PennState Behrend

Building A Question Answering System using Tweets
Team 23

Edward Lach: Major in SWENG

Luke Westfall: Major in SWENG

Shiang Jin, Chin: Major in SWENG

Zachery Smith: Major in SWENG

Course Instructor: Naseem Ibrahim
Faculty Advisor: Naseem Ibrahim

Industry Sponsor: Penn State Behrend - School of Engineering.
Project Mentor: Zhifeng Xiao

A capstone project report submitted to the faculty of Computer Science and Software Engineering Pennsylvania State University, Erie

May 2022
(Version 5.0)

1. Abstract

Social media such as Twitter are increasingly used by people as a primary source of news and current events, as well as a primary means to disseminate information they have. Hence, the ability of a software system to accurately answer questions about social media posts would be instrumental to the effectiveness of many applications that rely on it. However, Question Answering based on social media texts is still a challenging task in natural language processing, where currently no machine learning model can perform as best as humans. Training of the required model is also limited by the number of training data available. The existing situation leads to the following needs from the client, the primary need is to develop a robust machine learning model that can perform as good as a human on the aforementioned task, the secondary need is to have a web-based application that connects the trained model to the public, while enable crowdsourcing to evaluate model performance and gather new tweet, question and answer sets from the user for continuous training of the model.

Set with the aim to fulfill the client's needs, the team has trained a baseline machine learning model based on BERT, which achieved an average score of above 70% on the three-performance metrics. The team also developed a front-end web application connecting to the back-end web API which allows the visitor to interact with the baseline model. The main features of the system included the capability for the visitor to submit a new tweet-question pair or ask a new question based on a random tweet selected from the database. The system then provides a prediction answer from the model. Other features implemented includes allowing the visitor to provide feedback on whether the answer is correct and provide an alternative answer if the answer is incorrect. Other implemented features are summarized in section 11.1.

The potential benefits and impacts of the system can be summarized in two aspects: the system can help gather more tweet-question-answer triplet data that can help enlarge the current dataset, and the development and continue training process of the baseline model help provide insights into potential improvements for the model. The improved model can then be adapted and used in broader applications such as by government agencies to automatically monitor tweets in response to natural disasters and emergencies, or commercial applications to monitor trending events on social media that could influence financial markets.

This report provides the specifications, design, and context for this project. The report started with the problem statement in section 3 and continue to section 4 which documented the requirement engineering process and outcome. Section 5 of this report discussed the exploratory studies related to the machine learning model and documented the hyperparameter tuning results and potential impact of this project. Section 6 outlines the architectural design and structural design for this project, as well as additional design details for the User Interface, behavioral design, design patterns used, design alternatives, and decision rationale. Section 7 lists all applicable programming languages & tools, coding conventions and code version control, implementation alternatives, and decision rationale. Section 8 listed the test automation framework, test case design for test suites, unit test cases, integration test cases, and the test case execution report. Section 9 discusses the challenges faced by the team during the project development. Section 10 includes the instructions for system deployment. Section 11 presents the conclusions of this project. Section 12 lists all the references used in this report. The additional details for the use-cases, functional and non-functional user requirements, and functional and non-functional system requirements, test cases, and test executions are attached in the appendix.

2. Report Revision History

2.1. Changes in Version 1.5

A summary of all the changes for Version 1.5 of this document is provided below in **Table 2.1**.

Table 2.1: Change log for Version 1.5

Section	Change Log
1	Added the third paragraph discussing the structure of the report to the abstract
2	Added the change log for version 1.5
3	Revised section 3.2 Needs, section 3.3 Objectives. Updated citations to correct format.
4	Rewrote the user groups in Section 4.1.2 to clarify System Administrators as primary users. Removed UC-001 and renumber all other use cases accordingly in Section 4.1.3 Update user and system functional requirements definitions in Section 4.1.3.3, 4.2.1.1, 4.3. Converted non-functional system requirements SP-01-04, SP-02-02, SP-02-03, SP-02-04, SP-07-01, SP-07-2 from Report 1.0 into functional system requirements SF-P-1 to SF-P-6 Updated user and system non-functional requirements definitions and numbering in Sections 4.1.4 and 4.2.2. Removed non-functional user requirement UO-02 (redundant with UP-07). Add Information privacy requirements on UE-02, UE-02-01, SP-P-7 and UC-001 Update requirements trace tables in section 4.3
5	Combined section 5.1 & section 5.2 on relevant techniques and packages. Revised section 5.2 Broader Impacts. Updated citations to correct format.
6	Updated package diagram to include hyperparameter tuning in the Machine Learning Pipeline. Added to the explanation of the system architecture to clarify each of the steps in the Machine Learning Pipeline and to clarify the packages fit within the overall system.
7	Updated citations to correct format. Section 7.4.3 was added to explain the decision between popular web frameworks Angular and React.
8	No revision
9	Elaborated more on section 9.1 challenges faced in requirement engineering.
10	No revision
11	No revision

12	No revision
Appendix U	Remove UC-001 and renumber all other use cases accordingly Simplify Use Case tables
Appendix R	Update user and system functional requirements cards according to section 4.1.3 and section 4.2.1 and the use cases. Update user and system functional requirements cards according to section 4.1.4 and section 4.2.2 and the use cases. Renumber the requirements cards.

2.2. Changes in Version 2.0

A summary of all the changes for Version 2.0 of this document is provided below in **Table 2.2**.

Table 2.2: Change log for Version 2.0

Section	Change Log
1	Updated the structure of the report described in the Abstract section
2	Added Change log for Version 2.0
3	No revision.
4	Updated UC-002 and UC-006 in section 4.1.3.2 for the implementation details.
5	No revision
6	Added contents for section 6.2 Structural Design, section 6.3 User Interface Design, section 6.4 Behavioral Design, and section 6.5 for Design Alternatives and Decision Rational.
7	Added section 7.4.3 for React vs Angular implementation choice
8	Completed section 8, excluding 8.2.4, 8.2.5, 8.3.3, and 8.3.4
9	Added Challenges faced during the system development in section 9.2.
10	Updated section 10.1.1 on instructions to set up the development environment.
11	No revision
12	Added new reference for this revision
Appendix U	Updated UC-002, UC-006 and UC-007 for the implementation details.
Appendix R	Update requirements tables to include related test cases

Appendix T	Add test suite and test case tables for TC-001 thru TC-017
Appendix TE	Add test execution tables for TC-001 thru TC-017

2.3. Changes in Version 2.5

A summary of all the changes for Version 2.5 of this document is provided below in **Table 2.3**.

Table 2.3: Change log for Version 2.5

Section	Change Log
1	No revision
2	Added Change log for Version 2.5
3	No revision.
4	No revision
5	No revision
6	● Updated style in section 6.2.2. Updated sequence diagram for figure 20 & 21 ● Updated Form State Diagram and added explanation of state changes to section 6.4.1 ● Updated Diagram 6.2.3, 6.2.4 to be more readable ● Updated font size in all sequence diagrams ● Updated font size for some of the texts
7	No revision
8	Add references to tables in Appendix T and Appendix TE
9	No revision
10	No revision
11	No revision
12	No revision
Appendix U	No revision
Appendix R	Add reference to TC-016 to SF-A-2
Appendix T	Add reference to SF-A-2 in TC-016
Appendix TE	No revision

2.4. Changes in Version 3.0

A summary of all the changes for Version 3.0 of this document is provided below in **Table 2.4**.

Table 2.4: Change log for Version 3.0

Section	Change Log
1	Updated the structure of the report described in the Abstract section
2	Added Change log for Version 3.0
3	No revision
4	No revision
5	● Renumber sections 5.2 to 5.3 ● Include section 5.2 to explain the fine-tuning Process for the Machine Learning Models. Contents include exploring the TweetQA dataset, the preprocessing requirements, and processes, and the hyperparameters available for tuning.
6	● Renumber section 6.5 to 6.6 ● Add section 6.5 “Design Patterns”
7	No revision
8	Added additional 2 Test suites and associated test cases
9	● Added new challenges faced during the system development in section 9.2. ● Added some open issues and ideas for solutions in section 9.3
10	No revision
11	No revision
12	Added new references used in section 5.2
Appendix U	No revision
Appendix R	No revision
Appendix T	Added additional 2 Test suites and associated test cases
Appendix TE	Added test execution reports for additional test cases

2.5. Changes in Version 3.5

A summary of all the changes for Version 3.5 of this document is provided below in **Table 2.5**.

Table 2.5: Change log for Version 3.5

Section	Change Log
1	No revision
2	Added Change log for Version 3.5
3	No revision
4	No revision
5	Added a new figure for TweetQA training dataset sample
6	Added references used for information about each of the described design patterns
7	No revision
8	No revision
9	Updated challenges related to using Tensorflow versus PyTorch
10	No revision
11	No revision
12	Added new references used in section 6.5
Appendix U	No revision
Appendix R	No revision
Appendix T	No revision
Appendix TE	Added recent test execution runs. Fixed format

2.6. Changes in Version 4.0

A summary of all the changes for Version 4.0 of this document is provided below in **Table 2.6**.

Table 2.6: Changelog for Version 4.0

Section	Change Log
1	Revised the abstract as per final instructions.
2	Added Change log for Version 4.0
3	No revision
4	Added a new function system requirement as per the client's request.
5	Updated the exploratory study results on hyperparameter tuning using the TensorFlow library.
6	Updated design diagrams
7	Added key algorithm analysis for section 7.5
8	Added TC-028 and TC-029 unit tests Added TC-030, TC-031, TC-032 system tests
9	Added challenges into section 9.2 and open issues into section 9.3
10	Completed section 10.2 Completed section 10.3
11	Completed sections 11.1 and 11.2.
12	Added new references
Appendix U	Update the use case UC-001 to include new requirements from the client
Appendix R	Added a new function system requirement (SF-A-2) as per the client's request.
Appendix T	Added new test cases (TC-028 and TC-029) corresponding to the new function system requirement. Added new test cases (TC-030-032) for the system tests.
Appendix TE	Added recent test execution runs and system test execution

2.7. Changes in Version 5.0

A summary of all the changes for Version 5.0 of this document is provided below in **Table 2.7**.

Table 2.7: Changelog for Version 5.0

Section	Change Log
1	Revised the abstract to fit within one page.
2	Added Change log for Version 5.0
3	No revision
4	No revision
5	No revision
6	No revision
7	Updated key algorithm analysis for section 7.5
8	Added TC-033 for security testing. Added TS-008, TC-034, TC-035 for acceptance testing and corresponding test executions
9	Updated section 9.3 Open Issues and Ideas for Solutions
10	No revision
11	Revised section 11.1, completed section 11.3 for acknowledgment
12	No revision
Appendix U	No revision
Appendix R	Updated requirement/testing traceability the all the requirements and also for TC-033, TC-034, TC-035
Appendix T	Added the test tables needed for TS-008, TC-033, TC-034, TC-035
Appendix TE	Added test execution tables needed for TC-033, TC-034, TC-035

3. Problem Statement

3.1. Business Background

Twitter is a popular social media platform in which users can write "tweets", messages that are limited to only 280 characters. Tweets can contain all sorts of information; personal stories, humor, memes, social and political commentary, etc. Journalists will also often write tweets that contain news pertaining to current events, sometimes including links to articles with further details on their story.

Tweets can also be replied to by other users, and frequently by the original user to provide more context to the original tweet due to character limitations. Replies are also considered tweets and are subject to the same character limitations. A user can choose to write threads in which they will reply to the original tweet, then reply to that reply, and so on to provide a more complete story.

Journalists and writers have taken advantage of including tweets in their articles, blogs, books, and other written media from users who may be witness to, or somehow associated with, the story or topic they are writing about. TweetQA is a dataset provided by a group of researchers that contain tweets that are referenced in such articles [18]. The dataset also contains questions and answers for the tweets that are sourced from humans as a training/test set of labeled data for machine learning algorithms intent on being able to provide real-time question answering functionality.

3.2. Needs

The primary need of the client is a robust machine learning model that can perform as well as a human on Question Answering with Tweets. The model is needed to develop automated question answering systems which are critical to the effectiveness of many applications that rely on real-time knowledge from social media. One example application of the improved model is the creation of an improved news chatbot which can help the users quickly grasp what is happening on social media by simply asking a few questions.

Currently, question answering and reading comprehension over short and noisy social media data is rarely studied in NLP. There are several competing algorithms and fine-tuned models with varying degrees of success in the task as rated by three evaluation methods: BLEU, METEOR, and ROUGE-L[18]. So far, none of these come close to the abilities of humans to answer questions about tweets.

The secondary need of the client is to have a web-based application that connects the machine learning model to the public. This will enable crowdsourcing to evaluate model performance and gather new question and answer sets from user input via tweets for continuous training. Currently, there is no existing web application in the market that can perform the functions required by the client.

The tertiary need for the client is to have a site that can compare the output of different machine learning models side by side given the same tweet-question pair. Currently, other than the TweetQA provided leaderboard, it's difficult to directly compare how different models perform on a particular tweet-question type. The site is needed to allow the client to gain further insights into how different

models behave differently on the different types of tweet-question pairs. These insights are needed to help improve the model further.

3.3. Objectives

The main objective of this project is trying to achieve a higher score (or even reach the top place) in the three-performance metrics used to evaluate the machine learning model.

The second objective of this project is to build a web application with the machine learning model(s) to interact with the machine learning students. The students will be able to pick a tweet and pick one of the available models. ask a question, receive the answer from the application, rate the answer, and provide an alternative answer.

The third objective of this project is to use the feedback from the students to continuously retrain and fine-tune the machine learning models. This includes gathering new tweet-question-answer triplets from the students, using the new data to extend the TweetQA dataset, and continuously retraining the model.

In addition to the aforementioned objectives, the website also aims to provide a method for comparing one or more models and their resulting answers for a given tweet-question pair. The system will provide a dashboard allowing users to visualize the current dataset, review model performance over time, and provide an overview of model metrics. Overall, the website should provide an interactive and visually aesthetic interface that allows users to have an ample understanding of a given model.

4. Requirements

4.1. User Requirement

4.1.1. Glossary of Relevant Domain Terminology

- **Artificial Intelligence (AI)** - a machine that uses software and training data to exhibit intelligence.
- **Application Programming Interface (API)** - An API is a generic term to describe the interface between two pieces of software; usually, an API describes a service offered by one program to another
- **Bidirectional Encoder Representations from Transformers (BERT)** - an ML model created by researchers at Google that uses transformers to train on text in both left-to-right and right-to-left directions, which is known for its high performance in the question-answering task of NLP
- **Machine Learning (ML)** - the activity of developing the algorithms and training that allow a machine to improve its ability to perform a task
- **Model** - The current state of a machine learning system based on previous training. The model of a machine learning system can change as weights and biases change through training.
- **Natural Language Processing (NLP)** - The activity of AI to understand the human text and/or generate text that mimics the human language
- **Social Media** - the set of internet applications that provide a platform for its users to share ideas
- **Tweet** - A social media post on the Twitter platform that is limited to 240 characters
- **TweetQA** - a dataset containing over 10,000 labeled tweets used for training questions answering AIs

4.1.2. User Groups

4.1.2.1. Machine Learning Students

Machine learning students are the primary end-user of the system and are also a likely source of user feedback on the system. As such, machine learning students have the highest priority among user groups. The core requirements for a machine learning student are as follows:

- Copying and pasting a link to a tweet of their choice into the user interface of the system.
- Select a random tweet from the database to ask a question about
- Typing a question related to the tweet into the user interface.
- Confirming or rejecting the model's response to their question via the user interface.
- Select a machine learning model from a list of available models to analyze the performance of different models with different types of questions.
- Analyzing the accompanying model metrics and visualizations to better understand the functionality of the model.

At least a basic understanding of machine learning principles is expected from the machine learning student user group. Some machine learning background is important to understand the various visual elements of the user interface. Without some prior experience, much of the core functionality will be less effective. In addition, machine learning students are expected to be somewhat familiar with popular culture and Twitter. Without that background, it will be challenging for a user to provide the necessary inputs for the model.

Machine learning students should have a cursory understanding of core system concepts that will be displayed. Included among those concepts are the BERT model and the BLEU-1, METEOR, and ROUGE-L evaluation metrics.

There will likely be an ongoing relationship between machine learning students and the secondary user group (system administrators), as it is expected that machine learning students are intellectually engaged with the system and therefore more likely to provide feedback and suggestions for improvement.

4.1.2.2. System Administrator

The second group of users is System administrators, who are also primary users of the Tweet QA system. Responsibilities of the system administrator include the following:

- Monitoring the additional data points entered into the system by end-users to limit the data set to only valid entries.
- Ensuring that the continuous learning pipeline remains operational and continues to provide updated models to the end system.
- Monitoring the model accuracy levels to make sure large regressions do not occur.

System administrators should have a moderate knowledge of Twitter and popular culture to be able to successfully maintain and monitor the system. In addition, system administrators are expected to have a high level of technical knowledge related to machine learning, the entirety of the system, and the cloud infrastructure tools used to deploy and operate the system. System administrators are expected to have a strong background in software engineering principles as well.

System Administrators are likely to have been a part of the team that built the system, to have the most time invested in the system, and to have the most incentives for the system to be and stay successful.

4.1.3. Functional Requirements

4.1.3.1. Project Scope

As mentioned above, machine learning students and system administrators will be the primary users of the system. Machine learning students will be able to interact with the system via a web-based application and will be able to input tweet-question pairs, view metrics, change the ML model they submit a tweet-question pair to, and rate the ML predicted answer. Administrators will interact with the system by logging into the cloud service provider (Google Cloud Platform) to deploy trained models, train models on new data, and invite new machine learning students to try out the system. A summary of the aforementioned use cases is summarized below in **Figure 4.1**.

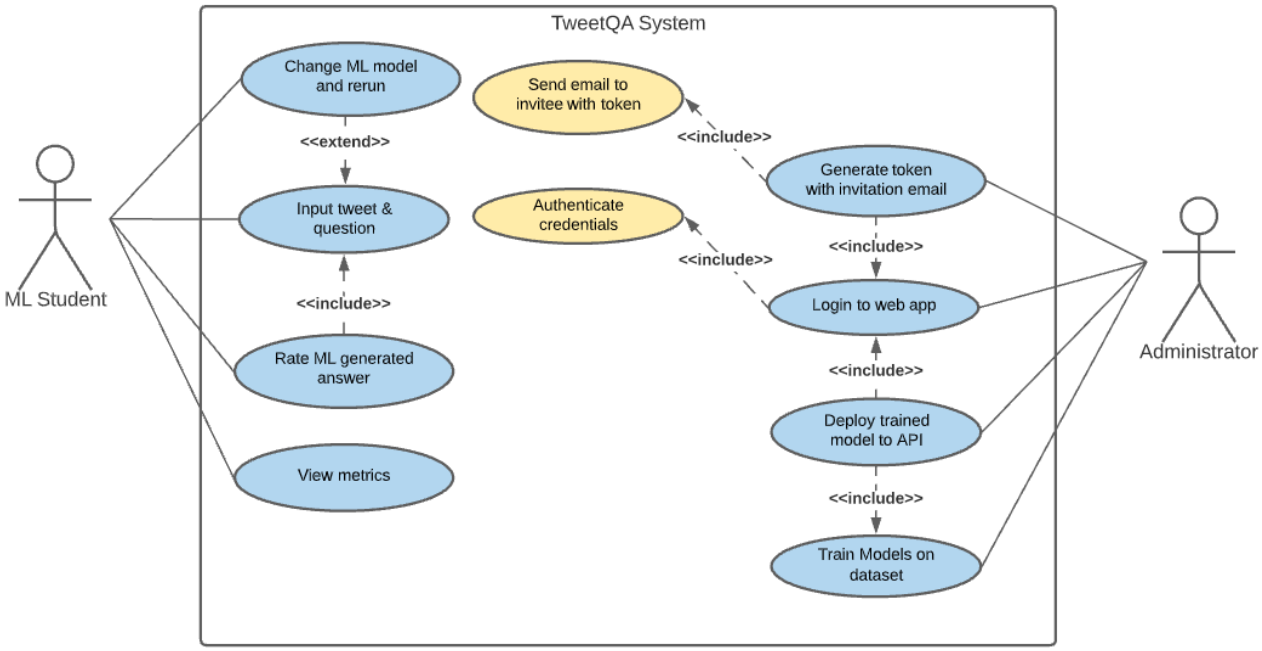


Figure 4.1: Use Case Diagram

4.1.3.2. User Scenarios

All use cases and user scenarios referenced can be found in **Appendix U** of this document.

UC-001: The ML student will be able to enter the selected tweet into the tweet input. Alternatively, the ML student can get a random tweet from the database to start asking a question. The user can also enter their question into the question input. The user can optionally choose a different ML model from a dropdown list. Before submitting the user will need to tick a checkbox providing their consent with regards to information privacy law. Upon clicking the submit button, the web app will validate these inputs, and if valid, will pass the inputs to the API. If invalid, an error message will be displayed, and the user will fix the inputs before continuing. When the API receives the request, it will run the tweet-question pair through the selected model and return the output answer and an ID, with status code 200. If an error is encountered, it will instead respond with whatever code is applicable. The web app will either display a success message or an error message depending on the response from the API.

UC-002: Upon completing UC-001 or UC-003, a feedback form will be provided to the ML student. The form will provide two radio buttons for the student to indicate whether the answer provided by the ML model is correct or not. There will be an optional text feedback field for an alternative answer if the ML student deems the answer is incorrect. The user fills in the inputs and clicks submit, and the web app sends the data to the API along with the ID of the tweet-question pair from the previous scenario. The API stores these details in a database table. The API responds with status code 200 if storage was successful, or an error status code if there was an error. The web app will either display a success message or an error message depending on the response from the API.

UC-003: After completing UC-001 and UC-002, the ML student has the option to repeat UC-001 with the same tweet-question pair but with a different ML model. Upon completion, UC-002 is repeated to get feedback for the new answer using the new model.

UC-004: The ML student can navigate to a page that displays various metrics, some of which include a report of the different machine learning models, their current performance for the 3 different metrics as well as the trends of these performance metrics over time as the models are retrained, etc. The data is presented in a modern and interactive format.

UC-005: The administrator can train the different models using the ML pipeline. The user logs into Google Cloud Platform and triggers the pipeline which pulls the TweetQA data as well as any user-submitted and labeled data (using the feedback mechanism in UC-002)**.** This data is passed to the model or models being trained. The training is executed and when finished, a weights file is produced to be uploaded in UC-006.

UC-006: The administrator can log in to the web app using their email and password. Upon hitting submit, the credentials are encrypted before being passed to the API, which triggers subfunction task UC-007. The API either responds with status 200 and authenticates the user as an administrator or it responds with an error code. If authorization is successful, the user is redirected to the administrator's home page. Otherwise, an error message is displayed, and the user can try again.

UC-007: The API receives a request to authenticate a set of encrypted credentials. It decrypts them and then attempts to pull the user record from the database, which contains the cryptographic salt for the password hash. If no user record exists for that username, a 404 error is returned, and the use case ends. The password that was entered by the user is hashed using this salt and compared against the password hash in the database. If the hashes don’t match, a 401 error is returned, and the use case ends. If they do match, an authorization token is generated and passed back to the web app.

UC-008: Upon completing UC-005 and UC-006 successfully, the administrator user can use an upload form to provide a new weights file to the API. The web app allows the user to browse their filesystem and select the weights file. They also select the ML model that this new weights file

applies to. Then the user clicks the submit button which begins an upload to the API. A progress bar is displayed and is continuously updated as the upload is performed. The API accepts the steam and the administrator authorization token and begins saving the weights file to the cloud server with a temporary filename. When the stream is complete, the API deletes the old weights file if it exists and renames the new file, so it becomes active. If the upload stream is interrupted, an error status is sent back to the web app. Otherwise, a success status is returned.

UC-009: The administrator user can navigate to an invitations control panel which has an input for an email address of a potential ML student to invite to access the system. The administrator enters the invitee's email address and clicks submit. A regular expression check is used to ensure that the email address is well-formed. If not, an error message is displayed. Otherwise, the email address is sent to the API and the subfunction task of UC-010 begins. If the API responds with a success code, a success message is displayed to the user. Otherwise, an error message is displayed.

UC-010: The API receives a request from the web app including the email address of the invitee. The API generates a token with high enough entropy that it is virtually impossible to guess. The API attempts to store the token in a database for future retrieval. If by some coincidence the token already exists, the API tries again until a unique one is formed and stored successfully. An email message is formed and includes a link with a query parameter containing the token. The email is sent over SMTP. If there is an error in sending the email, the API will retry up to 3 times before responding with an error status code. Otherwise, it responds with a success status code.

4.1.3.3. User Functional Requirements

All user functional requirements referenced can be found in **Appendix R** of this document.

UF-A: The user will be able to provide a tweet-question pair, then the system will provide an answer.

UF-B: The user will be able to provide feedback on the provided answer.

UF-C: The user can select different machine learning models to submit a tweet-question pair to.

UF-D: The user can generate a report of a machine learning model and the current dataset.

UF-E: The user can generate a comparison report between two different machine learning models.

UF-F: Users will be invited to interact with the system via a system-generated token.

4.1.4. Non-functional Requirements

All user non-functional requirements referenced can be found in **Appendix R** of this document.

4.1.4.1. Product: Usability Requirements

UP-01: The system should have a simplified front-end web page such that Machine Learning Students should require less than 5 minutes of training to use the system.

UP-02: System administrators should require less than 1 hour of training to use the system. The administrative view page should also be simplified with enough hints to guide the admins. The webpage should contain the tabs corresponding to all the main functions available to the admin.

4.1.4.2. Product: Performance Requirements

UP-03: The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on all the three performance metrics used (BLEU-1, METEOR, ROUGE-L scores).

UP-04: The machine learning model should achieve a more ambitious target of 70% score and above on all three performance metrics after continuous improvement.

UP-05: The webpage should return the result (answer to the question on the tweet) within less than one second after the user sends the question.

4.1.4.3. Product: Dependability/Reliability/Security

UP-06: The software system is expected to have an availability of at least 99%. The maximum allowed system failure rate will be less than 2 hours within one week.

UP-07: The system should have two different levels of access for the normal user and the administrative user. The ML student users will be invited to use the system and will gain access to the web interface via a system-generated token. The administrative user will need to log in to the system using their administrative account with the password.

4.1.4.4. Organizational: Development Requirements

UO-01: The machine learning model will need to be developed using Google Colab/Spyder Anaconda with Python as the main programming language. The model results are to be submitted to CodaLab for the competition.

Note there is currently no restriction on the programming language and scripts used to build the website hosting the machine learning model. The developer team is free to use any HTML, CSS, javascript, PHP, or other languages /scripts that are deemed suitable.

4.1.4.5. Organizational: Environmental Requirements

UO-02: The website should be displayed properly on the mainstream browsers run on the mainstream Operating System. The content of the website should be transmitted smoothly to the client browser with a connection speed of 512 kbps or better.

4.1.4.6. External: Legislative Requirements on Safety/Security

UE-01: The website should properly credit the TweetQA dataset used, and the use of any machine learning model developed by others.

UE-02: All the tweets used by the website, including those submitted by the users, should observe the information privacy law.

4.1.4.7. External: Cultural and Social Requirements

UE-03: The main language to be used by the model and the website should be English.

4.2. System Requirements

4.2.1. Functional Requirements

4.2.1.1. System Functional Requirements

All system functional requirements referenced can be found in **Appendix R** of this document.

SF-A-1: The system will utilize a web interface allowing users to submit tweet-question pairs to the system through the TweetQA API.

SF-A-2: The system will allow the user to select a random tweet from the database and start asking a question

SF-A-3: The system will predict an answer based on the user-provided tweet-question pair and will be communicated to the user through the TweetQA API.

SF-B-1: The system will prompt the user with a yes/no questionnaire regarding the validity of the predicted answer.

SF-B-2: The system will record all tweet-question-answer triples to the database that receives user validation.

SF-B-3: The system will prompt user feedback as needed to determine the correct answer for a tweet-question-answer triplet for use as an additional label in the dataset

SF-C-1: The system will route the user-submitted tweet-question pair to the appropriate machine learning model based on user selection in the web interface.

SF-D-1: The system will be able to provide the current performance metrics.

SF-D-2: The system will be able to provide previously recorded performance metrics vs time.

SF-D-3: The system will provide a useful visualization of the current dataset in the form of a word cloud and any other pertinent statistics related to the model (i.e. accuracy, precision, training loss, etc.).

SF-E-1: The system will be able to provide a comparison report between two models that highlight key differences related to performance between each model in an easy-to-read summary. Performance metrics including the BLEU, METEOR, and ROGUE-L should be included as well as any other metrics that could affect performance.
SF-E-2: The system will be able to provide a comparison of the difference in performance metrics vs time for the two models.

SF-E-3: The system will provide a comparison of any other pertinent statistics related to the two models.

SF-F-1: A user should not be able to access the system unless provided with a system-generated token.

SF-P-1: Hints should be visible to the students in guiding them through the activities such as submitting the tweet and question, viewing the answer returned by the model, and rating the answer returned by the model.

SF-P-2: The administrative view page should also be simplified with enough hints to guide the admins.

SF-P-3: There should be a help page providing the “How to” guides for all activities that can be performed by the administrator.

SF-P-4: The webpage should contain the tabs corresponding to actions available to the admin.

SF-P-5: The machine learning student users will need to access the system via the system-generated token.

SF-P-6: The administrative user will need to log in to the system using their administrative account with the password.

SF-P-7: The system will obtain consent on information privacy from the users when they submit new tweets in SF-A-1

4.2.1.2. Data Requirements

The TweetQA labeled dataset is provided by Xiong, W., et al. This data will be inserted into the SQL database unchanged. The same database will also store the tweet-question-answer triples along with their feedback data (if given) and the user token that requested the answer. These data combined will be pulled down by the ML pipeline and fed into subsequent model training to provide continuous learning of the models and to increase their scores.

The database will store user authentication details for system administrators, including first and last name, username, hashed password, and hashing salt. There will be no user data stored for ML student users but invite tokens will be stored along with their creation date, expiry date (if applicable), status flag for controlling whether the token is still active and to allow revocation, and the administrator username that generated the token.
The database will store the names of the models that can be selected for running tweet-question pairs along with their BLEU-1, METEOR, and ROUGE-L scores for each training run, as well as the amount of time it took to train each model.

4.2.2. Non-functional Requirements

All system non-functional requirements referenced can be found in **Appendix R** of this document.

4.2.2.1. Product: Usability Requirements

SP-01-01: Machine Learning Students should require less than 5 minutes of self-training to use the system.

SP-01-02: The first front-end web page visited by the students should contain all the information available to the students.

SP-01-03: The first front-end web page visited by the students should contain all the required interaction button/input text fields.

SP-02-01: For the administrator user, they should require less than 1 hour of guided- training to use the system.

4.2.2.2. Product: Performance Requirements

SP-03-01: The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on the BLEU metric.

SP-03-02: The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on the METEOR metric.

SP-03-03: The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on the ROUGE-L metric.

SP-04-01: The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on the BLEU metric.

SP-04-02: The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on the METEOR metric.

SP-04-03: The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on the ROUGE-L metric.

SP-05-01: The webpage should return the result (answer to the question on the tweet) within less than one second after the user sends the question.

4.2.2.3. Product: Dependability/Reliability/Security

SP-06-01: The software system is expected to have an availability of at least 99% measured by up time.

SP-06-02: The maximum allowed system failure rate will be less than 2 hours within any one week.

4.2.2.4. Organizational: Development Requirements

SO-01-01: The machine learning model will need to be developed using Python as the main programming language.

SO-01-02: The model results are to be submitted to CodaLab for the competition.

4.2.2.5. Organizational: Environmental Requirements

SO-02-01: The website should be displayed properly on the mainstream browsers (Chrome 50 or newer, Firefox 45 or newer, Edge, Safari 8) run on the mainstream Operating System (Windows 7 or newer, Mac OS X 10.6 or newer. IOS, Android). Note there are multiple possible combinations of browser and operating systems, they are combined to simplify the report. Each of the combinations should be tested out individually.

SO-02-02: The website should be completely loaded on the client browser with a connection speed of 512 kbps or better within 10 seconds.

4.2.2.6. External: Legislative Requirements on Safety/Security

SE-01-01: The credit to the TweetQA dataset used, and credit to the use of any machine learning model developed by others should be visible on the first page.

SE-01-02: All the tweets used by the website, including those submitted by the users, should observe the information privacy law.

4.2.2.7. External: Cultural and Social Requirements

SE-02-01: The main language to be used by the model and the website should be English.

4.3. Requirements Trace Tables

The engineering of user requirements into system requirements is summarized below in **Table 4.1** and **Table 4.2**, with the former providing functional requirements and the latter providing non-functional requirements.

Table 4.1: Functional Requirement Trace Table

Project Name:	**Building a Question Answering System using Tweets**		
User Requirements		**System Requirements**	
Req ID	**Description**	**Req ID**	**Description**
UF-A	The user will provide a tweet-question pair, then the system will return an answer.	**SF-A-1**	The system will utilize a web interface allowing users to submit tweet-question pairs to the system through the TweetQA API.
		SF-A-2	The system will allow the user to select a random tweet from the database and start asking question
		SF-A-3	The system will predict an answer based on the user provided question-answer pair, and will be communicated to the user through the TweetQA API.
UF-B	The user will be able to provide feedback on the provided answer.	**SF-B-1**	The system will prompt the user a yes/no questionnaire regarding the validity of the predicted answer.
		SF-B-2	The system will record all tweet-question-answer triples to the database that receive user validation.
		SF-B-3	The system will prompt for user feedback as needed to determine the correct answer for a tweet-question-answer triplet for use as an additional label in the dataset
UF-C	The user can select different machine learning models to submit a tweet-question pair to.	**SF-C-1**	The system will route the user submitted tweet-question pair to the appropriate machine learning model based on user selection in the web interface.
UF-D	The user can generate a report of a machine learning model and the current dataset.	**SF-D-1**	The system will be able to provide the current performance metrics
		SF-D-2	The system will be able to provide previously recorded performance metrics vs time
		SF-D-3	The system will provide a useful visualization of the current dataset in the form of a word cloud and any other pertinent statistics related to the model (i.e. accuracy, precision, training loss, etc.).

UF-E	The user can generate a comparison report between two different machine learning models.	**SF-E-1**	The system will be able to provide a comparison report between two models that highlights key differences related to performance between each model in an easy to read summary. Performance metrics including the BLEU, METEOR, and ROGUE-L should be included as well as any other metrics that could affect performance.
		SF-E-2	The system will be able to provide a comparison of the difference in performance metrics vs time for the two models.
		SF-E-3	The system will provide a comparison of any other pertinent statistics related to the two models.
UF-F	Users will be invited to interact with the system via a system generated token.	**SF-F-1**	A user should not be able to access the system unless provided with a system generated token.

Table 4.2: Non-functional Requirement Trace Table

User Requirements		**System Requirements**	
Req ID	**Description**	**Req ID**	**Description**
UP-01	The system should have a simplified front-end web page such that a Machine Learning Students should require less than 5 minutes of training to use the system	**SP-01-01**	Machine Learning Students should require less than 5 minutes of self-training to use the system.
		SP-01-02	The first front-end web page visited by the students should contain all the information available to the students
		SP-01-03	The first front-end web page visited by the students should contain all the required interaction button/input text field.
		SF-P-1	Hints should be visible to the students in guiding them through the activities such as submitting the question, view the answer returned by the model, and provide feedback on the answer.
UP-02	Administrative users should require less than 1 hour of training to use the system. The administrative view page should also be	**SP-02-01**	For the administrator user, they should require less than 1 hour of guided- training to use the system.

	simplified with enough hints to guide the admins. The webpage should contain the tabs corresponding to actions available to the admin	**SF-P-2**	The administrative view page should also be simplified with enough hints to guide the admins.
		SF-P-3	There should be a help page providing the "How to" guides for all activities that can be performed by the administrator.
		SF-P-4	The webpage should contain the tabs corresponding to actions available to the admin.
UP-03	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on all the three performance metrics used (BLEU-1, METEOR, ROUGE-L scores).	**SP-03-01**	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on BLEU metric.
		SP-03-02	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on METEOR metric.
		SP-03-03	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on ROUGE_L metric.
UP-04	The machine learning model should achieve a more ambitious target of 70% score and above on all three performance metrics after continuous improvement.	**SP-04-01**	The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on BLEU metric.
		SP-04-02	The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on METEOR metric.
		SP-04-03	The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on ROUGE_L metric.
UP-05	The webpage should return the result (answer to the question on the tweet) within less than one second after the user sends the question.	**SP-05-01**	The webpage should return the result (answer to the question on the tweet) within less than one second after the user sends the question.
UP-06	The software system is expected to have an availability of at least 99%. The maximum allowed system failure rate will be less than 2 hours within one week.	**SP-06-01**	The software system is expected to have an availability of at least 99% measured by up time.
		SP-06-02	The maximum allowed system failure rate will be less than 2 hours within any one week.
UP-07	The system should have two different levels of access for the normal user and the	**SF-P-5**	The machine learning student users will need to access the system via the system generated token.

	administrative user. The ML student users will be invited to use the system, and will gain access to the web interface via a system generated token. The administrative user will need to log in to the system using their administrative account with the password.		Use of the system is on an invitation only basis.
		SF-P-6	The administrative user will need to log in to the system using their administrative account with the password.
UO-01	The machine learning model will need to be developed using Google Colab/Spyder Anaconda with Python as the main programming language. The model results are to be submitted to CodaLab for the competition.	**SO-01-01**	The machine learning model will need to be developed using Python as the main programming language.
		SO-01-02	The model results are to be submitted to CodaLab for the competition.
UO-02	The website should be displayed properly on the mainstream browsers run on the mainstream Operating System. The content of the website should be transmitted smoothly to the client browser with a connection speed of 512 kbps or better.	**SO-02-01**	The website should be displayed properly on the mainstream browsers (Chrome 50 or newer, Firefox 45 or newer, Edge, Safari 8) run on the mainstream Operating System (Windows 7 or newer, Mac OS X 10.6 or newer. IOS, Android).
		SO-02-02	The website should be completely loaded on the client browser with a connection speed of 512 kbps or better within 10 seconds.
UE-01	The website should properly credit the TweetQA dataset used, and the use of any machine learning model developed by others.	**SE-01-01**	The credit to the TweetQA dataset used, and credit to the use of any machine learning model developed by others should be visible in the first page.
UE-02	All the tweets used by the website, including those submitted by the users, should observe the information privacy law.	**SE-02-01**	All the tweets used by the website, including those submitted by the users, should observe the information privacy law.
		SF-P-7	The system will obtain consent on information privacy from the users when they submit new tweets in SF-A-1
UE-03	The main language to be used by the model and the website should be English.	**SE-03-01**	The main language to be used by the model and the website should be English.

5. Exploratory Studies

5.1. Relevant Techniques and Packages

5.1.1 Available Pre-trained Models

The relevant techniques refer to how different Machine Learning Models for Natural Language Processing (NLP) tasks are trained. There are several pre-trained Natural Language Processing (NLP) models that are available in the market. They can be categorized into autoencoder (AE) language models and autoregressive (AR) language models. An AE language model aims to reconstruct the original data from corrupted input, the example of AE language models includes the various variations of the BERT model[16]. AR language models use the context word to predict the next word, but the context word is constrained to one direction only, either forward or backward[16]. Examples of AR language models include XLNET and GPT2/GPT3 from Open AI. These models will be briefly introduced here. The key principles of the BERT model that we will start with will also be discussed.

Starting with BERT, which was developed by Google, it was developed to address the problem of sequence transduction or neural machine translation. Thus, it best suits tasks that transform an input sequence to an output sequence such as speech recognition and text-to-speech transformation[15]. BERT was trained on 2,500 million Wikipedia words and 800 million words of the BookCorpus dataset. It is proven to perform 11 NLP tasks efficiently[15].

RoBERTa stands for Robustly Optimized BERT Pre-training approach. It modifies the hyperparameters in BERT such as training with larger mini-batches and removing BERT's next sentence pretraining objective. RoBERTa is known to outperform BERT in all individual tasks on the General Language Understanding Evaluation (GLUE) benchmark and can be used for NLP training tasks such as question answering[15]. Hence this model is likely to be the next model we will try on to improve the results.

ALBERT stands for A Lite BERT. As its name suggests, it aims to increase the training speed and lower the memory consumption of the traditional BERT model[15]. It utilizes two parameter-reduction techniques such as Factorized Embedding Parameterization (by separating the size of hidden layers from the size of vocabulary embeddings) and Cross-Layer Parameter Sharing (which prevents the number of parameters from growing with the depth of the network)[15].

StructBERT incorporates the language structures into the pre-training of BERT. The development of the model was inspired by the linearization exploration work of Elman. The model is said to perform surprisingly well on a variety of downstream tasks, including reaching a GLUE benchmark score of 89.0 (outperforming all published models), an F1 score on SQuAD v1.l question answering to 93.0, and the accuracy on SNLI to 91.7[15]. This model is worth exploring as part of our project.

DeBERTa stands for Decoding-enhanced BERT with disentangled attention. This model proposed by Microsoft Research includes two main improvements over BERT: the disentangled attention and an enhanced mask decoder[15]. Disentangled attention is concerned with the position-to-content self-attention, which is hypothesized to be needed to comprehensively model relative

positions in a sequence of tokens[15]. An enhanced mask decoder processes the absolute position of the token/word along with the relative information[15]. These two improvements allow the DeBERTa model to process self-attention of position-to-content, which is not present in the original BERT model[15].

OpenAI's GPT model stands for Generative Pre-Trained Transformer. The latest release is version 3 (GPT-3) which is an upgraded version of GPT2. The GPT3 is trained on 175 billion parameters (10x over the size of GPT2)[17] and is trained on 45TB of text from an open-source dataset called 'Common Crawl' and other texts from OpenAI such as Wikipedia entries[15]. The advantage of GPT-3 compared to BERT is that it does not require fine-tuning to perform downstream tasks. Its "text in, text out" API will allow the developers to reprogram the model using instructions[15].

Similar to OpenAI's GPT3, XLNet is an autoregressive pre-training model. However, it proposed a new objective during the pre-train phase called the Permutation Language Modeling[16]. This allows the model to gather information from all positions from both sides. Theoretically, it can utilize the techniques of the AE language model without the disadvantage of pretrain-finetune discrepancy due to the use of the MASK method during pre-training by the AE language model[16]. This pretrain-finetune discrepancy arises as the MASK method assumes the predicted (masked) tokens are independent of each other, while in reality, there could be a correlation among the predicted (masked) tokens that can be used by a model to predict the other[16].

5.1.2 Exploring the BERT model

Diving further into the BERT model, which will be the starting point for training the project's initial model, BERT stands for Bidirectional Encoder Representations from Transformers. The model utilized a technique called bi-directional reading as opposed to directional models[14]. In directional models, text input is read sequentially (left-to-right or right-to-left). In bi-directional reading, the Transformers encoder reads the entire sequence of words together. This technique allows the model to learn the context of a word based on all its surroundings (left and right of the word)[14].

The BERT model has two steps in the training framework: pre-training and fine-tuning[13]. During the pre-training, the BERT model is trained on unlabeled data over different pre-training tasks [13]. The tasks included during pre-training are Masked LM (MLM) and Next Sentence Prediction (NSP).

During the fine-tuning stage, the BERT model will be initialized with the pre-trained parameters, and all these parameters will be fine-tuned using labeled data from the downstream tasks [13]. Each of these downstream tasks will have separate fine-tuned models. Hence for this project, we will be adapting the BERT model into a fine-tuned model specifically for the question-answering task.

The Masked LM (MLM) training is done as follows, before feeding the word sequences into the BERT model, 15% of the words in each sequence are replaced with a [MASK] token [14].

The model then tries to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence [14]. In machine learning technical terms, the prediction of the output words will require [14]

- Adding a classification layer on top of the encoder output.
- Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
- Calculating the probability of each word in the vocabulary with softmax.

The Next Sentence Prediction (NSP) helps the machine learning model to understand the relationship between the two sentences, which is key to solving downstream tasks such as Question Answering (QA) and Natural Language Inference (NLI) [13]. To train a model that understands sentence relationships, the BERT is pre-trained for a binarized next sentence prediction task that can be trivially generated from any monolingual corpus [13]. Specifically, when choosing the sentences A and B for each pretraining example, 50% of the time B is the actual next sentence that follows A (labeled as IsNext), and 50% of the time it is a random sentence from the corpus (labeled as NotNext) [13]. During the training, the model will then predict the value of IsNext and NotNext.

In summary, different machine learning models used in NLP are trained differently and they have their strengths in a particular area. One of the objectives of this project is to explore different models, compare their results to understand, and perhaps even design a pre-trained fine-tuning strategy that will enhance the result.

