DSCI 644 Final Report

Rochester Institute of Technology

Michael Kogan
Michael Kitching
Akanksha Arora
Mohammadreza Shojaei Kol Kachi
Muhammad Fazalul Rahman

Nov 22, 2020

Table of Contents

Introduction

The purpose of this document is to detail the work done during all four phases of our software engineering project. In the first phase, we developed the requirements, established our collaboration environments to include Trello and GitHub, and developed our phase 1 report: the project proposal. During the second phase, we revisited and broadened the requirements, developed a system design, and created implementation, test, and support plans. At the end, we summarized our work in the phase 2 report: the design document. In the third phase, we implemented the model in Azure and further developed the test plan to include a specific list of tests. In the final phase we tested our model using the detailed test plan against the initial requirements, developed the presentation, and went through a team reflection.

Project Purpose

Your most unhappy customers are your greatest source of learning - Bill Gates.

Today's era is about reviews, customer satisfaction, and ensuring high product quality. Every business wants to be a leader in its own field and strives to expand into others. Customer feedback is a popular and effective way to measure business performance. Amazon, which is one of the largest companies in the world, uses a simple review approach for their applications. Ratings on a pre-defined scale with associated text reviews are provided by the users, and help gauge how well each application is received. There are many ways of analyzing these reviews, and the currently implemented multiclass neural network model is only able to attain an accuracy of about 60%. Furthermore, it does so by predicting the most common class, which is a very rudimentary machine.

Our customers require a robust and high-accuracy model that is able to interpret text reviews. The purpose of this project was to revisit the currently implemented approach and develop it into a high-quality model. In order to accomplish this, we redefined the problem as a binary classification problem, classifying reviews as good or bad. This is a coarser scale which allows for much higher accuracies while still enabling the client to draw conclusions about text reviews.

The goal was to develop a sentiment analysis model that determines the probability of a given text review being positive. The model was to be built in Azure ML and was required to attain an accuracy of at least 90% on test data. A project website has been created to act as a gateway to the project, containing all relevant details and links to all documents. This website is available on our project GitHub repository. The project followed the OpenUP process and was broken down into four sprints, the progress of which was managed through Trello.

Preliminary Plan

Our initial approach was to look at areas where the current model is lacking and improve them. From the preliminary analysis, we identified three areas for improvement:

1. Text pre-processing
2. Text representation
3. Modeling

Our plan was to improve on each of those areas to attain an enhanced model for this dataset.

Text Pre-Processing

Initially, the pre-processing of the text includes steps that may be unnecessary and may actually hurt the model's accuracy. One of the key focuses here was the selection of stopwords. Not all stopwords in the default set were applicable. For example, the word "not" is considered a stopword but can be really valuable, especially with N-Gram representation. As an example, there is a significant difference between "good" and "not good", and removing the word "not" makes those two statements identical.

Other stop words should be included but were not in the default set. Namely, these are the words "full" and "review", which appear at the end of every review and therefore offer no information. Further research was done to identify the correct pre-processing steps for our model.

Text Representation

Initially, the text was represented using Latent Dirichlet Analysis (LDA), meaning the entire set of reviews was broken down into topics, and each text was represented as a vector of percentages, where the percentages correspond to how much the text aligns with each identified topic. This may not be the ideal way to represent the text, and as such, we explored other representations, including the famous bag-of-words (N-Gram) representation.

Modeling

Initially, the output was generated by a multi-class neural network. Our hypothesis was that a better approach to this problem is sentiment analysis, meaning a model that generates the probability of a review being positive or negative. As such, we focused on binary classification with probability output. We investigated several models, including boosted trees, logistic regression, VADER, and neural networks. The majority of these models can work with various representations of the text.

Another weakness of the initial model was that it only uses a fraction of the available data for training. This is an issue, especially for a neural network that must tune many parameters and therefore needs a large amount of data. Our intention was to use the entirety of the dataset to develop our model.

Project Management

The project was managed using the OpenUp methodology. In this process, there are four phases with key deliverables during each phase. The first phase, the Inception Phase, outlines the project proposal and key requirements, which will be presented in the following section. The second phase, the Elaboration phase, delivered a Github repository, Trello board, a project webpage, and the design document. The following phase, called the Construction phase, was all about the implementation and delivery of the final product. The outputs of this phase were the implemented model and a detailed test plan. In the final phase, the Transition phase, we wrapped up the project and developed our presentation and team reflection.

