity versus fragmenting it into less meaningful parts, which could inadvertently skew the grading scale towards inaccuracies.

Looking ahead, we have decided to not use the dataset containing ten scores for each question. We have instead chosen to only use the concept-based scoring, and through our machine learning
model, we were able to achieve greater accuracy this way. This way of using the dataset is much more flexible since it allows for less setup and preprocessing to use the data.

In the future, it would be much easier to incorporate a dataset that does not need the teacher file, but instead just includes only the questions given as well as the student’s answers. By developing the machine learning model further, we could achieve the same results, if not better, with the use of a smaller and more flexible dataset.

In the image above, we can see an example of how the score-based dataset could become clunky and take a lot of time to develop. For each question, we would have ten answers, but in a more practical use case, a school would probably need more answers and scores to train from to ensure it is truly accurate when grading. This would take an immense amount of time, especially considering coming up with these example answers and figuring out the dilemma of who is to grade these answers. If you have only a few people grade the answers, it could limit other possible correct explanations, so you would need to have many people grading each answer to get a rough estimate of what each grade should be. Therefore, this is why we chose to use the other dataset.
6 Developer Manual

6.1 Overview

In the dynamic landscape of education, the imperative for efficient and accurate assessment practices has never been more pronounced. The evaluation of students' written responses, particularly in the realm of short answers, poses a formidable challenge for educators seeking comprehensive insights into their students' comprehension. This challenge prompted the inception of our project, titled "Automated Students' Short Answers Assessment using NLP and LLM," at the intersection of cutting-edge technology and the pedagogical imperative, poised to redefine the landscape of student assessments.

6.2 Purpose of the Developer Manual

This developer manual serves as a comprehensive guide for those involved in the creation, maintenance, and optimization of the automated tool designed for the formative assessment of students' short answers. It encapsulates the intricate technological processes, the theoretical underpinnings, and the practical considerations necessary for the successful implementation and evolution of this innovative solution.

6.3 Target Audience

This manual is tailored for a diverse audience encompassing developers, data scientists, educators, and stakeholders invested in the advancement of educational technology. Developers will find detailed insights into the algorithms, methodologies, and codebase, while educators will gain a nuanced understanding of how this tool can enhance their assessment practices. Data scientists will delve into the intricacies of natural language processing (NLP) and large language models (LLM) employed, and stakeholders will grasp the broader impact and potential applications of the project.

6.4 Prerequisites

Before embarking on the installation and configuration process, ensure that your Windows system meets the following prerequisites:

- Operating System: Windows 7, 8, or 10.
- Processor: Intel Core i5 or equivalent.
- RAM: Minimum 1 GB.
- Disk Space: Ensure at least 1 GB of free disk space for installation.
- Internet Connection: A stable internet connection is required for downloading necessary components and updates.
6.5 Installation

Follow these step-by-step instructions to install the Automated Students' Short Answers Assessment tool on your Windows system:

Download the VSCode:

- Visit the website of VSCode and download it [Download Visual Studio Code - Mac, Linux, Windows] in Windows version
- Make a new folder in VS code (I named mine Automated Students' Short Answers Assessment)

Download the Installer:

- Download the source code we provided into tar
- If you are using tar, use this line to untar > tar -xvf cs4624-automated-students-short-answers-assessment-main.tar
- Using command > cd cs*/a*/m*/src to get into the src
- Download the python from Microsoft Store and ten Python extensions

- Download the flask
  > pip install flask
- Download the flaskCors
  > pip install flask_Cors
- Download the sentence_transformers
  > pip install sentence_transformers

If you see the error like this:
ERROR: Could not install packages due to an OSError: [Errno 2] No such file or directory:
'C:\Users\Westin\AppData\Local\Packages\PythonSoftwareFoundation.python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\transformers\models\deprecated\trajectory_transformer\convert_trajectory_transformer_original_pytorch
ch_checkpoint_to_pytorch.py

Here is the solution:
Open the Group Policy Editor by pressing Win + R, typing gpedit.msc, and hitting Enter.
Double-click on the "Enable Win32 long paths" policy.
Set it to "Enabled" and click "OK".
Then type this in terminal
```bash
pip uninstall sentence_transformers
pip install sentence_transformers
```
- Download the pymongo
  ```bash
  > pip install pymongo
  ```
- Install Node.js and npm[Node.js (nodejs.org)]
  ```bash
  > npm install
  ```
- Download the MongoDB [https://www.mongodb.com/try/download/compass]
  And download this one

**MongoDB Compass Download (GUI)**

Easily explore and manipulate your database with Compass, the GUI for MongoDB.
Intuitive and flexible, Compass provides detailed schema visualizations, real-time performance metrics, sophisticated querying abilities, and much more.

Please note that MongoDB Compass comes in three versions: a full version with all features, a read-only version without write or delete capabilities, and an isolated edition, whose sole network connection is to the MongoDB instance.

For more information, see our documentation pages.
Download the MongoDB server [Download MongoDB Community Server | MongoDB]

Run the Installer:

Start the MongoDB Compass, the default should be like this:

Now change the 27017 to 12345
Then go to the address of where you installed your MongoDB then go into server\version of server\bin/, use my MongoDB address as an example
> C:\Program Files\MongoDB\Server\7.0\bin
Click the mongod.CFG
Change the port number to 12345
So it would be like:

```plaintext
port: 12345
bindIP: 125.0.0.1
```

Finally, click the connect on MongoDB Compass
- Start VSCode and using command to get into the src
  > cd cs*/a*/m*/src
- Start the project first run back-end server
  > python App.py
• Then start WebApp
  > npm start
• When you see the user interface, it means you successfully launched it!
  Good luck!
7 User Manual

7.1 Instruction

- Purpose of the Manual: This user manual instruction aims to guide users through the process of using the Automated Students' Short Answers Assessment Tool. The manual includes user interface overview, use environment, use cases/ tasks supported, tutorials on use.
- About the Tool: The Automated Students' Short Answers Assessment Tool includes the following features: display the brief score in Webapp, download the grade with details, view the previous submissions, download the previous submissions. The advantage of the product is that it helps educators grade quickly and accurately. At the same time students can get feedback and improve their assignments.