5.2 Fine-tuning Process for the Machine Learning Models

This section provides a summary overview of how the pre-trained base model (BERT) is further fine-tuned for the downstream task (question answering) using the TweetQA dataset. Section 5.2.1 explores the characteristics of the TweetQA dataset, section 5.2.2 discusses the preprocessing required to prepare the dataset for the fine-tune training process and section 5.2.3 explores the hyperparameters available for tuning during the fine-tune training process.

5.2.1 Exploring the data

The TweetQA dataset can be divided into three subsets, the training subset, the development subset and the test subset. The training subset has 10692 samples, each sample is a python dictionary consisting of four key:value pairs. The keys for the dict include the Question, the Answer, the Tweet and the qid. Figure 5.1 shows the first sample of the TweetQA training dataset. Table 5.1 & 5.2 and shows further details of the basic statistics of TweetQA dataset and the Question Type statistic of TweetQA Datasets.


```
[{"Question": "at which school were first responders on the scene for?",
"Answer": ["independence high school"],
"Tweet": "Our prayers are with the students, educators & families at Independence High School & all
 the first responders on the scene. #PatriotPride\u2014 Doug Ducey (@dougducey) February 12, 2016",
"qid": "0c871b7e5320d0816d5b2979d67c2649"},
```


Figure 5.1: First sample of TweetQA training dataset

Table 5.1 - Basic Statistics of TWEETQA dataset[18]

Dataset Statistics	
# of Training triples	10,692
# of Development triples	1,086
# of Test triples	1,979
Average question length (#words)	6.95
Average answer length (#words)	2.45

Table 5.2 - Question Type statistics of TWEETQA[18]

Question Type	Percentage
What	42.33%
Who	29.36%
How	7.79%
Where	7.00%
Why	2.61%
Which	2.43%
When	2.16%
Others	6.32%

Out of the 10692 samples in the training dataset, the answers for 6152 of them are extractive. This means for these 6152 samples the answer to the question has an exact match in the tweet context. It also means that the rest of the 4540 samples will not have an exact match in the tweet context and will introduce challenges in training the models. Deep diving into those answers that don't have an exact match in the tweet context, they appear to be divisible into two groups. The first group of tweet-question-answer triples are when the answer is not possible to be found from the tweet context, either due to the answer being wrong or non-exist in the tweet context. Table 5.3 shows some examples for the first group. The second group of tweet-question-answer triples are when a partial match to the answer can still be found in the tweet context, i.e. the answer is abstractive instead of extractive from the tweet context. Table 5.4 shows some examples for the second group.

Table 5.3 Examples of Question-Answer-Tweet where the answer is non-exist in the Tweet

Question	Answer	Tweet
what website is linked in the tweet?"	instagram	I can finally say it out loud and proud: I'm going to a galaxy far far away! Lupita Nyong'o (@Lupita_Nyongo) June 2, 2014
when does he say kaine is able?	5:09 pm- jul 22, 2016.	KAINE IS ABLE!!!\u2014 Cory Booker (@CoryBooker) July 23, 2016
who did they kill on tape	eric	Sean Abbott - thinking about you too, mate!\u2014 Kevin Pietersen (@KP24) November 27, 2014

Table 5.4 Examples of Question-Answer-Tweet where the answer is partially found in the Tweet

Question	Answer	Tweet
what will senator rand paul do if he keeps obama plan?	tax it	Shorter Obama: If you like your college savings plan, you can keep it. But I'm gonna tax the hell out of it. #sotu\u2014 Senator Rand Paul (@SenRandPaul) January 21, 2015
who was reported as having knowledge of abuse while at penn state?	greg schiano	In response to media reports from earlier today:I never saw any abuse, nor had reason to suspect any abuse, during my time at Penn State.\u2014 Greg Schiano (@OSUCoachSchiano) July 12, 2016
what did the white people do?	steal from us	"@jtimberlake @iJesseWilliams Did you like the part when Jesse talked about white people stealing from us? That should resonate with you.\u2014 sockruhtese (@sockruhtese) June 27, 2016
at what age is he suing the obama administration?	15 years old	Met this guy the other day in Paris. Now 15 and suing the Obama Administration over climate change. #2degrees\u2014 John D. Sutter (@jdsutter) November 30, 2015

One of the main reasons why an exact match of the answer cannot be found in the tweet context is due to the nature of the language used in the tweet. As the tweet has a character limitation of 280, hence the tweet language often uses acronyms, symbols to represent longer words, and eliminates spaces to reduce the character input. For example, tweet language will use ‘tbh’ to represent ‘to be honest’, vp to represent ‘vice president’ , and ‘POTUS’ to represent ‘President of the United State”. Skipping space also causes the normal names to consist of two or more words being read as a single word, like ‘Hillary Clinton’ becoming ‘HillaryClinton’. The use of tweet language above will pose a challenge in identifying the start and end position later during the preprocessing stage.

5.2.2 Preprocessing the Data

The TWEETQA datasets need to be further preprocessed before feeding them into the training process. Key steps involved during preprocessing include normalizing the question and tweet strings, identifying start and end position of the answer span inside the tweet context, and tokenizing the question, tweet context, start and end position of the answer into the tokenized dataset.

5.2.2.1 Preview of final dataset required for training

An example of the final tokenized dataset required for training is shown in figure 5.2.


```
{'input_ids': [[101, 2256, 12583, 2024, 2007, 1996, 2493, 1010, 19156, 1004, 2945, 2012, 4336, 2152, 2082, 1004,
2035, 1996, 2034, 6869, 2545, 2006, 1996, 3496, 1012, 1001, 16419, 18098, 5178, 1517, 8788, 26363, 3240, 1006,
1030, 8788, 8566, 25810, 1007, 2337, 2260, 1010, 2355, 102, 2012, 2029, 2082, 2020, 2034, 6869, 2545, 2006, 1996,
3496, 2005, 1029, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [101, 16220, 5498, 3662, 2149, 1996, 5053, 1997,
14784, 1012, 2009, 3936, 1996, 2190, 1999, 2149, 1012, 2085, 2009, 1005, 1055, 2039, 2000, 2149, 2000, 2562,
11131, 1012, 16220, 5498, 16846, 14287, 1006, 1030, 16220, 5498, 16846, 14287, 1007, 2244, 2321, 1010, 2418, 102,
2054, 2003, 2039, 2000, 2149, 2000, 2562, 2725, 1029, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
'start_positions': [12, 26], 'end_positions': [14, 26]}
```


Figure 5.2: Example of the tokenized datasets

Further explanations of the keys-values used in the tokenized datasets are as below:

- 'input_ids' contains the value of the tokenized id of all the question and tweet pairs in the dataset. Section 5.2.2.1 will explain further details on the tokenization process. The input_ids is a list of sub-lists. Each sub-lists stores the tokenized id of a particular question and tweet context pair. The sub-list has a fixed length determined by the maximum padding size during the tokenization process. If the combined token numbers from the question and tweet context pair is less than the maximum padding size, a padding token will be added to fill up the list until the maximum padding size. This padding token will have a value of 0 as shown in figure 1.

- 'token_type_ids' contains a list of sub-lists. Each sub-list corresponds to a sub-list in input_ids, and stores the value of 0 and 1 to tell which part of the input_ids is the tweet context span to find the answer. 0 indicates the token is non-context (like the question), 1 indicates the token is part of the tweet context span.

- 'attention_mask' contains a list of sub-lists. Each sub-list corresponds to a sub-list in input_ids, and stores the value of 0 and 1 to tell which part of the input_ids is important. 0 indicates the token in input_ids is not important, 1 indicates the token in the input_ids is important. The use of attention_mask basically tells the training process which part of the input_ids contains the unimportant padding tokens that can be ignored.

- ‘start_positions’ contains a list of values. Each value corresponds to a sub list in input_ids, and represents the index in the sub list where the answer span of tweet context starts.

- ‘end_positions’ is similar to the ‘start_positions’ key. It contains a list of values. Each value corresponds to the sub lists in input_ids, and represents the index in the sublist where the answer span of the tweet context ends. Together with the ‘start_positions’ key, these two keys tell the training process which part of the input_ids contains the answer span where the answer to the question lies.

5.2.2.2 The Tokenization Process

Tokenization is the process of breaking down the input sentences (of the question and tweet) into tokens available in the pre-trained model. Tokenization is done using the tokenizer of the pre-trained model. Each version of the pre-trained model will have their own tokenizer. Some of the common variants of the pre-trained model and tokenizer for BERT include “bert-base-uncased”, “bert-large-uncased”, "distilbert-base-uncased" and "vinai/bertweet-base". Their differences are explained below:

- the uncased keywords mean the model doesn't make a difference between lowercase and uppercase [22]

- “bert-large” is a larger version of “bert-base”, their difference is summarized in table 5.5 below. Although larger models tend to perform better, fine-turning and training such models will require much more computer processing power and memory[23]. For example, BERT-base was trained on 4 cloud TPUs for 4 days and BERT-large was trained on 16 TPUs for 4 days [23]

Table 5.5 Difference between “bert-large” and “bert-base” [23]

Parameters	bert-large	bert-base
number of encoder layers	12	24
number of parameters (weights)	110 million	340 million
number of attention heads	12	16
number of hidden layers	768	1024

- Contrary to “bert-large”, “distilbert-base” is a smaller version of the “bert-base” model. It is unclear on which aspects “distilbert-base” cut down from the “bert-base”. The “distilbert-base” model claims to be 40% smaller, 60% faster, and retains 97% of the language understanding capabilities compared to the “bert-base” model [24].

- "vinai/bertweet-base" is a customized version of Bert model for English Tweets. It has the same architecture as BERT-base and is trained using the RoBERTa pre-training procedure[25].

Different tokenizers will tokenize the same sentences differently. Table 5.6 shows the tokenization result of the same example using the four different tokenizers listed above. As shown in the table, for the majority of the sentences, tokenizers for “bert-base-uncased”, “bert-large-uncased” and "distilbert-base-uncased" will yield the same tokenization results. While the "vinai/bertweet-base" tokenizer will yield a different result. Note for each token word, it will have a corresponding token id in the tokenizer.

Table 5.6 Tokenization results of different tokenizers

Original Sentences	kaine took hundreds of thousands of dollars in what? Is it the same Kaine that took hundreds of thousands of dollars in gifts while Governor of Virginia and didn't get indicted while Bob M did?— Donald J. Trump (@realDonaldTrump) July 23, 2016
“bert-base” result	['kai', '##ne', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'what', '?', 'is', 'it', 'the', 'same', 'kai', '##ne', 'that', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'gifts', 'while', 'governor', 'of', 'virginia', 'and', 'didn', ''', 't', 'get', 'indicted', 'while', 'bob', 'm', 'did', '?', '—', 'donald', 'j', '.', 'trump', '(', '@', 'real', '##don', '##ald', '##trum', '##p', ')', 'july', '23', ',', '2016']
“bert-base” token_ids	[11928, 2638, 2165, 5606, 1997, 5190, 1997, 6363, 1999, 2054, 1029, 2003, 2009, 1996, 2168, 11928, 2638, 2008, 2165, 5606, 1997, 5190, 1997, 6363, 1999, 9604, 2096, 3099, 1997, 3448, 1998, 2134, 1005, 1056, 2131, 21801, 2096, 3960, 1049, 2106, 1029, 1517, 6221, 1046, 1012, 8398, 1006, 1030, 2613, 5280, 19058, 24456, 2361, 1007, 2251, 2603, 1010, 2355]
“distilbert-base” result	['kai', '##ne', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'what', '?', 'is', 'it', 'the', 'same', 'kai', '##ne', 'that', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'gifts', 'while', 'governor', 'of', 'virginia', 'and', 'didn', ''', 't', 'get', 'indicted', 'while', 'bob', 'm', 'did', '?', '—', 'donald', 'j', '.', 'trump', '(', '@', 'real', '##don', '##ald', '##trum', '##p', ')', 'july', '23', ',', '2016']
“distilbert-base” token_ids	[11928, 2638, 2165, 5606, 1997, 5190, 1997, 6363, 1999, 2054, 1029, 2003, 2009, 1996, 2168, 11928, 2638, 2008, 2165, 5606, 1997, 5190, 1997, 6363, 1999, 9604, 2096, 3099, 1997, 3448, 1998, 2134, 1005, 1056, 2131, 21801, 2096, 3960, 1049, 2106, 1029, 1517, 6221, 1046, 1012, 8398, 1006, 1030, 2613, 5280, 19058, 24456, 2361, 1007, 2251, 2603, 1010, 2355]
“bert-large” result	['kai', '##ne', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'what', '?', 'is', 'it', 'the', 'same', 'kai', '##ne', 'that', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'gifts', 'while', 'governor', 'of', 'virginia', 'and', 'didn', ''',

	't', 'get', 'indicted', 'while', 'bob', 'm', 'did', '?', '—', 'donald', 'j', '.', 'trump', '(', '@', 'real', '##don', '##ald', '##trum', '##p', ')', 'july', '23', ',', '2016']
“bert-large” token_ids	[11928, 2638, 2165, 5606, 1997, 5190, 1997, 6363, 1999, 2054, 1029, 2003, 2009, 1996, 2168, 11928, 2638, 2008, 2165, 5606, 1997, 5190, 1997, 6363, 1999, 9604, 2096, 3099, 1997, 3448, 1998, 2134, 1005, 1056, 2131, 21801, 2096, 3960, 1049, 2106, 1029, 1517, 6221, 1046, 1012, 8398, 1006, 1030, 2613, 5280, 19058, 24456, 2361, 1007, 2251, 2603, 1010, 2355]
"vinai/bertweet-base" result	['ka@@', 'ine', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'what@@', '?@@', 'Is', 'it', 'the', 'same', 'Kaine', 'that', 'took', 'hundreds', 'of', 'thousands', 'of', 'dollars', 'in', 'gifts', 'while', 'Governor', 'of', 'Virginia', 'and', "didn't", 'get', 'indicted', 'while', 'Bob', 'M', 'did@@', '?@@', '—', 'Donald', 'J.', 'Trump', '(@@', '@@@', 'real@@', 'Donald@@', 'Trump@@', ')', 'July', '23@@', ',', '2016']
"vinai/bertweet-base" token_ids	[2491, 1466, 598, 7686, 15, 4876, 15, 3672, 16, 9247, 50239, 281, 18, 6, 222, 50072, 25, 598, 7686, 15, 4876, 15, 3672, 16, 5362, 290, 6677, 15, 4976, 13, 44204, 51, 25218, 290, 4406, 665, 15428, 50239, 267, 2402, 14865, 394, 10994, 5238, 9724, 47104, 11462, 60, 1198, 2957, 7, 723]

5.2.2.3 Identifying the start & end position

The original TweetQA datasets didn't have the start and end position recorded and those values are the two key inputs required for the model training as shown in section 5.2.2.1. Hence one of the most important steps during preprocessing is to identify the start and end position of the tweet context span which contains the answer. For the 6152 samples where the answer has an exact match in the tweet context, identifying start and end positions can be done easily using the string.find() method. For the rest of the 4540 samples that do not have an exact match of the answer in the tweet context, a technique utilizing the fuzzy string matching in python is needed.

FuzzyWuzzy is a Python library used for string matching. The fuzzy string matching process tries to find strings that match a given pattern using different metrics. The first metric is string similarity which is a measurement of edit distance [26]. The second metric utilizes partial-string similarity, where the process tries to find the “best partial” when two strings are of noticeably different lengths. If the shorter string is of length m, and the longer string is of length n, the process will look for the length-m substring in the longer string with the best matching score[26]. An additional technique utilized by FuzzyWuzzy is the token sort approach, which tokenizes the strings in question, sorting the tokens alphabetically before rejoining them back into strings for comparison[26]. For strings of unequal lengths, the token set approach is used. The approach tokenizes both strings and splits the tokens into intersection groups (where the tokens exist in both strings) and remainder groups before using those sets to build up the comparison strings[26].

Armed with the FuzzyWuzzy library, the next step is to use the brute force search to look for the span in the tweet context with the highest FuzzyWuzzy score compared to the answer. Starting with the first word in the tweet, the algorithm will calculate the FuzzyWuzzy score, compared with the best score stored, and replace the best score if the calculated score is higher. The algorithm will try to loop through all possible combinations of consecutive words in the tweet string to find the best score of all the combinations. The returned score and associated best-matching string then allow the identification of the start and end position.

5.2.3 Hyperparameters tuning

There are three main hyperparameters that were explored by the team during the model fine-tune training process. These include the number of epochs, the batch size, and the learning rate. The following paragraphs give explanations of what the hyperparameter is about and the indicative effect of the hyperparameter tuning.

5.2.3.1 Number of Epochs

The number of epochs is a hyperparameter that defines the number of times where the learning algorithm will work through the entire training dataset [27]. One epoch means that each sample in the training dataset has had one opportunity to update the internal model parameters [27]. The number of epochs also determines how long the training process will take; generally, more epochs will result in a longer training process. Initially when the team started training on PyTorch with 16 Epochs, which means the learning algorithm worked through the entire training dataset 16 times. The resulting score is quite low with average BLEU, METEOR, and ROUGE scores of 17.49%. This indicates the potential effect of overfitting. Hence the team tried to lower the number of Epochs to 2 which resulted in approximately 50% improvement in the scores. After switching to the TensorFlow library for the training, the team run and compared the training results from 1 epoch to 4 epochs. The results show that a higher number of epochs tend to lead to lower average scores. The comparison of the training with 16 epochs and 2 epochs using the PyTorch library is shown in table 5.7. A comparison of training with 1 to 4 epochs using the TensorFlow library is shown in table 5.8.

Table 5.7 Comparison of the training results with 16 epochs and 2 epochs using the PyTorch library

Epochs	Batch Size (Train)	Batch Size (eval)	Learning Rate	BLEU	METEOR	ROUGE	Avg
16	8	8	5.00E-05	16.46%	15.83%	20.18%	17.49%
2	12	12	5.00E-05	23.81%	21.28%	28.15%	24.41%

Table 5.8 Comparison of the training results with 1 to 4 epochs using the TensorFlow library

Epochs	Batch Size (Train)	Batch Size (eval)	Learning Rate	BLEU	METEOR	ROUGE	Avg
1	8	8	2.90E-05	69.96%	66.06%	71.94%	69.32%
2	8	8	2.90E-05	69.08%	64.93%	70.75%	68.25%
3	8	8	2.90E-05	68.10%	63.99%	70.17%	67.42%
4	8	8	2.90E-05	67.53%	63.73%	69.82%	67.02%

5.2.3.2 Batch size

Continuing with the second hyperparameter, the batch size is the number of samples processed before the model is updated [27]. The batch size will have an impact on the memory consumption required. In the team's experience, larger batch sizes require larger memory, sometimes exceeding the GPU memory available when training using NVIDIA CUDA GPU. For the fine-tune training with the TweetQA dataset, it seems the batch size has minimal effect on the evaluation scores when using the PyTorch library. However, when training using the TensorFlow library, there seems to have an optimum batch size of around 10-14 as shown in figure 5.3. The comparison of training results with a batch size of 12 and 8 for the PyTorch library is shown in table 5.9. The comparison of training results starting with a batch size of 2 to 20 (with an increment of 2) for the TensorFlow library is shown in table 5.10.

Table 5.9 Comparison of the training results with batch sizes of 12 and 8 for PyTorch

Epochs	Batch Size (Train)	Batch Size (eval)	Learning Rate	BLEU	METEOR	ROUGE	Avg
2	12	12	5.00E-05	23.81%	21.28%	28.15%	24.41%
2	8	8	5.00E-05	23.65%	21.81%	27.71%	24.39%

Table 5.10 Comparison of training results starting with a batch size of 2 to 20 (with an increment of 2) for the TensorFlow library

Epochs	Batch Size (Train)	Batch Size (eval)	Learning Rate	BLEU	METEOR	ROUGE	Avg
2	2	2	1.00E-05	0.693762	0.655148	0.709835	68.62%
2	4	4	1.00E-05	0.705487	0.667424	0.721572	69.82%
2	6	6	1.00E-05	0.696724	0.654418	0.713717	68.83%
2	8	8	1.00E-05	0.709087	0.665326	0.724216	69.95%
2	10	10	1.00E-05	0.720067	0.675178	0.733605	70.96%
2	12	12	1.00E-05	0.712139	0.666696	0.728681	70.25%
2	14	14	1.00E-05	0.71805	0.67651	0.732975	70.92%
2	16	16	1.00E-05	0.702144	0.659774	0.718162	69.34%
2	18	18	1.00E-05	0.711764	0.670664	0.72754	70.33%
2	20	20	1.00E-05	0.708776	0.666742	0.72462	70.00%

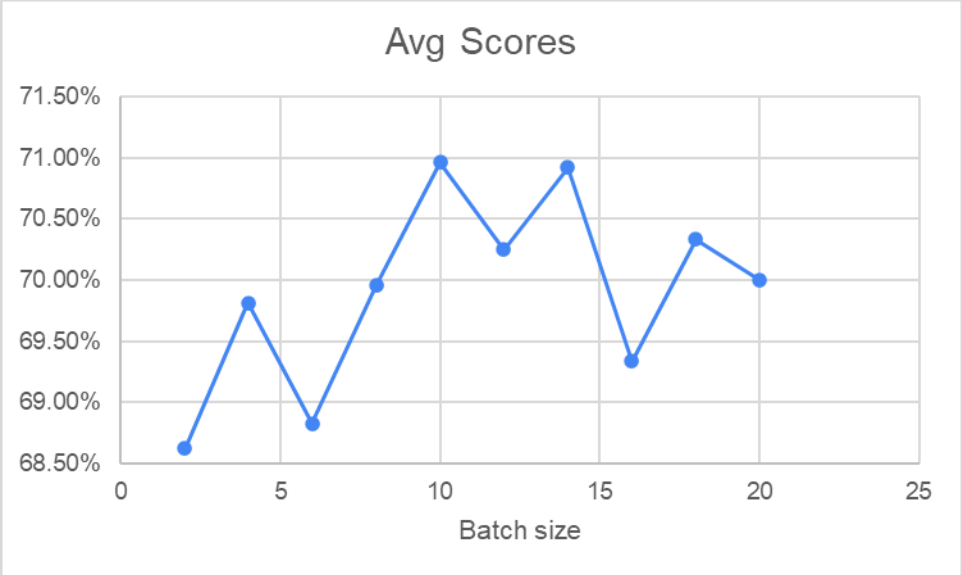


Figure 5.3: Graph of average scores against batch size for training using the TensorFlow library

5.2.3.3 Learning Rate

The third hyperparameter available for tuning is the learning rate. The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is supposed to be challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process[28]. The learning rate does not impact the length of the training process or the memory required for the training. For the fine-tune training with the TweetQA dataset using the PyTorch library, it seems the learning rate has minimal effect on the evaluation scores. However, when switching to the TensorFlow library, it seems that a lower learning rate can increase the average scores, where the optimal learning rate lies within the range of 0-1*e-06 as shown by Figures 5.4 & 5.5. The comparison of training results with different learning rates using the PyTorch library is shown in table 5.11. The comparison of training results with different learning rates using the TensorFlow library is shown in table 5.12.

Table 5.11 Comparison of the training results with different learning rates using the PyTorch library

Epochs	Batch Size (Trai	Batch Size (eva	Learning Rat	BLEU	METEOR	ROUGE	Avg
2	12	12	5.00E-05	23.81%	21.28%	28.15%	24.41%
2	12	12	5.00E-05	24.39%	22.17%	28.27%	24.95%
2	12	12	5.00E-05	22.97%	21.12%	27.25%	23.78%
2	12	12	4.60E-05	24.43%	22.16%	28.64%	25.08%
2	12	12	4.50E-05	24.46%	22.23%	28.73%	25.14%
2	12	12	4.30E-05	22.98%	21.11%	27.34%	23.81%
2	12	12	3.00E-05	23.44%	21.45%	27.68%	24.19%
2	12	12	1.00E-05	23.75%	21.64%	28.26%	24.55%

Table 5.12 Comparison of the training results with different learning rates using the TensorFlow library

Epochs	Batch Size (Train	Batch Size (eval	Learning Rat	BLEU	METEOR	ROUGE	Avg
2	8	8	1.00E-06	0.705775	0.666178	0.71986	69.73%
2	8	8	2.00E-06	0.704831	0.665102	0.720578	69.68%
2	8	8	3.00E-06	0.71025	0.66963	0.725936	70.19%
2	8	8	4.00E-06	0.704902	0.663677	0.721554	69.67%
2	8	8	5.00E-06	0.71045	0.667377	0.724577	70.08%
2	8	8	6.00E-06	0.712312	0.671403	0.729954	70.46%
2	8	8	7.00E-06	0.708688	0.665453	0.72648	70.02%
2	8	8	8.00E-06	0.71496	0.672102	0.729074	70.54%
2	8	8	9.00E-06	0.712156	0.666371	0.726931	70.18%
2	8	8	1.00E-05	71.23%	66.89%	72.84%	70.32%
2	8	8	1.00E-05	0.710581	0.668128	0.726671	70.18%
2	8	8	1.20E-05	0.710005	0.672603	0.725989	70.29%
2	8	8	1.40E-05	0.694451	0.652559	0.711635	68.62%
2	8	8	2.00E-05	69.77%	65.55%	71.52%	68.94%
2	8	8	3.00E-05	68.45%	64.68%	70.52%	67.88%
2	8	8	4.00E-05	66.61%	62.74%	68.50%	65.95%
2	8	8	5.00E-05	58.85%	55.27%	61.02%	58.38%

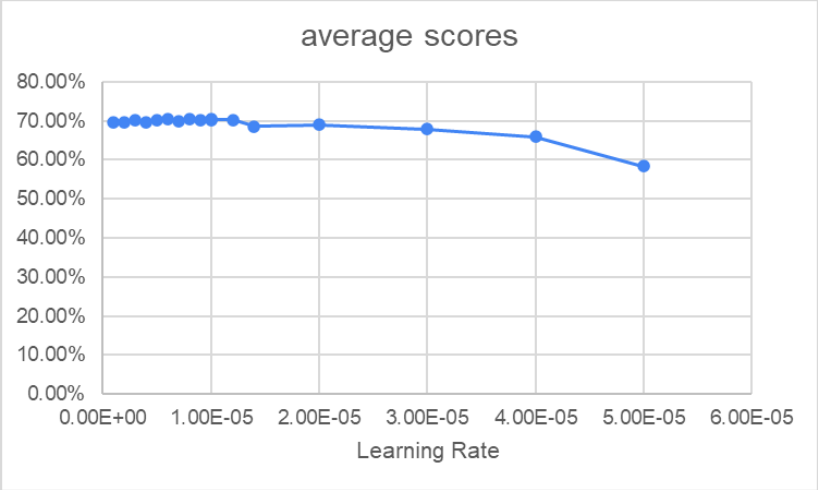


Figure 5.4 : Graph of average scores against learning rate for training using the TensorFlow library

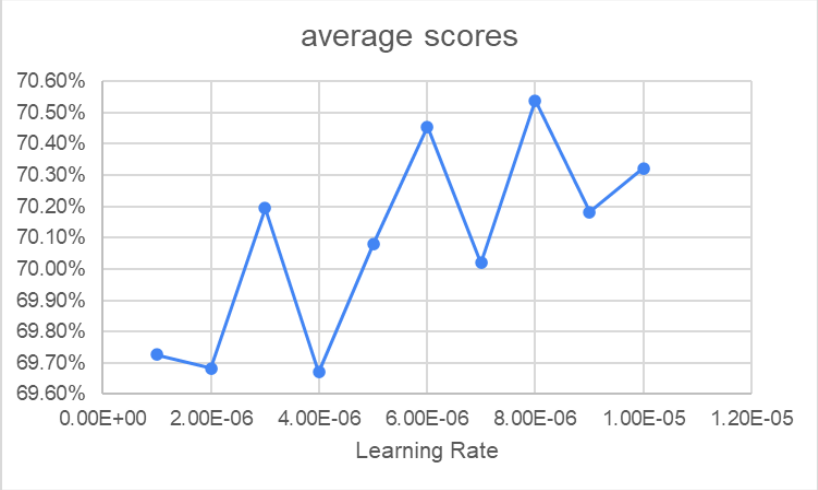


Figure 5.5 : Graph of average scores against learning rate for training using the TensorFlow library

5.3 Broader Impacts

The direct impact of this project is the contribution toward the advancement of the NLP processing capability in dealing with question answering over social media data. Currently, there is only one large-scale dataset for QA over twitter developed by the UCSB team. The best model on the TweetQA competition leaderboard can only achieve an average score of 75 plus on the three performance metrics (BLEU-1, METEOR, ROUGE-L). This project can help contribute to two aspects: gathering more tweet-question-answer triplet data that can help enlarge the current dataset and the continued improvement of the model, hopefully achieving higher average scores than the other existing models on the three performance metrics.

The improved model can be adapted and used in broader applications such as by government agencies to automatically monitor tweets in response to natural disasters and emergencies. Currently, government agencies must rely on human moderators to monitor social media during emergencies. This approach requires a lot of human resources and is slow and inefficient for identifying distress signals. With the automated machine learning models, the system could efficiently and effectively help agencies identify where the event happened, where the people are, and who may need help. This would allow the delivery of relief to those who need it most.

Other potential commercial applications include using the model to monitor trending events on social media that could influence financial markets and then notify traders when such events occur. The model can also be used by a company's customer service or public relations department to monitor for complaints about dissatisfied user experiences on social media. In all of these use cases, the potential end-user lacks the manpower required to effectively monitor social media and is therefore likely to respond slowly to negative events. A quicker response is likely to provide better and less widespread negative outcomes. Automated machine learning models will allow the above end users to put out sparks before they become a fire.

6. System Design

6.1. Architectural Design

The core architectural design of the system is client-server. Within the client-server model, the web application subsystem follows a Model View Controller (MVC) architecture. The Machine Learning (ML) Pipeline runs on the Server, but is not part of the MVC pattern and is independent of the Tweet QA Web App, which only communicates directly with the Tweet QA API. A high-level package diagram shown below in **Figure 6.1** summarizes the overall architectural design.

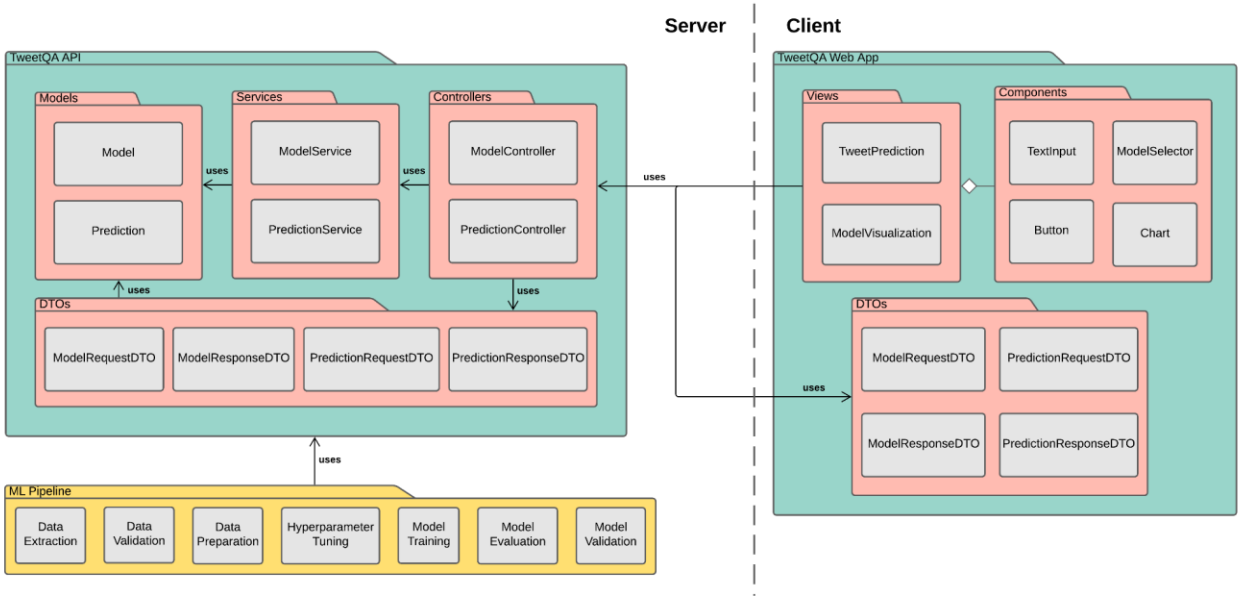


Figure 6.1: UML Package Diagram

The TweetQA web application acts as the client in our system. The web application consists of 3 core packages; views, components, and Data Transfer Objects (DTOs). The views are the presentation layer of our system. The views handle data returned from the TweetQA API (which resides in the server layer of our architecture) and display the appropriate information to end-users. Views are composed of components, which are small reusable UI elements. The DTOs are the objects that are passed between the client and server. Within the TweetQA web application, the DTOs also act as the data models for use within the views.

Client requests to the TweetQA API running on the server are handled by the API's controllers. The controllers handle the appropriate requests and make calls to the necessary services within the framework. The controllers accept data as a DTO and also return data to the client as a DTO. When making requests for the service layer, DTOs are first converted to model objects. The models mirror the data structure for persisted values in our database. The service layer handles the business logic within the API. The service layer is also responsible for retrieving and saving persisted data, which is maintained in a MySql database.

Within the server layer of our architecture, there also exists a package to handle the entire machine learning process. Machine Learning activities are handled in the Machine Learning Pipeline. The pipeline will retrieve data from the database and sequentially handle each machine learning operation, passing the results of each operation to the next in the pipeline. The first component of

the ML Pipeline is Data Extraction. The Data Extraction component's sole responsibility is to extract the data set from the database into a pandas Dataframe and share that object with the Data Validation component. The Data Validation component is responsible for any automated data integrity checks that are implemented. Since new data will be entering the system from the TweetQA web application, some automated checks will be necessary to make sure poorly formatted and incorrect data is not used to train the model. Once the data has passed through this component, it will be shared with the Data Preparation component. The Data Preparation component will take the data and transform it to a state that is usable within the model. A large part of this process will be to separate the data into a question, answer, and context (the tweet). In addition, the starting and ending token positions within the context (which correspond to the segment of the context most useful in answering the question) must be identified. Once the data is in the necessary format, it is passed along to the Hyperparameter Tuning component. The Hyperparameter Tuning component is responsible for setting the hyperparameters to be used in the next component, Model Training. The Model Training component will begin training on the provided data using the previously specified hyperparameters to build a new model. A completed model will be passed to the Model Evaluation component to determine the accuracy and precision of the new model. Finally, the model will be sent to the Model Validation component. Since this is a continuous training system where models are regularly being built, it is imperative that a model is checked for possible regression in performance before it is automatically put into production. If the model passes the validation step, it is saved to Cloud Storage where it can be accessed by the TweetQA API for use with the TweetQA web application.

Another important part of the ML Pipeline is how the process can be initiated. Initiation of the process can occur in one of two ways; through an automatic cloud trigger initiated by certain parameters (either elapsed time or a certain threshold of new data), or manually by a system administrator. For a system administrator to manually begin the ML Pipeline process, they will log in to Google Cloud Platform and use the GCP user interface to begin the process.

The overall system makes use of several cloud server features. The TweetQA Web App (the client) communicates with the server solely through the TweetQA API. The TweetQA API communicates with a MySql database to store and retrieve persisted data. In addition, the TweetQA API communicates with a Cloud Storage bucket to retrieve saved machine learning models. The Machine Learning Pipeline will save all completed models to the Cloud Storage bucket and log the details of the created model to the MySql database using the TweetQA API. The continuous process of training models will be initiated with a time-based cloud trigger and can also be manually triggered. A summary of the component level architecture is provided below in **Figure 6.2**.

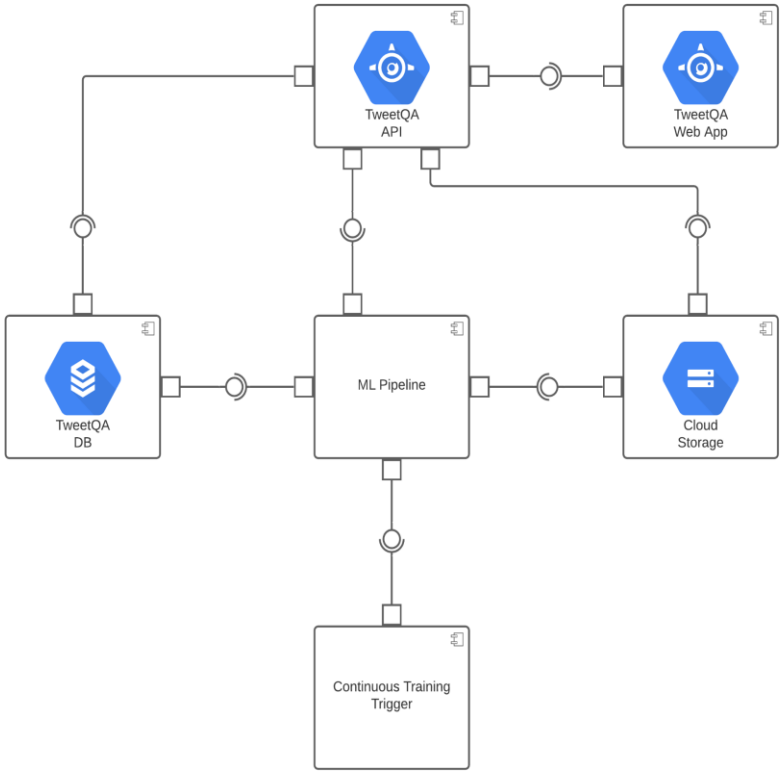


Figure 6.2: UML Component Diagram

6.2. Structural Design

6.2.1. Structural Relationship Diagrams

There are three primary structures contained within the overall system; the API, the user web application, and the machine learning pipeline. We will start by taking a deeper look at the class structure of the API.

The API is designed as a series of layers, each handling a particular responsibility. The most basic is the model layer as shown in **Figure 6.3**. Each primary class in the model layer maps directly to an entity in the database. Each model object consists of several Column objects. A Column is a class provided by SQLAlchemy that defines and relates directly to a field in a specified table. Because each object in the design is identifiable by an id value, each object inherits the id definition from the IdentifiableEntity class. The IdentifiableEntity class extends the SQLAlchemy Model base class, which provides each child object with several functions needed for communicating with the database about the object.

A number of the classes use one or more of the model classes in their attribute definitions. The Prediction model, for example, has model, datum, and visitor attributes. The values that are ultimately contained within each of those respective attributes are a Column element containing a QAModel, Data, and Visitor model object. The models are used extensively in the other layers of the API as they represent the persistent state of all core application components.

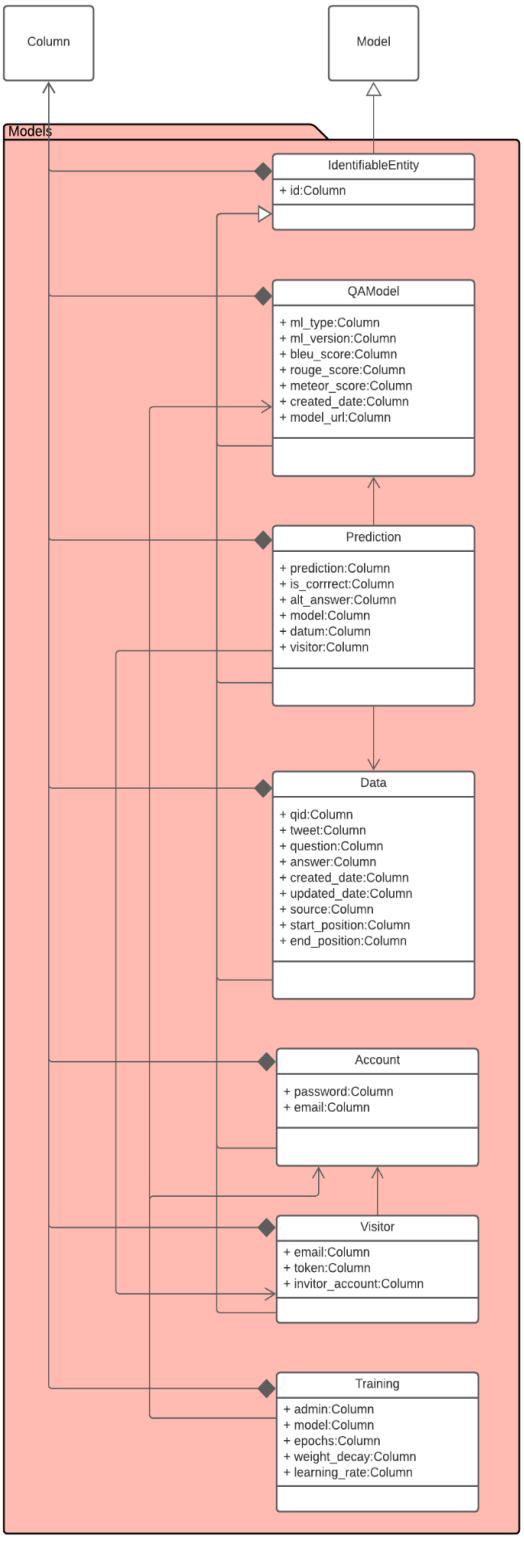


Figure 6.3: API Class Structure – Models

The second layer of the API is the service layer as shown in **Figure 6.4**. The service layer handles all business logic and manipulation of the various model objects. The main function of the API is CRUD (Create, Read, Update, Delete), though Delete functionality is not provided as it is not desired to permanently remove any data from the system once it has been created (only to further update as needed). Each service in the layer is responsible for handling the possible actions that can be taken for one specific model. There are many shared actions that can be taken, so abstracting a number of reusable actions was important. To this end, basic implementations of the create, read, and update methods are handled in respective abstract services. Because not every child service requires (or should have access to) all CRU methods, the functionality is divided into a hierarchy of abstract classes. At a core level, any service should have the ability to read an element of the object it is responsible for from the database, so the ReadService is at the top of the hierarchy. Extending the base ReadService class is the CreateReadService, which provides both read and create methods to any inheriting classes. Finally, the CreateReadUpdateService provides a basic implementation of the core CRU methods to any inheriting class. All inheriting classes provide the necessary object types that the parent class methods will work with. For instance, the PredictionService inherits from the CreateReadUpdateService, so must provide the Prediction class as the type of MODEL object that the create, read, and update methods will work with.

Because each service is responsible for the logic that can be performed on and with a specific model, there is an association relationship between a service and its corresponding model type. The models are used as inputs and outputs for the various methods in the service layer.

Lastly, because in this design the service layer handles access to the database, each service object has an attribute containing an instance of SQLAlchemy. This is an aggregation relationship in that the service is relying on the SQLAlchemy class to handle communication with the database for any requests the service may make.

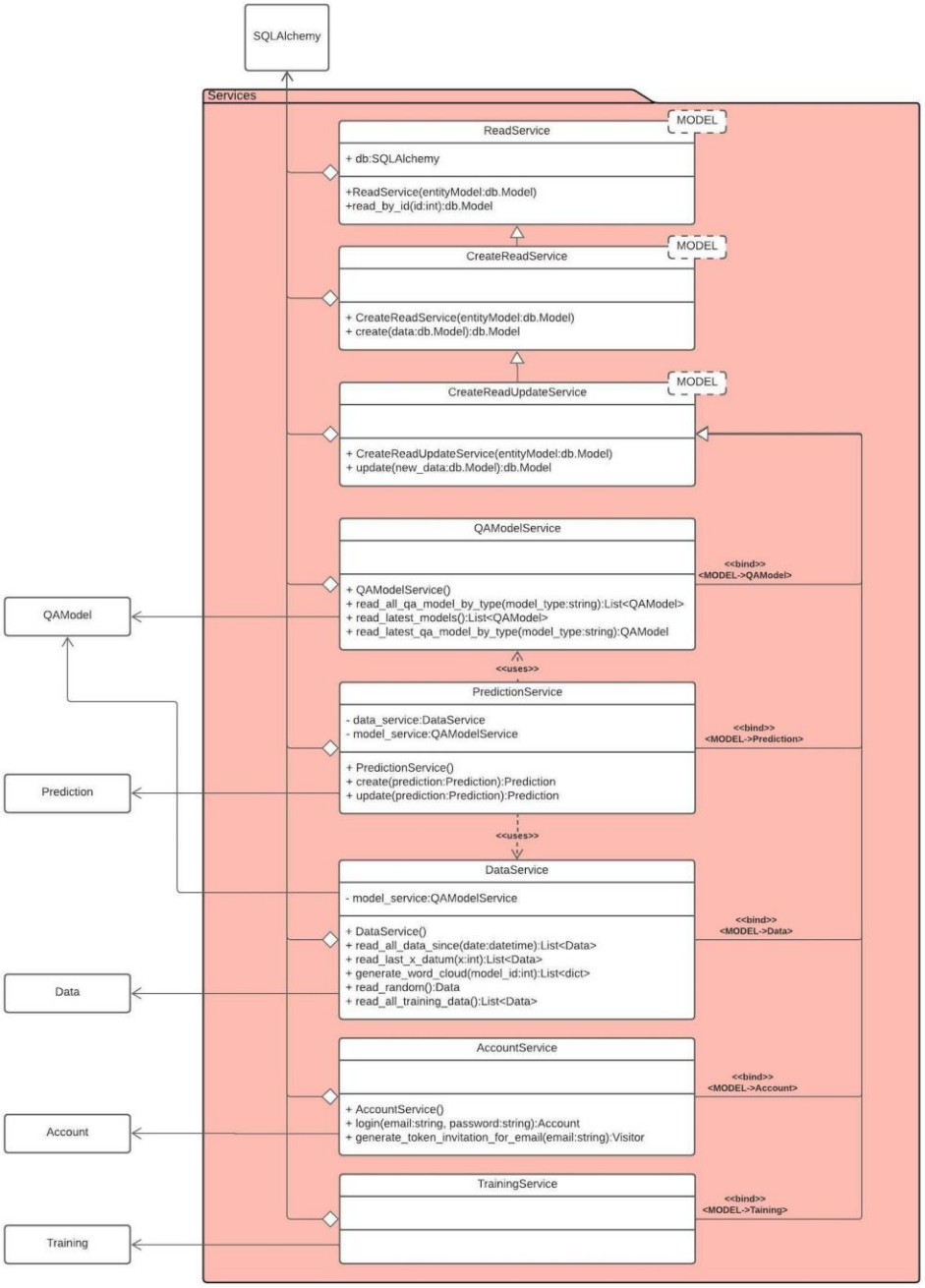


Figure 6.4: API Class Structure - Services

The third layer in the API architecture is the controller layer as shown in **Figure 6.5**. Each controller class's sole responsibility is to handle incoming requests to its designated resource, route those requests to the appropriate service, and respond to the requesting client after the service has completed its operation. In the same way that the standard create, read and update functionality was abstracted into higher-level classes for the services, those operations are abstracted in the controller layer. This results in each controller inheriting the appropriate create, read and update functionality from its parent class.