The team managed the project through informal communication in slack and formal project management in Trello. The deliverables are all stored in the Github repository and linked to the project webpage.

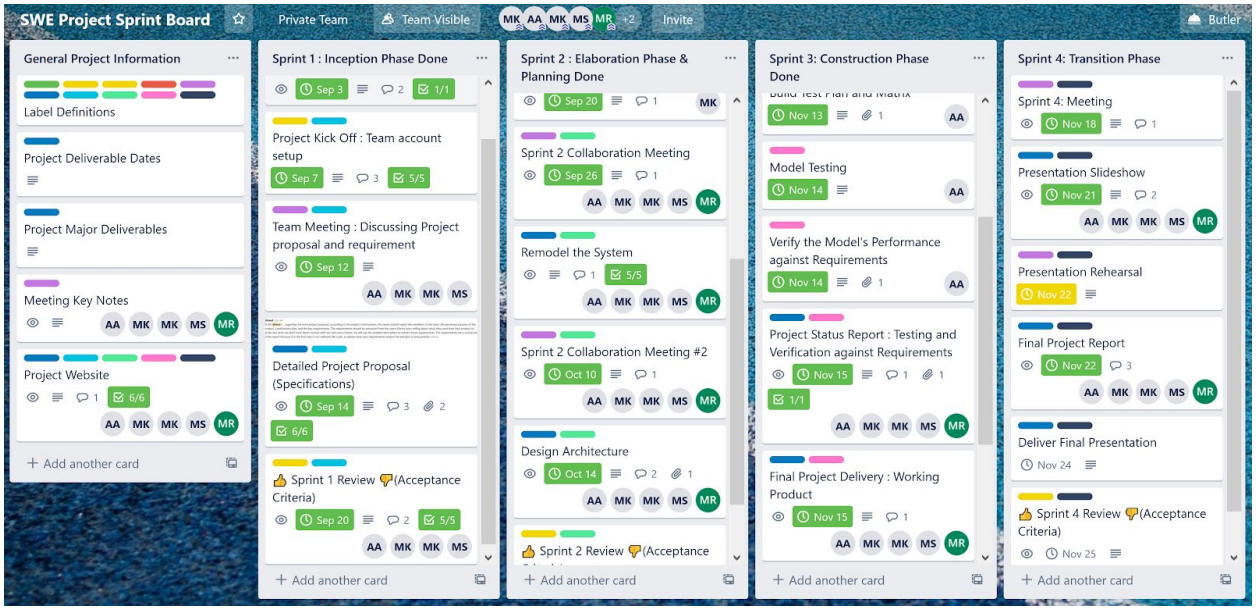

Figure 1 - Final Trello Board

Requirements

The specifications include requirements pertaining to the model as well as the hosting system. There are three types of requirements: functional, performance, and implementation.

Requirement	Title	Description
FR1	Enhanced Model	The provided model needs to be re-visited and enhanced to meet the performance requirements.
FR2	Model Input	The model shall receive as input only a text review. No other input variables should be considered.
FR3	Model Output	When given a text review, the model will output the probability that the review is positive. A review is considered positive if the associated rating is > 0.5.
FR4	Model Evaluation	The model will be evaluated using 0 - 1 loss on the test set.
FR5	Data Set	The model that will be used in production must be trained on the entirety of the provided dataset: “AppReview.csv”.
FR6	Software Hosting System	As part of the solution, a system will be provided to host the model on the client’s internal company network and provide controlled access.
FR7	Model Access (Frontend)	The model will be accessed through a web portal, which will serve as the frontend of the hosting system. This component will accept user inputs, convert them to the appropriate format, interface the model, and represent the results as required.
FR8	Single/Multiple User Input	The user will be able to either enter a single text review and obtain a predicted sentiment or upload a CSV file containing many reviews and download a corresponding file with associated predicted sentiments.
FR9	File Output Type	When the user uploads a CSV file as per FR8, the corresponding output file will also be a CSV file of ones and zeroes, representing positive and negative reviews.
FR10	User Authentication	Each user accessing the portal should be authenticated against the company’s Active Directory. This will be done using Windows Integrated Authentication.