7.2 Use Environment

This product meets most usage environments on the market, including Windows, Linux, and Mac systems. The ideal environments for this product include grading short answer tests, homework assignments. However, for some text-heavy assignments, grading time may become longer. Also, it is not possible to grade drawing assignments.

7.3 User Interface Overview
This is an overview of the Automated Students' Short Answers Assessment Tool homepage. On this page, the instructor/professor can provide the teacher's file and save. Then students can upload their assignments and click the Grade button. Educators can also view previous submissions in the homepage.

7.4 Use Cases/ Tasks Supported

- Display the Brief Score in Web App:

![Automated Short Answers Assessment](image)

**Figure 4: After Pressing the Grade Button Page**

After both the teacher and the student have submitted their documents, the student can click the GRADE button. At the bottom of the page the student's grade will appear, based on the differences between the student and teacher files.

- Download grade with details
Once grading is complete, students have the option to download submission details. In the downloaded file students can check the difference between their answers and the teacher's answers to improve their work.
• View the Previous Submissions

![Automated Short Answers Assessment](image)

**Figure 7: View Previous Submissions Page**

If a student submits an assignment multiple times, the student can see previous submissions under the Download Grade button. Click to see the submission history.

• Download the Previous Submissions
Figure 8: Teacher File Page

Students can also download grade files when viewing their submission history.

7.5 Tutorials on Use

Step 1: Log In to the System

Open the ASSAA (Automated Students' short answers assessment) website.
Log in to the system using your credentials.

Step 2: Upload Files

Select the "Upload Files" option.
Choose the file containing questions, standard answers, and concepts (typically a single file) and upload it.
Select the file containing questions and student answers and upload it.

Step 3: Questions, Standard Answers, and Concepts File

- The questions, standard answers, and concepts file typically includes:
  - Questions: The questions or tasks provided by the teacher.
  - Standard Answers: The correct answers to the questions.
  - Concepts: Relevant concepts or standards related to the questions.

Step 4: Student Answers File

- The student answers file includes answers provided by students.
Step 5: Submit and Get Feedback

After uploading the files, click the "Submit" button. The system will automatically assess student answers and generate a score for each student response. You can review the score feedback for student answers.
8 Lessons learned

Throughout the journey of developing our automated grading system, we've encountered a myriad of challenges and triumphs. This section reflects on our experience and distills it into wisdom for future endeavors.

Timeline/Schedule:

1. **Initial Planning:** Project began with enthusiasm, setting ambitious goals for completion. General concepts and ideas for the project were thrown around, but the final decisions would happen further down the road.
2. **Mid-Project Adjustments:** Encountered inevitable delays requiring schedule revisions such as changes in the dataset, the large language model, and how the frontend and backend should connect.
3. **Final Stages:** Accelerated progress as pieces fell into place, although closer to the deadline than anticipated.

Problems and Solutions:

1. **Data Complexity:** Initially, our model struggled with the nuanced language of student answers. **Solution:** We refined our NLP algorithms and enhanced our dataset to improve comprehension and context analysis.
2. **Integration Hurdles:** Frontend and backend communication struggled as we were learning this integration during the project. **Solution:** We reached out to our client as well as educators to learn more about connecting the two. This was a large step forward.
3. **User Experience Issues:** Early user feedback within our group highlighted a non-intuitive interface. **Solution:** We reevaluated our design, placing user journey and simplicity at the forefront of our interface redesign.

Unforeseen Insights:

1. **The Importance of Flexibility:** Our rigid initial timeline didn't account for iterative testing and feedback incorporation. We learned to build in time for these critical stages.
2. **Cross-Disciplinary Collaboration:** Working closely with educators not only provided domain-specific insights but also highlighted the importance of usability in educational technology.

Future Work:

1. **Scalability:** Our system currently caters to Computer Science related questions. Future work includes expanding to all STEM fields as well as testing for questions from other fields and majors, requiring a more versatile range of answer analysis.
2. **Adaptive Learning Integration:** We aim to not just grade but also provide tailored feedback to students, paving the way for an adaptive learning platform.
3. **Community Engagement:** To continually improve, we hope our project can be expanded upon in the future, inviting contributions and critiques from the wider educational technology community.

4. **File Type Accessibility:** We would like our web application to expand on the idea of accessibility, allowing for more file types to be uploaded and processed instead of the standard .tsv we currently have implemented.

5. **In-App Typing:** To allow for a more streamlined workflow, we would like the users to be able to type their answers in for a question straight into a text box on our website, and use this answer as the one the ML model will grade.

Reflecting on this experience, we've learned that the journey is as important as the destination. The lessons learned here will guide not only the future development of this project but also our approach to problem-solving and innovation in all our future endeavors.
9 Acknowledgement

Mohamed Farag
mmagdy@vt.edu
Dr. Farag is a research associate in the Center for Sustainable Mobility (CSM):
Ph.D. degree in Computer Science from Virginia Tech in 2016
M.Sc. degree in Computer Science from the Arab Academy for Science, Technology, and Maritime Transport (AAST) in Alexandria, Egypt, in 2010
10 Reference