Every HTTP request is mapped to a specific method within the appropriate controller. The controller methods are defined to accept and return specific data types to and from the requesting client. The data types are defined in the DTO (Data Transfer Object) layer that is described in detail later. This creates an association relationship between each controller class and the various dtos that its methods use.

As mentioned earlier, each controller method accepts an input to share with an appropriate service to handle the actual work required for the request. This requires a dependency relationship between a controller and any of the services required for that controller to perform its necessary operations.

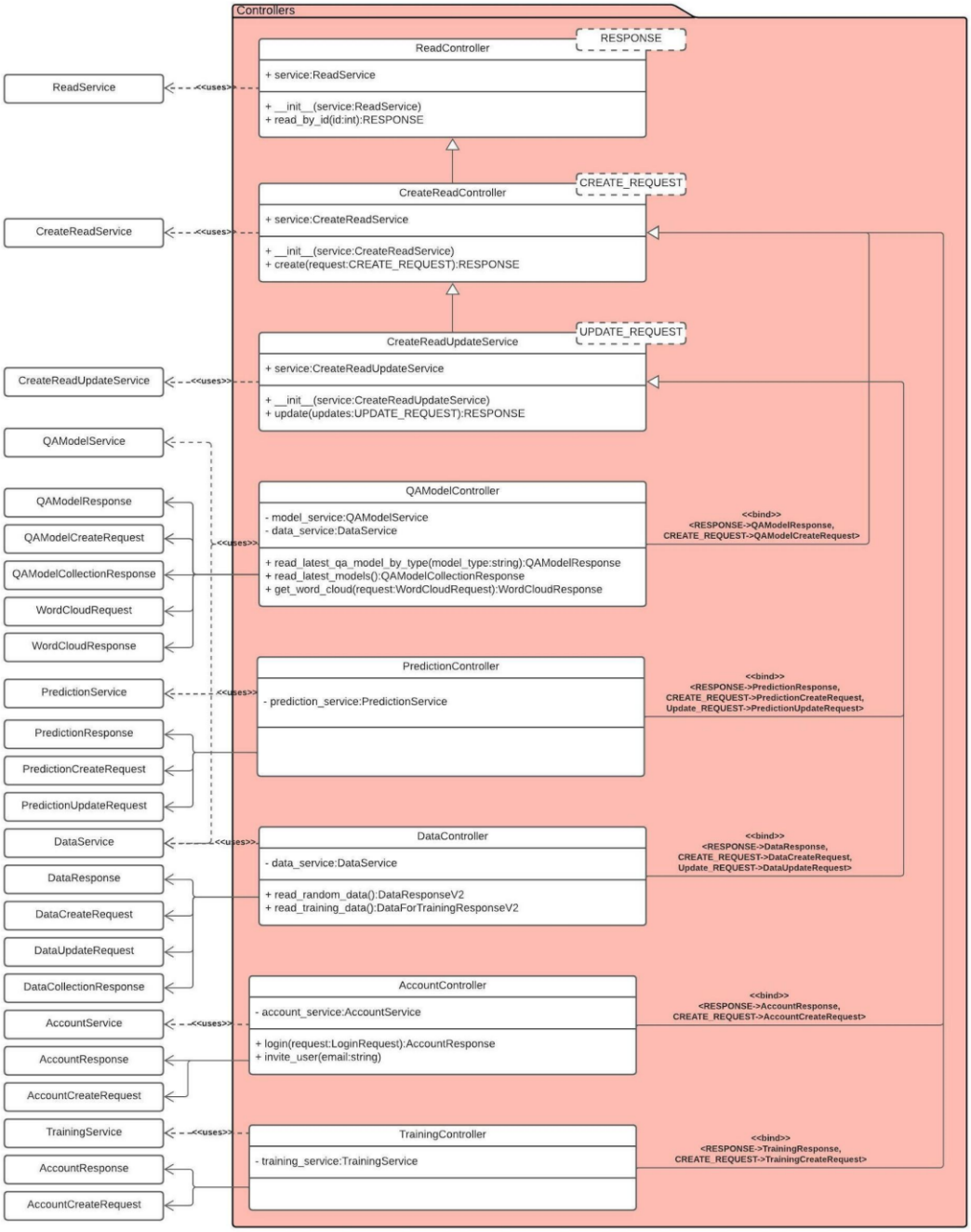


Figure 6.5: API Class Structure - Controllers

The final layer of the API is the DTO (Data Transfer Object) layer as shown in **Figure 6.6**. This layer defines the object structures that a client can send and receive to and from the API. Four primary types are defined in the system; a CreateRequest, a Response, an UpdateRequest, and a CollectionResponse. An appropriate DTO for any resource that is managed by the API is defined for each relevant primary type. A CreateRequest contains only the attributes necessary to create a new instance of a resource. A Response contains the attributes a client expects in any response

from the API. A UpdateRequest contains only the attributes necessary to update an existing instance of a resource. A CollectionResponse contains a list of Response objects along with any summary level information about the response. Not every resource requires each type of DTO, so only those that are required are defined. Each DTO inherits from an abstract class that defines the common methods and attributes of the specific DTO type. Each abstract DTO inherits from the python dataclass base class. The dataclass class provides some simple methods often used by data structure classes such as DTOs.

For any DTO object used by a client to make a request, it must have a toModel method that converts the DTO object to the appropriate model object to be passed to a service. All response classes have a constructor that accepts a model object so that the response from a service can be converted from a model to an appropriate DTO. Those two behaviors create an association relationship between a DTO and its respective model.

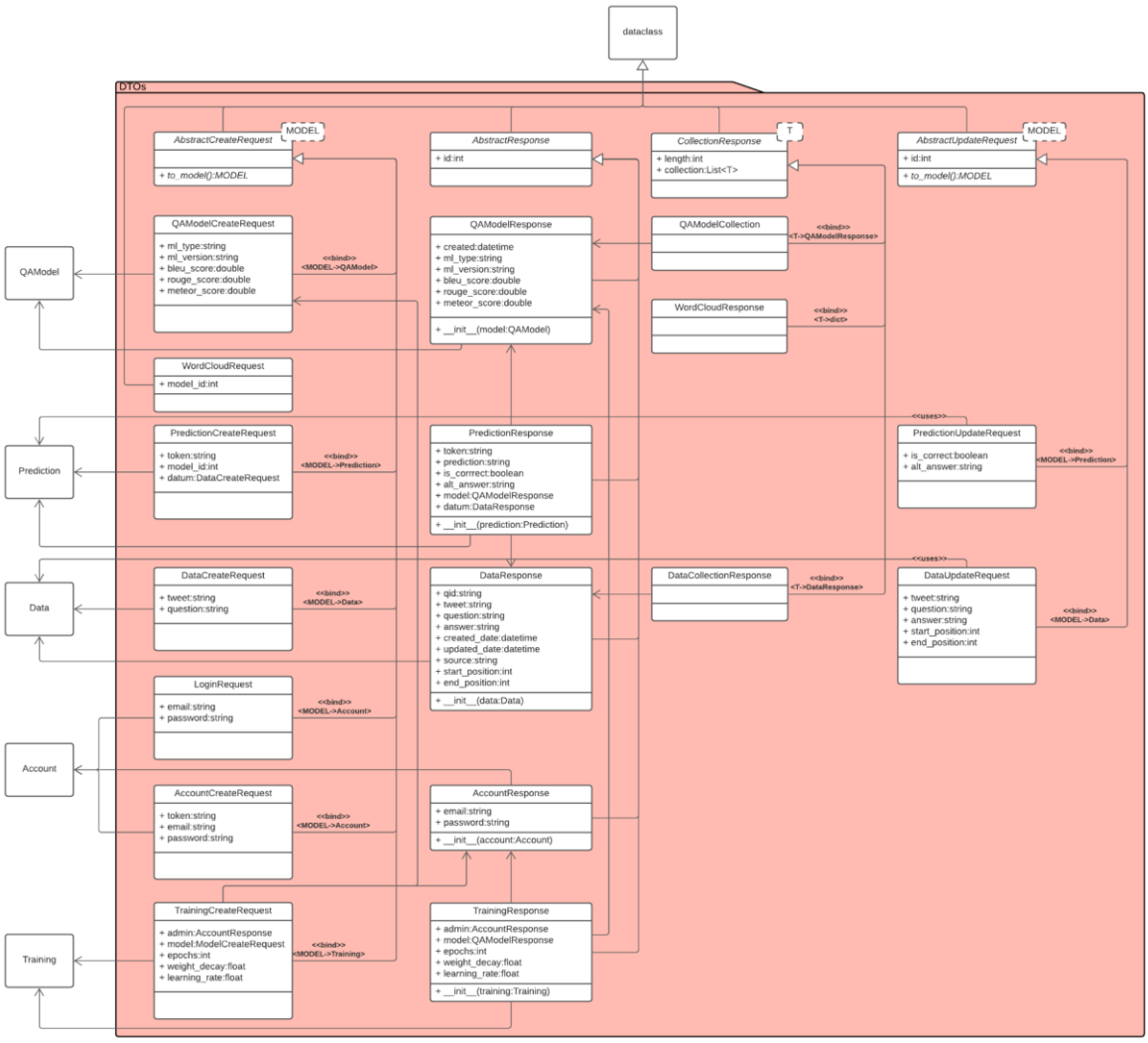


Figure 6.6: API Class Structure - DTOs

Next, we will examine the structural elements of the Web Application. The core elements of the Web Application include the DTOs, the services, the components, and the state.

The DTOs in the Web Application exactly mirror the corresponding DTOs from the API. As shown in Figure 6.7, these objects are defined as interfaces. In Typescript, interfaces are used to define the structure of a class and are allowed to contain any attribute and definition desired, so are often used to define data structures. The DTOs are the primary data type used throughout the Web Application.

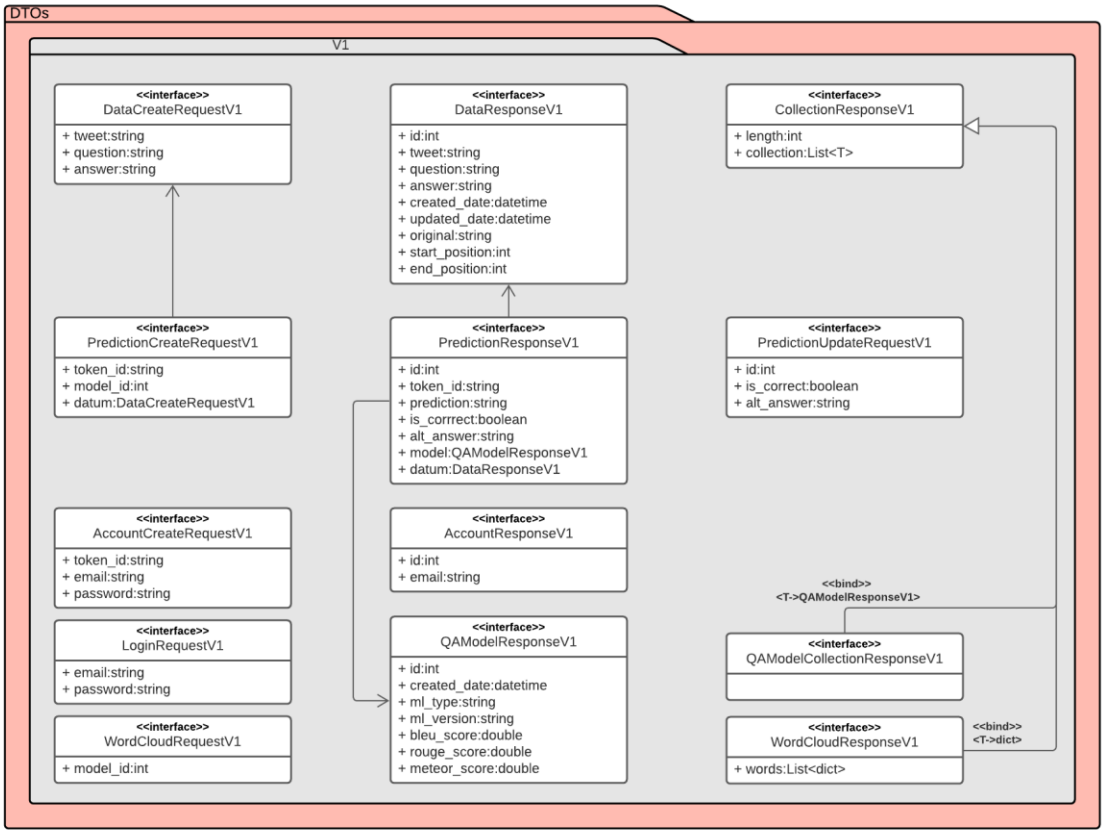


Figure 6.7: Web App Class Structure - DTOs

The services in the Web App are primarily responsible for making HTTP requests from the client to an external API. **Figure 6.8** shows all the service classes. In this application, those requests are all directed to the TweetQA API. The Web App is capable of creating, reading, and updating resources through the TweetQA API. As such, the corresponding functionality is again abstracted away in the same way that was done in the API service and controller layers. There is a service class for each resource that the Web App uses and each service class inherits from an appropriate abstract parent service class. Each method in a concrete service class uses a combination of DTOs as input and output parameters. As such, each service class has some number of association relationships with appropriate DTO classes.

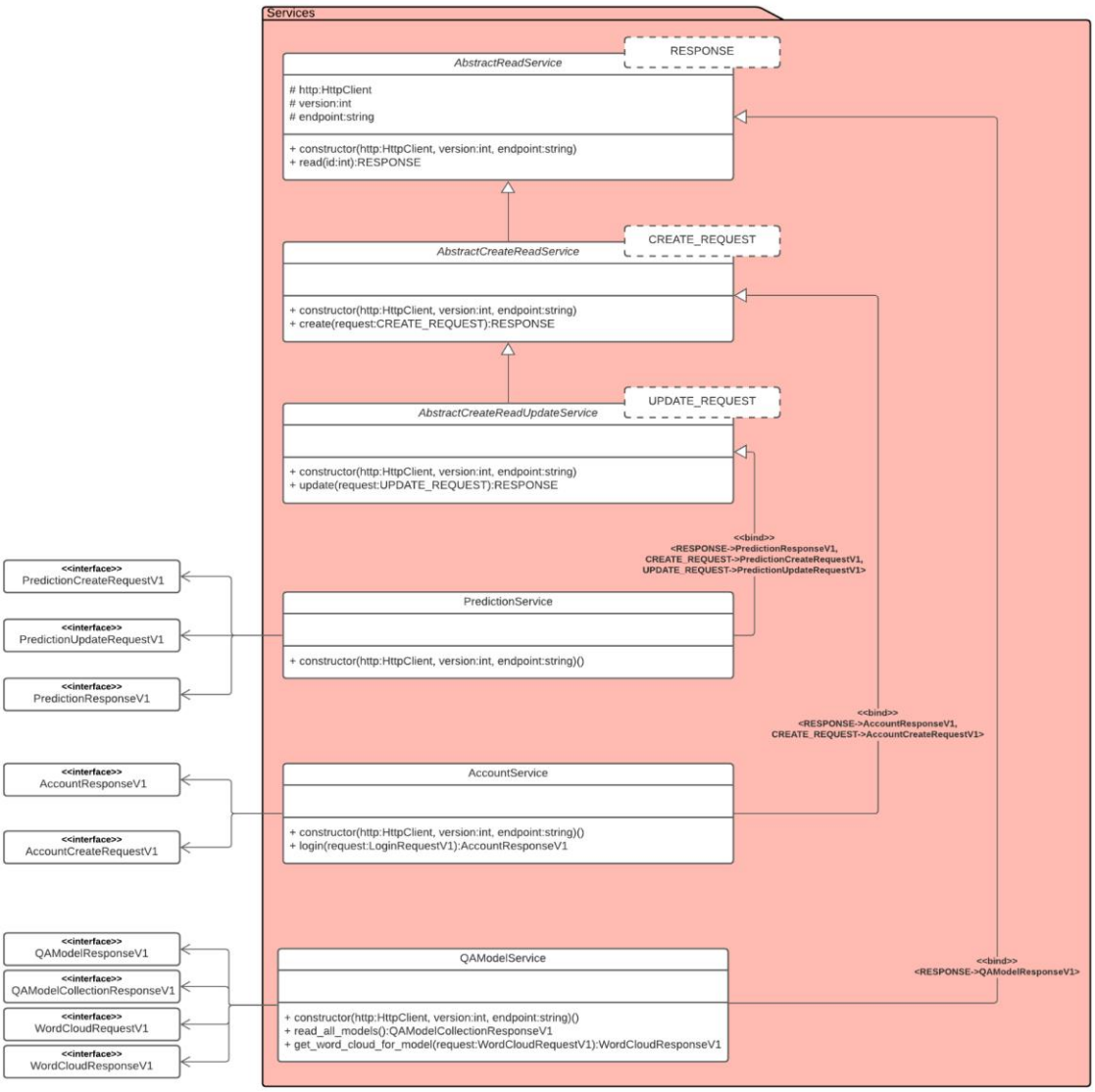


Figure 6.8: Web App Class Structure - Services

The components in the Web App are the stateful building blocks of the User Interface. **Figure 6.9** shows all components in the Web App. Every component has an association with an HTML file that defines the structure of the web page that will be displayed to the user and an scss file that defines the style of the HTML elements.

The components are also often the consumers of the defined services and DTOs. The PredictionFormComponent, for instance, uses the PredictionService class to create new prediction requests and update existing predictions with user feedback. It relies on a PredictionRequestFormState object to manage the state of the various inputs. All components implement the Angular defined OnInit interface which defines an ngOnInit method that is called during the Angular application's construction of a specific component and allows for any

necessary setup before displaying the component to a user.

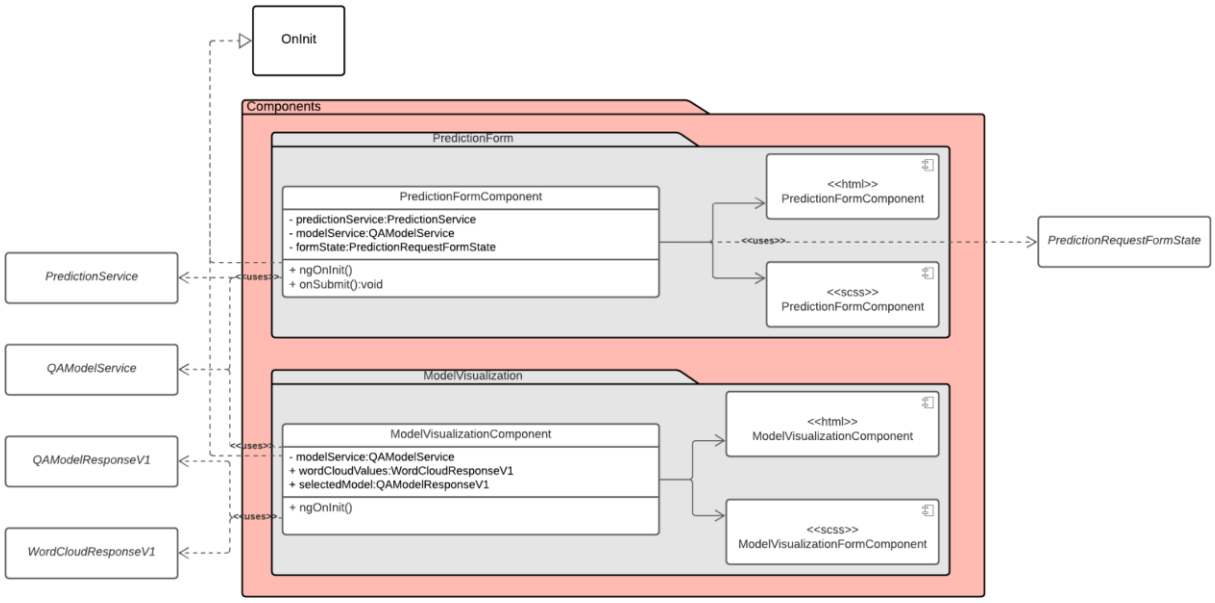


Figure 6.9: Web App Class Structure - Components

The final element of the Web App is its state. The only state currently designed is for the PredictionRequestForm. The classes use the State design pattern. The PredictionRequestFormState is an abstract class that exposes a public nextState method, a protected abstract nextStateAction method, and a protected abstract enter method. The nextStateAction method is a template method that each child class is required to implement to define allowable state transitions based on a specific Action. The nextState accepts an action argument and returns a new PredictionRequestFormState object by calling the nextStateAction method. The nextState element is dynamic as the state transitions are dependent on the current state. Each concrete state object inherits from the abstract state and so is of type PredictionRequestFormState. All possible states for the PredictionRequestForm are shown in **Figure 6.10.**

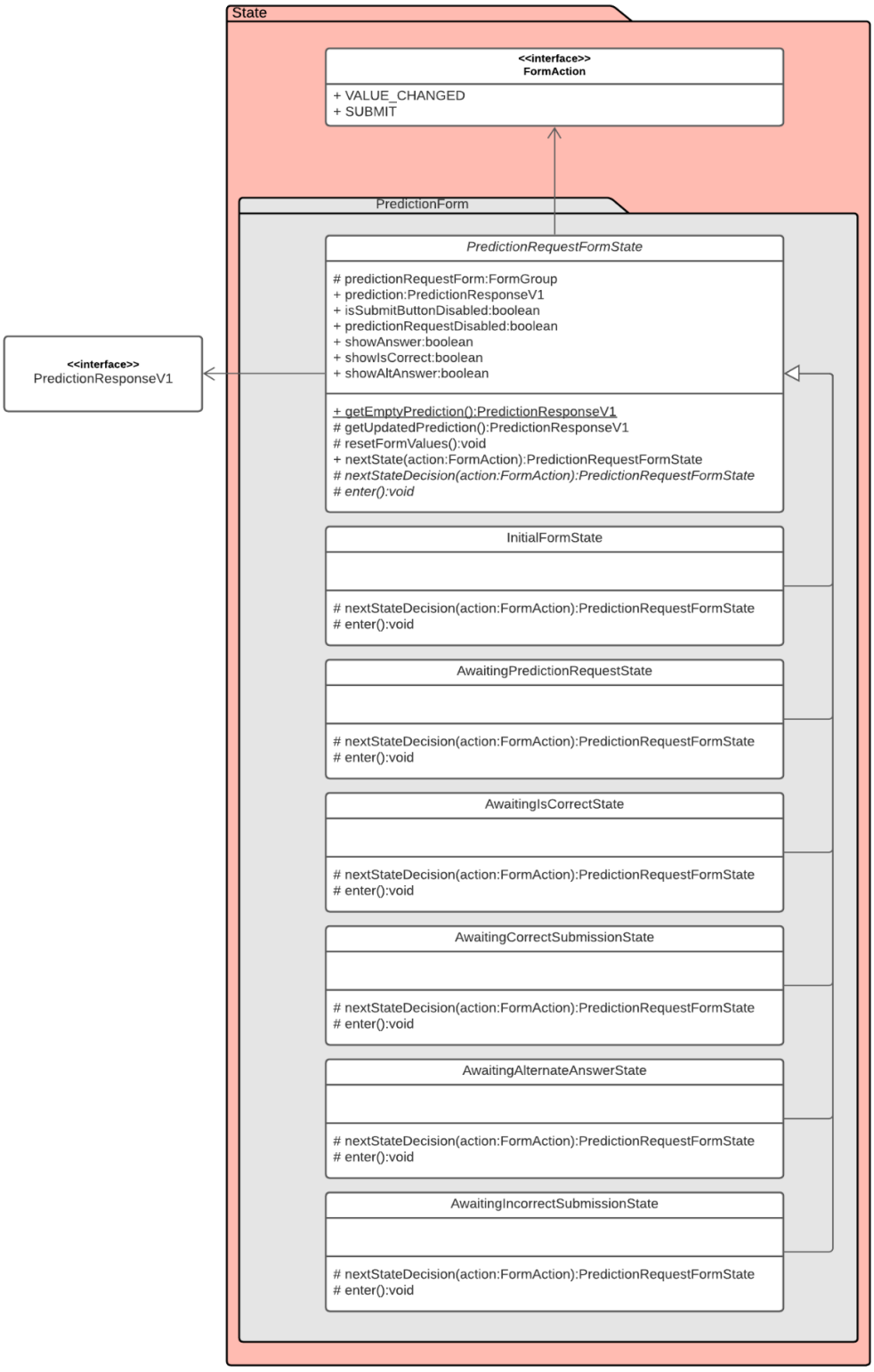


Figure 6.10: Web App Class Structure - State

The final system structure is the Machine Learning Pipeline as shown in **Figure 6.11**. This particular system is a series of orchestrated scripts that each run in their own container in a Kubernetes Cluster and communicate with one another by passing messages from one container to another as one process finishes and another begins. The messages are either simple data types or location references to where files containing the output of the previous container's work were saved. The process begins by running the pipeline method of the Pipeline class. Each subsequent step has a dependency on the previous step in that it requires the output of the previous step as its input. These relationships are all defined in the Pipeline class and the pipeline method will call each step in the process to move from one component to the next.

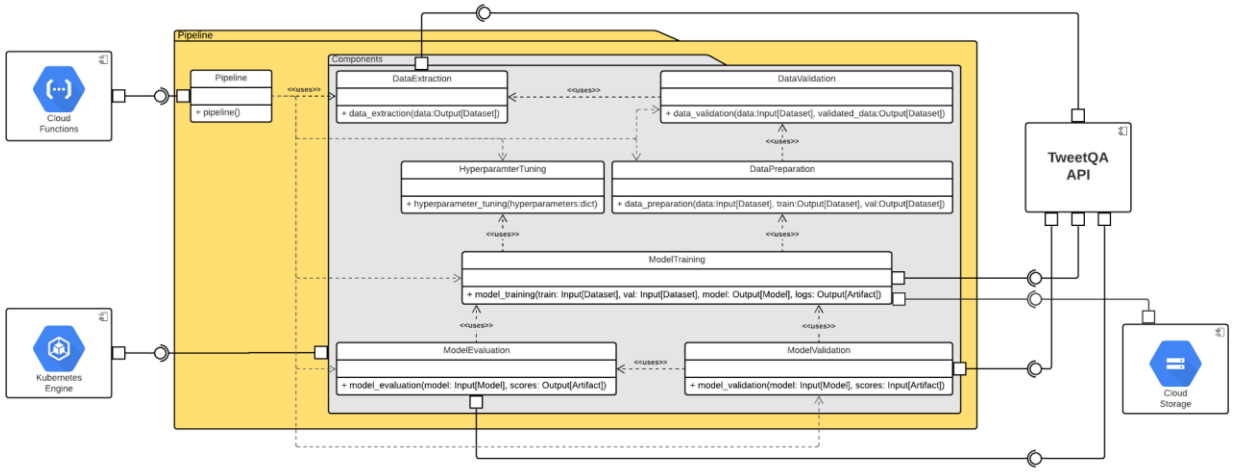


Figure 6.11: Machine Learning Pipeline Class Structure

6.2.2. Entity Relationship Diagram

Figure 6.12 shows the overall Entity Relationship Diagram for the database used by this project. All the requirements for the database are described in the following paragraphs.

The database will store each prediction (which can be an alphanumeric value of up to 280 characters) returned by the machine learning model given each tweet and question set. The tweet will be an alphanumeric value of up to 400 characters and the question can be an alphanumeric value of up to 280 characters. The user will enter both tweet and question sets. The database will keep the information on which visitor is requesting the prediction, as well as a boolean value indicating whether the prediction given by the modal is correct. If the prediction is incorrect, the database will store the alternative answer (which can be an alphanumeric value of up to 280 characters) given by the visitor. For each of the predictions, the database will also store the information of the machine learning model used. The prediction set will have an auto-incrementing integer id as the primary key.

The database will store the data set for future training. This is done by keeping the tweet and question sets asked by the visitor, and also the chosen answer for this data set. In addition, the database will store the date this dataset is created, and the date where this dataset is updated. Both

dates will be recorded using the DATE format. A tweet, question, and answer triplet need to have a unique qid for running the BLEU, METEOR, and ROUGE score analysis. The dataset will store the start and end position (both as integers) for each triplet to be used by the Machine Learning models. The database will store the source for this data set. The source will indicate if the data was from the original Tweet QA dataset, from user input, or from any other source that may be introduced in the future. The data set will have an auto-incrementing integer id as the primary key.

The database will store the information required to authenticate an administrator for the website, this includes the login email and password (both can be an alphanumeric value of up to 60 characters). The administrator will have an auto-incrementing integer id as the primary key.

The database will store the information required to verify the invited visitor for the website, this will include an email and a token_id (both can be an alphanumeric value of up to 60 characters) generated for the visitor. The visitor will also have an auto-incrementing integer id as the primary key.

The database will store information for each machine learning model run. This will include the unique id to identify the model, the name of the model, the version of the model, the BLEU, METEOR, and ROUGE score for the model (as performance record), and the created_date for the model. The id will be an auto-incrementing integer. The name of the model will be stored as an alphanumeric value of up to 100 characters, while the version of the model will be stored as an alphanumeric value of up to 20 characters. All the BLEU, METEOR, and ROUGE scores will be stored as floating numbers, while the created_date will be stored using the DATE format. The database will store the URL for the model as well (which can be an alphanumeric value of up to 200 characters).

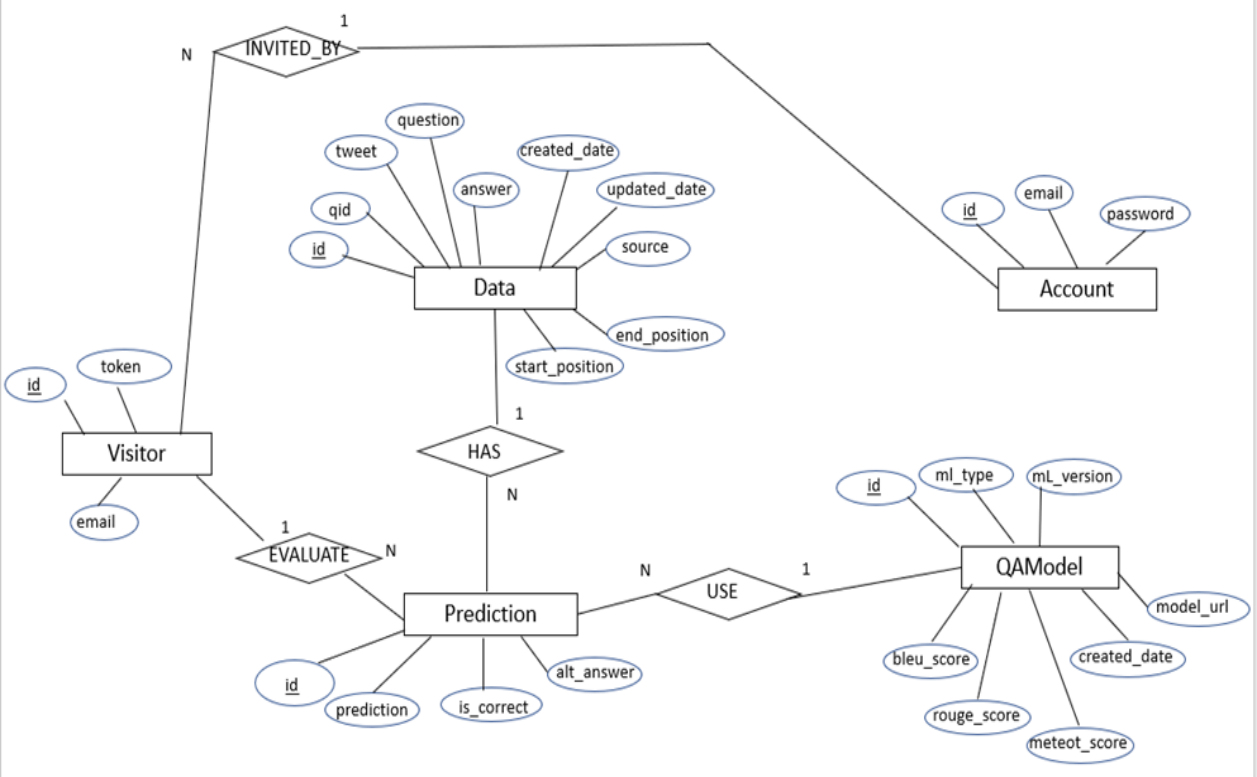


Figure 6.12: Entity Relationship Diagram for the database

A database schema designed for the relational database is presented below, a further normalization analysis of the database schema is presented in section 6.5.4.

Table DATA

Data_id	qid	tweet	question	answer	created_date	updated_date	source	start_position
end_position								

Table PREDICTION

Prediction_id	datum	model	visitor	prediction	is_correct	alt_answer

Note: datum, model and visitor are foreign keys that refer to the Data_id, QAModel_id, and Visitor_id respectively.

Table QAMODEL

QAModel_id	ml_type	ml_version	model_url	created_date	bleu_score	rouge_score
metero_score						

Table Account

Account_id	email	password

Table Visitor

Visitor_id	email	token	invitor_account

6.3. User Interface Design

6.3.1. Admin UI

Admins will have to have an associated email and password to access the system. When requesting the admin login page the admin will be prompted to provide both the email and password to authenticate. This interface is captured below in **Figure 6.13**.

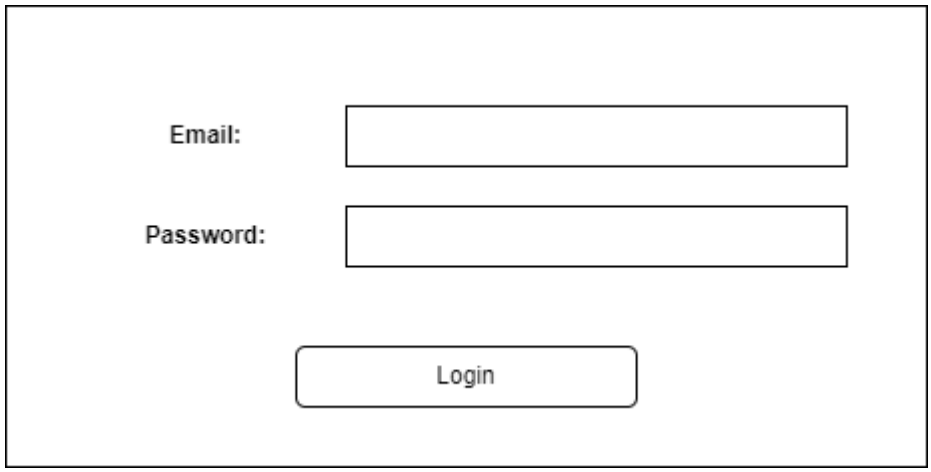


Figure 6.13: Admin Login UI

Once logged in, the system will default to the Invite Users page with a navigation pane on the right side to navigate to other admin-specific pages. The Invite Users page interface is captured below in **Figure 6.14**, which will prompt the admin to provide a given email to invite new users to the system.

Figure 6.14: Admin Invite Users UI

Using the navigation pane an admin can navigate to the Deploy Training page to allow the admin to specify a model and set of hyperparameters. On submission the backend will automatically deploy the machine learning pipeline to train the model with the given hyperparameters. The Deploy Training interface is captured below in **Figure 6.15**.

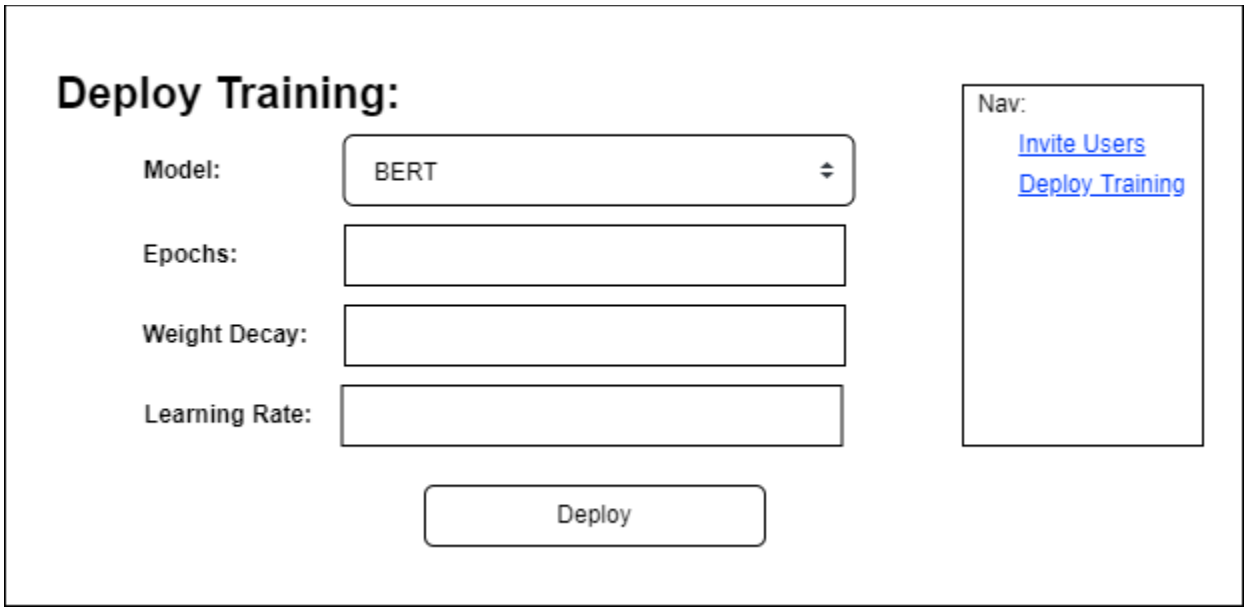


Figure 6.15: Admin Deploy Training UI

6.3.2. User UI

From the user side, the web application will utilize a single page with three different states. Upon accessing the website the user will be presented with statistics describing the default model including current performance metrics and historic metrics as the model has been trained over time. A word cloud describing the current dataset will also be visible. On the right side, the user will see a form to select a different model, besides the default model, and a form to submit a tweet-question pair. This initial user UI state is captured in **Figure 6.16**

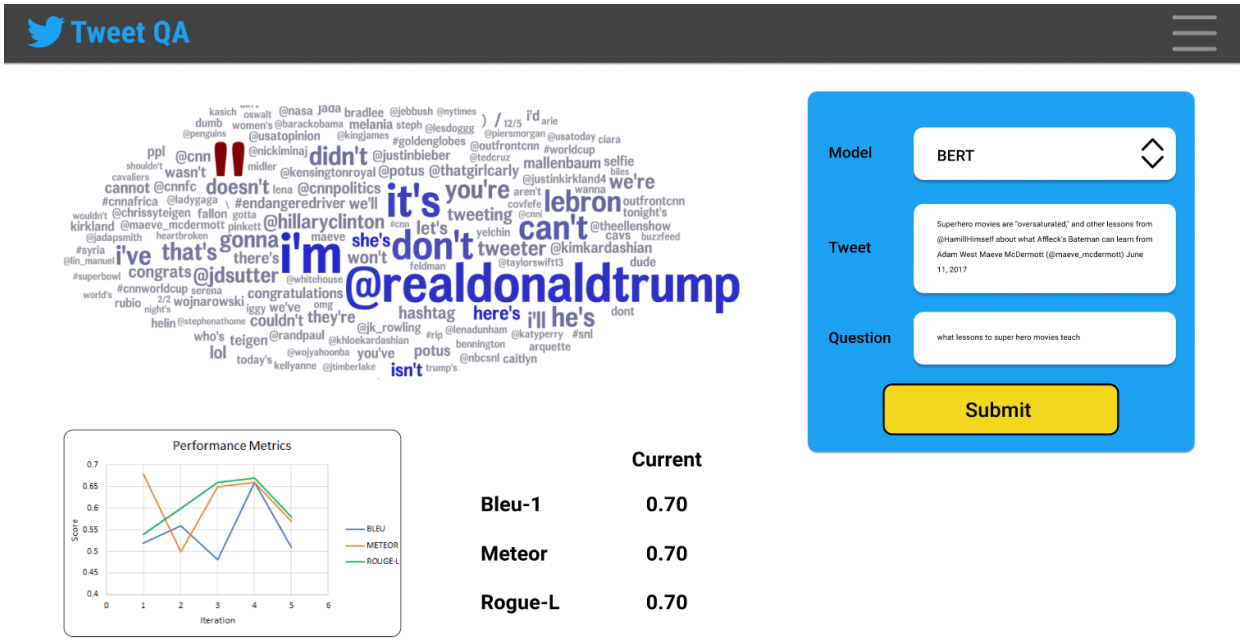


Figure 6.16: User UI - State 1

Upon submission of the user input form, the webpage will provide the machine learning generated answer, and then a radio button field to indicate whether the derived answer was correct or not. This user UI state is shown below in **Figure 6.17**. If the user selects yes and submits, the webpage will return to the previous state, but if the user selects no then the webpage will advance to a third state prompting the user to provide the correct answer. This user UI state is captured in **Figure 6.18**. Once submitted, the webpage will return to the initial state displayed in **Figure 6.16**.

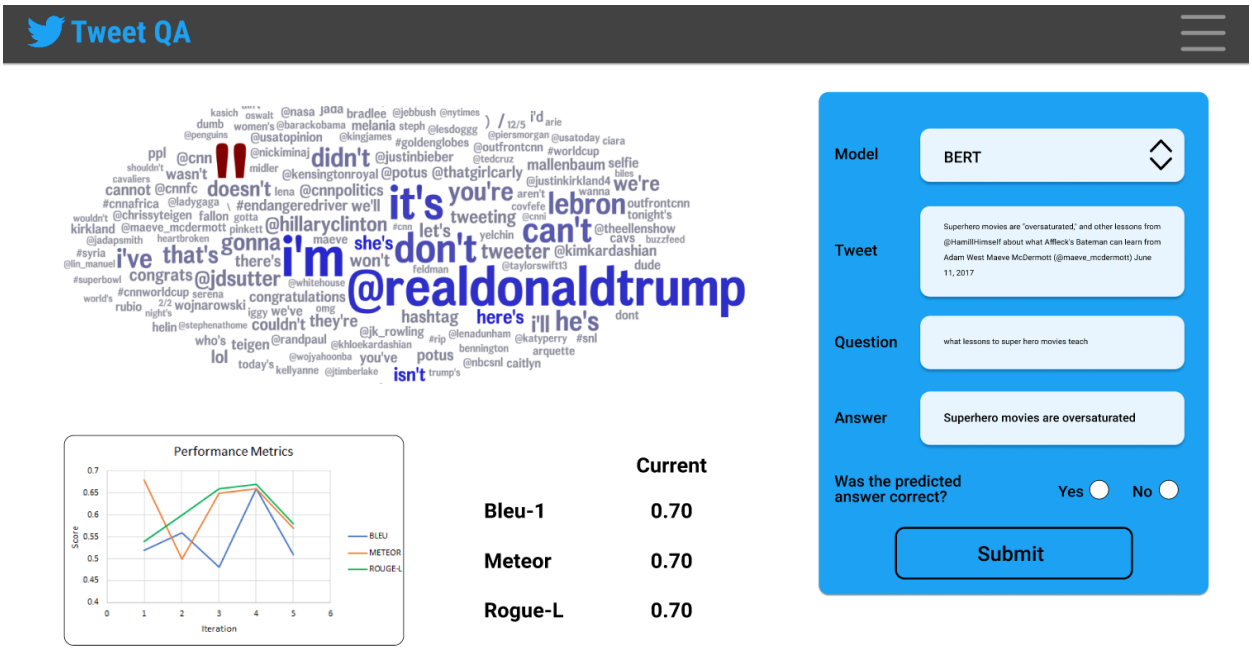


Figure 6.17: User UI - State 2

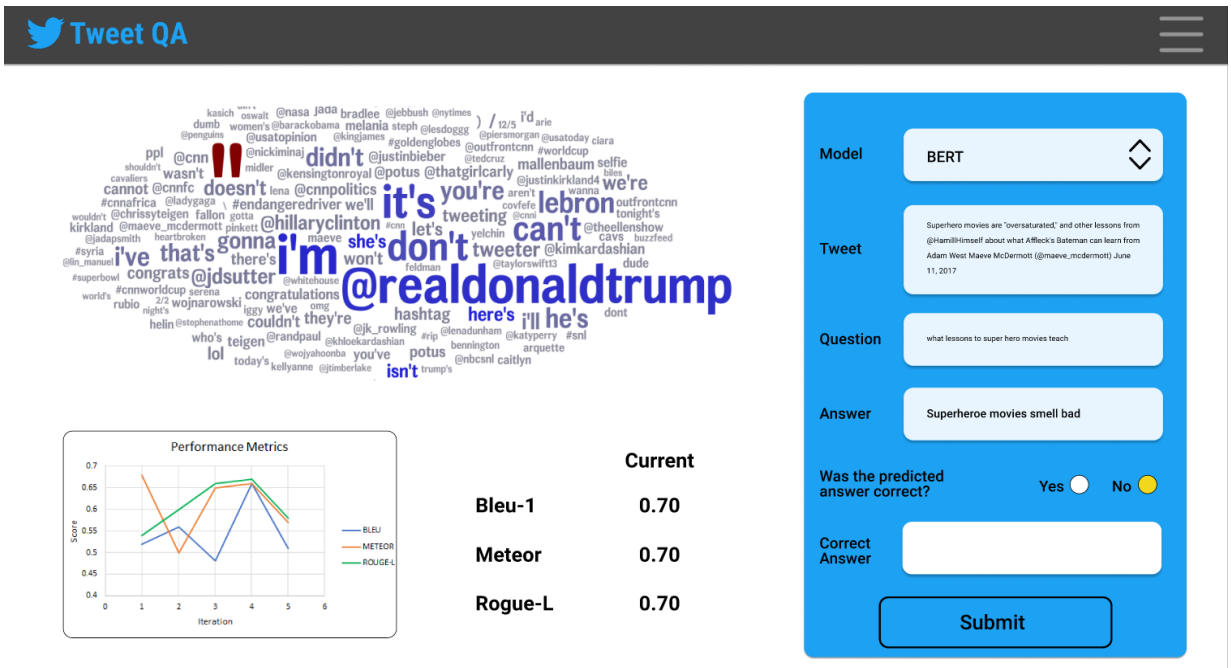
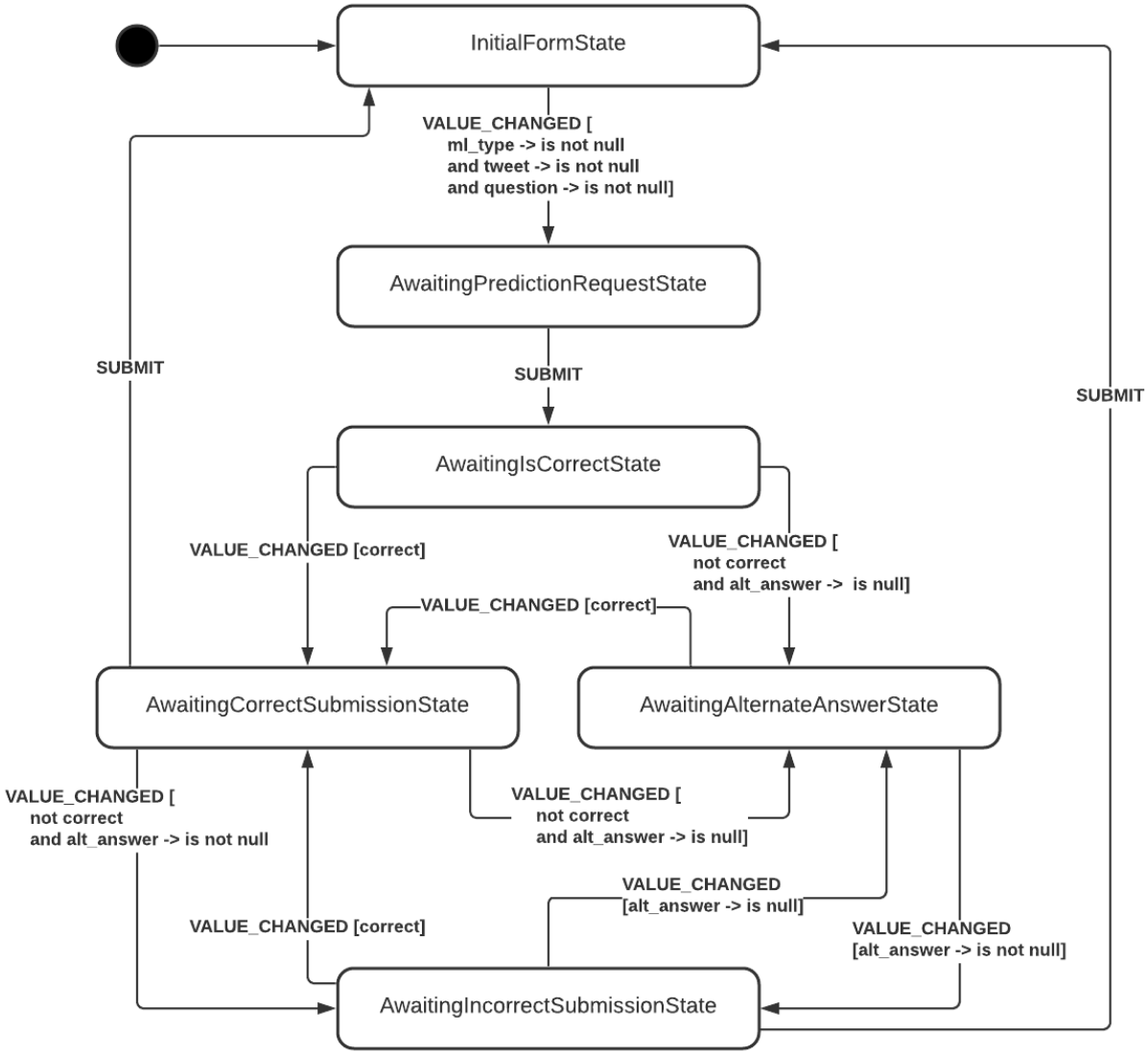


Figure 6.18: User UI - State 3

6.4. Behavioral Design

6.4.1. Web Application State Diagram

As mentioned above the web application front end utilizes a stateful design. **Figure 6.19** below illustrates the possible state transitions. Each concrete state ensures that the form is displayed and behaves properly when in a specific state based on user actions and input.



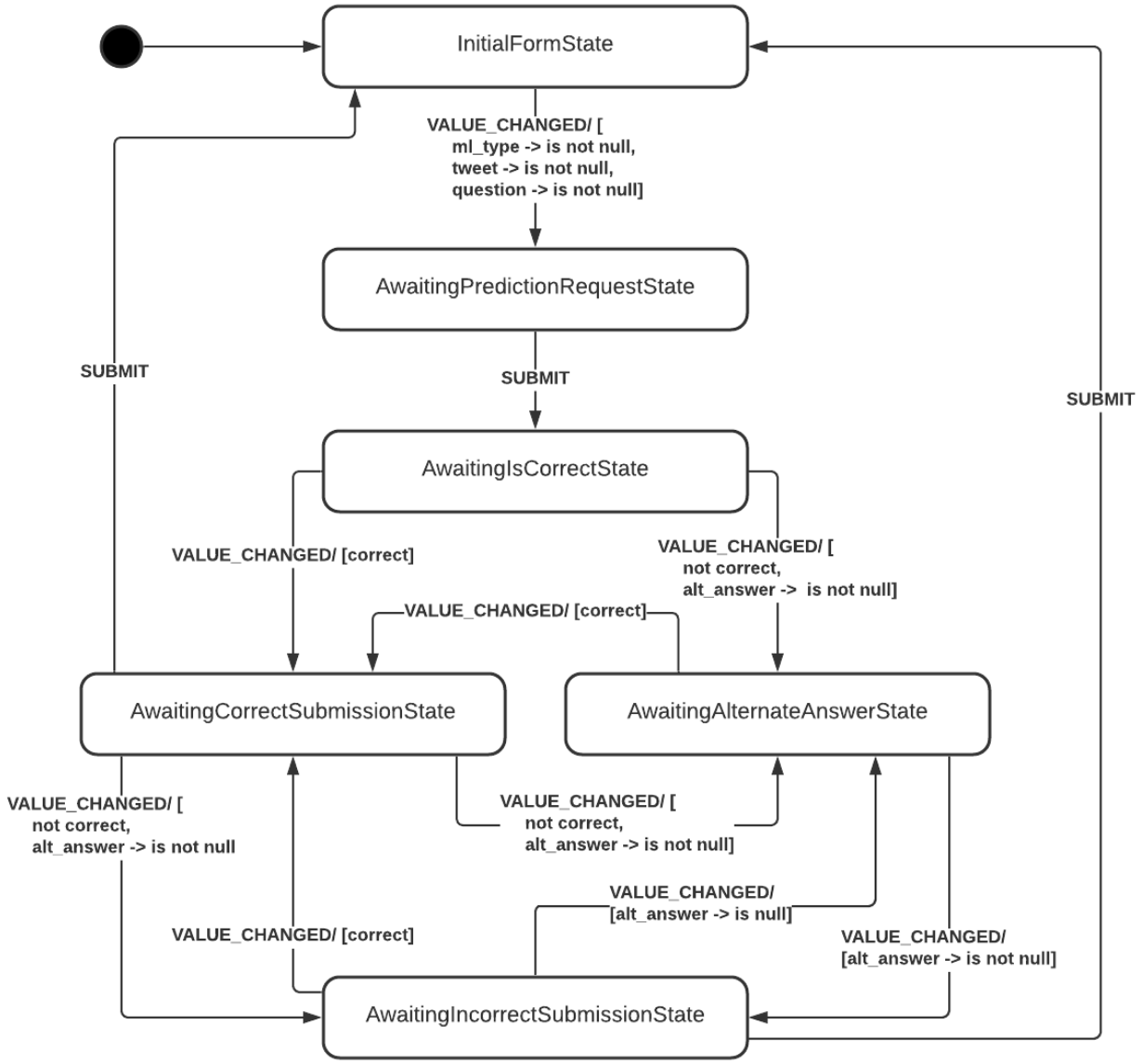


Figure 6.19: Web App Prediction Form State Diagram

When the form is first loaded, it is in the InitialFormState and only the model dropdown selections and the tweet and question fields are available. The submit button is not initially enabled. On a value change action to any of the available fields, the form state may transition to the AwaitingPredicitonRequestState if the machine learning model type, tweet and question values have been set by the user. In the AwaitingPredicitonRequestState, the submit button becomes enabled. Once in AwaitingPredicitonRequestState, the only possible transition is to AwaitingIsCorrrectState on a button submission. The AwaitingPredictionRequestState displays the returned prediction and enables a radio button to collect user feedback on the correctness of the prediction, but all other fields and buttons are disabled. Once the value for the correctness of the prediction is set, the form transitions to the AwaitingCorrectSubmissionState if the user chooses that the prediction was correct or to the AwaitingAlternateAnswerState if they choose that

the prediction was incorrect. In the AwaitingCorrectSubmissionState, the submission button is again enabled, but the correct button also remains enabled. If the user chooses to submit the answer, the form returns to InitialFormState. However, the user can change their mind and change the value of the correct value. When this value changes, if the user had previously entered an alternate answer, they go to the AwaitingIncorrectSubmissionState and can then submit their incorrect answer. If the user had not previously entered an alternate answer, they go to the AwaitingAlterenateAnswerState. AwaitingAlternateAnswerState is also where the form would be if the user initially chose that the prediction was incorrect. In this state, a text field to collect an alternate answer from the users is visible. The submission button is not enabled. A user can always navigate back to AwaitingCorrectSubmissionState by choosing that the prediction was correct. Once a user enters an alternate answer, they enter the AwaitingIncorrectSubmissionState, where the submit button is enabled. If the response is submitted from this state, the form again returns to the InitialFormState.

6.4.2. Query model, Evaluate Prediction and Rerun Sequence Diagram

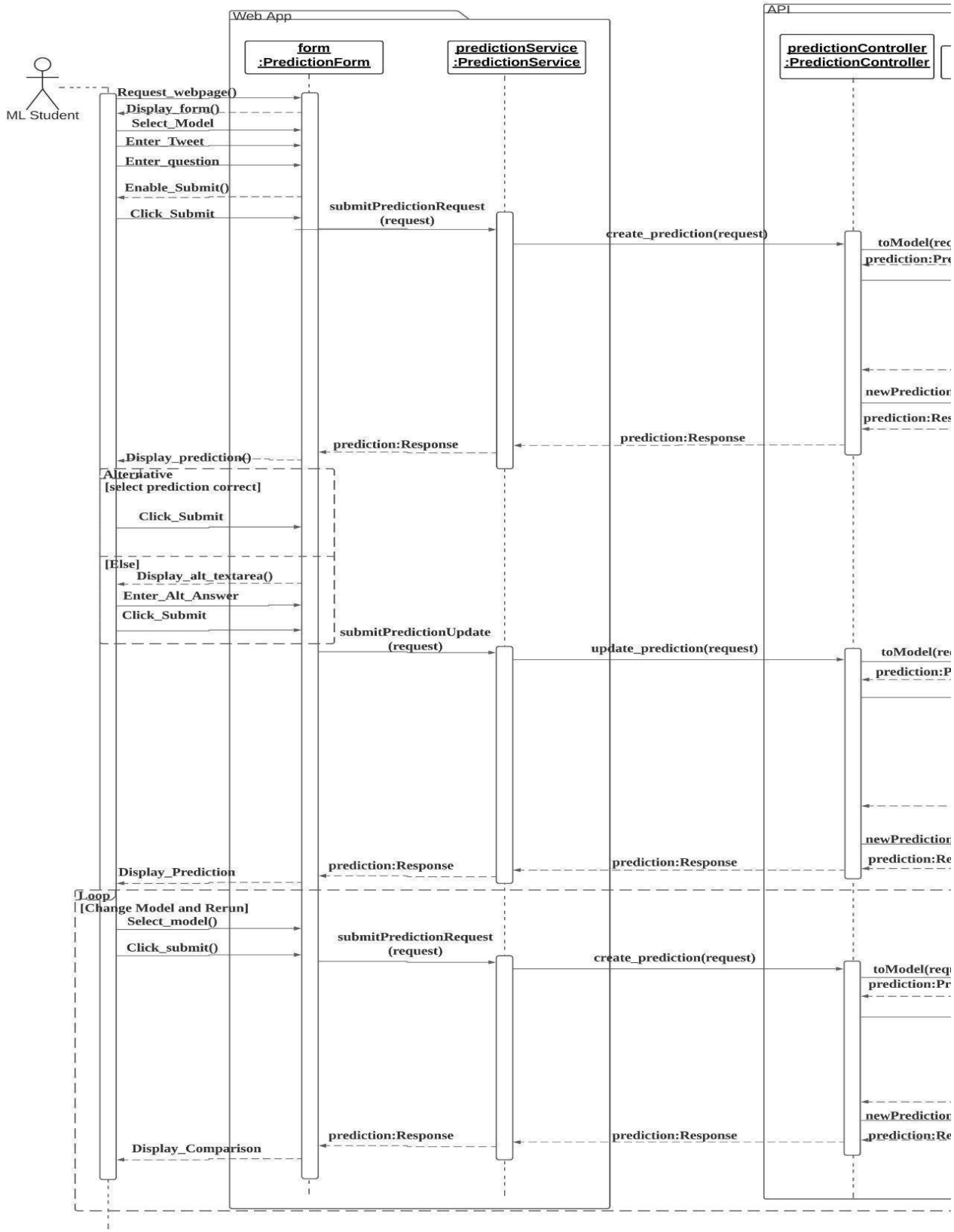


Figure 6.20 Web App side of the sequence diagram

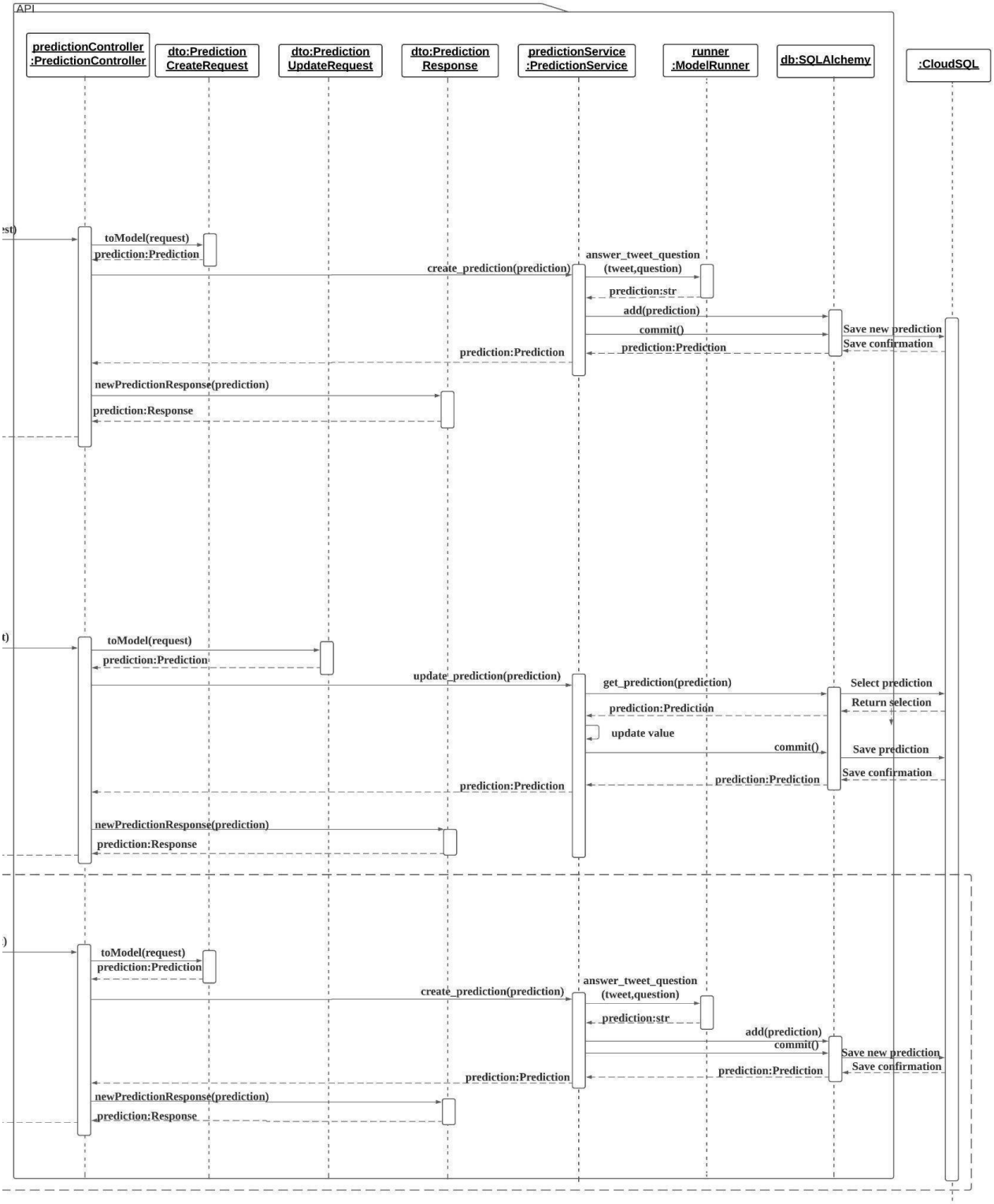


Figure 6.21 API side of the sequence diagram

Figures **6.20** and **6.21** show the interactions between the objects inside the web package and the objects inside the API package when the ML student is running use cases 1, 2, and 3. To start, by entering the URL in the invitation email, the ML student will be requesting a webpage from the website with an interface similar to **Figure 6.16** of the user UI. The ML student will proceed to select the model and enter the tweet and the question set. Once the web form object validates the entries, the student is then allowed to submit the form inputs. Upon submission, the prediction form object will pass the submission details as a request to the PredictionService object inside the web package. The PredictionService object will call the PredictionController objects' create_prediction(request) method to pass the request details to the API side. Inside the API package, the PredictionController object will first transform the request into the Prediction class using the dto object of PredictionCreateRequest. The PredictionController object then calls the create_prediction(prediction) method of the PredictionService object. The PredictionService object will first use the ModelRunnner object to get a response based on the tweet and question from the selected machine learning model. Next, the PredictionService object will add the new prediction and commit to the SQLAlchemy database object. The SQLAlchemy db object is responsible for communicating with the CloudSQL instance to implement the changes to the database. Once the addition is successful, the SQLAlchemy db object will return the saved Prediction object to the PredictionController object. The PredictionController object will then call the PredictionResponse object to transform the Prediction object into response format, which is then passed back to the PredictionService object on the web package. Inside the web package, the PredictionService object will continue to return the response to the PredictionForm object, which then displays the prediction response from the ML model to the user.

The ML student can evaluate if the prediction given by the ML model is correct or not as per use case 2. There are two alternative scenarios here: if the ML student deems the prediction incorrect, an alternative answer can be provided by the student; otherwise, the ML student can click submit to confirm the prediction is indeed correct. In both cases, after the ML student clicks the submit button, the PredictionForm object will pass the request to the PredictionService object using the submitPredictionRequest(request) method. The PredictionService object will pass the request to the PredictionController object in the API using the update_prediction(request) method. Inside the API package, the PredictionController object will use the dto object of PredictionUpdateRequest to convert the request into a Prediction class object. After this, the PredictionController object will call the PredictionService object's update_prediction(prediction) method. This will first retrieve the current prediction using the get_prediction(prediction) method of the SQLAlchemy db object and update the values of the retrieved prediction based on the Prediction object passed from the PredictionController object. After that, the PredictionService object will commit the changes to the SQLAlchemy db object, which will update the prediction record in the SQL instance. Similar to use case 1, the SQLAlchemy db object will return the prediction to the PredictionService object upon successful update confirmation. The PredictionService object will then pass the prediction back to the PredictionController object, which will use the PredictionResponse object to transform it back into the response format. The PredictionController object will then return the response to the PredictionService object on the web side. The PredictionService object will return the response to the PredictionForm object, which will display the update confirmation to the user.

The last part of the sequence diagram corresponds to use case 3 when the ML student decides to choose a different model and rerun it. In this case, the interactions between the system objects are similar to use case 1 as described before. One thing to note is a new prediction record will be created every time the ML student chooses a new model and reruns the prediction. Once the response is returned and displayed to the student, the ML student can evaluate the response as per use case 2.

6.4.3. View Metrics Sequence Diagram

When accessing the web application or selecting a new model type ML student users will be able to view the metrics and statistics related to a particular ML model and a visualization of the current dataset used to train the models. **Figure 6.22** below describes the interaction between system objects to collect and display those metrics and data visualization. Upon requesting the webpage or selecting a new model type, the TweetQAForm instance will dispatch the QAModelService object to request the metrics by model type from the API. The model type is either the default model on a webpage request or a user-selected model. On the receiving end of the API, the QAModelController object will pass the request to the QAModelService object, which will then utilize the SQLAlchemy instance to query the database. The selected model object will then be passed back to the API's QAModelCotnroller object. The controller object will then extract the necessary fields and package them into a response to send to the client-side. Upon receiving the response, the webpage will populate the given model's metrics and the data visualization.

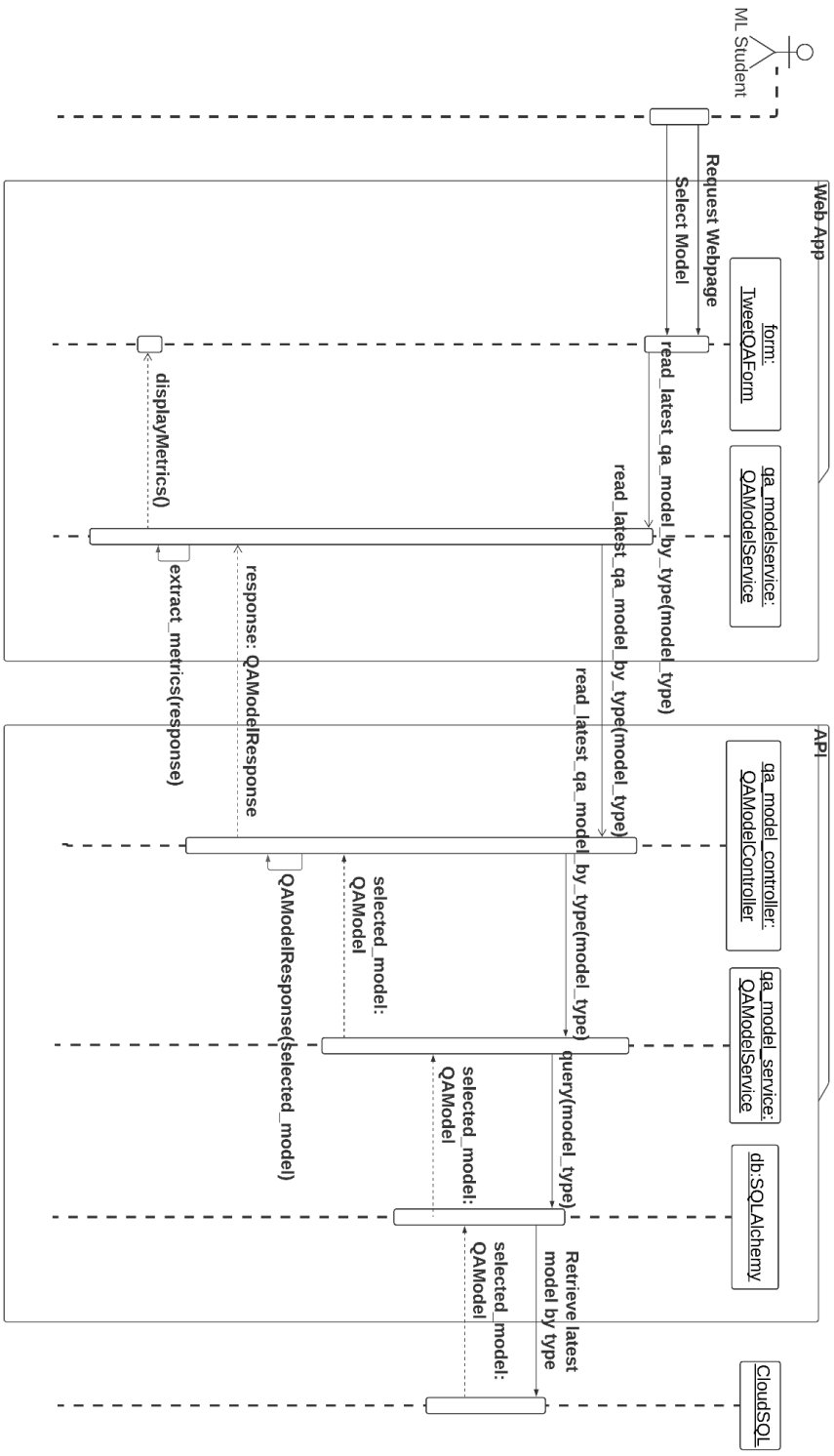


Figure 6.22: View Metrics Sequence Diagram

6.4.4. Admin Login Sequence Diagram

Figure 6.23 describes the interactions between system objects when the admin tries to log in to the website. When an administrator would like to log in to gain access to administrator-only functions in the web application, they must first enter their credentials into the login form on the website. The admin user is then required to submit their credentials, again using the login form. The LoginForm object calls the login method from the accountService object and passes the login credentials (in the form of a LoginRequest object) to the method. The accountService object then makes a LoginRequest to the API which is handled by the login method of the AccountController object.

The AccountController passes the email and password from the LoginRequest to an AccountService object's login method. The login method of the AccountService object will first call another method within the service to get an account by the provided email. As a part of that process, the AccountService object will use its instance of the SQLAlchemy class to query the database using the email. Upon retrieval of the account from the database, the AccountService object will make a call to its authenticate method to verify that the provided email and password match those of the retrieved account.

The response from the authentication call is sent back to the AccountController object. The AccountController object uses the AuthenticationResponse class to create a new instance of AuthenticationResponse to return to the client. The Controller object then responds to the client's initial request with the AuthenticationResponse object.

In the AccountService object on the client, upon receiving the AuthenticationResponse object there are two possible outcomes. If the login was successful the AccountService object will save the returned token for use in subsequent API calls. The AccountService object will respond to the LoginForm object and then the LoginForm object will display a login success message to the user letting them know everything worked as expected. If the login was unsuccessful, the AccountService object will notify the LoginForm object and the LoginForm object will display a login error message to the user.

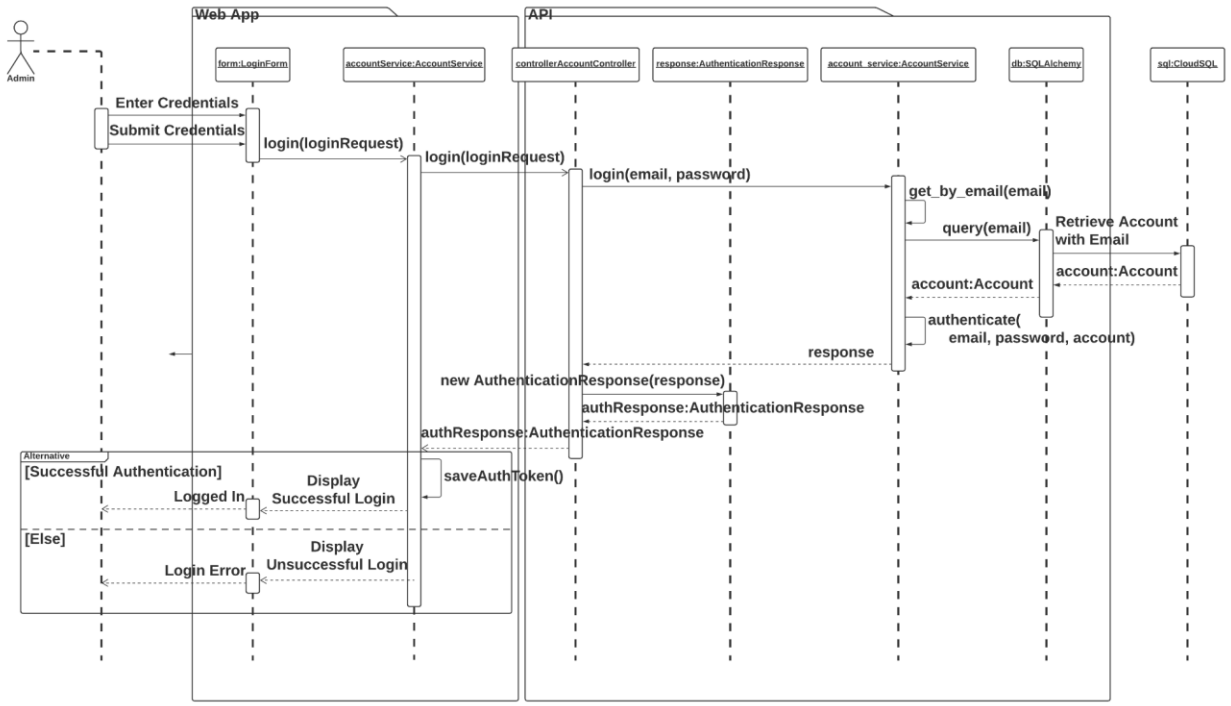


Figure 6.23: Admin Login Sequence Diagram

6.4.5. Admin Model Training Sequence Diagram

Before an administrator can initiate a new model training, they must first log in. The admin user will follow the Admin Login behavior outlined above and referenced in **Figure 6.23**. Once successfully logged in, **Figure 6.24** shows the interactions between the system objects for the admin to train the model. The admin will enter the training hyperparameters, select a base model and select the date parameters for the desired training data using the TrainingForm object. Once complete, the admin will submit the request using the TrainingForm object. Upon submission, the TrainingForm object will use an object of TrainingService to create a new training request. The TrainingService object will send the TrainingCreateRequest object to the API where the TrainingController object's create method will handle the request.

The TrainingController object will rely on the received TrainingCreateRequest object's toModel method to convert the data structure to a Training model object to be used by the service. The TrainingController object will then call the create method from an instance of TrainingService. The TrainingService object will first use an instance of an MLPipeline object to initiate a new machine learning training. The TrainingService object will not wait for the training to complete (this could take days), so only waits to get a response that the training was successfully started.

Upon notification of success, the TrainingService object will save the parameters used to initiate the training to the database. This is done by using the add method from the TrainingService's instance of the SQLAlchemy class. Once added, the SQLAlchemy class's commit() method must be called to persist the change to the database. Upon receiving confirmation of a successful save

from the database, the TrainingSercice object will query the database for the newly created Training object to later return it to the client. Once the object has been retrieved, it is passed back from the TrainingService object to the TrainingController object.

The TrainingController object uses the TrainingResponse object's constructor to create a new instance of TrainingResponse from the received Training object. The TrainingController object then responds to the client by sending it the TrainingResponse object. Upon receipt of the TrainingRequest object, the TrainingService object notifies the TrainingForm instance which in turn displays the success to the admin.

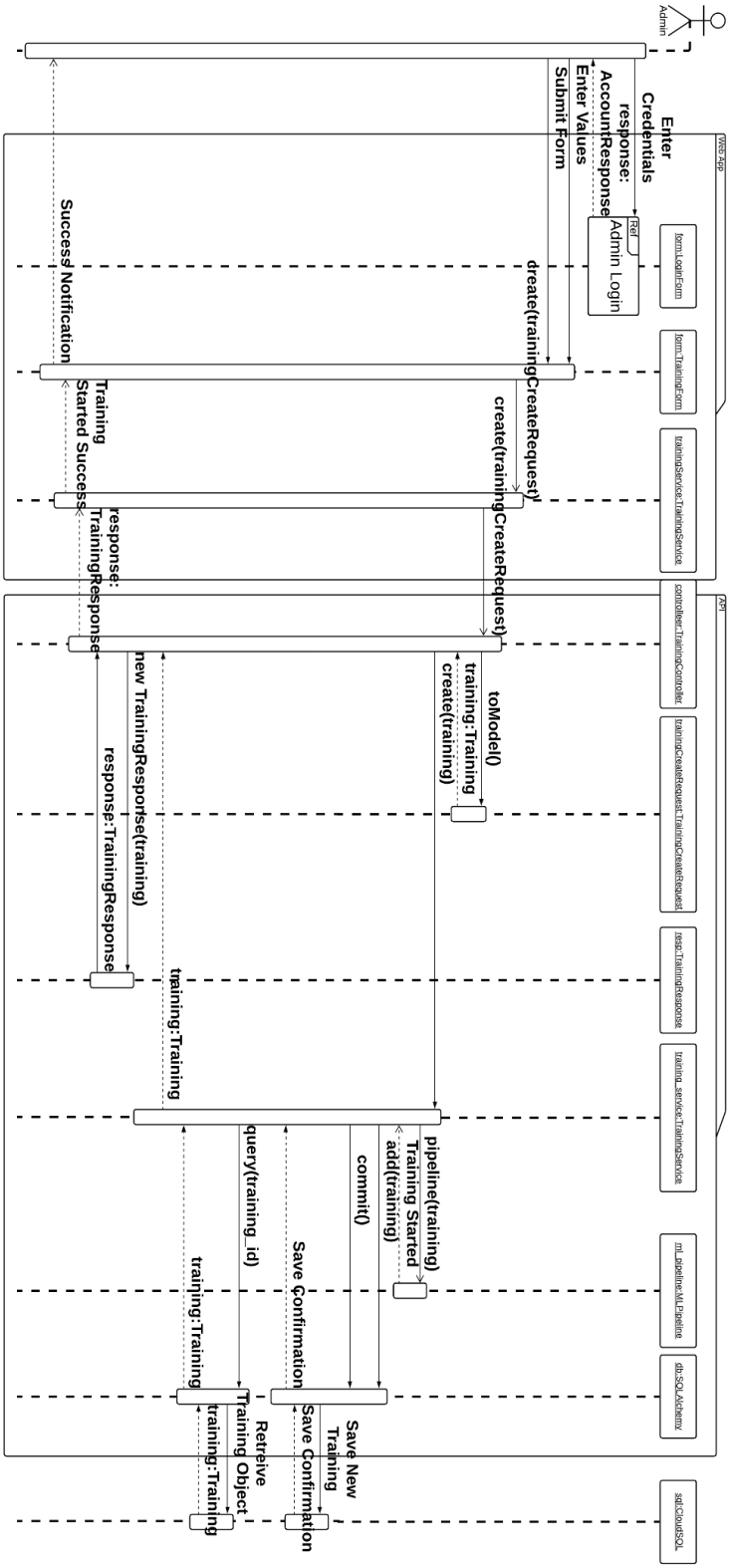


Figure 6.24: Admin Model Training Sequence Diagram

6.4.6. Visitor Create and Invitation Sequence Diagram

Similar to the new model training sequence described in the previous section, the administrator must begin by logging in (as referenced in Figure 6.23), which is displayed at the beginning of the Visitor Create / Invitation Sequence Diagram in **Figure 6.25**. Once logged in, the administrator enters the email address into the InvitationForm for the visitor they wish to invite to the system. Afterward, they can submit the form which passes the email address for the new visitor to the VisitorService object of the frontend in the form of a VisitorCreateRequest object. The service takes this request object and sends it to the API endpoint that handles the VisitorCreateRequest object.

Upon receipt of the request, the API endpoint in the VisitorController object uses the toModel method on the Request DTO to create an instance of the VisitorModel class. This model instance is passed to the create method of the backend VisitorService instance. The VisitorService generates a UUID for the token, which is virtually guaranteed to be unique, and stores it in the model's token property. The VisitorService instance passes the model object to the SQLAlchemy database object's add method, and then the commit method is run to tell the ORM to solidify the changes to the database. The ORM handles the forming of the SQL insert query and sends it to the actual database in the CloudSQL object, which response to the SQLAlchemy object with a confirmation of success. The confirmation is passed back to the VisitorService instance.

The VisitorService instance queries the database via the SQLAlchemy ORM object requesting to retrieve the Visitor entity that was just stored. The ORM again translates the request to SQL and sends it to the CloudSQL object which response with the new Visitor object. At this point, the VisitorService instance can finally send the email to the new visitor, containing a URL with the token as a query parameter.

The new Visitor model is passed back to the controller object. It converts the model to a VisitorResponse DTO and sends that back to the frontend VisitorService object. Upon receipt of the response, the VisitorService object indicates to the InvitationForm object that the request was successful. Finally, the form indicates to the administrator user that the Visitor was created and the invitation containing the new token has been sent to the email address entered.

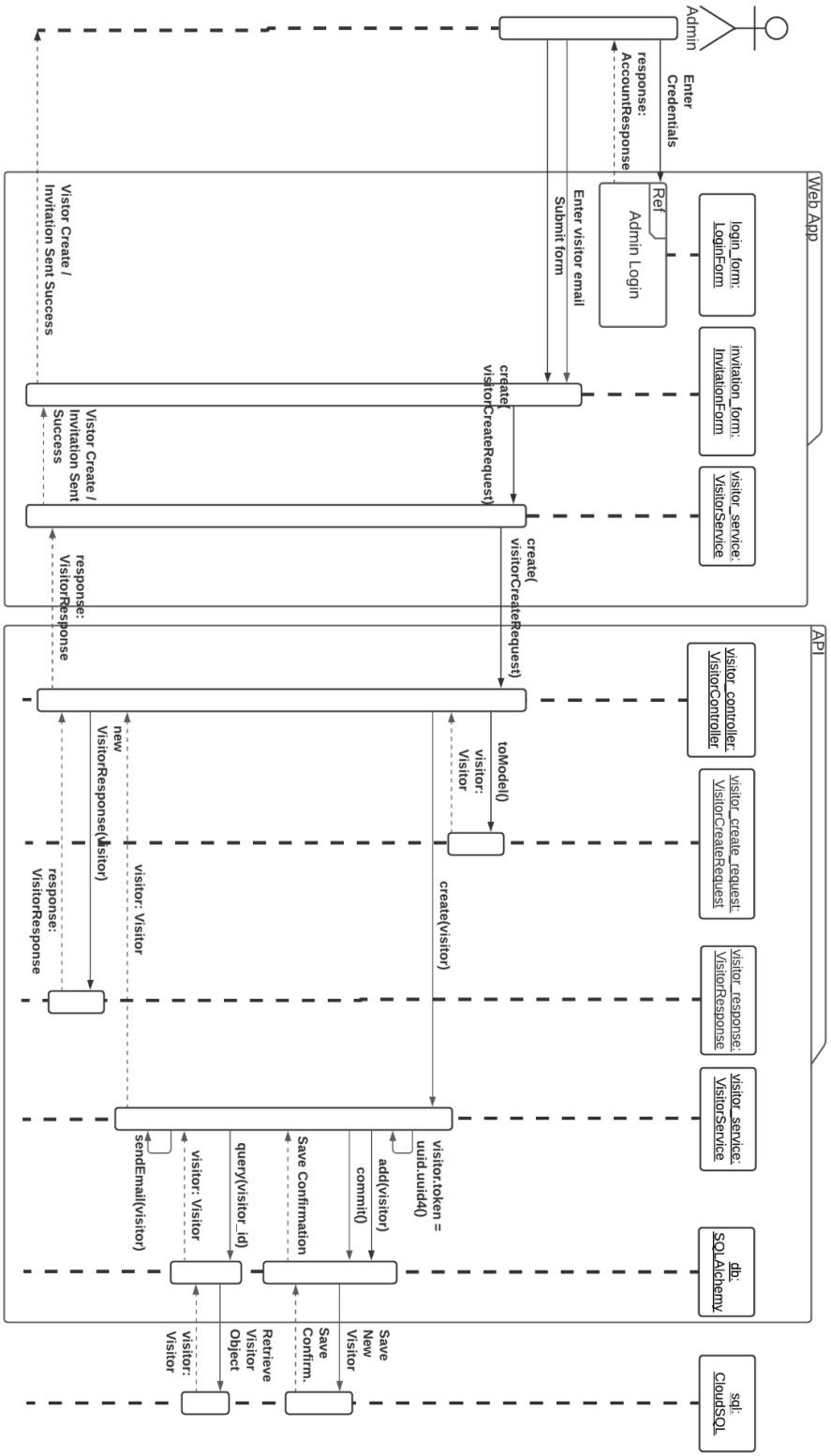


Figure 6.25: Admin Visitor Creation / Invitation Sequence Diagram

6.5. Design Patterns

6.5.1 Web App

6.5.1.1 State

The State pattern is used to manage the various possible states the prediction input form can be in as there are several discrete possible states for the form [30]. The State pattern consists of an abstract class that defines the attributes of each stateful object, an abstract method to determine the next state object, and another abstract method to define the necessary state changes as the next state object is entered. When a class uses a state object, it initializes to a subclass that has inherited from the abstract state class. Each time an action happens that may affect the state, the next state method is called and returns an instance of the class of an object representing the next state (which will also inherit from the abstract state class). Finally, the enter method of the next state is run to adjust the state attributes of the class.

In our design, the PredictionRequestFormState is the abstract class that fulfills the abstract class role. Each possible state that the form can be in is captured as a class that inherits from the abstract PredictionRequestFormState class. Each state class defines certain attributes, such as the visibility of certain inputs and whether visible components are enabled for the user or not. A PredictionRequestFormState object is used within our Prediction form component via the formState attribute and is initialized to the InitialFormState class upon the app's first loading. Each time the user interacts with the form by changing a value in a field or pressing a button, an action is passed to the next state method to determine if a new class type of the PredictionRequestFormState is needed. If it is, an instance of that type is returned and the formState attribute is updated. Via this pattern, we can ensure that only the elements of the form that a user should be able to interact with are enabled at any given time. In addition, because each form state is captured by a unique class type, unit testing the form is much simpler. After certain inputs have occurred, we can be sure that the correct UI is presented to the user if the class is of the correct form state.

6.5.1.2 Observer

To better manage our application state, we rely on the Angular NGRX framework, which is heavily influenced by the Observer design pattern. In any of our UI components that have stateful elements, the component will subscribe to the object (also known as an Observable). The component then becomes an Observer and any changes to the Observable in the future will notify the Observer of those changes [30]. In this way, changes will propagate through the system in real-time. This also enables a single source of truth for all application states, which means that all components will be working with the same, up-to-date data.

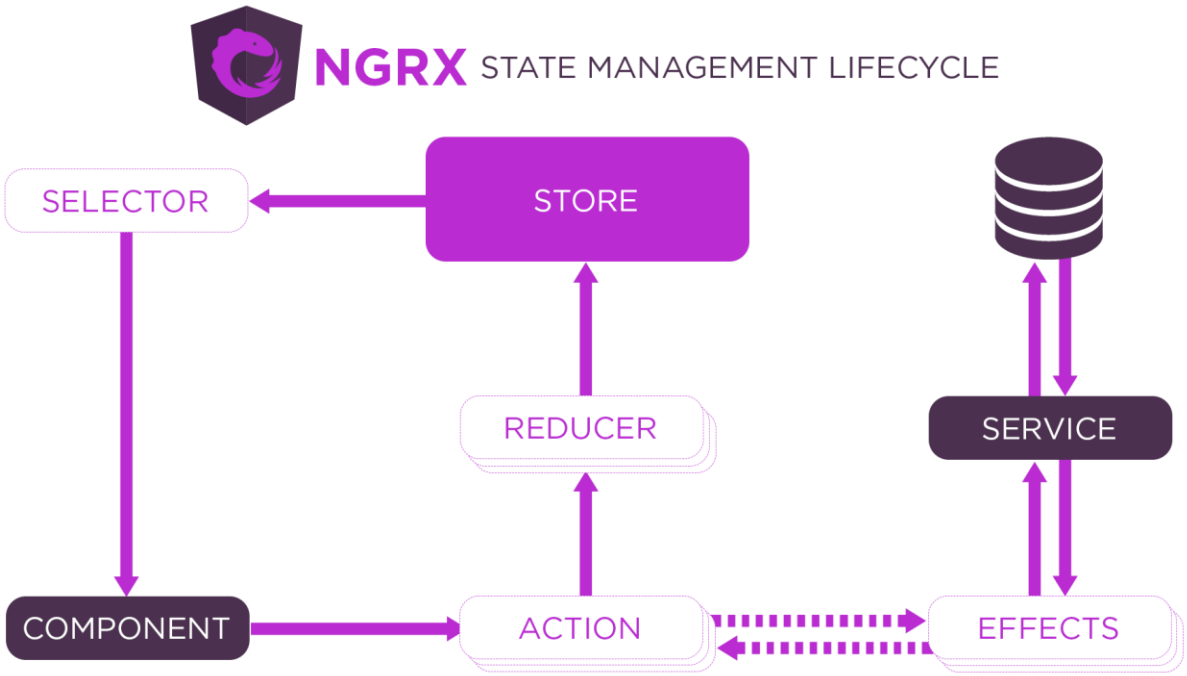


Figure 6.26: Observer Pattern via the NGRX State Management Framework [21]

In our application, using NGRX, the underlying concepts of the Observer pattern as described above are present, but additional elements are also present. All application state is defined in the Store object. A series of functions, called selectors, are defined that specify what segment of the application state should be returned to the observing component. A component will subscribe to a selector, which will be called and return a segment of the state from the Store anytime the underlying state is updated. Another important element of the framework, however, is that the state objects being observed can only be updated by calling an action. Actions are defined with a type attribute and any properties that should be passed along. Actions can be initiated by any component in the application. There are then a series of functions known as reducers that listen for certain types of actions. If the specific type of action a reducer is listening for is identified, then the reducer will update the state in the Store as defined in the reducer. The change of state in the Store will then notify any applicable selector, which in turn updates the state of any observing components to complete the observable life cycle.

6.5.1.3 Factory

Our web application makes use of a third-party library called Cooky Cutter to create mock objects of our data types for use during testing. The Cooky Cutter library allows us to easily define a factory for a specific data type [31]. Cooky Cutter's defined method takes a Typescript object as an argument which defines the object the factory will return. You can later call the factory method that Cooky Cutter generated using the provided object definition to create an instance of the object.

6.5.2 Web API

6.5.2.1 Abstract Superclass

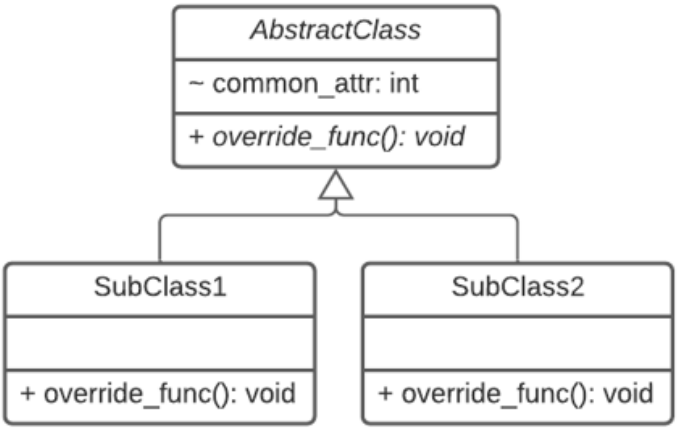


Figure 6.27: UML class diagram showing simple example Abstract Superclass pattern

The abstract superclass design pattern is used to create generalization relationships between entities, especially when the general type that is inherited does not make sense to instantiate on its own [30]. In this case, the general type would be marked as “abstract” which means that the type cannot be directly instantiated; rather, one of its subclasses must be instantiated. Another strong benefit of this pattern is that it allows subclasses to override superclass members so that custom implementation differs based on the child type, which is the most powerful aspect of polymorphism.

The web API makes use of the abstract superclass pattern in multiple areas, the services layer is one example. There are several different service classes, which are illustrated in figure 6.4. The concrete classes perform business logic and interact with the database via the SQLAlchemy framework, and some classes are restricted only to reading from the database, while others can read and create entities in the database, and then others can read, create, and update entities.

There are three abstract superclasses in the service layer: ReadService is the most general superclass, CreateReadService inherits from ReadService, and CreateReadUpdateService inherits from CreateReadService. Each of the abstract superclasses has an abstract method that enables an inheriting class to read, create, and update data in the database. The current design doesn’t contain any subclasses that directly inherit from ReadService. QAModelService inherits from CreateReadService. Since the QAModel entities are designed to be immutable once stored, the design specifically doesn’t allow for a QAModel to be updated, but it does need to be able to read and create entities. The DataService, on the other hand, inherits from CreateReadUpdateService because it must be able to update entities after they already exist. The inheritance from the abstract classes forces the developer to implement the abstract methods.

6.5.2.2 Controller-Service-Repository Pattern

The controller-service-repository design pattern is common in programs that are built on data storage and retrieval, especially when combined with a client-server architecture [32]. This pattern enforces encapsulation and single responsibility for each layer. The controller objects handle the data transfer between client and server and are typically what handle the requests and responses. The controller should ideally contain no business logic whatsoever and should pass the request data (if applicable) to its corresponding service for that purpose. The service object validates and transforms the request data as needed. If the request includes data storage or retrieval action, and if the requested data is valid, it will call upon the repository object to perform the direct interaction with the database. The repository layer doesn't contain any business logic, it is strictly responsible for connection with the database and forming the query, sending the query, and then passing back the database response to the service. The service transforms that response (see section 6.5.2.4), sends the transformed data up to the controller, and the controller sends the status of the request, as well as any relevant stored data, back to the client.

The web API uses this design pattern since it acts as the server in the client-server architecture and because data storage and retrieval are core requirements of the system. The upper-left portion of the package diagram in figure 6.1 shows the dependency between the controllers and the services. The repository layer isn't shown in the figures since it is handled by the third-party SQLAlchemy framework. The use of this pattern in the web API means that there is no confusion about the role of each layer, making the development process more straightforward. Different developers can implement different classes in this pattern and there will be consistency because of the rules that this pattern imposes.

6.5.2.3 Singleton

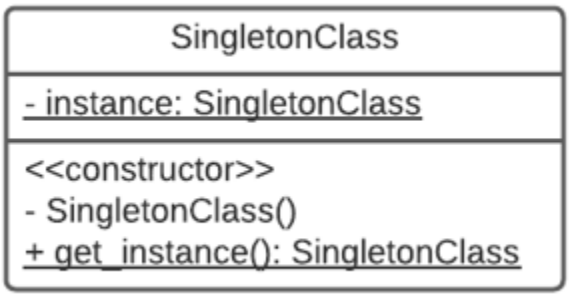


Figure 6.28: UML class diagram showing simple Singleton pattern example with static members

The singleton pattern is used when the maximum number of instances for a given class is one [30]. This is often used by utility classes and those which don't represent an element in a collection of entities. This is especially desirable in cases where synchronization is important in multiprocessing, when resources have to be reserved for the lifetime of the object, and when an object uses large amounts of resources, especially memory. The instance count is limited specifically for the purpose of handling concurrent requests that can be placed into a singular queue and/or to control the allocation of system resources. Typically the class will contain a private constructor and static fields that make it impossible to instantiate more than one instance

accidentally (reference **Figure 6.28** for an example).

The web API uses the singleton pattern for a couple of different objects. The primary one is the "app" object which represents the Flask server instance itself. If multiple instances of this object were allowed, there would be failure to bind to the network ports as only one process is allowed to do so. The "db" object is another singleton that represents the repository object explained in section 6.5.2.2. This SQLAlchemy object handles the connections and transactions with the database. Only one instance of this object should be allowed to exist as the object itself has its object pooling to deal with resource management and queuing for transactions.

6.5.2.4 Data-Transfer-Object

Figure 6.29: UML class diagram showing an example of a Model class and its DTO class

The data-transfer-object (DTO) pattern requires the existence of classes with the sole responsibility of handling data transfer and includes no business logic, other than perhaps a convenience method for mapping to its corresponding model object [33]. Multiple classes of DTOs can be created to fulfill multiple request or response purposes. For example, one DTO might be just for create requests by a client. This DTO should contain the minimum amount of data to fulfill this purpose in order to keep the payloads as lightweight as possible. Another example is a response type of DTO, which similarly would contain only the relevant information needed by the client, and would not contain any sensitive data that might otherwise be related to the object on the server which shouldn't be exposed to the client for security reasons.

The web API makes use of the DTO pattern to handle requests and responses with the web application client. The controller layer receives DTOs from the client and passes them to the service object which maps the DTO to the model object. The service object also maps model objects received by the database to DTOs, which are passed to the controller object to send to the client in response.

6.5.2.5 Asynchronous Processing

The asynchronous processing design pattern allows for threads to be created for handling concurrent operations [30]. This is also known as multiprocessing, which has the benefit of using the operating system's built-in scheduling algorithms to perform multiple computations at any given time. This can be complicated especially when shared resources are involved which can lead

to deadlocking, a state in which the process cannot complete tasks due to two or more threads requiring the same locked resources at the same time. Fortunately, this can be effectively mitigated through prevention and detection techniques. The use of asynchronous processing is extremely common in client-server architectures since typically any given server process can have multiple client connections at any given time. The server must be able to accept and handle requests in a timely manner, so usually, the server process will start a new thread when a request is received so the OS can schedule the processing.

The web API in our system, as a web server, uses this design pattern in order to be able to allow multiple instances of the web application client to make multiple simultaneous requests at the same time. The requests are handled concurrently so that no response takes too long which would make for a poor user experience.

6.6. Design Alternatives & Decision Rational

6.6.1. Architectural Design: Client-server versus SPA versus desktop application

A client-server architecture proved to be the most feasible pattern for this system of applications. A web-based system made the most sense to be easily reached by users around the world without having to manage the deployment of updates to many clients. With a web-based approach, you can deploy once on a server for each version change. In a client-server architecture, most of the business logic is performed on the server. For this system, this is important as the ML models consist of large files. Keeping them on the server means that they don't need to be delivered to each client.

A Single-Page-Application (or "SPA") rather than a client-server was considered. A SPA is a type of web application that delegates all of the business logic to the client side. The major benefit to such an architecture is that users don't have to compete for server-side resources as the computation is done on the user's machine. The major pitfall for a SPA is that up-front loading times are longer as the entire application has to be downloaded to the client browser when the website is visited. Since this application consists of several ML models and each of these models consist of several gigabyte-sized files, a SPA would be an infeasible architecture for this system. Downloading these files could take tens of minutes or hours on slower connections which is unacceptable for the system requirements.

Another architecture that was considered was a desktop application. The benefit to desktop applications versus web-based systems is that the application can take advantage of native operating system functionality such as direct control over hardware, interactions with the user filesystem, etc. There are several downsides that include the large download size of the ML model files also present in a SPA, more complicated deployment, and more difficult updating of the application on the user's system. Furthermore, the operating system interactivity provided by a desktop application simply doesn't have any compelling use in the system described in this document.

6.6.2. Structural/Behavioral Design: ORM vs DAO Layer

Using an object-relational mapping (ORM) library such as SQLAlchemy allows us to create rich relationships between our persistent data, and be able to quickly write complex queries against the database. The alternative to using an ORM library would be to implement a low-level DAO layer where classes would handle different create, retrieve, update, and delete operations to the database and the necessary operations would exist in functions as predefined SQL queries. The latter may offer performance benefits in the form of fewer CPU cycles, but would also be at the cost of increased complexity seeing as how each database operation would have to be predefined. In the case of having non-complex queries or relationships, the DAO layer seems to be the better choice. Although, for our given application the simplicity and high-level abstraction offered by an ORM far outweighed the performance benefits of a DAO layer.

The SQLAlchemy library was chosen as the designated ORM because it offers a simplistic approach for defining model objects, and database access similar to standard OOP operations in Python. For instance, relationships between models can be defined as object attributes similar to a class definition in OOP. When an object is retrieved from the database the other object that is defined as an attribute can simply be accessed through the object's dot operation. Other benefits include being able to utilize polymorphism and inheritance with the database schema. Overall, the SQLAlchemy library allows easy access to our persistent data at the cost of a little more overhead.

6.6.3. Structural/Behavioral Design: GCP ML Pipeline versus Deployed Package

While many machine learning programs are run in a single process that executes from start to finish in one environment, we decided against that approach in favor of developing a machine learning pipeline. In this context, a pipeline is a series of machine learning steps that are executed one by one, each in its own environment.

There are numerous key components of any machine learning process. Data must be extracted from a source, prepared so that it is in the correct format to run through a training process, a model must be trained, the trained model must be scored and validated, etc. When initially developing a new training process, it is common for each of these steps to be handled by a series of methods and run in one continuous program (often on a local machine or in a Google Colab environment). This is a good way to quickly iterate and get a working program. Many training processes do not go beyond this setup. The design is packaged into a single program and can be deployed as-is.

We chose to go a step further. We broke each core segment of the training process apart. Using a tool called Kubeflow, we are able to deploy each segment to its own container with a unique environment. Each process can run independently and share its results with the next step. Because of this, if one step fails, the prior steps will not need to be rerun as their outputs will still be valid. In addition, a development team has the ability to customize the environment within a container. If one part of the process requires a heavier library that another does not, the package only needs to be loaded in the environment in which it is required. The last core benefit is that each individual component of the pipeline can be updated without affecting the other components, as long as the input and output match up with the steps before and after. Each component is unaware of how any

other component is implemented and only relies on the interface between components. This allows for a layered architecture where changes can more easily be made and tracked without affecting existing components that are already working.

While deploying the training process as a single package would have been a more straightforward approach, we felt that spending the time and effort to implement the process as a pipeline would save us time in the future as we will be better able to quickly iterate through new processes and approaches.

6.6.4 Normalization Analysis for the Database Schema

First Normal Analysis

Examining the table and the attributes for the ERD diagram presented in section 6.2, we can list the functional dependencies for the attributes as follows:

Data_id -> qid
Data_id -> tweet
Data_id -> question
Data_id -> answer
Data_id -> created_date
Data_id -> updated_date
Data_id -> source|
Data_id -> start_position
Data_id -> end_position
Prediction_id -> prediction
Prediction_id -> is_correct
Prediction_id -> alt_answer
Prediction_id -> Data_id
Prediction_id -> Visitor_id
Prediction_id -> QAModel_id
QAModel_id ->ml_type
QAModel_id ->ml_version
QAModel_id ->model_url
QAModel_id ->created_date
QAModel_id ->bleu_score
QAModel_id ->rouge_score
QAModel_id ->meteor_score
Account_id -> email
Account_id -> password
Visitor_id -> token
Visitor_id-> email
Visitor_id -> invitor_account

All attributes above contain only atomic values. Next, we begin by creating a universal relation containing all attributes. To remove the need for multi-valued attributes, the following attributes will be added as part of the primary keys:

- Data_id
- Prediction_id
- QAModel_id
- Account_id
- Visitor_id

The entire relation is shown in the following schema.

Data_id	Prediction_id	QAModel_id	Account_id	Visitor_id	qid	tweet	question

Continue:

answer	created_date	updated_date	source	start_position	end_position	prediction

Continue:

Is_correct	alt_answer	ml_type	ml_version	model_url	created_date	bleu_score

Continue:

rouge_score	meteor_score	email	password	token	email	invitor_account

Second Normal Analysis

The second normal form requires that the 1NF table be decomposed into tables where each nonprime attribute is fully functionally dependent on its key. Hence to decompose the table into the 2NF form, we need to select the key and attributes that are fully dependent on it only. This leads to the following tables:

Key selected: Data_id
Table DATA

Data_id	qid	tweet	question	answer	created_date	updated_date	source	start_position	end_position

Key selected: Prediction_id
Table PREDICTION

Prediction_id	datum	model	visitor	prediction	is_correct	alt_answer

Note: datum, model, and visitor are the foreign key, reference to the Data_id, QAModel_id, and Visitor_id respectively.

Key selected: QAModel_id
Table QAMODEL

QAModel_id	ml_type	ml_version	model_url	created_date	bleu_score	rouge_score	metero_score

Key selected: Account_id
Table Account

Account_id	email	password

Key selected: Visitor_id
Table Visitor

Visitor_id	email	token	invitor_account

Note: invitor_account is the foreign key, reference to the Account_id in table Account.

Third Normal Analysis

To ensure all the schemas are in third normal form (3NF), we need to ensure no non-key attributes are used to define other non-key attributes, or in other words, no non-key attribute can be functionally dependent on another non-key attribute.

Starting with the first table DATA, the attributes tweet, question, and answer may not be unique. There could be a different question and answer set based on the same tweet, hence the same tweet will appear in two different records. Similarly, the questions or answers could be repetitive for a different tweet. Hence the tweet, question, and answer cannot be used to define other non-key attributes. Similarly, other attributes (created_date, updated_date, source, start_position, and end position) are not unique and cannot be the key. Note, the last attribute qid can be a primary key candidate, but we select the data_id as the primary key as it will save storage when referenced as a foreign key compared to qid. Note, this table might not be in third normal form as the other attributes can be all transitively dependent on the data_id via the qid. However, as it will not be efficient to further break the table, we will keep the current schema.

The second table PREDICTION is in third normal form with the assumptions that the prediction, is_correct, and alt_ans will not be unique and thus cannot be the key. As all attributes other than foreign keys are non-unique, they wouldn't be transitively dependent on the primary key.

The third table QAModel is in the third normal form with the assumptions that there could be a different model of the same type recorded in the database (thus ml_type and model_url are not unique), or there could be a different model with the same version number recorded in the database (thus ml_version is not unique). The three scores (BLEU, METEOR, ROUGE) also may not be unique. Therefore, there are no attributes that could be transitively dependent on the primary key.

For the Fourth table ACCOUNT, the administrator email can be a primary key candidate, but we select the account_id as the primary key as it will save storage when referenced as a foreign key compared to email. The table might not be in third normal form as the password can be transitively dependent on the account_id via the email, but it will not be efficient to further break the table so we will keep the current schema.

For the fifth table VISITOR, the visitor email may not be unique. The invitation token can be a primary key candidate, but we select the Visitor_id as the primary key as it will save storage when referenced as a foreign key compared to the token. The table might not be in third normal form as the visitor email can be transitively dependent on the visitor_id via the token, but it will not be efficient to further break the table so we will keep the current schema.

6.6.5 Design rationale for the User Interfaces

The User Interfaces for the Admin and the normal user are designed based on the following UI Design principles:

- Consistency Principle - where comparable operations will be activated in the same way. For the admin and user's interactive forms, the rows for inputs and selections will be placed on the top, while the button for submission will always be placed on the bottom.

- Recoverability Principle – where the form will only enable control when the user is permitted to use it. For example, in the User UI, the submission button for the tweet and question set will only be enabled after the input validation has been performed for the inputs. Furthermore, the alternative answer text box will only be enabled when the user clicks not_correct when evaluating the return response from the ML model.

- Color Design Principle – following the color economy where a maximum of 5 +/-2 colors will be used in the UI. Aside from black and white, blue will be used as a general background. A more delightful color such as yellow will be used for user interaction buttons.

- Navigability Principle – following the system's non-functional requirements (**SP-01-02 and SP-01-03),** for the normal user UI, all potential objects for user interactions are presented on the first front page. Thus avoiding the user from getting lost in the pages.

- Screen Layout Principle – which helps focus the viewer's attention on the most important areas. In the admin UIs, all form interactions are presented in the center of the screen. In the User UIs, the main form interaction is highlighted in a blue box contrasting from the background

7 System Implementation

7.1. Programming Languages & Tools

7.1.1. Python

Python is the primary language that will be used within this project. Python is the ideal choice for a machine learning project due to its first-class integrations with core machine learning packages such as Tensorflow and Keras. In addition, Python provides several well-documented packages that assist with large-scale data processing, which is a core component of Machine Learning projects. Python will be used to implement the various segments of the Machine Learning Pipeline. In addition, the TweetQA API will be written in Python using the Flask framework (which is outlined in more detail below).

7.1.1.1. *Tensorflow*

Tensorflow [1] is an end-to-end machine learning package that can be easily used as an imported python package. Tensorflow provides several services for model building, data validation, data extraction, model evaluation, etc. Tensorflow will be the primary package for the machine learning aspects of the system.

7.1.1.2. *Keras*

Keras [2] extends the functionality of Tensorflow by wrapping it in a high-level, easy-to-use API to provide machine learning services with a focus on Deep Learning.

7.1.1.3. *Pandas*

Pandas [3] is a Python library that allows for fast and efficient DataFrame object creation and manipulation. Pandas is used as a part of the Python Machine Learning stack for efficient manipulation of large data sets required for model training.

7.1.1.4. *Flask*

Flask [4] is a lightweight web application framework for Python. Flask will be used as the basis for designing the TweetQA API and will be used extensively in the controller and service layers.

7.1.1.5. *Connexion*

Connexion [12] is a framework that adds additional layers to the Flask framework to easily handle HTTP requests defined using OpenAPI. This package allows for easy documentation of the API.

7.1.1.6. *SQLAlchemy*

SQLAlchemy [5] is a Python SQL toolkit and Object Relational Mapper that will be used to create the model layer and to allow the service layer to communicate with a MySql Database running on the Google Cloud Platform. SQLAlchemy [6] integrates seamlessly with the Flask framework when building a full REST API.

7.1.2. Typescript

Typescript adds additional features to the JavaScript language, mainly typing. Typescript allows Javascript to behave more like an object-oriented language such as Java. Typescript is the language used as a part of the Angular framework, discussed in more detail below. Typescript is the second language used for the development of the system (along with Python) and is used exclusively for the development of the TweetQA Web Application.

7.1.2.1. *Angular*

Angular [7] is a robust component-based framework for building web applications built on Typescript. Angular provides libraries for several essential web-based features such as routing, form management, client-server communications, etc. The TweetQA Web Application is developed using the Angular framework.

7.1.3. Google Cloud Platform (GCP)

Google Cloud Platform is a cloud service provider that is used for all system cloud services. This includes the serving of the running web application and api, storage of all data, storage of completed machine learning models, continuous training of machine learning models via the machine learning pipeline, and automatic triggering of the machine learning pipeline.

7.1.3.1. *Cloud Storage*

Cloud Storage [9] is an object storage service provided by Google Cloud Platform. Completed machine learning models will be saved to a cloud storage bucket.

7.1.3.2. *App Engine*

App Engine [8] is a GCP service that allows for application deployment on a fully managed serverless platform. Both the TweetQA API and Web Application are deployed using App Engine.

7.1.3.3. *SQL*

SQL is a managed relational database service that is hosted with GCP.

7.1.3.4. *AI Platform*

AI Platform [10] provides cloud services for model training, evaluation, deployment, and predictions. AI Platform services are used as a part of the Machine Learning Pipeline.

7.1.4 Google CoLab

Google CoLab is an interactive notebook that is set up with the necessary python packages for machine learning. CoLab notebooks are useful for initial model development as they require little setup and allow for rapid early development.

7.1.5 Pycharm

Pycharm is an Integrated Development Environment for Python development. Pycharm is the recommended environment for all Python development within the project.

7.1.6 WebStorm

WebStorm is an Integrated Development Environment for web development. WebStorm is particularly well suited to Javascript and Typescript applications using frameworks such as Angular. WebStorm is the recommended environment for all Angular development for the project.

7.2. Coding Conventions

Having a coding convention allows for improved readability when reviewing code, so the development team has opted to adopt a coding convention for both the Typescript-based client-side and the Python-based server-side. The details for each coding convention are provided in **Table 7.1** below.

Table 7.1: Coding Convention Details

	Python	Typescript
class / type / enum / decorator / type parameters	UpperCamelCase	UpperCamelCase
interface	UpperCamelCase, should not be specifically marked; iie Model, not ModelInterface or IModel.	UpperCamelCase, should not be specifically marked; iie Model, not ModelInterface or IModel.
variable / parameter / function / method / property / module alias	lower_snake_case	lowerCamelCase
global constants	CONSTANT_CASE	CONSTANT_CASE
abbreviations	Capitalize all letters of the abbreviation, ie modelDTO, not modelDto	Treated like whole words, ie modelDto, not modelDTO
imports	Separate lines, at the top of the file, absolute imports are recommended, avoid wildcard imports	lowerCamelCase, at the top of the file
file names	lower_snake_case	lower_snake_case
indentation	4 spaces per indentation level	2 spaces per indentation level
continuation line	Align wrapped elements vertically or use a hanging indent. If using a hanging indent, no arguments should be on the first line	Align wrapped elements vertically or use a hanging indent. If using a hanging indent, no arguments should be on the first line
maximum line length	79 characters	80 characters
blank lines	Top level function and class definitions should be surrounded	A file should end with one blank line

	with two blank lines. Blank lines can be used within a function to separate logical sections.	
binary operator line break	Break before so that the operator is the leading character of the new line	Break before so that the operator is the leading character of the new line
link to full style guide	https://www.python.org/dev/peps/pep-0008/	https://google.github.io/styleguide/tsguide.html

7.3. Code Version Control

Code for the TweetQA system uses Github for version control. Repositories for each section of the codebase are grouped into an organization that can be found here, https://github.com/sweng480-team23. The development team checks out feature branches to work on changes to any of the repositories. All merges of those feature branches back to the master branch require an approved pull request by at least one other team member. System deployment will happen from the master branch.

7.4. Implementation Alternatives & Decision Rationale

7.4.1. Google Cloud Platform vs AWS

Google Cloud Platform and AWS were the leading cloud service providers discussed for the project. Ultimately, GCP was determined to be the best solution for the project due to its more extensive and polished AI services. In addition, the primary model used as a starting point for the System's own models was BERT, which was developed by Google. As such, it was determined that it would more easily integrate with GCP services.

7.4.2. A Flask/SQLAlchemy/Connexion Stack vs Django

It was determined early in the project that the API should be developed in Python to limit the language complexity of the project and to better integrate with the Machine Learning code base. There are two predominant frameworks for developing RESTful APIs in Python; Django and Flask [11]. The ultimate decision between the two came down to the fact that Flask has a lower learning curve and is a lighter-weight framework than Django. For a large enterprise scaled system, Django may be the better choice, but the TweetQA project did not require many of the bells and whistles that accompany Django. Lowering the development curve and maintaining an easier-to-follow codebase was more important for the context of this system. Flask leaves the implementation of some common functionality up to the developer, which is why SQLAlcheemy was included in the stack to handle data mapping between the API and the database. Object-relational mapping is one of many features that are included with Django, along with a significant amount that would not be used for this project.

7.4.3. Angular vs React

Angular and React are both well established web application frameworks with large, active communities. Either option would have been a good choice for development of our project. Ultimately, Angular was chosen because the team had slightly more past experiences using that framework compared to React. In addition, Angular is more structured and opinionated in how an application should be built, and the team preferred that approach to the larger degree of freedom that React allows. The team believes that greater standardization ultimately leads to more readable code and a simpler learning curve for new developers in a project.

7.5. Analysis of Key Algorithm

7.5.1. String Matching for FuzzyWuzzy

The use of FuzzyWuzzy has been discussed in section 5.2.2.3. The main challenge in utilizing the FuzzyWuzzy to find the answer span is dealing with all possible substrings which can be generated from a particular tweet. Referring to the pseudocode for the initial algorithm attached next to this paragraph, the initial algorithm designed using the Brute Force method will have two for loop. The first for loop will loop through the first index of the tweet till the last index of the tweet to extract a substring start on the for loop variable index(let it be i). The second for loop will loop through from the first loop variable index(i) to the remaining length of the tweet (length of the tweet – 1 – i), thus extracting the substring of variable length from 1 to the maximum length possible starting on index i.

```
Algorithm fuzzy_match_nosplit (tweet:str, answer:str)
        // Return the start and end position of the string span with the best scores compared to the
        // answer
        // input: a tweet in string form, an answer in string form
        // output: the start and end position (as the index of the tweet string)

        canidates = []
        n = len(tweet) - 1
        for i ← 0 to n do:
            for j ← i + 1 to (n- i) do:
                    //extract the substring and append to canidates list
                    canidates.append(tweet[i:j])

        // Find the best match combination
        best_matches = process.extractBests(answer, candidates, scorer=fuzz.token_sort_ratio,
        score_cutoff = 75)

        if best_matches != null:
            start_position = tweet.find(best_match)
            end_position = start_position + len(best_match)
            return start_position, end_position
        else:
            return -1, -1
```

The analysis of the algorithm above is as follows:

- For this algorithm the **input size is n**. Where n is the length of the tweet.

- There are two **basic operations**, the first one is the append operation to append the extracted substring into the list, and the second operation is to score the substring using FuzzyWuzzy and compared it with the best score available.

- There is **no worst-case or best case**. In the **base case** to complete the two for loop, for a string with length n, all the possible substrings that can be generated from it will be n*(n + 1)/2. Hence for a tweet with up to 280 characters, there will be up to 280*(280+1)/2 = 39340 substrings that need to be evaluated using the FuzzyWuzzy.

- Hence, the time efficiency will be $T(n) \in \Theta(n^2)$

- Since the basic operation will store each substring in the list, the space efficiency will also be $\in \Theta(n^2)$

The alternative to evaluating every possible substring is to take advantage that each tweet can be broken down into words, and the answer must consist of words. Hence, we can generate only all the possible combinations of words for the FuzzyWuzzy evaluation. Assuming each tweet contains an average of 50 words, the number of word combinations that need to be evaluated will be 50*(50+1)/2 = 1275, which is 1/30th compared to the previous evaluation using all the substrings. The use of words instead of characters combination is expected to reduce the time complexity and space complexity of the algorithm to 1/30th of the original. The pseudocode for the algorithm is shown after this paragraph.

```
Algorithm fuzzy_match(tweet:str, answer:str)
        // Return the start and end position of the string span with the best scores compared to the
        // answer
        // input: a tweet in string form, an answer in string form
        // output: the start and end position (as the index of the tweet string)

        canidates = []
        //use python str.split() function to split the tweet and answer
        tweet_split = tweet.split()
        answer_split = answer.split()
        n = len(answer_split)
        m = len(tweet_split)

        for i ← 0 to (m-n) do:
            for j ← i + 1 to (m-n-i) do:
                    //extract the substring and append to canidates list
                    canidates.append(tweet[i:j])


        // Find the best match combination
```

```
best_matches = process.extractBests(answer, candidates, scorer=fuzz.token_sort_ratio,
score_cutoff = 75)

if best_matches != null:
        start_position = tweet.find(best_match)
        end_position = start_position + len(best_match)
        return start_position, end_position
else:
        return -1, -1
```

The analysis of the algorithm above is as follows:

- For this algorithm the **input size is n**. Where n is now the length of the words of the tweet.

- There are two **basic operations**, the first one is the append operation to append the extracted substring into the list, and the second operation is to score the substring using FuzzyWuzzy and compared it with the best score available.

- In the **base case** to complete the two for loop, for a string with length n, all the possible substrings that can be generated from it will be n*(n + 1)/2. The **worst case** is when each character of the tweet is a word where the number of substrings that can be generated is still n*(n + 1)/2. The best case is when the entire tweet and answer consists of one word, where the algorithm will only run once.

- Hence, the time efficiency will be $T(n) \in \Theta(n^2)$

- Since the basic operation will store each substring in the list, the space efficiency will also be $\in \Theta(n^2)$

- Although the time efficiency and space efficiency remain in the same magnitude as the first algorithm designed using Brute-Force, the overall efficiency of the algorithm can still be improved by x^2, where x is the average number of characters of a word inside the tweet and answer.

8 System Testing

8.1. Test Automation Framework

8.1.1. Steps for Installing Test Frameworks

8.1.1.1. Angular (Frontend Web Application)

Testing an Angular application depends on the Jasmine test framework, along with the Karma test runner [20]. Both of these dependencies are included with the Angular CLI installation, the procedure of which is outlined in section 10.1.1.1, specifically in step 4.

This project also depends on a couple of third-party packages that make it easier to create mock objects for testing purposes. The following commands need to be run from the Angular project root to install these dependencies, which aren't included when the Angular CLI is installed:

```
npm i --save-dev cooky-cutter
npm i --save-dev @types/faker
```

8.1.1.2. Python / Flask (Backend Web API)

The official Flask documentation recommends using the popular Python testing framework called *pytest* [19]. Pytest typically needs to be installed using a package manager such as pip. Reference section 10.1.1.2 *TweetQA API Development Environment Setup*. The pytest framework will be installed automatically in this step as it is included in the requirements.txt file.

8.1.2. Steps for Running Test Cases

8.1.2.1. Angular (Frontend Web Application)

1. Running the test cases for the frontend is performed by running the following command from the root directory of the Angular project:

```
ng test
```

2. The tests are executed and the Angular CLI outputs the results of the testing upon completion. A sample of the test report output is provided in the console. Additionally, the user's browser opens a locally-served webpage with more details of the tests that were run and their failures (if any) in a more visually appealing format:

Figure 8.1: Chrome screenshot with test result details

3. The Karma test runner will continue until it is terminated manually. While the test runner is waiting between test executions, it will watch for filesystem changes in the project and rerun tests automatically when it sees a file change [20]. This allows the developer to quickly retest during the development and debug process.
4. When testing is complete, use Ctrl+C to end the test runner.

8.1.2.2. Python / Flask (Backend Web API)

1. Running the test cases for the API is performed by simply running the following command from the project root directory:

```
pytest
```

2. The tests are executed automatically and the pytest script outputs the results of the testing. A sample of the test report output is as follows:

```
===================== test session starts =====================
platform win32 -- Python 3.9.9, pytest-6.2.5, py-1.11.0, pluggy-
1.0.0
rootdir: C:\Users\lwest\Source\tweetqa-api
collected 1 item

tests\app_test.py .                                      [100%]

=====================  8 passed in 16.09s =====================
```

3. Optionally, the verbose flag can be supplied to the test script to provide more details in the report:

```
pytest -v
```

4. The verbose output sample is as follows:

```
==================== test session starts ====================
platform win32 -- Python 3.9.9, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 --
C:\Users\lwest\Documents\py-venv\tqa\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\lwest\Source\tweetqa-api
collected 1 item

tests/app_test.py::dataservice_creates_data PASSED                [13%]
tests/app_test.py::dataservice_reads_data PASSED                  [25%]
tests/app_test.py::dataservice_updates_data PASSED                [38%]
tests/app_test.py::qamodelcontroller_creates_qamodel PASSED       [50%]
tests/app_test.py::qamodelcontroller_reads_qamodel_by_id PASSED   [63%]
tests/app_test.py::qamodelcontroller_reads_latest_qamodel PASSED  [75%]
tests/app_test.py::qamodelcontroller_reads_latest_qamodels PASSED [88%]
tests/app_test.py::test_root PASSED                               [100%]


==================== 8 passed in 14.14s ====================
```

5. The pytest command must be rerun (repeat step 1 or step 3) each time the source code is updated and tests need to be run again.

8.2. Test Case Design

8.2.1. Test Suites

All test suites described can be referenced in **Appendix T** of this document.

TS-001: Tests related to the DataService object in the API responsible for creating, reading, and updating Data model entities in the database. Reference **Table 15.1** in **Appendix T.**

TS-002: Tests related to the QAModelController in the API responsible for handling HTTP requests with the client, converting DTOs to model instances, interacting with the QAModelService layer, and responding to the client. Reference **Table 15.2** in **Appendix T.**

TS-003: Tests related to the PredictionForm of the web application frontend responsible for accepting user input of an ML model choice, tweet text, question text, feedback on the prediction, and the alternate answer if the prediction was incorrect. Also responsible for displaying the prediction returned by the PredictionService layer and for handling the various states of the Prediction form. Reference **Table 15.3** in **Appendix T.**

TS-004: Tests related to the PredictionService layer of the web application frontend responsible for sending HTTP requests containing tweets, questions, feedback, and alternate answers to the API, and for handling responses from the API containing predictions on the data submitted. Reference

Table 15.4 in **Appendix T.**

TS-005: Tests related to the QAModelService object in the API responsible for creating and reading QAModel model entities in the database. Reference **Table 15.5** in **Appendix T.**

TS-006: Tests related to the PredictionService object in the API responsible for creating and updating Prediction model entities in the database. Reference **Table 15.6** in **Appendix T.**

TS-007: End-to-end system-level testing for user invitation, training deployment, and ML prediction functionality. Added in also security testing for accessing the system with/without invitation token, and accessing administrative features with/without login. Reference **Table 15.7** in **Appendix T.**

TS-008: Acceptance level testing that ensures quality delivery of the machine learning model, proper submission to the CodaLab competition, and proper accreditation to the TweetQA dataset founders. Reference **Table 15.8** in **Appendix T.**

8.2.2. Unit Test Cases

All unit test cases described can be referenced in **Appendix T** of this document.

TC-001: This unit test of the API verifies that the DataService object can accurately serialize the Data instance to the database and that it returns the object that was serialized. Reference **Table 15.9** in **Appendix T.**

TC-002: This unit test of the API verifies that the DataService object can retrieve a serialized Data object from the database with a specified id and that no property values are missed in translation. Reference **Table 15.10** in **Appendix T.**

TC-003: This unit test of the API verifies that the DataService object can update a serialized Data instance in the database and that all properties in the newly serialized version match those of the object that was passed in. Reference **Table 15.11** in **Appendix T.**

TC-008: This unit test of the web application frontend verifies that the prediction form's submit button is disabled in the initial state of the form. Reference **Table 15.16** in **Appendix T.**

TC-009: This unit test of the web application frontend verifies that the prediction form's submit button is enabled when the initial form is filled out and that the state transition to AwaitingPredictionRequestState has occurred. Reference **Table 15.17** in **Appendix T.**

TC-010: This unit test of the web application frontend verifies that after a prediction request is submitted, the prediction form is ready to collect feedback from the user and that the state transition to AwaitingIsCorrectState has occurred. Reference **Table 15.18** in **Appendix T.**

TC-011: This unit test of the web application frontend verifies that the prediction form is ready to resubmit after the user has entered their feedback for a correct prediction and that the state transition

to AwatingCorrectSubmissionState has occurred. Reference **Table 15.19** in **Appendix T.**

TC-012: This unit test of the web application frontend verifies that the prediction form is ready to accept an alternate answer after the user has entered their feedback for an incorrect prediction and that the state transition to AwatingAlternateAnswerState has occurred. Reference **Table 15.20** in **Appendix T.**

TC-013: This unit test of the web application frontend verifies that the prediction form is ready to resubmit after the user has entered their alternate answer for an incorrect prediction and that the state transition to AwatingIncorrectSubmissionState has occurred. Reference **Table 15.21** in **Appendix T.**

TC-014: This unit test of the web application frontend verifies that the prediction form state returns to InitialFormState after feedback has been submitted for the prediction. Reference **Table 15.22** in **Appendix T.**

TC-015: This unit test of the web application frontend verifies that the prediction service's create() method sends a POST request to the proper endpoint of the API and returns a response that matches the request. Reference **Table 15.23** in **Appendix T.**

TC-016: This unit test of the web application frontend verifies that the prediction service's read() method queries the proper endpoint of the API and returns a response matching the id of the query. Reference **Table 15.24** in **Appendix T.**

TC-017: This unit test of the web application frontend verifies that the prediction service's update() method sends a PUT request to the proper endpoint of the API and that it returns a response that matches the request. Reference **Table 15.25** in **Appendix T.**

TC-018: This unit test of the API verifies that the QAModelService object can accurately serialize the QAModel instance to the database and that it returns the object that was serialized. Reference **Table 15.26** in **Appendix T.**

TC-019: This unit test of the API verifies that the QAModelService object can retrieve a serialized QAModel object from the database with a specified id and that no property values are missed in translation. Reference **Table 15.27** in **Appendix T.**

TC-020: This unit test of the API verifies that the DataService object can retrieve all serialized Data objects from the database after a given date. Reference **Table 15.28** in **Appendix T.**

TC-021: This unit test of the API verifies that the DataService object can retrieve n number of serialized Data objects from the database. Reference **Table 15.29** in **Appendix T.**

TC-022: This unit test of the API verifies that the DataService object can generate a list of unique words with the frequency of all serialized Data objects from the database. Reference **Table 15.30** in **Appendix T.**

TC-023: This unit test of the API verifies that the QAModelService object can provide a list of all instances of QAModel that share the same model_type attribute. Reference **Table 15.31** in **Appendix T.**

TC-024: This unit test of the API verifies that the QAModelService object can provide a list of the latest instances of QAModel that have unique model_type attributes. Reference **Table 15.32** in **Appendix T.**

TC-025: This unit test of the API verifies that the QAModelService object can provide the latest QAModel instance of a given model type. Reference **Table 15.33** in **Appendix T.**

TC-026: This unit test of the API verifies that the PredictionService object can persist a given Prediction model in the database. Reference **Table 15.34** in **Appendix T.**

TC-027: This unit test of the API verifies that the PredictionService object can update a given Prediction model in the database. Reference **Table 15.35** in **Appendix T.**

TC-028: This unit test of the API verifies that the DataService object can return a random tweet when requested. Reference **Table 15.36** in **Appendix T**

TC-029: This unit test of the web application frontend verifies the prediction form displays a new random tweet after the user clicks the Get-Random Tweet button. Reference Table **15.37** in **Appendix T**

8.2.3. Integration Test Cases

All integration test cases described can be referenced in **Appendix T** of this document.

TC-004: This integration test of the API verifies that the POST endpoint of the QAModelController can accept a request to create a new QAModel object, that the response contains a matching QAModelResponse object to the one that was requested, and that the response code of the endpoint is 200 for success. Reference **Table 15.12** in **Appendix T.**

TC-005: This integration test of the API verifies that the GET endpoint of the QAModelController for reading a QAModel by id accepts an id for a known existing QAModel instance, that the response contains an object with valid values, that the id returned matches the one given, and that the response code of the endpoint is 200 for success. Reference **Table 15.13** in **Appendix T.**

TC-006: This integration test of the API verifies that the GET endpoint of the QAModelController for retrieving the most recently created QAModel, of a given model type, returns the object as requested and that the response code of the endpoint is 200 for success. Reference **Table 15.14** in **Appendix T.**

TC-007: This integration test of the API verifies that the GET endpoint of the QAModelController for retrieving the most recently created QAModel of each model type returns an array of objects as requested and that the response code of the endpoint is 200 for success. Reference **Table 15.15** in **Appendix T.**

8.2.4. System & Security Test Cases

TC-030: This system-level test verifies the ability of an admin to invite users to the system using a unique, system-generated URL. Access to the system without the URL should have limited functionality, such as being unable to receive a prediction or provide feedback. Reference **Table 15.38** in **Appendix T.**

TC-031: This system-level test verifies the ability of an admin to trigger the machine learning pipeline, which will take the provided hyperparameter inputs and train a new model. After training is complete the system will update. Reference **Table 15.39** in **Appendix T.**

TC-032: This system-level test verifies the end-to-end function of the main user interaction which is to provide a tweet question, submit that tweet-question pair to the ML model, receive the predicted answer, and provide feedback. All information should be persisted in the database. Reference **Table 15.40** in **Appendix T.**

TC-033: This security test verifies that only the administrator user who is logged in can access features available to the administrator, such as the page to send out invitation tokens to visitor emails, and the page to select the hyperparameters and request for new training of the machine learning model.

8.2.5. Acceptance Test Cases

TC-034: This acceptance test ensures the machine learning model delivered by the team is meeting the desired performance metrics for the BLUE, ROUGE, and METEOR scoring metrics. In addition, it also ensures proper submission of the model to the CodaLab competition. Reference **Table 15.41** in **Appendix T.**

TC-035: This acceptance test ensures the proper accreditation to the TweetQA dataset founders that was utilized for machine learning development. Reference **Table 15.42** in **Appendix T.**

8.3. Test Case Execution Report

8.3.1. Unit Test Report

All test unit test case execution reports described can be referenced in **Appendix TE** of this document.

TC-001 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object to prevent corruption of the production data with temporary test data. Instances of the DataService class and Data class are created. The Data object is initialized with mock data and then passed into the create_data() method of the DataService object. Finally, the test asserts that the mock database contains the new record, that all fields in the record exactly match that of the Data object passed in, that the returned object isn't a null value, and that the returned object exactly matches the Data object passed in. Reference **Table 16.1** in **Appendix TE.**

TC-002 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. Instances of the DataService class and Data class are created. The Data object is initialized with mock data and then saved to the database. The read_data_by_id() method is called on the DataService object, passing in the id of the mock Data object just created. The test asserts that the returned object is not a null value and that it exactly matches the mock data passed into the database (except for the id property, which is automatically generated by the database when the record is created). Reference **Table 16.2** in **Appendix TE.**

TC-003 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. Instances of the DataService class and Data class are created. The Data object is initialized with mock data and then saved to the database. The Data object is then mutated by changing its property values, and then the object is passed to the update_data() method of the DataService object. The database is queried to retrieve the record created and updated in this test. The test asserts that the returned object isn't a null value and that all properties on the returned object match the Data object that was changed. Reference **Table 16.3** in **Appendix TE.**

TC-008 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A reference to the submit button DOM element is found. This reference allows the test to assert that the submit button is disabled and that the form state is InitialFormState. Reference **Table 16.8** in **Appendix TE.**

TC-009 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu and mock tweet text and question text are entered into their corresponding text fields of the form. A reference to the submit button DOM element is found and the test asserts that the button is enabled and that the form state is AwaitingPredictionRequestState. Reference **Table 16.9** in **Appendix TE.**

TC-010 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu and mock tweet text and question text are entered into their corresponding text fields of the form. A reference to the submit button DOM element is found and a click event on the button is simulated. The test asserts that the onSubmit() function was called, that the button is disabled after the click event, and that the current state is AwaitingIsCorrectState. Reference **Table 16.10** in **Appendix TE.**

TC-011 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu and mock tweet text and question text are entered into their corresponding text fields of the form. A reference to the submit button DOM element is found and a click event on the button is simulated. A reference to the feedback radio button DOM element is found and the "Yes" option is selected. The test asserts that the submit button is enabled and that the current state is AwaitingCorrectSubmissionState. Reference **Table 16.11** in **Appendix TE.**

TC-012 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu and mock tweet text and question text are entered into their corresponding text fields of the form. A reference to the submit button DOM element is found and a click event on the button is simulated. A reference to the feedback radio button DOM element is found and the "No" option is selected. The test asserts that the submit button is disabled and that the current state is AwaitingAlternateAnswerState. Reference **Table 16.12** in **Appendix TE.**

TC-013 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu and mock tweet text and question text are entered into their corresponding text fields of the form. A reference to the submit button DOM element is found and a click event on the button is simulated. A reference to the feedback radio button DOM element is found and the "No" option is selected. A mock alternate answer is provided in the text field. The test asserts that the current state is AwaitingIncorrectSubmissionState. Reference **Table 16.13** in **Appendix TE.**

TC-014 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu and mock tweet text and question text are entered into their corresponding text fields of the form. A reference to the submit button DOM element is found and a click event on the button is simulated. A reference to the feedback radio button DOM element is found and the "Yes" option is selected. Another click event on the submit button is simulated. The test asserts that the submit button is now disabled, that the form state is now InitialFormState, and that the tweet input has a null value. Reference **Table 16.14** in **Appendix TE.**

TC-015 Execution: This unit test begins by configuring the TestBed testing module with all required components. The PredictionService, HttpClient, and HttpTestingController dependency objects are all injected into the TestBed. Mock PredictionResponseV1 and PredictionCreateRequestV1 objects are instantiated. The PredictionCreateRequestV1 mock object

is sent in the body of a POST request to the v1/predictions endpoint using the PredictionService. The test asserts that there is a matching POST HTTP request emitted and that the response matches the mock PredictionResponseV1 object. Reference **Table 16.15** in **Appendix TE.**

TC-016 Execution: This unit test begins by configuring the TestBed testing module with all required components. The PredictionService, HttpClient, and HttpTestingController dependency objects are all injected into the TestBed. A mock PredictionResponseV1 object is instantiated and the PredictionService is used to send a GET request to the v1/predictions/{id} endpoint. The test asserts that there is a matching GET HTTP request emitted and that the response matches the mock PredictionResponseV1 object. Reference **Table 16.16** in **Appendix TE.**

TC-017 Execution: This unit test begins by configuring the TestBed testing module with all required components. The PredictionService, HttpClient, and HttpTestingController dependency objects are all injected into the TestBed. Mock PredictionResponseV1 and PredictionUpdateRequestV1 objects are instantiated. The PredictionUpdateRequestV1 mock object is sent in the body of a PUT request to the v1/predictions/{id} endpoint using the PredictionService. The test asserts that there is a matching PUT HTTP request emitted and that the response matches the mock PredictionResponseV1 object. Reference **Table 16.17** in **Appendix TE.**

TC-018 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object to prevent corruption of the production data with temporary test data. Instances of the QAModelService class and QAModel class are created. The QAModel object is initialized with mock data and then passed into the create() method of the QAModelService object. Finally, the test asserts that the mock database contains the new record, that all fields in the record exactly match that of the QAModel object passed in, that the returned object isn't a null value, and that the returned object exactly matches the QAModel object passed in. Reference **Table 16.18** in **Appendix TE.**

TC-019 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. Instances of the QAModelService class and QAModel class are created. The QAModel object is initialized with mock data and then saved to the database. The read() method is called on the QAModelService object, passing in the id of the mock QAModel object just created. The test asserts that the returned object is not a null value and that it exactly matches the mock data passed into the database (except for the id property, which is automatically generated by the database when the record is created). Reference **Table 16.19** in **Appendix TE.**

TC-020 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A list of sequential DateTime objects will be created of length 100, and then 100 data instances will be pulled down from the dataset and instantiated as Data objects. The DateTime list produced will be assigned to the data instances and then persisted in the mock database. The DataService object will be instantiated and a random DateTime value will be selected from the previously mentioned DateTime list. The DataSerivices' read_all_data_since

method will be called providing the randomly selected DateTime object. The returned list of data instances will be ensured to have a known length based on the selected DateTime object. The Reference **Table 16.20** in **Appendix TE.**

TC-021 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A 100 data instances will be pulled down from the dataset and instantiated as Data objects. The DataService object will be instantiated and a random integer value will be selected in the range of 1-100. The DataServices' read_x_data method will be called providing the randomly selected integer. The returned list of data instances will be ensured to have a length of n. The Reference **Table 16.21** in **Appendix TE.**

TC-022 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A 100 data instances will be pulled down from the dataset and instantiated as Data objects. The DataService object will be instantiated and the generate_word_cloud method will be called. A list of word and frequency tuples will be ensured to be returned. The Reference **Table 16.22** in **Appendix TE.**

TC-023 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A number of QAModel instances are committed to the mock database with different model_type attributes. The QAModelService object will be instantiated and the read_all_qa_model_type method will be called with a given model_type string that matches a known number of instances added to the mock database. A list of all QAModel instances is returned; it will be ensured that the length of the list matches what was committed and that all instances share the same model_type attribute. The Reference **Table 16.23** in **Appendix TE.**

TC-024 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A number of QAModel instances are committed to the mock database with different model_type attributes. The QAModelService object will be instantiated and the read_latest_models method will be called. A list of all QAModel instances is returned; it will be ensured that the length of the list matches the number of unique model types persisted in the mock database and that they all have unique model_type attributes. The Reference **Table 16.24** in **Appendix TE.**

TC-025 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A number of QAModel instances are committed to the mock database with different model_type attributes. The QAModelService object will be instantiated and the read_latest_model_by_type will be called passing in a known model_type string. An instance of QAModel is returned and will be ensured it will have a matching model_type attribute as the one requested. In addition, it should be ensured that the creation date is the latest of the persisted QAModel instances of the requested model_type. The Reference **Table 16.25** in **Appendix TE.**

TC-026 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object to prevent corruption of the production data with temporary test data. Instances of the PredicitonService class and Prediction class are created. The Prediction object is initialized with mock data and then passed into the create() method of the Prediction object. Finally, the test asserts that the mock database contains the new record, that all fields in the record exactly match that of the Prediction object passed in, that the returned object isn't a null value, and that the returned object exactly matches the Prediction object passed in. Reference **Table 16.26** in **Appendix TE.**

TC-027 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object to prevent corruption of the production data with temporary test data. Instances of the PredicitonService class and Prediction class are created. The Prediction object is initialized with mock data and then passed into the create() method of the Prediction object. The Prediction object then makes changes to the mock data and attempts to update the record using the update() method of the Prediction object. Finally, the test asserts that the mock database contains the updated record, that all fields in the record exactly match that of the Prediction object passed in, that the returned object isn't a null value, and that the returned object exactly matches the Prediction object passed in. Reference **Table 16.27** in **Appendix TE.**

TC-028 Execution: This unit test begins by importing the application and database object from the controllers' module. An SQLite mock database object is instantiated and overwrites the reference of the existing application database object. A 100 data instances will be pulled down from the dataset and instantiated as Data objects. The DataService object will be instantiated and called the read_random method to return random data. Reference **Table 16.28** in **Appendix TE.**

TC-029 Execution: This unit test begins by configuring the TestBed testing module with all required components and instantiating a mock PredictionResponseV1 object. A model is selected from the dropdown menu. A click event on the Get Random Tweet button is simulated. The test asserts that the tweet text field is populated with random tweet feedback from the API. Reference **Table 16.29** in **Appendix TE.**

8.3.2. Integration Testing Report

All test integration test case execution reports described can be referenced in **Appendix TE** of this document.

TC-004 Execution: This integration test begins by importing the application and instantiating the test_client object from the application object. A dictionary object containing mock data is created to represent a POST body from a client. The address for the controller endpoint that handles QAModel creates requests and the dictionary object is passed into the post function of the test_client object. The test asserts that the response DTO object matches the dictionary object of the request and that the response status code is 200. Reference **Table 16.4** in **Appendix TE.**

TC-005 Execution: This integration test begins by importing the application and instantiating the test_client object from the application object. The address for the controller endpoint that handles QAModel-by-id read requests and the id for a known existing QAModel record is passed to the get function of the test_client object. The test asserts that the response contains a complete QAModelResponse DTO, that the id property matches the id requested, and that the response status code is 200. Reference **Table 16.5** in **Appendix TE.**

TC-006 Execution: This integration test begins by importing the application and instantiating the test_client object from the application object. The address for the controller endpoint that handles the latest-QAModel-by-type read requests and a sample model_type are passed to the get function of the test_client object. The test asserts that the response contains a complete QAModelResponse DTO, that the model_type property of the response matches the model_type requested, and that the response status code is 200. Reference **Table 16.6** in **Appendix TE.**

TC-007 Execution: This integration test begins by importing the application and instantiating the test_client object from the application object. The address for the controller endpoint that handles the latest-QAModels is passed into the get function of the test_client object without any other parameters. The test asserts that the response contains an array of complete QAModelResponse DTOs, that the model_type property of the array elements in the response is all unique, and that the response status code is 200. Reference **Table 16.7** in **Appendix TE.**

8.3.3. System & Security Testing Report

TC-030 Execution: This system test begins by opening the TweetQA web application and logging into the system as an admin user. The admin user will then access the invite visitor page and submit the prepared email. It should be ensured that the system emails the provided email address, the provided URL works appropriately, and removal of the token from the URL results in the inability to submit data to the system. Reference **Table 16.30** in **Appendix TE.**

TC-031 Execution: This system test begins by opening the TweetQA web application and logging into the system as an admin user. The admin user will then access the deploy training page and submit the prepared hyperparameters. It should be ensured that the machine learning pipeline is triggered (can be viewed from the google cloud console), the training information is persisted in the database, and after completion that the weights files are saved to cloud storage. Reference **Table 16.31** in **Appendix TE.**

TC-032 Execution: This system test begins by opening the TweetQA web application and providing the prepared tweet-question pair. After submitting the pair the system should return a machine learning predicted response. Provide feedback on the validity of the answer and then submit the form again. Ensure that all data is recorded in the database. Reference **Table 16.32** in **Appendix TE.**

TC-033 Execution: This security test begins by trying to access the page accessible only to the administrator without login as an administrator. After copying and pasting the URL for the specific page to the web browser, the browser should automatically redirect the user to the home page if they are not logged in as the administrator. Reference **Table 16.33** in **Appendix TE.**

8.3.4. Acceptance Testing Report

TC-033 Execution: This acceptance test begins by importing the score_model method from the tqa-train-lib/model_scoring_lib library, and then calling that method pointing to the locally saved model and ensuring the save local boolean value is true. The output file is then submitted to the CodaLab competition. Reference **Table 16.34** in **Appendix TE.**

TC-034 Execution: This acceptance test begins by accessing the deployed web application, navigating to the appropriate hyperlinks, and clicking them. The TweetQA Dataset link should redirect the user to the TweetQA website and the privacy statement hyperlink should redirect the user to the privacy statement. Reference **Table 16.35** in **Appendix TE.**

9 Challenges & Open Issues

9.1 Challenges Faced in Requirements Engineering

The challenges faced in requirements engineering are listed as follows:

- The first challenge is deriving a clear and concise project scope that not only satisfies our client needs, but also the needs of the SWENG 480 project requirements. At the first meeting, our client states his need is primarily about an improved machine learning model that can achieve a high score (more than 70%) on the three performance metrics. However, this need alone is insufficient for SWENG 480 project requirements. Hence in subsequent requirements solicitation meetings, the team managed to agree with the client and also the faculty advisor on a set of requirements relevant to the client's needs while fulfilling SWENG 480 project requirements.

- The second challenge is related to conducting initial research of the problem domain, specifically on Natural Language Processing (NLP). The entire development team had various levels of machine learning experience ranging from none at all to introductory understanding. The team had to read through quite a number of research papers and articles to grasp the key fundamental concepts for NLP and its model.

- A follow-up challenge from the second challenge is understanding what existing ML models are available and how they differ from each other. This is required to form a shortlist of options to evaluate for the question-answering task.

- The third challenge is Identifying intended user groups and user scenarios for the desired ML model deliverable. The team was initially confused about who should be the primary users for the system developed by this project. Further clarification with the sponsor helped resolve the confusion.

- Additional challenges were faced by the team when trying to meet as a group for client, faculty, and sprint meetings. The team members have varying personnel commitments which block out numerous time slots and one of the team members is living in a different time zone. The team overcomes these challenges by using Zenhub to distribute and manage sprint tasks. Through continued communication, the team has also found several meeting times that work for each meeting stakeholder. To enable continuous communication across the team despite the varying time restrictions, Discord is used for asynchronous conversations to help each other keep track of the current progress of the project.

9.2 Challenges Faced in System Development

The challenges faced in system development are listed as follows:

- The first challenge is overcoming the learning curve required to build the machine learning model and the prototype system. The lists of tools chosen for the system include PyTorch, Flask, Flask-SQLAlchemy, SQLAlchemy, Connexion, and Angular. These tools are fairly new to some of the team members who just started to program using Python and its library of tools. Hence half of the development time was spent on setting up the environment, learning the new tools, troubleshooting bugs, and understanding the behavior of the tools and how to integrate them before proceeding to implement the desired system features. The team also starts with the Google Cloud Platform (GCP) for hosting the SQL instance and machine learning pipelines. It takes time to learn to set up the hosting and connection properly as well.

- The second challenge is controlling the development cost of the project while ensuring that common resources are available to all the team members. Initially, the team hosted the machine learning model with the GCP Compute Engine and Kubernetes Engine and hosted the SQL instance using Cloud SQL. However, both services cost about $9 per day, and in about a month the team finished spending the $300 free credit given to new GCP users. As the cost of running the GCP services is not sustainable for now, the team had to shut down the Compute Engine and Kubernetes Engine and moved the SQL instance to one of the member's private servers. Moving forward, the team will have to pool all the available grants together and budget the usage of GCP services.

- The third challenge is the process of implementing the system design while simultaneously revising the system design based on issues/ideas encountered during the development process. Following agile practice, the team started with an initial design of the overall system and proceeded to implement parts of the system during each sprint. However, during the implementation, some of the functions and parameters initially designed were found to be less useful. The team had discussions and decided to amend some of the functions and parameters devised during the initial design. Some class/function/variable names were standardized with naming conventions as well. These changes required the team to revisit the part of the system that had been implemented and tested and required refactoring and retesting.

- The fourth challenge is trying to divide the coding tasks into blocks or sections that could be finished by the individuals while synchronizing the development process for the integration units. As shown in the system structural design, the majority of the classes have associations or dependencies among each other. To distribute the loads evenly the team members each focus on different classes. However, some of the members will have to wait for others to finish a particular class, before they can import and use it in the class they are currently developing. This resulted in some inefficiency in developing and testing the code.

- The fifth challenge faced by the development team is trying to iteratively train new models to find the best set of hyperparameters, working around the hardware limitations. Initially, the development team utilized Google Colab to try and train new models, but the Google Colab was easily prone to connection timeout due to the lengthy duration of the training process.

  The team then proceeded to set up the Google Cloud Platform (GCP) machine learning pipeline instance. Training models using the GCP machine pipeline instance have two main limitations. First, due to the lack of GPU resources available, it took a long time to train the models. Second, running the machine learning pipeline consumes valuable GCP credits very fast, where a $300 GCP credit can be spent within a few weeks of training.

  Finally, it was decided to train the models using local hardware, which allowed the development team to utilize some of their NVIDIA CUDA GPUs hardware to quickly train the models and iteratively develop the machine learning pipeline. However, the major drawback of using consumer-grade GPUs is that they have limited graphics memory capacity for the machine learning application. This will limit the maximum padding size and batch size (two key hyperparameters discussed in section 5.2.3) that can be trained, potentially missing out on the best combinations of hyperparameters. The alternative solution to explore a larger maximum padding size and batch size is to train the model using the local CPU hardware. The drawback of using a CPU is up to 10 times longer training duration is needed compared to GPU.

  Moving forward the development team will continue to make use of the GPU to fast train the model, and conduct supplement training for hyperparameters that are memory hungry using the CPU and the GCP production environment.

- The sixth challenge is dealing with the TweetQA dataset and preprocessing it for input to the model fine-tuning training process. As explained in section 5.2, the use of tweet language will make it more difficult to identify the starting and ending position of the answer span in the tweet, and potentially causes the model to misread the tweet and answer if they are tokenized differently due to the uses of acronyms or skip of whitespace. Efforts are needed to ‘normalize’ the data to obtain better training results.

- A significant challenge up to the time of writing report Version 3.0 was getting any of the BLEU, METEOR, or ROUGE scores to an acceptable level using PyTorch. The team was unsuccessful in breaching 30% on any of these, clearly far below the stated goal in the requirements of 60-70%. An alternative training route was developed that uses the Tensorflow library from Google. After fixing a few bugs, it immediately returned promising results in the 48-57% range in all three metrics. With some further fine-tuning, the scores were increased to the 66-72% range. It appears that all of the work surrounding the PyTorch path such as training data normalization, in pursuit of generating an acceptable model, led to a very viable training pipeline once PyTorch was swapped for TensorFlow.

- Some challenges were met with the deployment of the full system to Google Cloud. Most team members have little or no experience with this cloud provider prior to the start of the project. The team has to learn how to deploy the full system to Google Cloud, including how to apply containerization to the applications which are complex in nature. The team also went through a lot of trial and error and experienced numerous pain points. Some pain points include the passing of environment variables, request routing between subsystems, request timeouts, and unhandled exceptions. With persistence and effort in continuing learning, the team manages to deploy all parts of the system.

9.3 Open Issues & Ideas for Solutions

The list of open issues and ideas for solutions are briefly discussed below:

- The first open issue is that currently only a single ML model can be loaded at a time by providing the weights file in the model runner "model" directory. Therefore, all prediction requests are routed to the single runner container, regardless of which model is selected on the front end when the user is submitting the prediction request. For the potential solution, support for multiple simultaneous runners has been conceptualized by running more instances of the runner container at different addresses, which the API would track and delegate prediction requests to the corresponding models based on the model selected by the user.

- The second open issue is that currently only 1 type of machine learning model based on the BERT architecture was trained. Hence it is unknown if the trained model is the best that can be obtained. It is suggested to train additional models based on a different pre-trained model such as XLNet. Even if the BERT-based model remained superior, having models from different pre-trained models would provide more interesting comparison data.

- The third open issue is that if more than one training is attempted at the same time, the Kubernetes Cluster (which processes the training) will run out of memory and kill one of the running training processes to restore memory. To resolve this, a training queue can be implemented. If a training job is currently running in the Cluster, then any subsequent training requests should be queued and then run in the order in which they were requested.

- The fourth open issue is that upon submitting a new training request through the UI, the user is alerted that training has successfully started, but does not get any type of alert upon the failure of the training during the training process or the completion of the training. For the solution, an email notification system will be useful to notify the admin who initiated the training upon successful completion of a new training job or after any error that occurred during training that would prevent completion.

- The fifth open issue is that the training loss from a new training job is not displayed in any way outside of the logs in the Kubernetes Cluster. As the training loss may provide insights into what went wrong during the training process, for the solution the use of a library such as Tensorboard is suggested to capture the staged output of training to display the accuracy, precision, and loss curves throughout the training.

- The sixth open issue is that the training currently happens on a Kubernetes Cluster built on nodes running CPUs. This results in lengthy training times of up to 24 hours. If funding permitted, the cluster can be reconfigured to run on GPUs or even TPU, which would decrease the training time to as little as 2 or 3 hours.

- The last open issue is that the user is currently comparing different models by toggling between the available models on the home page of the UI. This comparison method is not straightforward and could be troublesome if the number of models grows large. The potential solution will be to have additional graphical components added to the home page UI to allow a user to directly compare two or more different models at the same time.

10 System Manuals

10.1. Instructions for System Development

10.1.1. How to set up the development environment

10.1.1.1. *TweetQA Web Application Development Environment Setup*

1. Run the following from the command line to clone the appropriate repo to your local file system:

```
git clone git@github.com:sweng480-team23/tweetqa-web.git
```

2. Download Node from https://nodejs.org/en/
3. Check that node is installed and that an npm package manager is available by running the command:

```
npm -v
```

4. To install the angular command line tools, run the following command:

```
npm install -g @angular/cli
```

5. To install all required node packages for the project, from your command line, navigate to the root of the tqa-web-app package within the recently cloned tweet-qa package and run the following:

```
npm install
```

6. To verify that the setup is correct, from the root of the tqa-web-app package, user your command line to run the following:

```
ng serve --open
```

10.1.1.2. *TweetQA API Development Environment Setup*

1. Run the following from the command line to clone the appropriate repo to your local file system:

```
git clone git@github.com:sweng480-team23/tweetqa-api.git
```

2. If not using Pycharm, you will need to create a python virtual environment. If using Pycharm, a virtual environment for your new project will be set up when you create a new python project in the Pycharm IDE.

3. To install all requirements for the package, use your command line to navigate to the root of tweetqa-api and run the following:

```
pip install -r requirements.txt
```

4. To set up the connection with the SQL instance, you will need to create the .env file in the root directory and add a SECRET_KEY value pair into it. Contact any of the team members for the SECRET_KEY.
5. To set up the model, you will need to download the weights file and save them into the directory under utils/model/bert/first for the current version. Contact any of the team members for the weights file.
6. To confirm the setup is correct, run the main.py file in python and navigate to 0.0.0.0:8080/ localhost:8080 in any web browser.

10.1.1.3. *ML Pipeline Development Environment Setup Steps*

1. Run the following from the command line to clone the appropriate repo to your local file system:

```
git clone git@github.com:sweng480-team23/tweetqa-pipeline.git
```

2. If not using Pycharm, you will need to create a python virtual environment. If using Pycharm, a virtual environment for your new project will be set up when you create a new python project in the Pycharm IDE.
3. A number of the package requirements require the most up-to-date versions from the package manager, which means you must update your package manager before installing the requirements. To update your package manager, pip, from the command line run the following:

```
pip install --update pip
```

4. To install all necessary requirements, from the command line navigate to the root of the tweetqa-pipeline package and run the following:

```
pip install -r requirements.txt
```

10.1.1.4. *GCP Development Environment Setup*

1. Request access from a system Administrator by emailing edl5040@psu.edu and providing your google email address.
2. After permissions have been granted by the System Administrator, log in to your Google account at https://console.cloud.google.com.
3. Select the TweetQA project from your cloud console projects dropdown menu.

10.2. Instructions for System Deployment

10.2.1. Platform Requirements

The only requirement for the user is to have a modern web browser such as Chrome, Edge, Safari, Firefox, etc. As most of the computation is done server-side (resources managed by predefined cloud configurations), the client system can have modest resource limits. For example, a client system with the following specifications should have no problem interacting with the cloud system:

- 1 GB RAM
- Dual-core 2 GHz CPU
- Windows, macOS, Linux, or any smartphone/tablet operating system

10.2.2. System Installation

10.2.1.1 *GCP Cloud Build / Cloud Run Deployment*

Prerequisites (these only need to be performed once):

1. Install Google Cloud CLI: https://cloud.google.com/sdk/docs/install
2. Authorize your Google Cloud CLI: https://cloud.google.com/sdk/docs/authorizing
3. Create a new Google Cloud project: https://cloud.google.com/resource-manager/docs/creating-managing-projects
4. In the cloudbuild.yaml file in the root of whichever component you wish to deploy, replace all instances of "tweetqa-338418" with your specific project ID from step 3.

Note: The following instructions apply to any of the containers that are deployed to Google Cloud Run, which includes the web API, the training pipeline, and the model runner. The procedure is left generic enough to apply to all three of these components of the system as they are virtually identical.

Deployment procedure:

1. In your terminal, navigate to the root directory of the component you wish to deploy (this directory should contain a cloudbuild.yaml file).
2. Run the following command:

```
gcloud builds submit
```

3. Wait for the command to complete. This can take several minutes. Once complete, the service should be running in Google Cloud.

10.3. Instructions for System End Users

10.3.1 Instruction for the Machine Learning Student

In order to use the TweetQA web application as a student user, you must first be invited to the system, simply reach out to one of the system admins who will then provide you with a unique URL to access the system via a system-generated email. Failure to do so will result in limited functionality from the web application because unauthorized users are not permitted to make any requests that require persisting data to the database.

After accessing the web application you will see a form to the right that allows you to select a given model and submit a tweet-question pair shown below in **Figure 10.1**. After selecting an available model you have the option to either get an existing tweet randomly from the database or copy/paste your own. Upon doing so simply ask a question with regards to the context of the tweet, and select submit. After submitting the web application will then route your tweet-question pair to the machine learning model and return a predicted answer.

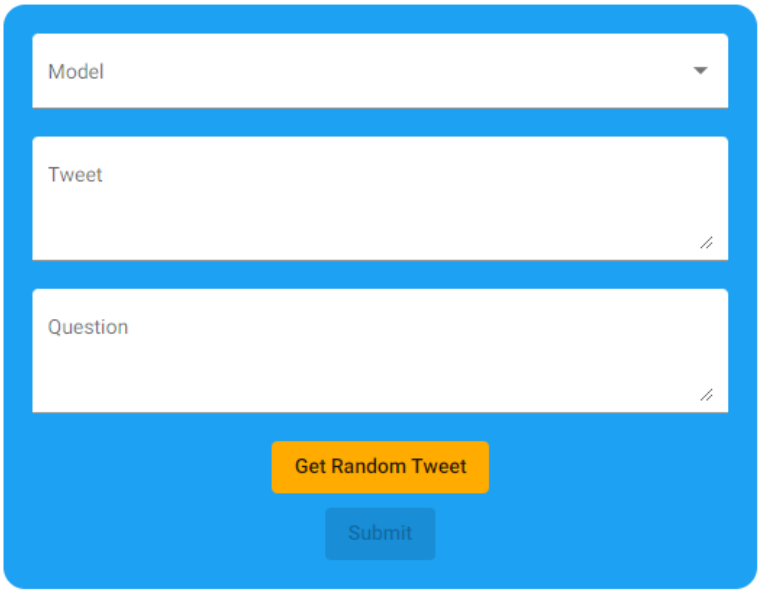


Figure 10.1: TweetQA Form

An example, tweet-question-answer set is shown below in **Figure 10.2**. It should be noted that the system utilizes an extractive NLP model for question answering tasks. You should try to ask questions that can be answered explicitly from the context of the tweet for the best results.

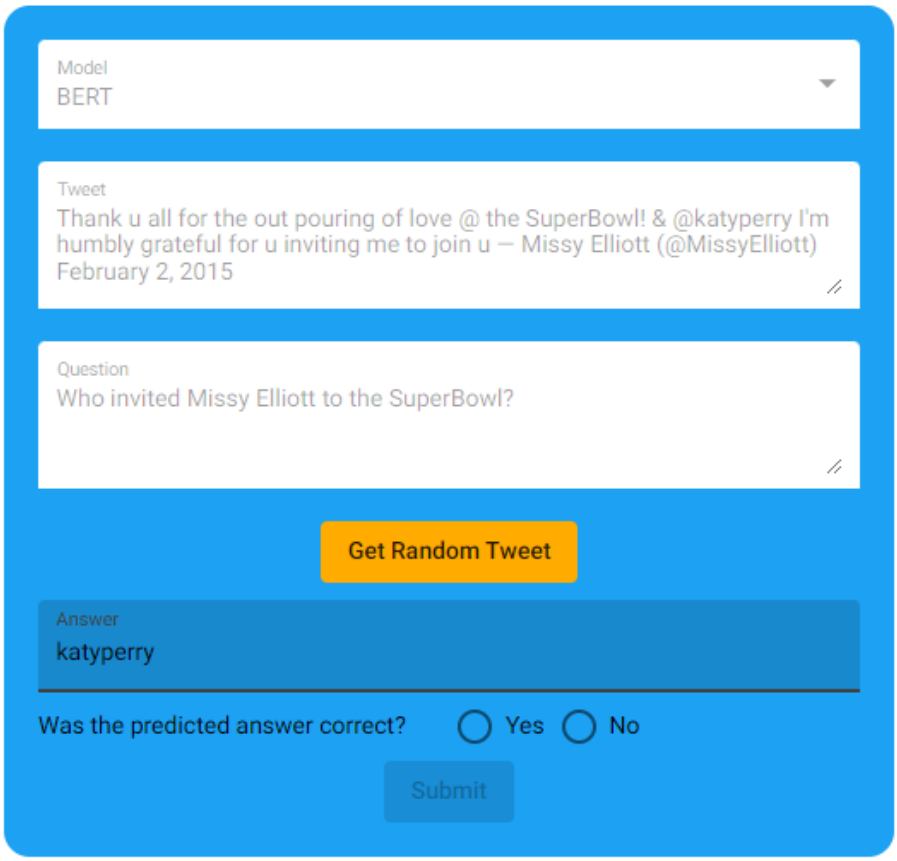


Figure 10.2: Tweet-Question-Answer Set Example

After submitting the tweet-question pair and receiving a predicted response the user will then be asked for feedback on the validity of the answer. Please make note that these forms should be completed with accuracy because the collected data is persisted to train future models, in order to produce a more accurate model the system needs to collect quality data. For the example above, the user would select yes and then submit. If the model provides an incorrect prediction, the user will select no and then be prompted for the correct answer.

As a user, you will also notice that upon selection of a model some graphics appear on the left side of the web application. The first is a word cloud that can be used to quickly observe the frequency of different words that describe the dataset. The second is a summary of the training metrics of the selected model over time. This allows the user to quickly see the data the model is trained on and also a metric of its performance.

10.3.2 Instruction for the System Admins

As a system admin, there are two main functionalities that can be accessed using the web application, which includes inviting more student users and triggering the training pipeline with a new set of hyperparameters. In order to access these, you must first log in to the application with your admin credentials. To do so load the web page and select the drop-down in the top right and select the "Admin Login" option. You will then be routed to the admin login page shown below in **Figure 10.3**.

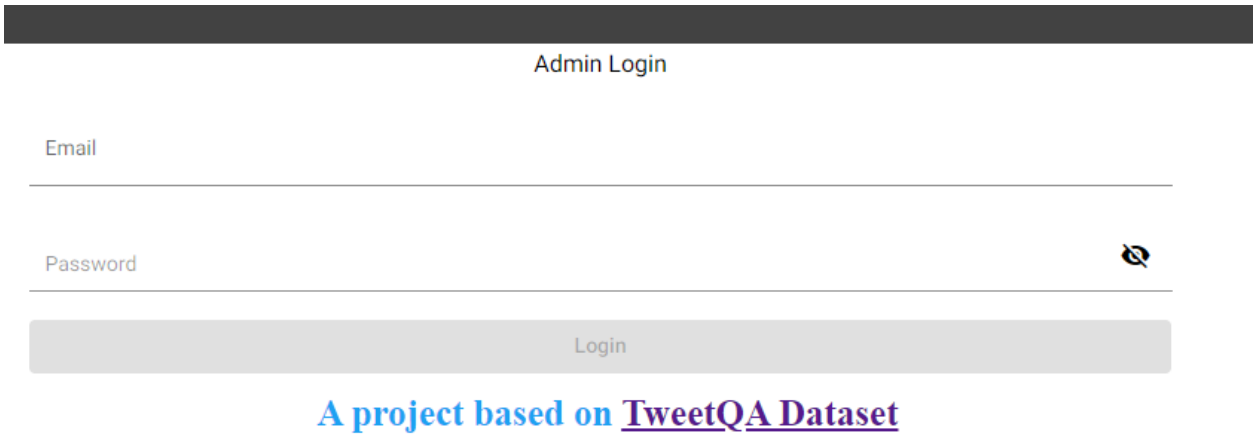


Figure 10.3: Admin Login Form

Upon successful login, you will be routed back to the main page, although now when you click the drop-down in the top right you will see the additional "Invite Visitors" and "New Training" options. Select the invite visitors selection and you will see the form shown in **Figure 10.4** which allows you to provide a single or comma-delimited list of emails. Upon submission, the system will send out invites with unique URLs to all the listed emails granting access to the individuals.

Figure 10.4: Visitor invite form

Now navigate to the "New Training" option from the drop-down in the top right of the page. You will be prompted with a form shown below in **Figure 10.5**. The form will be populated with default values, but you can select a set of hyperparameter values to utilize for the new training. Upon submission, the system will trigger the training pipeline and deploy a new model.

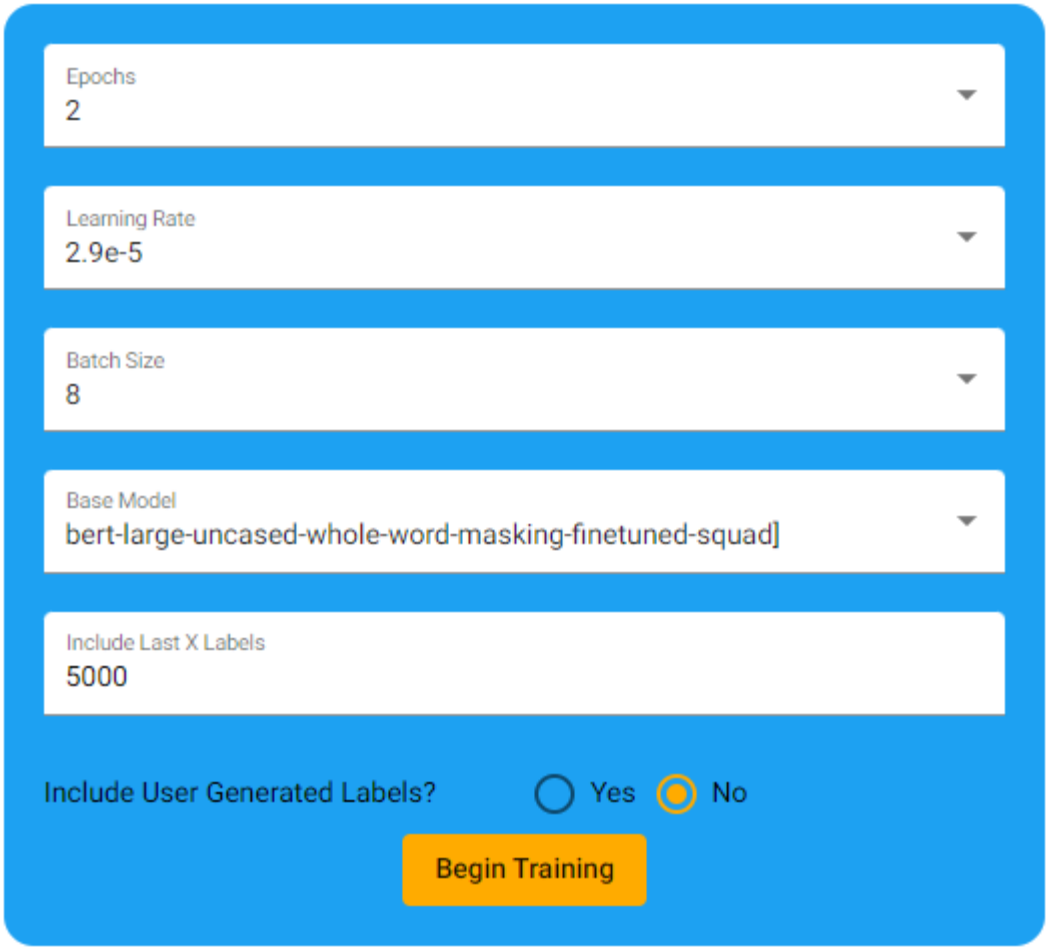


Figure 10.5: New Training Form

11 Conclusion

11.1 Achievement

The team managed to train a machine learning model for the TweetQA task with average scores of 71% for the BLEU-1, METEOR, and ROUGE-L metrics, which places the team in fifth place in the TweetQA Competition.

The team also managed to deploy a web application front end and the corresponding web API to service it onto the Google Cloud Platform with the following key implemented features:

- The web application is accessible to machine learning students with an invitation token.
- The students can provide a tweet-question pair and get an answer from the machine learning model.
- The students can also select a random tweet, ask a question about it and get an answer from the selected machine learning model
- The students can provide feedback on whether the answer is correct and provide an alternative answer if the answer is incorrect.
- Responses from the students will be recorded in the database.
- The system will generate a word cloud to display the characteristics of the dataset used and a scoring graph to showcase the performance of the baseline model.
- The web application also allows the administrator user to perform several functions. These include training a new machine learning model using hyperparameters chosen, or sending a new invitation email to other students.

11.2 Lessons Learned

Most of the lessons learned during the development of this project are related to how the team managed to overcome the challenges listed in section 9. A summary of the key lessons learned is listed below:

- Fine-tuning the base model on the TweetQA data set using the TensorFlow library yields better performance compared to using the PyTorch library.
- The team needs to be cost-conscious when developing a similar project in the future when deploying using the Google Cloud Platform. With a better understanding of which cloud components have a higher unit cost, the team will need to keep the deployment to the

components in the end.

- One lesson learned during the agile practice is that the team needs to agree on the layout of the UML class diagram, and the sequence diagram on how the different components will interact with each other, before proceeding to develop the section of code assigned. The alignment is needed to avoid conflicting code, bugs, or incompatible interfaces.

- Another lesson learned during the agile practice is when working on the Github project, the pull request of a new commit needs to be reviewed and amended soonest possible so that other team members can start working on the new commit instead of the old code. This will help eliminate conflicting code or bugs.

- Although the team is practicing agile practice, the how-to documentation, especially those related to system deployment is still useful in saving the time and efforts of other team members. This is because the team members can avoid the trial and error phase and directly learn from the others on how to deploy the system components.

11.3 Acknowledgment

This project would not have been possible without the TweetQA authors from the University of California Santa Barbara: Wenhan Xiong, Jiawei Wu, Hong Wang, Vivek Kulkarni, and William Yang Wang; as well as the co-authors from IBM research: Mo Yu, Shiyu Chang, and Xiaoxiao Guo. The TweetQA Dataset paved the way for the success of our training processes and the accompanying paper gave many insights that led to the choice of transformer model to use.

We thank the creators and contributors of the Huggingface community for the amazing framework and documentation that made the training development much simpler than it would’ve been without them.

We also acknowledge and thank the advisors from Penn State Behrend who supported the team throughout the project, especially during the periods where progress was slow before the switch to Tensorflow. Dr. Naseem Ibrahim, Dr. Zhifeng Xiao, and Dr. Ziyun Huang provided us with the encouragement and confidence to continue working toward our goal of reaching minimum scores in the 60-70% range when the outlook was looking grim going into the second semester of the project.

Finally, we acknowledge the Penn State Behrend School of Engineering for providing us with the education required to complete the project, as well as sponsoring the project itself. Another acknowledgment goes to Dr. Xiao, who is the contact from the school for this project.

12 References

[1] *Why tensorflow*. TensorFlow. (n.d.). Retrieved October 2, 2021, from https://www.tensorflow.org/about.

[2] Team, K. (n.d.). *Keras documentation: About keras*. Keras. Retrieved October 2, 2021, from https://keras.io/about/.

[3] *About Pandas*. pandas. (n.d.). Retrieved October 2, 2021, from https://pandas.pydata.org/about/.

[4] *Flask*. Pallets. (n.d.). Retrieved October 2, 2021, from https://palletsprojects.com/p/flask/.

[5] *The Python SQL Toolkit and Object Relational Mapper*. SQLAlchemy. (n.d.). Retrieved October 2, 2021, from https://www.sqlalchemy.org/.

[6] Farrell, D. (2021, June 13). *Python rest apis with flask, Connexion, and SQLAlchemy*. Real Python. Retrieved October 2, 2021, from https://realpython.com/flask-connexion-rest-api/.

[7] Angular. (n.d.). Retrieved October 2, 2021, from https://angular.io/guide/what-is-angular.

[8] Google. (n.d.). *App Engine Application Platform | google cloud*. Google. Retrieved October 2, 2021, from https://cloud.google.com/appengine#all-features.

[9] Google. (n.d.). *Cloud storage | google cloud*. Google. Retrieved October 2, 2021, from https://cloud.google.com/storage.

[10] Google. (n.d.). *Introduction to ai platform | google cloud*. Google. Retrieved October 2, 2021, from https://cloud.google.com/ai-platform/docs/technical-overview.

[11] Singh, V. (n.d.). *Flask vs Django in 2021: Which framework to choose?* Hackr.io. Retrieved October 2, 2021, from https://hackr.io/blog/flask-vs-django.

[12] *Welcome to connexion's documentation!*. Welcome to Connexion's documentation! - Connexion 2020.0.dev1 documentation. (n.d.). Retrieved October 3, 2021, from https://connexion.readthedocs.io/en/latest/.

[13] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, June). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Retrieved September 20, 2021, from https://aclanthology.org/N19-1423.pdf.

[14] Horev, R. (2018, November 17). BERT explained: State of the art language model for NLP. Medium. Retrieved September 20, 2021, from https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270.

[15] Daffodil Software. (2021, June 25). *Top 10 pre-trained NLP language models*. Software Development Services Company. Retrieved October 3, 2021, from https://insights.daffodilsw.com/blog/top-5-nlp-language-models.

[16] LIANG, X. (2020, May 31). What is XLNet and why it outperforms Bert. Medium. Retrieved October 3, 2021, from https://towardsdatascience.com/what-is-xlnet-and-why-it-

outperforms-bert-8d8fce710335.

[17] Vu, K. (2021, February 25). *GPT-2 (GPT2) vs. GPT-3 (GPT3): The openai showdown - dzone AI*. dzone.com. Retrieved October 3, 2021, from https://dzone.com/articles/gpt-2-gpt2-vs-gpt-3-gpt3-the-openai-showdown.

[18] Xiong, W., Wu, J., Wang, H., Kulkarni, V., Yu, M., Chang, S., Guo, X., & Wang, W. Y. (2019). Tweetqa: A Social Media Focused Question answering dataset. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. https://doi.org/10.18653/v1/p19-1496

[19] *Testing Flask Applications*. Pallets (2010). Retrieved November 21, 2021 from https://flask.palletsprojects.com/en/2.0.x/testing/

[20] *Developer guides, Testing*. Google (2021). Retrieved November 22, 2021 from https://angular.io/guide/testing

[21] "@Ngrx/Store." *NgRx Docs*, https://v7.ngrx.io/guide/store

[22] Bert-base-uncased · hugging face. bert-base-uncased · Hugging Face. (n.d.). Retrieved February 20, 2022, from https://huggingface.co/bert-base-uncased

[23] Akhtar, Z. (2021, January 12). Bert Base vs Bert Large. OpenGenus IQ: Computing Expertise & Legacy. Retrieved February 20, 2022, from https://iq.opengenus.org/bert-base-vs-bert-large/

[24] Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

[25] Nguyen, D. Q., Vu, T., & Nguyen, A. T. (2020). BERTweet: A pre-trained language model for English Tweets. arXiv preprint arXiv:2005.10200.

[26] Cohen, A. (2011, July 8). FuzzyWuzzy: Fuzzy String Matching in Python. Chairnerd. Retrieved February 20, 2022, from https://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/

[27] Brownlee, J. (2019, October 25). Difference between a batch and an epoch in a neural network. Machine Learning Mastery. Retrieved February 20, 2022, from https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/#:~:text=The%20number%20of%20epochs%20is%20a%20hyperparameter%20that%20defines%20the,update%20the%20internal%20model%20parameters.

[28] Brownlee, J. (2020, September 11). Understand the impact of learning rate on neural network performance. Machine Learning Mastery. Retrieved February 20, 2022, from https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-

learning-neural-networks/

[29] Grand, Mark. *Patterns in Java*. Second ed., vol. 1, Wiley Publishing, Inc, 2002.

[30] "Cooky-Cutter." *Npm*, Skovy, https://www.npmjs.com/package/cooky-cutter.

[31] Collings, Tom. "Controller-Service-Repository." *Medium*, Medium, 10 Aug. 2021, https://tom-collings.medium.com/controller-service-repository-16e29a4684e5.

[32] Baeldung. "The DTO Pattern (Data Transfer Object)." *Baeldung*, 27 Jan. 2022, https://www.baeldung.com/java-dto-pattern.

[33] Geron, Aurelien. "Introduction to Artificial Neural Networks with Keras." *Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow*, 2nd ed., O'Reilly Media, Inc, Sebastopol, CA, 2019, pp. 279–292.

Appendix U

Appendix U focuses on providing specific details for user scenarios. A summary of all use cases is provided in **Table 13.1** below and **Tables 13.2 - 13.11** provide specific details for each individual use case.

Table 13.1: Usecase Summary

Project Name: Building a Question Answering System using Tweets				
Use Case ID	Use Case Name	Level	Author	Version
UC-001	Input tweet & question	Primary	Luke Westfall	0.3
UC-002	Rate ML generated answer	Primary	Luke Westfall	0.2
UC-003	Change ML model and rerun	Primary	Luke Westfall	0.2
UC-004	View metrics	Primary	Luke Westfall	0.2
UC-005	Train models on dataset	Summary	Luke Westfall	0.2
UC-006	Deploy trained model to backend	Primary	Luke Westfall	0.2
UC-007	Login to frontend	Primary	Luke Westfall	0.2
UC-008	Authenticate credentials	Subfunction	Luke Westfall	0.2
UC-009	Generate token with invitation email	Primary	Luke Westfall	0.2
UC-010	Send email to invitee with token	Subfunction	Luke Westfall	0.2

Table 13.2: UC-001

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-001
Use Case Name:	Input tweet & question
User Goal:	Get ML generated answer to question about tweet
Scope:	Web app (client) & Web API
Level:	Primary Task
Primary Actor:	ML Student
Precondition:	The user has an access token from an invite email
Minimal Guarantee:	No intelligible answer is given by the system
Success Guarantee:	The system provides an answer to the question that is meaningful
Trigger:	The user inputs the tweet text/The user click the get random tweet button
Success Scenario:	**Step Action**
	1 The user inputs the tweet text/The user click the get random tweet button
	2 The user inputs the question text
	3 The user selects the ML model they want to answer
	4 The user provides information privacy consent on tweet used.
	5 The user submits their inputs
	6 The web app (client) sends the tweet, question, and desired model to the API
	7 The API generates an answer using the selected ML model
	8 The API responds to the web app with the answer and ID
	9 The web app (client) displays the answer to the user
Extensions:	**Branching Scenarios**
1A,2A	**Condition: The user leaves the tweet or question text blank**
	Step Actions
	1 The user tries to submit the form
	2 The web app (client) displays an error message
	3 Go back to step 1
1B	**Condition: The user click the get random tweet button**
	Step Actions
	1 The web app(client) get a random tweet from the API

	2 The web app(client) display the tweet to the user
9A	**Condition: The API responds with error**
	Step Actions
	1 The web app (client) displays an error message with a reason based on API response
	2 Go back to step 2 to retry

Table 13.3: UC-002

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-002
Use Case Name:	Rate ML generated answer
User Goal:	Provide feedback on the answer for storage in the database
Scope:	Web app (client) & Web API
Level:	Primary Task
Primary Actor:	ML Student
Precondition:	The user has received an answer to their question in UC-001
Minimal Guarantee:	Feedback is not stored
Success Guarantee:	The user's feedback is saved to the database for the tweet and question that was previously answered
Trigger:	The user completes UC-001 or UC-003
Success Scenario:	**Step Action**
	1 The user completes UC-001 or UC-003
	2 The web app (client) displays the feedback form
	3 The user evaluates if the answer is correct or incorrect
	4 (Optional) The user fills in the optional explanation text field
	5 The user submits the feedback
	6 The web app (client) sends the feedback to the API along with the ID returned in UC-001
	7 The API stores the feedback data for the ID given
	8 The web app (client) acknowledges the success
Extensions:	**Branching Scenarios**
7A	**Condition: The API responds with an error status**
	Step Actions
	1 The web app (client) displays an error message with a reason based on API response
	2 Go back to step 5 to retry

Table 13.4: UC-003

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-003
Use Case Name:	Change ML model and rerun
User Goal:	Compare the quality of ML answer with the previous model/answer
Scope:	Web app (client) & Web API
Level:	Primary Task
Primary Actor:	ML Student
Precondition:	The user has received an answer to their question in UC-001
Minimal Guarantee:	No intelligible answer is given by the system
Success Guarantee:	The system provides an answer to the question that is meaningful
Trigger:	The user completes UC-001
Success Scenario:	**Step Action**
	1 The user completes UC-001
	2 (Optional) The user completes UC-002
	3 The user selects a different model to repeat the tweet and question
	4 The user submits the tweet and question with the new model
	5 The web app (client) sends the tweet, question, and desired model to the API
	6 The API generates an answer using the selected ML model
	7 An ID is generated by the system for the stored data
	8 The API responds to the web API with the answer and ID
	9 The web app (client) displays the answer to the user along with any previous answers and their models to the same question
Extensions:	**Branching Scenarios**
8A	**Condition: The API responds with an error status**
	Step Actions
	1 The web app (client) displays an error message with a reason based on API response
	2 Go back to step 4 to retry

Table 13.5: UC-004

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-004
Use Case Name:	View Metrics
User Goal:	Get insights on the performance of the ML models, statistics of the system, etc.
Scope:	Web app (client) & Web API
Level:	Primary Task
Primary Actor:	ML Student
Precondition:	None
Minimal Guarantee:	Metrics are not displayed due to error
Success Guarantee:	Users can see all metrics provided
Trigger:	The web app requests the current metrics from the API
Success Scenario:	**Step Action**
	1 The web app requests the current metrics from the API
	2 The API responds to the web app with the data
	3 The web app presents the data in an aesthetic manner
Extensions:	**Branching Scenarios**
2A	**Condition: The API responds with an error status**
	Step Actions
	1 The web app (client) displays an error message with a reason based on the API response
	2 Go back to step 1

Table 13.6: UC-005

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-005
Use Case Name:	Train models using dataset
User Goal:	Generate updated weights to deploy to the API for improved question answering
Scope:	ML Pipeline
Level:	Summary Task
Primary Actor:	Administrator
Precondition:	Dataset has data (TweetQA, plus optional user-submitted data)
Minimal Guarantee:	No new weights are produced
Success Guarantee:	Training yields updated weights
Trigger:	The user signs into Google Cloud Platform (GCP)
Success Scenario:	**Step Action**
	1 The user signs into Google Cloud Platform (GCP)
	2 (Optional) The user tweaks hyperparameters for fine-tuning
	3 The ML pipeline is executed
	4 The system produces an output weights file
	5 User downloads weights file for deployment to API (see UC-008)
Extensions:	**Branching Scenarios**
4A	**Condition: An exception occurs within the ML pipeline**
	Step Actions
	1 The exception is logged in detail
	2 The user diagnoses the exception
	3 The user fixes the error that led to the exception
	4 Go back to step 3
4B	**Condition: GCP times out on the training**
	Step Actions
	1 Timeout is logged
	2 User diagnoses the timeout
	3 Go back to step 3

Table 13.7: UC-006

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-006
Use Case Name:	Login to the web app
User Goal:	Gain access to administrator features within the web app
Scope:	Web App
Level:	Primary Task
Primary Actor:	Administrator
Precondition:	None
Minimal Guarantee:	The user fails to log in to the system and lacks access to administrator areas of the app
Success Guarantee:	User is authenticated by the system and can access administrator areas of the app
Trigger:	The user navigates to the login page of the web app
Success Scenario:	**Step Action**
	1 The user enters their email and password and submits the form
	2 The web app sends the encrypted credentials to the API
	3 The API performs the authentication (see UC-007)
	4 The API responds with an authorization token
	5 The web app redirects the user to the administrator screen
Extensions:	**Branching Scenarios**
1A	**Condition: One or both credential fields are left blank**
	Step Actions
	1 The web app displays an error message
	2 The user fills in the missing information
	3 Go to step 1
4A	**Condition: The API responds with an error status**
	Step Actions
	1 The web app (client) displays an error message with a reason based on the API response
	2 Go to step 1

Table 13.8: UC-007

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-007
Use Case Name:	Authenticate credentials
User Goal:	Gain access to administrator features within the web app
Scope:	API
Level:	Subfunction Task
Primary Actor:	None
Precondition:	None
Minimal Guarantee:	No response is given (timeout)
Success Guarantee:	Credentials are correct, the authorization token is granted and given in 200 response
Trigger:	The API receives a request to log in with encrypted credentials
Success Scenario:	**Step Action**
	1 The API receives a request to log in with encrypted credentials
	2 The API decrypts the email and password from the request
	3 The API looks up the email in the database with its hashed password and salt
	4 The API uses the salt to hash the password from the request
	5 The API checks that the hashed passwords are a match
	6 A match is found
	7 The API generates an authorization token
	8 The API responds with the authorization token
Extensions:	**Branching Scenarios**
3A	**Condition: Email is not found in the database**
	Step Actions
	1 The API returns resource not found
6A	**Condition: The hashed passwords do not match**
	Step Actions
	1 The API returns unauthorized

Table 13.9: UC-008

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-008
Use Case Name:	Deploy trained model to API
User Goal:	Upload new, updated weights for better question answering
Scope:	Web app & API
Level:	Primary Task
Primary Actor:	Administrator
Precondition:	The user has downloaded weights (UC-005) and is authorized as an administrator (UC-006)
Minimal Guarantee:	Weights are not updated, old weights are retained
Success Guarantee:	New weights file is uploaded and the previous one is discarded
Trigger:	User selects the model that they desire to update
Success Scenario:	**Step Action**
	1 User selects the model that they desire to update
	2 The user selects the weights file they wish to upload
	3 The web app starts uploading the weights file to the API
	4 The API accepts the file stream upload and stores the weights file
Extensions:	**Branching Scenarios**
4A	**Condition: The upload stream is interrupted**
	Step Actions
	1 The API responds with an error
	2 The web app displays an error message asking that the user retry
	3 Go to step 1

Table 13.10: UC-009

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-009
Use Case Name:	Generate a token with an invitation email
User Goal:	Enter an email address for their invitee and have the system send them an invitation email including a token for access
Scope:	Web app & API
Level:	Primary Task
Primary Actor:	Administrator
Precondition:	The user is authorized as an administrator (UC-006)
Minimal Guarantee:	No email invitation is sent
Success Guarantee:	An email invitation is sent
Trigger:	The user enters the email address of the invitee and submits
Success Scenario:	**Step Action**
	1 The user enters the email address of the invitee and submits
	2 The web app sends the email address to the API
	3 The API sends the email invitation (see UC-010)
	4 The web app acknowledges the success
Extensions:	**Branching Scenarios**
2A	**Condition: The email address is malformed**
	Step Actions
	1 The web app asks the user to correct the email address
	2 Go to step 1
6A	**Condition: The API responds with an error**
	Step Actions
	1 The web app (client) displays an error message with a reason based on the API response
	2 Go to step 2

Table 13.11: UC-010

Project Name:	**Building a Question Answering System using Tweets**
Use Case ID:	UC-010
Use Case Name:	Send email to invitee with token
User Goal:	An access token is generated and an email with the token is sent to the invitee
Scope:	API
Level:	Subfunction Task
Primary Actor:	None
Precondition:	UC-009 has been completed
Minimal Guarantee:	No email invitation is sent
Success Guarantee:	An email invitation is sent
Trigger:	API receives a request from the administrator on the web app to generate an email
Success Scenario:	**Step Action**
	1 API receives a request from the administrator on the web app to generate an email
	2 API randomly generates a token with sufficient entropy
	3 The API sends the email invitation
	4 The API responds to the web app with status 200 (success)
Extensions:	**Branching Scenarios**
3A	**Condition: Email fails to send due to SMTP error**
	Step Actions
	1 API retries, if success then go to step 6
	2 Otherwise API responds to the web app with an error

Appendix R

Appendix R provides details on all requirements engineered by the development team. **Section 14.1** provides further details on all user functional requirements, **Section 14.2** provides further details on all user non-functional requirements, **Section 14.3** provides further details on all system functional requirements, and **Section 14.4** provides further details on all system non-functional requirements.

User Functional Requirements

Tables 14.1 - 14.6 below provide further details for all the user functional requirements including description, priority, and traceability information.

Table 14.1: UF-A

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	UF-A		Type	Functional	Non-Functional
Creation:	Sep 30 2021 4:30 PM		User	X	
Modification:	Sep 30 2021 4:30 PM		System		
Description:	The user will provide a tweet-question pair, then the system will return an answer.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SF-A-1, SF-A-2, SF-A-3				
Justify why UF-A can be completely covered by SF-A-1, SF-A-2, SF-A-3	SF-A-1, SF-A-2, SF-A3 and describe the interaction between the user utilizing the web interface to submit a tweet-question pair and receiving an answer from the system utilizing the TweetQA API.				
Traceability:	Use cases	UC-001			
	Test cases	TC-001, TC-005, TC-006, TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-015, TC-018, TC-026, TC-027, TC-32			

Table 14.2: UF-B

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	UF-B			Type	Functional	Non-Functional
Creation:	Sep 30 2021 4:30 PM			User	X	
Modification:	Sep 30 2021 4:30 PM			System		
Description:	The user will be able to provide feedback on the provided answer.					
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SF-B-1, SF-B-2, SF-B-3				
Justify why UF-A can be completely covered by SF-B-1, SF-B-2, SF-B-3	SF-B-1, SF-B-2, and SF-B-3 describe how user feedback will be used to create additional labels in the system's dataset to utilize for future machine learning training.				
Traceability:	Use cases	UC-003			
	Test cases	TC-003, TC-005, TC-006, TC-010, TC-011, TC-012, TC-013, TC-014, TC-017, TC-018, TC-019, TC-026, TC-027, TC-032			

Table 14.3: UF-C

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	UF-C			Type	Functional	Non-Functional
Creation:	Sep 30 2021 4:30 PM			User	X	
Modification:	Sep 30 2021 4:30 PM			System		
Description:	The user can select different machine learning models to submit a tweet-question pair to.					
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SF-C-1				
Justify why UF-A can be completely covered by SF-C-1	SF-C-1 describes how the system will route the submission of a tweet-question pair based on user selection.				
Traceability:	Use cases	UC-004			
	Test cases	TC-002, TC-004, TC-007, TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-015, TC-018, TC-026, TC-027			

Table 14.4: UF-D

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	UF-D			Type	Functional	Non-Functional
Creation:	Sep 30 2021 4:30 PM			User	X	
Modification:	Sep 30 2021 4:30 PM			System		
Description:	The user can generate a report of a machine learning model and the current dataset.					
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SF-D-1, SF-D-2, SF-D-3				
Justify why UF-D can be completely covered by SF-D-1, SF-D-2, SF-D-3	SF-D-1, SF-D-2, and SF-D-3 describe the different parts of the report that will be generated by the system.				
Traceability:	Use cases	UC-005			
	Test cases	TC-005, TC-006, TC-007, TC-018, TC-019, TC-021, TC-022, TC-023			

Table 14.5: UF-E

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	UF-E			Type	Functional	Non-Functional
Creation:	Sep 30 2021 4:30 PM			User	X	
Modification:	Sep 30 2021 4:30 PM			System		
Description:	The user can generate a comparison report between two different machine learning models.					
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SF-E-1, SF-E-2, SF-E-3				
Justify why UF-E can be completely covered by SF-E-1, SF-E-2, SF-E-3	SF-E-1, SF-E-2, and SF-E-3 describe the different parts of the comparison report that will be generated by the system.				
Traceability:	Use cases	UC-005			
	Test cases	TC-005, TC-006, TC-007, TC-018, TC-019, TC-021, TC-022, TC-023			

Table 14.6: UF-F

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	UF-F			Type	Functional	Non-Functional
Creation:	Sep 30 2021 4:30 PM			User	X	
Modification:	Sep 30 2021 4:30 PM			System		
Description:	Users will be invited to interact with the system via a system generated token.					
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SF-F-1				
Justify why UF-F can be completely covered by SF-F-1	SF-F-1 describes how user access will be granted based on invitation.				
Traceability:	Use cases	UC-010, UC-011			
	Test cases	TC-032			

User Non-functional Requirements

Tables 14.7 - 14.18 provide further details for all user non-functional requirements including description, priority, subtype, and traceability information.

Table 14.7: UP-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UP-01			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	For the first group of users which are the Machine Learning Students, they should require less than 5 minutes of training to use the system. The system should have a simplified front-end web page to interact with the users				Product(sub-type below)	
					Usability Requirements	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SP-01-01, SP-01-02, SP-01-03, SP-01-04					
Justify why UP-01 can be completely covered by SP-01-01, SP-01-02, SP-01-03, SP-01-04	SP-01-02, SP-01-03, SP-01-04 describe the specific requirements for the front-end web page to be considered simple and user-friendly to the students. So that they can learn to use the system within 5 minutes (SP-01-01)					
Traceability	Use Cases	UC-001				
	Test Cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.8: UP-02

Project Name	**Building a Question Answering System using Tweets**				
Requirement #	UP-02		Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM		User		X
Modification	Sep 29 2021 4:20 PM		System		
Description	System administrators should require less than 1 hour of training to use the system. The administrative view page should also be simplified with enough hints to guide the admins. The webpage should contain the tabs corresponding to all the main functions available to the admin.			Product(sub-type below)	
				Usability Requirements	
Priority	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SP-02-01, SP-02-02, SP-02-03, SP-02-04				
Justify why UP-02 can be completely covered by SP-02-01, SP-02-02, SP-02-03, SP-02-04	SP-02-02, SP-02-03, SP-02-04 describe the specific requirements for the administration of the web-page to be simple to use to the administrator. So that they can learn to use the system with less than 1 hour of training (SP-02-01)				
Traceability	Use Cases	N/A			
	Test Cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.9: UP-03

Project Name	**Building a Question Answering System using Tweets**				
Requirement #	UP-03		Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM		User		X
Modification	Sep 29 2021 4:20 PM		System		
Description	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on all the three performance metrics used (BLEU-1, METEOR, ROUGE-L scores).			Product(sub-type below)	
				Performance Requirements	
Priority	Highest	High	Medium	Low	Lowest
This Req. is Refined Into	SP-03-01. SP-03-02, SP-03-03				
Justify why UP-03 can be completely covered by SP-03-01, SP-03-02, SP-03-03	SP-03-01. SP-03-02, SP-03-03 are the breakdown requirements of the three performance metrics.				
Traceability	Use Cases	UC-005, UC-006			
	Test Cases	TC-034			

Table 14.10: UP-04

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UP-04			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The machine learning model should achieve a more ambitious target of 70% score and above on all three performance metrics after continuous improvement.				Product(sub-type below)	
					Performance Requirements	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SP-04-01. SP-04-02, SP-04-03					
Justify why UP-04 can be completely covered by SP-04-01. SP-04-02, SP-04-03	SP-04-01. SP-04-02, SP-04-03 are the breakdown requirements of the three performance metrics.					
Traceability	Use Cases	UC-005, UC-006				
	Test Cases	TC-034				

Table 14.11: UP-05

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UP-05			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The webpage should return the result (answer to the question on the tweet) within less than one second after the user sends the question.				Product(sub-type below)	
					Performance Requirements	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SP-05-01					
Justify why UP-05 can be completely covered by SP-05-01.	SP-05-01 is equivalent to UP-05					
Traceability	Use Cases	UC-001				
	Test Cases	TC-016				

Table 14.12: UP-06

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UP-06			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The software system is expected to have an availability of at least 99%. The maximum allowed system failure rate will be less than 2 hours within one week.				Product(sub-type below)	
					Reliability	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SP-06-01, SP-06-02					
Justify why UP-06 can be completely covered by SP-06-01, SP-06-02	The two requirements (availability and maximum failure rate) in UP-06 is further breakdown into SP-06-01, SP-06-02 respectively.					
Traceability	Use Cases	All				
	Test Cases	All				

Table 14.13: UP-07

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UP-07			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The system should have two different levels of access for the normal user and the administrative user. The ML student users will be invited to use the system, and will gain access to the web interface via a system generated token. The administrative user will need to log in to the system using their administrative account with the password.				Product(sub-type below)	
					Security	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SF-P-5, SF-P-6					
Justify why UP-07 can be completely covered by SF-P-5, SF-P-6	The two requirements (normal user authentication and administrator authentication) in UP-07 are further broken into SF-P-5, SF-P-6 respectively.					
Traceability	Use Cases	UC-007, UC-008, UC-009, UC-010				
	Test Cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.14: UO-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UO-01			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The machine learning model will need to be developed using Google Colab/Spyder Anaconda with Python as the main programming language. The model results are to be submitted to CodaLab for the competition.				Product(sub-type below) Organizational Development	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SO-01-01, SO-01-02					
Justify why UO-01 can be completely covered by SO-01-01, SO-01-02	The two requirements (use of Python and submission to CodaLab) in UO-01 are further broken into SO-01-01, SO-01-02 respectively.					
Traceability	Use Cases	UC-005. UC-006				
	Test Cases	TC-034				

Table 14.15: UO-02

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UO-02			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Oct 23 2021 4:20 PM			System		
Description	The website should be displayed properly on the mainstream browsers run on the mainstream Operating System. The content of the website should be transmitted smoothly to the client browser with a connection speed of 512 kbps or better.				Product(sub-type below) Organization Environmental	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SO-02-01, SO-02-02					
Justify why UO-03 can be completely covered by SO-03-01, SO-03-02	The two requirements (proper display on mainstream browsers/OS combinations and loading requirement) in UO-02 are further broken into SO-02-01, SO-02-02 respectively.					
Traceability	Use Cases	All except UC-005, UC-006				
	Test Cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.16: UO-03

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UO-03			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The website should be displayed properly on the mainstream browsers (Chrome 50 or newer, Firefox 45 or newer, Edge, Safari 8) run on the mainstream Operating System (Windows 7 or newer, Mac OS X 10.6 or newer. IOS, Android). The content of the website should be transmitted smoothly to the client browser with a connection speed of 512 kbps or better.				Product(sub-type below)	
					Organization Environmental	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SO-03-01, SO-03-02					
Justify why UO-03 can be completely covered by SO-03-01, SO-03-02	The two requirements (proper display on mainstream browsers/OS combinations and loading requirement) in UO-03 are further broken into SO-03-01, SO-03-02 respectively.					
Traceability	Use Cases	All except UC-006, UC-009, UC-011				
	Test Cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.17: UE-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UE-01			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The website should properly credit the TweetQA dataset used, and the use of any machine learning model developed by others.				Product(sub-type below)	
					Intellectual Property Protection	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into	SE-01-01					
Justify why UE-01 can be completely covered by SE-01-01	SE-01-01 describes the specific details on where to show the credit for any Intellectual Property used/referenced.					
Traceability	Use Cases	N/A				
	Test Cases	TC-035				

Table 14.18: UE-02

Project Name	**Building a Question Answering System using Tweets**					
Requirement #	UE-02			Type	Functional	Non-Functional
Creation	Sep 29 2021 4:20 PM			User		X
Modification	Sep 29 2021 4:20 PM			System		
Description	The main language to be used by the model and the website should be English.				Product(sub-type below)	
					Cultural and Social	
Priority	Highest	High	Medium	Low		Lowest
This Req. is Refined Into		SE-02-01				
Justify why UE-02 can be completely covered by SE-02-01		SE-02-01 is equivalent to UE-02.				
Traceability	Use Cases	All				
	Test Cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

System Functional Requirements

Tables 14.19 - 14.39 provide further details for all the system functional requirements including description, priority, and traceability information.

Table 14.19: SF-A-1

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-A-1		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will utilize a web interface allowing users to submit tweet-question pairs to the system through the TweetQA API.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-A				
Justify why meeting SF-A-1 can contribute to the fulfillment of UF-A	SF-A-1 describes the necessary interface needed to allow the user to submit tweet-question pairs to the machine learning model.				
Traceability:	Use cases	UC-001			
	Test cases	TC-001, TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-015, TC-018, TC-026, TC-027			

Table 14.20: SF-A-2

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-A-2		Type	Functional	Non-Functional
Creation:	Mar 20 2022 12:30 PM		User		
Modification:	Mar 20 2022 12:30 PM		System	X	
Description:	The system will allow the user to select a random tweet from the database and start asking question				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-A				
Justify why meeting SF-A-2 can contribute to the fulfillment of UF-A	SF-A-2 describes one of the options to allow the user to submit tweet-question pairs to the machine learning model.				
Traceability:	Use cases	UC-001			
	Test cases	TC-001, TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-015, TC-018, TC-026, TC-027			

Table 14.21: SF-A-3

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-A-3		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will predict an answer based on the user provided question-answer pair, and will be communicated to the user through the TweetQA API.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-A				
Justify why meeting SF-A-3 can contribute to the fulfillment of UF-A	SF-A-3 describes the necessary interface needed to allow the user to receive a machine learning predicted answer based on their submitted tweet-question pair.				
Traceability:	Use cases	UC-002			
	Test cases	TC-005, TC-006, TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-015, TC-016, TC-018, TC-026, TC-027, TC-032			

Table 14.22: SF-B-1

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SF-B-1			Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM			User		
Modification:	Oct 2 2021 12:30 PM			System	X	
Description:	The system will prompt the user a yes/no questionnaire regarding the validity of the predicted answer.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From		UF-B				
Justify why meeting SF-B-1 can contribute to the fulfillment of UF-B		SF-B-1 describes the necessary user interface needed to allow the user to provide feedback on if the predicted answer was correct or not.				
Traceability:	Use cases	UC-002				
	Test cases	TC-010, TC-011, TC-012, TC-013, TC-014, TC-032				

Table 14.23: SF-B-2

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-B-2		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will record all tweet-question-answer triples to the database that receive user validation.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-B				
Justify why meeting SF-B-2 can contribute to the fulfillment of UF-B	SF-B-2 describes how tweet-question-answer tuples will be recorded or discarded based on user feedback.				
Traceability:	Use cases	UC-002			
	Test cases	TC-017, TC-032			

Table 14.24: SF-B-3

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-B-3		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will prompt for all other user feedback that is needed to determine if a tweet-question-answer triple is valid to create an additional label to the dataset.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-B				
Justify why meeting SF-B-3 can contribute to the fulfillment of UF-B	SF-B-3 describes how any other feedback that will be needed to determine if a given tweet-question-answer tuple is a valid label or not will be prompted to the user.				
Traceability:	Use cases	UC-003			
	Test cases	TC-003, TC-005, TC-006, TC-012, TC-013, TC-017, TC-032			

Table 14.25: SF-C-1

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SF-C-1			Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM			User		
Modification:	Oct 2 2021 12:30 PM			System	X	
Description:	The system will route the user submitted tweet-question pair to the appropriate machine learning model based on user selection in the web interface.					
Priority:	Highest	High	Medium	Low	Lowest	
This Req. is Engineered From	UF-C					
Justify why meeting SF-C-1 can contribute to the fulfillment of UF-C	SF-C-1 describes how the user will provide selection of a machine learning model through the web interface, and the system will route the submitted tweet-question pair to the appropriate model.					
Traceability:	Use cases	UC-001				
	Test cases	TC-002, TC-004, TC-007, TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-015, TC-018, TC-032				

Table 14.26: SF-D-1

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SF-D-1			Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM			User		
Modification:	Oct 2 2021 12:30 PM			System	X	
Description:	The system will be able to provide the current performance metrics					
Priority:	Highest	High	Medium	Low	Lowest	
This Req. is Engineered From	UF-D					
Justify why meeting SF-D-1 can contribute to the fulfillment of UF-D	SF-D-1 describes how the performance metrics (BLEU-1, METEOR, ROUGE-L) will be available to the user in the generated report.					
Traceability:	Use cases	UC-004				
	Test cases	TC-005, TC-006, TC-018, TC-020, TC-023, TC-024, TC-025				

Table 14.27: SF-D-2

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-D-2		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will be able to provide previously recorded performance metrics vs time				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-D				
Justify why meeting SF-D-2 can contribute to the fulfillment of UF-D	SF-D-2 describes how the historical performance metrics (BLEU-1, METEOR, ROUGE-L) will be available to the user in the report.				
Traceability:	Use cases	UC-004			
	Test cases	TC-005, TC-006, TC-018, TC-019, TC-023, TC-024, TC-025			

Table 14.28: SF-D-3

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-D-3		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will provide a useful visualization of the current dataset in the form of a word cloud and any other pertinent statistics related to the model (i.e. accuracy, precision, training loss, etc.).				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-D				
Justify why meeting SF-D-3 can contribute to the fulfillment of UF-D	SF-D-3 describes how a data visualization of the current dataset and other important machine learning statistics will be available to the user in the report.				
Traceability:	Use cases	UC-004			
	Test cases	TC-005, TC-006, TC-007, TC-018, TC-019, TC-021, TC-023, TC-024, TC-025			

Table 14.29: SF-E-1

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-E-1		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will be able to provide a comparison report between two models that highlights key differences related to performance between each model in an easy to read summary. Performance metrics including the BLEU, METEOR, and ROGUE-L should be included as well as any other metrics that could affect performance.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-E				
Justify why meeting SF-E-1 can contribute to the fulfillment of UF-E	SF-E-1 describes how a comparison of the performance metrics (BLEU-1, METEOR, ROUGE-L) will be available to the user in the comparison report.				
Traceability:	Use cases	UC-004			
	Test cases	TC-005, TC-006, TC-007, TC-018, TC-019, TC-023, TC-024, TC-025			

Table 14.30: SF-E-2

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-E-2		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will be able to provide a comparison of the difference in performance metrics vs time for the two models.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-E				
Justify why meeting SF-E-2 can contribute to the fulfillment of UF-E	SF-E-2 describes how a comparison of the historical performance metrics (BLEU-1, METEOR, ROUGE-L) will be available to the user in the comparison report.				
Traceability:	Use cases	UC-005			
	Test cases	TC-005, TC-006, TC-007, TC-018, TC-019, TC-023, TC-024, TC-025			

Table 14.31: SF-E-3

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-E-3		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	The system will provide a comparison of any other pertinent statistics related to the two models.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-E				
Justify why meeting SF-E-3 can contribute to the fulfillment of UF-E	SF-E-3 describes how any other pertinent statistics related to the two machine learning models will also be available to the user in the comparison report.				
Traceability:	Use cases	UC-004			
	Test cases	TC-005, TC-006, TC-007, TC-018, TC-019, TC-021, TC-023, TC-024, TC-025			

Table 14.32: SF-F-1

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-F-1		Type	Functional	Non-Functional
Creation:	Oct 2 2021 12:30 PM		User		
Modification:	Oct 2 2021 12:30 PM		System	X	
Description:	A user should not be able to access the system unless provided with a system generated token.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UF-F				
Justify why meeting SF-F-1 can contribute to the fulfillment of UF-F	SF-F-1 describes how user accessibility will be limited to those invited to use the system.				
Traceability:	Use cases	UC-001, UC-002, UC-003, UC-004, UC-005			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-030			

Table 14.33: SF-P-1

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-P-1		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	Hints should be visible to the students in guiding them through the activities such as submitting the question, view the answer returned by the model, and provide feedback on the answer.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-01				
Justify why meeting SF-P-1 can contribute to the fulfillment of UP-01	SF-P-1 specify one of the detail requirements for the User Interface to meet UP-01 target				
Traceability:	Use cases	UC-001, UC-002, UC-003, UC-004			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.34: SF-P-2

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-P-2		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The administrative view page should also be simplified with enough hints to guide the admins.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-02				
Justify why meeting SF-P-2 can contribute to the fulfillment of UP-02	SF-P-2 specify one of the detail requirements for the User Interface to meet UP-01 target				
Traceability:	Use cases	UC-007, UC-008, UC-009, UC-010			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.35: SF-P-3

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SF-P-3			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	There should be a help page providing the “How to” guides for all activities that can be performed by the administrator.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UP-02					
Justify why meeting SF-P-3 can contribute to the fulfillment of UP-02	SF-P-3 specify one of the detail requirements for the User Interface to meet UP-01 target					
Traceability:	Use cases	UC-008				
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.36: SF-P-4

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SF-P-4			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The webpage should contain the tabs corresponding to actions available to the admin, such as generating the invitation code for the user to test the system, display the statistical result of the models, approve new questions answer set to input by the user, and rerun the pre-training process for the model if required.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UP-02					
Justify why meeting SF-P-4 can contribute to the fulfillment of UP-02	SF-P-4 specify one of the detail requirements for the User Interface to meet UP-01 target					
Traceability:	Use cases	UC-007, UC-008, UC-009, UC-010				
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.37: SF-P-5

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-P-5		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 24 2021 12:30 PM		System	X	
Description:	The machine learning student users will need to access the system via the system-generated token.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-07				
Justify why meeting SF-P-5 can contribute to the fulfillment of UP-07	SF-P-5 narrows down UP-07 requirements on security to one of the two stated.				
Traceability:	Use cases	UC-010			
	Test cases	TC-30			

Table 14.38: SF-P-6

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-P-6		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The administrative user will need to log in to the system using their administrative account with the password.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-07				
Justify why meeting SF-P-6 can contribute to the fulfillment of UP-07	SF-P-6 narrow down UP-07 requirements on security to one of the two stated.				
Traceability:	Use cases	UC-007, UC-008, UC-009, UC-010			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014, TC-033			

Table 14.39: SF-P-7

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SF-P-7		Type	Functional	Non-Functional
Creation:	Oct 24 2021 12:30 PM		User		
Modification:	Oct 24 2021 12:30 PM		System	X	
Description:	The system will obtain consent on information privacy from the users when they submit new tweets in SF-A-1				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UE-02				
Justify why meeting SF-P-7 can contribute to the fulfillment of UP-07	SF-P-7 specify the UE-02 requirements on how and when to obtain consent for adhering to information privacy law				
Traceability:	Use cases	UC-001			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

System Non-functional Requirements

Tables 14.40 - 14.58 provide further details for all the system's non-functional requirements including description, priority, and traceability information.

Table 14.40: SP-01-01

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SP-01-01		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	For the Machine Learning Students, they should require less than 5 minutes of self-training to use the system.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-01				
Justify why meeting SP-01-01 can contribute to the fulfillment of UP-01	SP-01-01 is the specific requirement of UP-01 with a measurable target.				
Traceability:	Use cases	UC-001			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.41: SP-01-02

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SP-01-02		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The first front-end web page visited by the students should contain all the information available to the students				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-01				
Justify why meeting SP-01-02 can contribute to the fulfillment of UP-01	SP-01-02 specify one of the detail requirements for the User Interface to meet UP-01 target				
Traceability:	Use cases	UC-001			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.42: SP-01-03

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SP-01-03		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The first front-end web page visited by the students should contain all the required interaction button/input text field.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-01				
Justify why meeting SP-01-03 can contribute to the fulfillment of UP-01	SP-01-03 specify one of the detail requirements for the User Interface to meet UP-01 target				
Traceability:	Use cases	UC-001, UC-002, UC-003, UC-004			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.43: SP-02-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-02-01			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	For the administrator user, they should require less than 1 hour of guided- training to use the system.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UP-02					
Justify why meeting SP-02-01 can contribute to the fulfillment of UP-01	SP-02-01 is the specific requirement of UP-02 with a measurable target					
Traceability:	Use cases	UC-005, UC-006, UC-007, UC-008, UC-009, UC-010				
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014				

Table 14.44: SP-03-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-03-01			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on BLEU metric.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UP-03					
Justify why meeting SP-03-01 can contribute to the fulfillment of UP-03	SP-03-01 narrows down the requirement of UP-03 into one of the performance metric					
Traceability:	Use cases	UC-006				
	Test cases	TC-034				

Table 14.45: SP-03-02

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SP-03-02		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on METEOR metric.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-03				
Justify why meeting SP-03-02 can contribute to the fulfillment of UP-03	SP-03-02 narrows down the requirement of UP-03 into one of the performance metric				
Traceability:	Use cases	UC-005			
	Test cases	TC-034			

Table 14.46: SP-03-03

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SP-03-03		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The machine learning model used to predict the answers given the tweets and questions should achieve a minimum score of 60% on ROUGE_L metric.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UP-03				
Justify why meeting SP-03-03 can contribute to the fulfillment of UP-03	SP-03-02 narrows down the requirement of UP-03 into one of the performance metric				
Traceability:	Use cases	UC-005			
	Test cases	TC-034			

Table 14.47: SP-04-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-04-01			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on the BLEU metric.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From		UP-04				
Justify why meeting SP-04-01 can contribute to the fulfillment of UP-04		SP-04-01 narrows down the requirement of UP-04 into one of the performance metric				
Traceability:	Use cases	UC-006				
	Test cases	TC-034				

Table 14.48: SP-04-02

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-04-02			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on METEOR metric.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From		UP-04				
Justify why meeting SP-04-02 can contribute to the fulfillment of UP-04		SP-04-02 narrows down the requirement of UP-04 into one of the performance metric				
Traceability:	Use cases	UC-006				
	Test cases	TC-034				

Table 14.49: SP-04-03

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-04-03			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The machine learning model used to predict the answers given the tweets and questions should achieve an ambitious score of 70% on ROUGE_L metric.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From		UP-04				
Justify why meeting SP-04-03 can contribute to the fulfillment of UP-04		SP-04-03 narrows down the requirement of UP-04 into one of the performance metric				
Traceability:	Use cases	UC-006				
	Test cases	TC-034				

Table 14.50: SP-05-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-05-01			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	For the web page, it should return the result (answer to the question on the tweet) within less than one second after the user sends the question.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From		UP-05				
Justify why meeting SP-04-03 can contribute to the fulfillment of UP-05		SP-05-01 is the specific requirement of UP-05 with a measurable target.				
Traceability:	Use cases	UC-002				
	Test cases	All				

Table 14.51: SP-06-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-06-01			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The software system is expected to have an availability of at least 99% measured by up time.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UP-06					
Justify why meeting SP-06-01 can contribute to the fulfillment of UP-06	SP-06-01 narrowed down UP-06 requirements on reliability to one of the two stated.					
Traceability:	Use cases	All				
	Test cases	All				

Table 14.52: SP-06-02

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SP-06-02			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The maximum allowed system failure rate will be less than 2 hours within any one week.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UP-06					
Justify why meeting SP-06-02 can contribute to the fulfillment of UP-06	SP-06-02 narrows down UP-06 requirements on reliability to one of the two stated.					
Traceability:	Use cases	All				
	Test cases	All				

Table 14.53: SO-01-01

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SO-01-01		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The machine learning model will need to be developed using Python as the main programming language.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UO-01				
Justify why meeting SO-01-01 can contribute to the fulfillment of UO-01	SO-01-01 narrows down UO-01 requirements on development to one of the two stated.				
Traceability:	Use cases	UC-005, UC-006			
	Test cases	All			

Table 14.54: SO-01-02

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SO-01-02		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The model results are to be submitted to CodaLab for the competition.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UO-01				
Justify why meeting SO-01-02 can contribute to the fulfillment of UO-01	SO-01-02 narrows down UO-01 requirements on development to one of the two stated.				
Traceability:	Use cases	N/A			
	Test cases	TC-034			

Table 14.55: SO-02-01

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SO-02-01		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 24 2021 12:30 PM		System		X
Description:	The website should be displayed properly on the mainstream browsers run on the mainstream Operating System .				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UO-02				
Justify why meeting SO-03-01 can contribute to the fulfillment of UO-03	SO-02-01 narrows down UO-02 requirements on environment to one of the two stated. Note there are multiple possible combinations of browser and operating systems, they are combined to simplify this report. Examples of mainstream browsers include (Chrome 50 or newer, Firefox 45 or newer, Edge, Safari 8), examples of mainstream operating systems include Windows 7 or newer, Mac OS X 10.6 or newer. IOS, Android. Each of the combinations should be tested out individually.				
Traceability:	Use cases	All			
	Test cases	TC-008, TC-009, TC-010, TC-011, TC-012, TC-013, TC-014			

Table 14.56: SO-02-02

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SO-03-02		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The website should be completely loaded on the client browser with a connection speed of 512 kbps or better within 10 seconds.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UO-02				
Justify why meeting SO-03-02 can contribute to the fulfillment of UO-03	SO-02-02 narrows down UO-03 requirements on environment to one of the two stated.				
Traceability:	Use cases	All except UC-005, UC-008, UC-010			
	Test cases	All			

Table 14.57: SE-01-01

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SE-01-01		Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM		User		
Modification:	Oct 4 2021 12:30 PM		System		X
Description:	The credit to the TweetQA dataset used, and credit to the use of any machine learning model developed by others should be visible in the first page.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UE-01				
Justify why meeting SE-01-01 can contribute to the fulfillment of UE-01	SE-01-01 specifies the requirement of UE-01 so that it is testable.				
Traceability:	Use cases	N/A			
	Test cases	TC-035			

Table 14.58: SE-02-01

Project Name	**Building a Question Answering System using Tweets**				
Requirement #:	SE-02-01		Type	Functional	Non-Functional
Creation:	Oct 24 2021 12:30 PM		User		
Modification:	Oct 24 2021 12:30 PM		System		X
Description:	All the tweets used by the website, including those submitted by the users, should observe the information privacy law.				
Priority:	Highest	High	Medium	Low	Lowest
This Req. is Engineered From	UE-02				
Justify why meeting SE-01-01 can contribute to the fulfillment of UE-01	SE-02-01 restates the requirement of UE-02				
Traceability:	Use cases	UC-001			
	Test cases	TC-035			

Table 14.59: SE-03-01

Project Name	**Building a Question Answering System using Tweets**					
Requirement #:	SE-02-01			Type	Functional	Non-Functional
Creation:	Oct 4 2021 12:30 PM			User		
Modification:	Oct 4 2021 12:30 PM			System		X
Description:	The main language to be used by the model and the website should be English.					
Priority:	Highest	High	Medium	Low		Lowest
This Req. is Engineered From	UE-02					
Justify why meeting SE-02-01 can contribute to the fulfillment of UE-02	SE-02-01 is equivalent to UE-02					
Traceability:	Use cases	All				
	Test cases	All				

Appendix T

Table 15.1: Test Suite Table TS-001

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API-DataService	
Test Case ID	**Test Stage**	**Test Case Description**
TC-001	Unit	The DataService object can create a datum entity in the database
TC-002	Unit	The DataService object can read a datum entity from the database by id
TC-003	Unit	The DataService object can update an existing datum entity in the database
TC-020	Unit	The DataService object can read all data since a given date
TC-021	Unit	The DataService object can read n number of last datum entries
TC-022	Unit	The DataService object can provide a word cloud
TC-028	Unit	The DataService object can provide a random datum entity from the database

Table 15.2: Test Suite Table TS-002

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-002: API-QAModelController	
Test Case ID	**Test Stage**	**Test Case Description**
TC-004	Integration	The create model end point is being reached and providing the appropriate response
TC-005	Integration	The read model endpoint being reached and providing the appropriate response
TC-006	Integration	The read latest model by type end point is being reached and providing the appropriate response
TC-007	Integration	The read latest models endpoint is being reached and providing the appropriate response

Table 15.3: Test Suite Table TS-003

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	**Test Stage**	**Test Case Description**
TC-008	Unit	The prediction form submit button should initially be disabled
TC-009	Unit	After filling out the initial form, the prediction should be ready to submit
TC-010	Unit	After submitting a prediction, the form should be ready to collect prediction correctness feedback.
TC-011	Unit	After receiving feedback for a correct prediction, the form should be ready to submit
TC-012	Unit	After receiving feedback for an incorrect prediction, the form should be ready to collect an alternate answer from the user
TC-013	Unit	After collecting an alternate answer from the user, the form should be ready to submit
TC-014	Unit	After submitting user feedback, the form should return to its initial state
TC-029	Unit	After clicking the Get Random Tweet button, a random tweet should fill the tweet text field

Table 15.4: Test Suite Table TS-004

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-004: WebApp-PredictionService	
Test Case ID	**Test Stage**	**Test Case Description**
TC-015	Unit	The create() method should POST a PredictionCreateRequestV1 and receive a PredictionResponseV1 in the response
TC-016	Unit	The read() method should GET a PredictionResponseV1 is the response
TC-017	Unit	The update() method should PUT a PredictionUpdateRequestV1 and receive a PredictionResponseV1 in the response

Table 15.5: Test Suite Table TS-005

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-005: API-QAModelService	
Test Case ID	**Test Stage**	**Test Case Description**
TC-018	Unit	The QAModelService object can create a datum entity in the database
TC-019	Unit	The QAModelService object can read a datum entity from the database by id
TC-023	Unit	The QAModelService object can read all instances of a model for a given type
TC-024	Unit	The QAModelService object can read the latest models
TC-025	Unit	The QAModelService object can read the latest model by type

Table 15.6: Test Suite Table TS-006

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-006: API-Prediction Service	
Test Case ID	**Test Stage**	**Test Case Description**
TC-026	Unit	The PredictionService object can create a prediction entity in the database
TC-027	Unit	The PredictionService object can update a prediction entity in the database

Table 15.7: Test Suite Table TS-007

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-007: System & Security Testing	
Test Case ID	**Test Stage**	**Test Case Description**
TC-030	System	The admin invitation is forwarding unique URLs
TC-031	System	The admin training deployment is working correctly
TC-032	System	The user question answer submission and machine learning prediction is working in production environment
TC-033	Security	The invitation page and training deployment page are only accessible to administrator who has logged in.

Table 15.8: Test Suite Table TS-008

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-008: Acceptance Testing	
Test Case ID	**Test Stage**	**Test Case Description**
TC-034	Acceptance	Evaluate the perfomance of the given model
TC-035	Acceptance	Ensure proper accreditation

Table 15.9: Test Case Table for TC-001

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: DataService	
Test Case ID	TC-001 (Unit Test)	
What to Test	The DataService object can create a datum entity in the database	
Test Data Input	An instance of Data class	
Expected Result	The database successfully adds a record containing the details of the Data instance input and returns the instance that was inserted.	
Traceability	Relevant User Req.(s)	UF-A
	Relevant System Req.(s)	SF-A-1
	Relevant Use Case.(s)	UC-001

Table 15.10: Test Case Table for TC-002

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API-DataService	
Test Case ID	TC-002 (Unit Test)	
What to Test	The DataService object can read a datum entity from the database by id	
Test Data Input	A valid id for an existing datum entity in the database	
Expected Result	The database successfully fetches the corresponding record having the id input and returns an instance of the Data class	
Traceability	Relevant User Req.(s)	UF-C
	Relevant System Req.(s)	SF-C-1
	Relevant Use Case.(s)	UC-003

Table 15.11: Test Case Table for TC-003

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API-DataService	
Test Case ID	TC-003 (Unit Test)	
What to Test	The DataService object can update an existing datum entity in the database	
Test Data Input	An instance of Data class	
Expected Result	The database successfully updates the record corresponding to the Data instance input (by id) with the details of the instance input.	
Traceability	Relevant User Req.(s)	UF-B
	Relevant System Req.(s)	SF-B-3
	Relevant Use Case.(s)	UC-002

Table 15.12: Test Case Table for TC-004

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-002: API-QAModelController	
Test Case ID	TC-004 (Integration Test)	
What to Test	The create model end point is being reached and providing the appropriate response	
Test Data Input	A POST request to the /v1/models endpoint with the requested creation object embedded in the body	
Expected Result	The API returns a QAModelResponse DTO of the requested object and 200 status code on successful creation	
Traceability	Relevant User Req.(s)	UF-C
	Relevant System Req.(s)	SF-C-1
	Relevant Use Case.(s)	UC-003

Table 15.13: Test Case Table for TC-005

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-002: API-QAModelController	
Test Case ID	TC-005 (Integration Test)	
What to Test	The read model endpoint being reached and providing the appropriate response	
Test Data Input	A GET request to the /v1/models/{id} end point with the requested id in the path	
Expected Result	The API returns a QAModelResponse DTO of the requested object and 200 status code on successful read	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-D, UF-E
	Relevant System Req.(s)	SF-A-3, SF-B-3, SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-003, UC-005

Table 15.14: Test Case Table for TC-006

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-002: API-QAModelController	
Test Case ID	TC-006 (Integration Test)	
What to Test	The read latest model by type end point is being reached and providing the appropriate response	
Test Data Input	A GET request to the /v1/models/{model_type} end point with the requested model_type in the path	
Expected Result	The API returns a QAModelResponse DTO of the requested object and 200 status code on successful read	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-D, UF-E
	Relevant System Req.(s)	SF-A-3, SF-B-3, SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-003, UC-005

Table 15.15: Test Case Table for TC-007

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-002: API-QAModelController	
Test Case ID	TC-007 (Integration Test)	
What to Test	The read latest models endpoint is being reached and providing the appropriate response	
Test Data Input	A GET request to the /v1/models/latest end point	
Expected Result	The API returns a QAModelCollection DTO of the latest models for each type and 200 status code on successful read	
Traceability	Relevant User Req.(s)	UF-C, UF-E
	Relevant System Req.(s)	SF-C-1, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-004

Table 15.16: Test Case Table for TC-008

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-008 (Unit Test)	
What to Test	That the prediction form submit button should initially be disabled	
Test Data Input	N/A	
Expected Result	That the disabled variable has a boolean value of true and the formState value is an instance of the InitialFormState class	
Traceability	Relevant User Req.(s)	UF-A, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001

Table 15.17: Test Case Table for TC-009

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-009 (Unit Test)	
What to Test	That after filling out the initial form, the prediction should be ready to submit	
Test Data Input	Random values for model, tweet and question fields	
Expected Result	That the disabled variable has a boolean value of false and the formState value is an instance of the AwaitingPredictionRequestState class	
Traceability	Relevant User Req.(s)	UF-A, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2,SF-A-3, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001, UC-002

T**able 15.18:** Test Case Table for TC-010

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-010 (Unit Test)	
What to Test	After submitting a prediction, the form should be ready to collect prediction correctness feedback.	
Test Data Input	Random values for model, tweet and question fields and the submit button is pressed	
Expected Result	The submit button has been called, the disabled variable has been reset to true, and the formState variables is an instance of the AwaitingIsCorrectState class	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3, SF-B-1, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001, UC-002

Table 15.19: Test Case Table for TC-011

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-011 (Unit Test)	
What to Test	That after receiving feedback for a correct prediction, the form should be ready to submit	
Test Data Input	Random values for model, tweet and question fields, submit button is pressed and correct radio button selected	
Expected Result	The disabled variable is false and the formState is an instance of the AwatingCorrectSubmissionState class	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3, SF-B-1, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001, UC-002

Table 15.20: Test Case Table for TC-012

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-012 (Unit Test)	
What to Test	That after receiving feedback for an incorrect prediction, the form should be ready to collect an alternate answer from the user	
Test Data Input	Random values for model, tweet and question fields, submit button is pressed and incorrect radio button selected	
Expected Result	The disabled variable is true and the formState variable is an instance of the AwatingAlternateAnswerState class	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2,SF-A-3, SF-B-1, SF-B-3, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001, UC-002

Table 15.21: Test Case Table for TC-013

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-013 (Unit Test)	
What to Test	That after collecting an alternate answer from the user, the form should be ready to submit	
Test Data Input	Random values for model, tweet and question fields, submit button is pressed, incorrect radio button selected and random value for alternate answer field entered	
Expected Result	The disabled variable is false and the formState variable is an instance of the AwatingIncorrectSubmissionState class	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3,SF-B-1, SF-B-3, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001, UC-002

Table 15.22: Test Case Table for TC-014

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-014 (Unit Test)	
What to Test	That after submitting user feedback, the form should return to its initial state	
Test Data Input	Random values for model, tweet and question fields, submit button is pressed correct radio button selected and submit button is pressed again	
Expected Result	The disabled variable should be true, the form's tweet field should be empty and the formState variable should be an instance of the class InitialFormState	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-C, UP-01
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3,SF-B-1, SF-C-1, SF-P-1
	Relevant Use Case.(s)	UC-001, UC-002, UC-003

Table 15.23: Test Case Table for TC-015

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-004: WebApp-PredictionService	
Test Case ID	TC-015 (Unit Test)	
What to Test	That the create() method should post a predictioncreaterequestv1 and receive a predictionresponsev1 in the response	
Test Data Input	POST request to /v1/predictions and a mock object of PredictionCreateRequestV1	
Expected Result	A responses of type PredictionResponseV1 should be returned and match the complimentary values in the create request	
Traceability	Relevant User Req.(s)	UF-A, UF-C
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3,SF-C-1
	Relevant Use Case.(s)	UC-001, UC-003

Table 15.24: Test Case Table for TC-016

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-004: WebApp-PredictionService	
Test Case ID	TC-016 (Unit Test)	
What to Test	That the read() method should get a predictionresponsev1 is the response	
Test Data Input	GET request to /v1/predictions/{id} and an id value for a mocked prediction	
Expected Result	A responses of type PredictionResponseV1 should be returned and match the id of the request	
Traceability	Relevant User Req.(s)	UP-05
	Relevant System Req.(s)	SF-A-3, SP-05-01
	Relevant Use Case.(s)	UC-001, UC-003

Table 15.25: Test Case Table for TC-017

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-004: WebApp-PredictionService	
Test Case ID	TC-017 (Unit Test)	
What to Test	That the update() method should put a predictionupdaterequestv1 and receive a predictionresponsev1 in the response	
Test Data Input	PUT request to /v1/predictions/{id} and a mock object of PredictionUpdateRequestV1	
Expected Result	A responses of type PredictionResponseV1 should be returned and match the complimentary values in the update request	
Traceability	Relevant User Req.(s)	UF-B
	Relevant System Req.(s)	SF-B-2, SF-B-3
	Relevant Use Case.(s)	UC-001, UC-002, UC-003

Table 15.26: Test Case Table for TC-018

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-005: API - QAModelService	
Test Case ID	TC-018 (Unit Test)	
What to Test	The QAModelService object can create a datum entity in the database	
Test Data Input	An instance of the QAModel Class	
Expected Result	The database successfully adds a record containing the details of the QAModel instance input and returns the instance that was inserted.	
Traceability	Relevant User Req.(s)	UF-A, UF-C, UF-D, UF-E
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-C-1, SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-001, UC-003, UC-004, UC-005

Table 15.27: Test Case Table for TC-019

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-005: API - QAModelService	
Test Case ID	TC-019 (Unit Test)	
What to Test	The QAModelService object can read a datum entity from the database by id	
Test Data Input	A valid id for an existing QAModel entity in the database	
Expected Result	The database successfully fetches the corresponding record having the id input and returns an instance of the QAModel class	
Traceability	Relevant User Req.(s)	UF-A, UF-C, UF-D, UF-E
	Relevant System Req.(s)	SF-A-1,SF-A-2, SF-C-1, SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-001, UC-003, UC-004, UC-005

Table 15.28: Test Case Table for TC-020

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API - DataService	
Test Case ID	TC-020 (Unit Test)	
What to Test	The DataService object can read all data since a given date	
Test Data Input	A valid datetime object	
Expected Result	The database successfully fetches all data instances with a given creation date greater than the provided one	
Traceability	Relevant User Req.(s)	UF-C
	Relevant System Req.(s)	SF-C-1
	Relevant Use Case.(s)	UC-003

Table 15.29: Test Case Table for TC-021

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API - DataService	
Test Case ID	TC-021 (Unit Test)	
What to Test	The DataService object can read n number of last datum entries	
Test Data Input	An integer value	
Expected Result	The database successfully fetches a list of the last n number of data objects	
Traceability	Relevant User Req.(s)	UF-D, UF-E
	Relevant System Req.(s)	SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-005

Table 15.30 Test Case Table for TC-022

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API-DataService	
Test Case ID	TC-022 (Unit Test)	
What to Test	The DataService object can provide a word cloud	
Test Data Input	N/A	
Expected Result	The data_service object successfully generates a dictionary with word count of all unique words in the data instances	
Traceability	Relevant User Req.(s)	UF-D, UF-E
	Relevant System Req.(s)	SF-D-3, SF-E-3
	Relevant Use Case.(s)	UC-005

Table 15.31: Test Case Table for TC-023

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-005: API-QAModelService	
Test Case ID	TC-023 (Unit Test)	
What to Test	The QAModelService object can read all instances of a model for a given type	
Test Data Input	The requested model type string	
Expected Result	The QAModelService can provide a list of all QAModel instances persisted in the database that are of the requested type	
Traceability	Relevant User Req.(s)	UF-D, UF-E
	Relevant System Req.(s)	SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-005

Table 15.32: Test Case Table for TC-024

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-005: API-QAModelService	
Test Case ID	TC-024 (Unit Test)	
What to Test	The QAModelService object can read the latest models	
Test Data Input	N/A	
Expected Result	The QAModelService object can provide a list of the latest QAModels for each unique model type	
Traceability	Relevant User Req.(s)	UF-D, UF-E
	Relevant System Req.(s)	SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-005

Table 15.33: Test Case Table for TC-025

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-005: API-QAModelService	
Test Case ID	TC-025 (Unit Test)	
What to Test	The QAModelService object can read the latest model by type	
Test Data Input	The requested model type string	
Expected Result	The QAModelService object can provide the latest QAModel instance of the requested model_type	
Traceability	Relevant User Req.(s)	UF-D, UF-E
	Relevant System Req.(s)	SF-D-1, SF-D-2, SF-D-3, SF-E-1, SF-E-2, SF-E-3
	Relevant Use Case.(s)	UC-005

Table 15.34: Test Case Table for TC-026

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-006: API-Prediction Service	
Test Case ID	TC-026 (Unit Test)	
What to Test	The PredictionService object can create a prediction entity in the database	
Test Data Input	A Prediction class instance	
Expected Result	The database successfully adds a record containing the details of the Prediction instance input and returns the instance that was inserted.	
Traceability	Relevant User Req.(s)	UF-A, UF-B
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3, SF-B-1
	Relevant Use Case.(s)	UC-002, UC-003

Table 15.35: Test Case Table for TC-027

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-006: API-Prediction Service	
Test Case ID	TC-027 (Unit Test)	
What to Test	The PredictionService object can update a prediction entity in the database	
Test Data Input	A Prediction class instance	
Expected Result	The database successfully updates a record containing the details of the Prediction instance input and returns the instance that was updated	
Traceability	Relevant User Req.(s)	UF-A, UF-B
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-A-3, SF-B-1
	Relevant Use Case.(s)	UC-002

Table 15.36: Test Case Table for TC-028

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-001: API-Data Service	
Test Case ID	TC-028 (Unit Test)	
What to Test	The DataService object can provide a random datum entity from the database	
Test Data Input	A Data class instance	
Expected Result	The database successfully return a random record from the database	
Traceability	Relevant User Req.(s)	UF-A
	Relevant System Req.(s)	SF-A-2
	Relevant Use Case.(s)	UC-001

Table 15.37: Test Case Table for TC-029

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-003: WebApp-PredictionForm	
Test Case ID	TC-029 (Unit Test)	
What to Test	After clicking the Get Random Tweet button, a random tweet should fill the tweet text field	
Test Data Input	N/A	
Expected Result	The prediction form's tweet text field will be filled with a random tweet	
Traceability	Relevant User Req.(s)	UF-A
	Relevant System Req.(s)	SF-A-2
	Relevant Use Case.(s)	UC-001

Table 15.38: Test Case Table for TC-030

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-007: System & Security Testing	
Test Case ID	TC-030 (System Test)	
What to Test	The admin invitation is forwarding unique URLs	
Test Data Input	A valid email	
Expected Result	The admin can login into the system, provide an email and a system generated URL is delivered to the appropriate email	
Traceability	Relevant User Req.(s)	UF-F, UO-2
	Relevant System Req.(s)	SF-F-1, SP-07-01
	Relevant Use Case.(s)	UC-009, UC-011

Table 15.39: Test Case Table for TC-031

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-007: System & Security Testing	
Test Case ID	TC-031 (System Test)	
What to Test	The admin training deployment is working correctly	
Test Data Input	A set of hyperparameters to a train a model	
Expected Result	The admin can login into the system, and trigger the machine learning pipeline to train a model on the existing dataset or user submitted data	
Traceability	Relevant User Req.(s)	UP-03, UP-04
	Relevant System Req.(s)	SP-03-01, SP-03-02, SP-03-03, SP-04-01, SP-04-2
	Relevant Use Case.(s)	UC-006, UC-009

Table 15.40: Test Case Table for TC-032

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-007: System & Security Testing	
Test Case ID	TC-032 (System Test)	
What to Test	The user question answer submission and machine learning prediction is working in production environment	
Test Data Input	A tweet question pair	
Expected Result	A user can access the system provide a tweet question pair, recieve a machine learning generated answer, provide feedback on that answer and the data is persisted to the database	
Traceability	Relevant User Req.(s)	UF-A, UF-B, UF-C
	Relevant System Req.(s)	SF-A-1, SF-A-2, SF-B-1, SF-B-2, SF-B-3, SF-C-1
	Relevant Use Case.(s)	UC-002, UC-003

Table 15.41: Test Case Table for TC-033

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-007: System & Security Testing	
Test Case ID	TC-033 (Security Test)	
What to Test	The invitation page and training deployment page are only accessible to administrators who has logged in.	
Test Data Input	N/A	
Expected Result	The web browser will redirect the user to the home page if they attempt to access the pages accessible to administrator only without logged in as the administrator	
Traceability	Relevant User Req.(s)	UP-07
	Relevant System Req.(s)	SF-P-6
	Relevant Use Case.(s)	UC-006, UC-007, UC-008, UC-009

Table 15.42: Test Case Table for TC-034

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-008: Acceptance Testing	
Test Case ID	TC-034 (Acceptance)	
What to Test	Ensure the machine learning model is performing to the desired quality	
Test Data Input	A trained machine learning model	
Expected Result	A minimum score of 60% of achieved for the BLUE, ROGUE, METEOR scoring metrics, and an ambitious goal of 70% is achieved also	
Traceability	Relevant User Req.(s)	UP-03, UP-04, UO-01
	Relevant System Req.(s)	SP-03-01, SP-03-02, SP-03-03, SP-04-01, SP-04-02, SP-04-03, SO-01-02
	Relevant Use Case.(s)	UC-004

Table 15.43: Test Case Table for TC-035

Project Name	**Building a Question Answering System Using Tweets**	
Test Suite	TS-008: Acceptance Testing	
Test Case ID	TC-035 (Acceptance)	
What to Test	Ensure proper accreditation to the TweetQA dataset founders	
Test Data Input	Deployed system web application	
Expected Result	Ensure the accreditation properly redirects the user to the TweetQA website	
Traceability	Relevant User Req.(s)	UE-01
	Relevant System Req.(s)	SE-01-01, SE-02-01
	Relevant Use Case.(s)	N/A

Appendix TE

Table 16.1: Test Execution Table for TC-001

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-001	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate DataService object *dataService*
	4	Create a mock Data class instance *datum*
	5	Call *create_data* method of *dataService*, passing in *datum.*
	6	Assert sqlite database contains new entry
	7	Assert new entry columns match returned object from *create_data* call
	8	Assert returned object is not None
	9	Assert returned object from *create_data* call matches mock input *datum*

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Luke Westfall	11/14/2021	Sqlite database creation failed	Fail	Malformed sqlite URI	11/14/2021 by Luke Westfall
2	Luke Westfall	11/14/2021	Entry not created in DB	Fail	Missing *commit()*call	11/14/2021 by Luke Westfall
3	Luke Westfall	11/14/2021	Pass	Pass		
4	Zachery Smith	3/1/2022	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function creates the database entry and returns a corresponding object, all matching the input object

Table 16.2: Test Execution Table for TC-002

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-002	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate DataService object *dataService*
	4	Create a mock Data class instance *datum*
	5	Add and commit *datum* to mock db
	6	Call *read_data_by_id* method of *dataService*, passing in id from mock object created in step 5
	7	Assert returned object is not None
	8	Assert all properties on returned object match that of *datum* except id

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Luke Westfall	11/14/2021	Ids didn't match in step 8	Fail	Didn't exclude id from equivalence assertion	11/14/2021 by Luke Westfall
2	Luke Westfall	11/14/2021	Pass	Pass		
3	Zachery Smith	3/1/0202	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		
Execution Summary	The function returns the correct Data instance with corresponding id					

Table 16.3: Test Execution Table for TC-003

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-003	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate DataService object *dataService*
	4	Create a mock Data class instance *datum*
	5	Add and commit *datum* to mock db
	6	Change property values of *datum* object
	7	Call *update_data* method of *dataService*, passing in modified *datum* object
	8	Query database to retrieve updated *datum* instance by id from step 5
	9	Assert returned object is not None
	10	Assert all properties on returned object match that of modified *datum* except id

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Luke Westfall	11/15/2021	Entry not created in DB	Fail	Missing *commit()*call	11/15/2021 by Luke Westfall
2	Luke Westfall	11/15/2021	Pass	Pass		
3	Zachery Smith	3/1/2022	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function correctly updates an existing Data object in the database and returns the updated object

Table 16.4: Test Execution Table for TC-004

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-004	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Integration	
Execution Steps:	1	Import the application from the controllers module
	2	Instantiate test_client object from the application
	3	Create a dictionary for the requested QAModel object
	4	Call the test_client's post function passing the correct path and dictionary
	5	Extract the QAModelResponse DTO object from the response
	6	Assert the requested object attributes match the response object attributes
	7	Assert the response status code is 200

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	11/10/2021	Key Error	Fail	Misspelling of rouge	11/10/2021 by Zachery Smith
2	Zachery Smith	11/10/2021	Pass	Pass		
3	Zachery Smith	3/1/2022	Failed to serialize dto	Fail	Nested mock dto not returning dto	3/1/2021 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The endpoint responds with the correct object

Table 16.5: Test Execution Table for TC-005

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-005	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Integration	
Execution Steps:	1	Import the application from the controllers module
	2	Instantiate test_client object from the application
	3	Call the test_client's get function passing the correct path with a known id
	4	Extract the QAModelResponseDTO object from the response
	5	Assert all response attributes are not None
	6	Assert the id attribute matches the request id value
	7	Assert the response status code is 200

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	11/10/2021	404 Response	Fail	End point overwritten by another	11/10/2021 by Zachery Smith
2	Zachery Smith	11/10/2021		Pass		
3	Zachery Smith	3/1/2022	Failed to serialize dto	Fail	Nested mock dto not returning dto	3/1/2021 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The endpoint responds with a valid object and matching id

Table 16.6: Test Execution Table for TC-006

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-006	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Integration	
Execution Steps:	1	Import the application from the controllers module
	2	Instantiate test_client object from the application
	3	Call the test_client's get function passing the correct path with a known model_type
	4	Extract the QAModelResponseDTO object from the response
	5	Assert all response attributes are not None
	6	Assert the model_type attribute matches the requested model_type
	7	Assert the response status code is 200

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	11/11/2021	Didn't retrieve the latest model	Fail	Incorrect query logic	Corrected query logic
2	Zachery Smith			Pass		
3	Zachery Smith	3/1/2022	Failed to serialize dto	Fail	Nested mock dto not returning dto	3/1/2021 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The endpoint responds with a valid object and matching model_type

Table 16.7: Test Execution Table for TC-007

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-007	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Integration	
Execution Steps:	1	Import the application from the controllers module
	2	Instantiate test_client object from the application
	3	Call the test_client's get function passing the correct path
	4	Extract the QAModelCollection object from the response
	5	Assert all response attributes are not None
	6	Assert the model_type attributes are unique
	7	Assert the response status code is 200

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	11/11/2021	Single object in collection	Fail	Incorrect query logic to get collection of latest models	Fixed the query with a subquery
2	Zachery Smith	11/11/2021		Pass		
3	Zachery Smith	3/1/2022	Failed to serialize dto	Fail	Nested mock dto not returning dto	3/1/2021 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	A collection of objects is returned with unique model types

Table 16.8: Test Execution Table for TC-008

Project Name	**Building a Question Answering System Using Tweets**						
Test Case ID	TC-008						
Testing Tools Used	Karma and Jasmine via Angular						
Testing Type	Unit						
Execution Steps:	1	Configure TestBed testing module with required components					
	2	Create mock PredictionResponseV1 object					
	3	Get submit button component					
	4	Assert submit button is currently disabled					
	5	Assert form state is currently InitialFormState					
Test Execution Records:							
#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**	
1	Ed Lach	11/14/2021	Expected false to be true	Fail	Submit button value incorrectly set	11/14/2021 by Ed Lach	
2	Ed Lach	11/14/2021	Pass	Pass			
3	Ed Lach	3/1/2022	Pass	Pass			
4	Ed Lach	4/15/2022	Pass	Pass			
Execution Summary	Correct state is observed						

Table 16.9: Test Execution Table for TC-009

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-009	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Create mock PredictionResponseV1 object
	3	Select Model and enter tweet and question values into form
	4	Assert submit button is currently enabled
	5	Assert form state is currently AwaitingPredictionRequestState

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Ed Lach	11/14/2021	Expected false to be true	Fail	Submit button state did not adjust after inputs	11/14/2021 by Ed Lach
2	Ed Lach	11/14/2021	Expected false to be true	Fail	Submit button was always enabled	11/14/2021 by Ed Lach
3	Ed Lach	11/14/2021	Pass	Pass		
4	Ed Lach	3/1/2022	Pass	Pass		
5	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct state is observed

Table 16.10: Test Execution Table for TC-010

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-010	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Create mock PredictionResponseV1 object
	3	Select Model and enter tweet and question values into form
	4	Submit prediction request
	5	Assert onSubmit() was called
	6	Assert submit button is disabled
	7	Assert formState is AwaitingIsCorrectState

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Ed Lach	11/15/2021	Expected false to be true	Fail	The correct toggle was still hidden	11/15/2021 by Ed Lach
2	Ed Lach	11/15/2021	Pass	Pass		
3	Ed Lach	1/3/2022	Pass	Pass		
4	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct state is observed

Table 16.11: Test Execution Table for TC-011

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-011	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Create mock PredictionResponseV1 object
	3	Select Model and enter tweet and question values into form
	4	Submit prediction request
	5	Select is correct toggle
	6	Assert submit button is enabled
	7	Assert formState is AwaitingCorrectSubmissionState

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Ed Lach	11/15/2021	Expected false to be true	Fail	The submit button was not enabled after selecting the correct toggle	11/15/2021 by Ed Lach
2	Ed Lach	11/15/2021	Pass	Pass		
3	Ed Lach	3/1/2022	Pass	Pass		
4	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct state is observed

Table 16.12: Test Execution Table for TC-012

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-012	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Create mock PredictionResponseV1 object
	3	Select Model and enter tweet and question values into form
	4	Submit prediction request
	5	Select is incorrect toggle
	6	Assert submit button is disabled
	7	Assert formState is AwaitingAlternateAnswerState

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Ed Lach	11/15/2021	Expected false to be true	Fail	The submit button was enabled after selecting the incorrect toggle but should not have been	11/15/2021 by Ed Lach
2	Ed Lach	11/15/2021	Expected false to be true	Fail	The alternate answer form was not visible	11/15/2021 by Ed Lach
3	Ed Lach	11/15/2021	Pass	Pass		
4	Ed Lach	3/1/2022	Pass	Pass		
5	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct state observed

Table 16.13: Test Execution Table for TC-013

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-013	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Create mock PredictionResponseV1 object
	3	Select Model and enter tweet and question values into form
	4	Submit prediction request
	5	Select is incorrect toggle
	6	Enter alternate answer into form
	7	Assert submit button is enabled
	8	Assert formState is AwaitingIncorrectSubmissionState

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Ed Lach	11/15/2021	Expected false to be true	Fail	The submit button was still disabled after input	11/15/2021 by Ed Lach
2	Ed Lach	11/15/2021	Pass	Pass		
3	Ed Lach	3/1/2022	Pass	Pass		
4	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct state is observed

Table 16.14: Test Execution Table for TC-014

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-014	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Create mock PredictionResponseV1 object
	3	Select Model and enter tweet and question values into form
	4	Submit prediction request
	5	Select is correct toggle
	6	Submit prediction feedback by pressing submit button
	7	Assert submit button is disabled
	8	Assert formState is InitialFormState
	9	Assert tweet input is now null

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Ed Lach	11/15/2021	Expected 'Soluta sed est necessitatibus. Est vita...' to be "	Fail	The tweet input field still contained the previously entered values	11/15/2021 by Ed Lach
2	Ed Lach	11/15/2021	Pass	Pass		
3	Ed Lach	3/1/2022	Pass	Pass		
4	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct state is observed

Table 16.15: Test Execution Table for TC-015

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-015	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Inject PredictionService into TestBed
	3	Inject HttpClient into TestBed
	4	Inject HttpTestingController into TestBed
	5	Create mock PredictionResponseV1
	6	Create mock PredictionCreateRequestV1
	7	Send POST request to v1/predictions
	8	Assert one matching request for POST request to v1/predictions
	9	Assert response is equal to PredictionResponseV1

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Ed Lach	11/17/2021	Expected one matching request for criteria, found none.	Fail	The create() method in the PredictionService class had not been implemented yet.	11/17/2021 by Ed Lach
2	Ed Lach	11/17/2021	Pass	Pass		
3	Ed Lach	3/1/2022	Pass	Pass		

Execution Summary	Correct response is observed

Table 16.16: Test Execution Table for TC-016

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-016	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Inject PredictionService into TestBed
	3	Inject HttpClient into TestBed
	4	Inject HttpTestingController into TestBed
	5	Create mock PredictionResponseV1
	6	Send GET request to v1/predictions/{id}
	7	Assert one matching request for GET request to v1/predictions/{id}
	8	Assert response is equal to PredictionResponseV1

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Ed Lach	11/17/2021	Expected one matching request for criteria, found none.	Fail	The read() method in the PredictionService class had not been implemented yet.	11/17/2021 by Ed Lach
2	Ed Lach	11/17/2021	Pass	Pass		
3	Ed Lach	3/1/2022	Pass	Pass		
4	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct response is observed

Table 16.17: Test Execution Table for TC-017

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-017	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Inject PredictionService into TestBed
	3	Inject HttpClient into TestBed
	4	Inject HttpTestingController into TestBed
	5	Create mock PredictionResponseV1
	6	Create mock PredictionUpdateRequestV1
	7	Send PUT request to v1/predictions/{id}
	8	Assert one matching request for PUT request to v1/predictions/{id}
	9	Assert response is equal to PredictionResponseV1

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Ed Lach	11/17/2021	Expected one matching request for criteria, found none.	Fail	The create() method in the PredictionService class had not been implemented yet.	11/17/2021 by Ed Lach
2	Ed Lach	11/17/2021	Pass	Pass		
3	Ed Lach	3/1/2022	Pass	Pass		
4	Ed Lach	4/15/2022	Pass	Pass		

Execution Summary	Correct response is observed

Table 16.18: Test Execution Table for TC-018

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-018	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate QAModelService object *qa_model_service*
	4	Create a mock QAModel class instance *qa_model_model*
	5	Call *create* method of *qa_model_service*, passing in *qa_model_model.*
	6	Assert sqlite database contains new entry
	7	Assert new entry columns match returned object from *create_data* call
	8	Assert returned object is not None
	9	Assert returned object from *create* call matches mock input *qa_model_model*

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Zachery Smith	1/17/2022	Object ID not Instantiated	Fail	Need to add to DB before return	1/17/2022 by Zachery Smith
2	Zachery Smith	1/17/2022	Pass	Pass		
3	Zachery Smith	3/1/2022	VistorEnforced verification Failure	Fail	Didn't add logic to simulate visitor verification in test suite	3/1/2022 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function creates the database entry and returns a corresponding object, all matching the input object

Table 16.19: Test Execution Table for TC-019

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-019	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate QAModelService object *qa_model_service*
	4	Create a mock QAModel class instance *qa_model_model*
	5	Add and commit *qa_model_model* to mock db
	6	Call *read* method of *qa_model_service*, passing in id from the mock object created in step 5
	7	Assert returned object is not None
	8	Assert all properties on returned object match that of *qa_model_model* except id

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	1/17/2022	None object returned	Fail	Didn't commit instance to database	1/17/2022 by Zachery Smith
2	Zachery Smith	1/17/2021	Pass	Pass		
3	Zachery Smith	3/1/2022	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function returns the correct QAModel instance with corresponding id

Table 16.20: Test Execution Table for TC-020

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-020	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Create a list of known dates
	4	Add 100 data objects to the mock db from the dataset with known date ranges
	5	Instantiate DataService object *data_service*
	6	Select a random date form the list generated in step 3
	7	Call *read_all_data_since* method of the *data_service* object providing the selected date
	8	Asser the object return is a list
	9	Assert the length of the given list is as expected based on the original datetime objects

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	1/19/2022	Empty List returned	Fail	Datetime comparison logic was incorrect	1/19/2022 by Zachery Smith
2	Zachery Smith	1/19/2022	Pass	Pass		
3	Zachery Smith	3/1/2022	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function returns a list of data objects with created_date values greater then the requested

Table 16.21: Test Execution Table for TC-021

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-021	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Add 100 data objects to the mock db from the dataset
	4	Instantiate DataService object *data_service*
	5	Select a random integer in the range of 1-100
	6	Call *read_x_datum* method of the *data_service* object providing the selected integer value
	7	Assert the returned list is of length n

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	1/19/2022	List length didn't match n	Fail	Range was inclusive vs exclusive	1/19/2022 by Zachery Smith
2	Zachery Smith	1/19/2022	Pass	Pass		
3	Zachery Smith	3/1/2022	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function returns a list of Data objects of length n

Table 16.22: Test Execution Table for TC-022

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-022	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Add 100 data objects to the mock db from the dataset
	4	Instantiate DataService object *data_service*
	5	Call the *generate_word_cloud* method of the *data_service* class
	6	Assert the returned object is a dictionary of strings with frequencies

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	1/19/2022	Pass	Pass		
2	Zachery Smith	3/1/2022	Pass	Pass		
3	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function returns a list of unique word with frequency count

Table 16.23: Test Execution Table for TC-023

Project Name		**Building a Question Answering System Using Tweets**					
Test Case ID		TC-023					
Testing Tools Used		pytest and flask.test_client()					
Testing Type		Unit					
Execution Steps:		1	Import the application and database object from the controllers module				
		2	Instantiate mock sqlite database object and overwrite the existing database object				
		3	Create mock data instance of type QAModel and persist them to the mock database				
		4	Instantiate QAModelService object *qa_model_service*				
		5	Call the *read_all_qa_model_type* method of the *qa_model_service* class with given *model_type* string				
		6	Assert the number of returned QAModel instances matches the number persisted to the mock database				
		5	Assert the *model_type* string matches the model type for each QAModel instance				
Test Execution Records:							
#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**	
1	Zachery Smith	1/19/2022	Empty list returned	Fail	Incorrect model type provided	1/19/2022 by Zachery Smith	
2	Zachery Smith	1/19/2022	Pass	Pass			
3	Zachery Smith	3/1/2021	Pass	Pass			
4	Zachery Smith	4/15/2022	Pass	Pass			
Execution Summary		The function returns a list of QAModel instances with matching *model_type*					

Table 16.24: Test Execution Table for TC-024

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-024	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Create mock data instance of type QAModel and persist them to the mock database
	4	Instantiate QAModelService object *qa_model_service*
	5	Call the *read_latest_models* method of the *qa_model_service* class
	6	Assert the number of returned QAModel instances matches the number of unique model types persisted to the mock database
	5	Assert the *model_type* attributes are unique

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	1/19/2022	Model types not unique	Fail	SQL query was incorrect	1/19/2022 by Zachery Smith
2	Zachery Smith	1/19/2022	Pass	Pass		
3	Zachery Smith	3/1/2022	Pass	Pass		
4	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function returns a list of the latest unique model types

Table 16.25: Test Execution Table for TC-025

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-025	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Create mock data instance of type QAModel and persist them to the mock database
	4	Instantiate QAModelService object *qa_model_service*
	5	Call the *read_latest_qa_model_by_type* method of the *qa_model_service* class passing in a known model type
	6	Assert the model_type matches the requested model_type
	5	Assert the QAModel is the latest persisted instance for the requested model_type

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	1/19/2022	Pass	Pass		
2	Zachery Smith	3/1/2022	Pass	Pass		
3	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function returns a list of the latest unique model types

Table 16.26: Test Execution Table for TC-026

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-026	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate PredictionService object *prediction_service*
	4	Create a mock Prediction class instance *prediction_model*
	5	Call *create* method of *prediction_service*, passing in *prediction_model.*
	6	Assert sqlite database contains new entry
	7	Assert new entry columns match returned object from *create* call
	8	Assert returned object is not None
	9	Assert returned object from *create* call matches mock input *prediction_model*

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Zachery Smith	1/19/2022	SQLAlchemy error	Fail	Non-nullable field not added to prediction_model	1/19/2022 by Zachery Smith
2	Zachery Smith	1/19/2022	Pass	Pass		
3	Zachery Smith	3/1/2022	Visitor enforced verification failure	Fail	Didn't simulate visitor token in test suite	3/1/2022 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function creates the database entry and returns a corresponding object, all matching the input object

Table 16.27: Test Execution Table for TC-027

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-027	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Instantiate PredicitonService object *prediction_service*
	4	Create a mock Prediction class instance *prediciton_model*
	5	Add and commit *prediction_model* to mock db
	6	Change property values of *prediciton_model* object
	7	Call *update* method of *prediction_service*, passing in modified *prediction_model* object
	8	Query database to retrieve updated *prediction_model* instance by id from step 5
	9	Assert returned object is not None
	10	Assert all properties on returned object match that of modified *prediction_model* except id

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Zachery Smith	1/19/2022	Assertion error	Fail	Original model not added to the database	1/19/2022 by Zachery Smith
2	Zachery Smith	1/19/2022	Pass	Pass		
3	Zachery Smith	3/1/2022	Visitor enforced verification failure	Fail	Didn't simulate visitor token in test suite	3/1/2022 by Zachery Smith
4	Zachery Smith	3/1/2022	Pass	Pass		
5	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function correctly updates an existing Data object in the database and returns the updated object

Table 16.28: Test Execution Table for TC-028

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-028	
Testing Tools Used	pytest and flask.test_client()	
Testing Type	Unit	
Execution Steps:	1	Import the application and database object from the controllers module
	2	Instantiate mock sqlite database object and overwrite the existing database object
	3	Add 100 data objects to the mock db from the dataset
	4	Instantiate DataService object *data_service*
	5	Call *read_random* method of the *data_service* object
	6	Assert a random data is return

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Chin Shiang Jin	3/20/2022	No data return	Fail	Incorrect SQL Query	3/20/2022 by Chin Shiang Jin
2	Chin Shiang Jin	3/20/2022	Pass	Pass		
3	Chin Shiang Jin	4/15/2022	Pass	Pass		

Execution Summary	The function returns a random Data from the database

Table 16.29: Test Execution Table for TC-029

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-029	
Testing Tools Used	Karma and Jasmine via Angular	
Testing Type	Unit	
Execution Steps:	1	Configure TestBed testing module with required components
	2	Select Model
	3	Click the Get Random Tweet button
	4	Assert tweet input field now contain a random tweet extracted from the database

Test Execution Records:

#	Tester	Test Date	Actual Result	Status	Defect	Correction
1	Chin Shiang Jin	03/27/2022	No tweet is filling the tweet input field	Fail	The Data response from API was not read properly	03/27/2022 by Chin Shiang Jin
2	Chin Shiang Jin	03/27/2022	Pass	Pass		
3	Chin Shiang Jin	4/15/2022	Pass	Pass		

Execution Summary	A random tweet filled the tweet input field

Table 16.30: Test Execution Table for TC-030

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-030	
Testing Tools Used	Web browser	
Testing Type	System	
Execution Steps:	1	Open the TweetQA web application using a unique URL
	2	Right click the drop down menu in the top right corner
	3	Select the admin login option (you will be rerouted to the main page)
	4	Right click the drop down menu in the top right corner
	5	Select the invite visitors option
	6	Provide the prepared email address to the form
	7	Click the invite button
	8	Ensure the system sent an email with a unique URL to the email
	9	Ensure the url allows access and submission to the system
	10	Ensure removing the token from the URL prevents submission to the system

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	4/15/2022	404 when accessing the webpage	Fail	Deployment failure	4/15/22 Zachery Smith
2	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function correctly updates an existing Data object in the database and returns the updated object

Table 16.31: Test Execution Table for TC-031

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-031	
Testing Tools Used	Web browser	
Testing Type	System	
Execution Steps:	1	Open the TweetQA web application using a unique URL
	2	Right click the drop down menu in the top right corner
	3	Select the admin login option (you will be rerouted to the main page)
	4	Right click the drop down menu in the top right corner
	5	Select the new training option
	6	Provide the prepared set of hyperparameters
	7	Click the begin training button
	8	Login to the google cloud platform console and ensure the machine leanring pipeline started
	9	Ensure the training session information is persisted to the database
	10	Ensure the weights file for the model are saved to cloud storage upon completition

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	4/15/2022	Failure to trigger pipeline	Fail	Deployment failure	4/15/22 Zachery Smith
2	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function correctly updates an existing Data object in the database and returns the updated object

Table 16.32: Test Execution Table for TC-032

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-032	
Testing Tools Used	Web browser	
Testing Type	System	
Execution Steps:	1	Open the TweetQA web application using a unique URL
	2	Provide the prepared tweet-question pair
	3	Click the submit button
	4	Ensure a machine learning predicted answer is provided by the system
	5	Provide feedback on the validity of the answer
	6	Click the submit button again
	7	Ensure the tweet-question-answer triple is persisted to the database along with the feedback

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Zachery Smith	4/15/2022	Pass	Pass		

Execution Summary	The function correctly updates an existing Data object in the database and returns the updated object

Table 16.33: Test Execution Table for TC-033

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-033	
Testing Tools Used	Web browser	
Testing Type	Security	
Execution Steps:	1	While logged in as the administrator, copy the URL for new user invitation page and also the new training initiation page.
	2	Logged out from the web application.
	3	Paste both URLs into the web browser separately.
	4	Ensure the user is redirected to the homepage when pasting both URLs into the web browser and unable to access the new user invitation page or the new training initiation page.

Test Execution Records:

#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Chin Shiang Jin	3/10/2022	Both URLs accessible, no redirection	Fail	No checking on authentication status	Implement checking on authentication status
2	Chin Shiang Jin	3/10/2022	Redirect to home page	Pass		
3	Chin Shiang Jin	4/15/2022	Redirect to home page	Pass		

Execution Summary	The security feature ensure checking of authentication status when URLs of secured page is entered into the web browser and redirect the users to the home page if they are not logged in as administrator.

Table 16.34: Test Execution Table for TC-034

Project Name	**Building a Question Answering System Using Tweets**	
Test Case ID	TC-034	
Testing Tools Used	tqa-train-lib library	
Testing Type	Acceptance	
Execution Steps:	1	Ensure you have a fine tuned model to evaluate saved locally
	2	Import the score_model method from the tqa-train-lib/model_scoring_lib library
	3	Run the method providing a path to the model, and to save the file locally
	4	Open the produced file to ensure answers were generated
	5	Access the CodaLab competition website and sign in
	6	Submit the output file and ensure the quality metrics are being met

Test Execution Records:						
#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**
1	Luke Westfall	4/27/2022	Pass	Pass		
Execution Summary	The model are meeting an average score of both 60% and 70%					

Table 16.35: Test Execution Table for TC-035

Project Name	**Building a Question Answering System Using Tweets**						
Test Case ID	TC-035						
Testing Tools Used	Web browser						
Testing Type	Acceptance						
Execution Steps:	1	Open the tweetqa web application					
	2	Navigate to the bottom of the screen where it says "A project based on TweetQA Dataset"					
	3	Click the embedded hyperlink under "TweetQA Dataset"					
	4	Ensure the page is redirected to the TweetQA website					
	5	Return to the previous page					
	6	Click the privacy policy hyperlink on the bottom of form					
	7	Ensure the page is redirected to the privacy policy					
Test Execution Records:							
#	**Tester**	**Test Date**	**Actual Result**	**Status**	**Defect**	**Correction**	
1	Zachery Smith	4/27/2022	Pass	Pass			
Execution Summary	Both hyperlinks work appropriately						