FR11	Model Hosting (Backend)	The model will be hosted on a backend server. The only task of this component is to respond to queries, replying with one or more predicted sentiments. As per FR7, the appropriate presentation of these outputs will be handled by the frontend.
FR12	Model Interface	The model will be interfaced via a custom-developed RESTful API. This will allow the client to easily access the model programmatically if required.
FR13	System Reproducibility	The system should be easily reproducible so that it can be locally staged at the various branches the client's company has. This will ensure that all associated traffic is kept local.
FR14	System Support	The system will be updated on a monthly basis as required and will be managed according to the support plan presented in this document.
FR15	Project Website	A website needs to be created that will host the project and act as a gateway to all deliverables. This is separate from the front-end web portal used to interface with the model.
FR16	Project Website Content	Project details, including the project's purpose, requirements, goals, and architecture, should be included on the website. In addition, all project documentation and the Azure ML Studio model need to be linked.
FR17	Project Documentation	Each sprint is to be capped by a report which will include all deliverables associated with that sprint. The deliverables are listed in the "Team Project Instructions" document. These documents will be made available on the project website.
PR1	Prediction Latency	A single prediction should take no more than 0.1 seconds.
PR2	Model Accuracy	The model should achieve test error of less than 10%.
PR3	Number of Users	Although the anticipated number of users is small as this is a specialized application, each deployment should be able to comfortably handle up to 10 simultaneous users.
PR4	Availability	The system should be highly available and should be down for no more than one day every month to allow for regular maintenance (as per FR14).

ImpR1	GitHub	The project website must be hosted on GitHub.
ImpR2	Trello	The project must be managed through Trello.
ImpR3	Azure ML Studio	The model must be initially implemented in Azure ML Studio for demonstration purposes before being integrated into the wider system.
ImpR4	Project Lifecycle	The project is to follow the OpenUP process and be broken down into four sprints: Initiation, Elaboration, Construction, and Transition.

Table 1 - Requirements

Design and Architecture

The system design based on the specifications listed above is conveyed in this section. We will first describe the hosting system in detail and then proceed to explain our approach for coming up with what we believe is the best possible model for this problem.

Software Hosting System

This subsection describes the design and architecture of the system that will host the model and web portal. The decision was made to follow a client-server architecture with a frontend and a backend server to host the web portal and model, respectively. These will be deployed on the client company's internal network in the same security zone as other company servers. Users on the client company's network will then be able to access the web portal as they would any other internal web application.

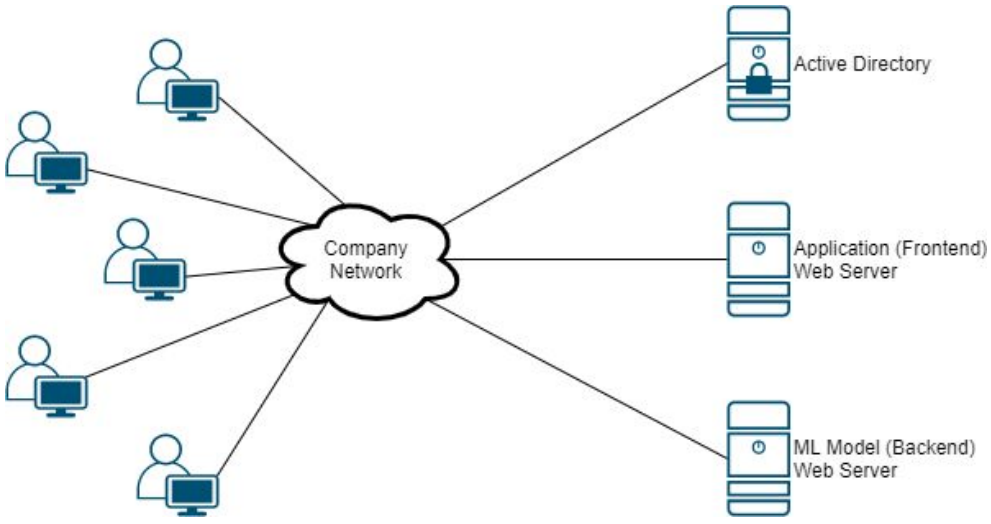

Figure 2 - Deployment Architecture

The software will be distributed across the frontend and the backend. The frontend will take care of user authentication, data representation, and sending queries to the backend. It will provide avenues for user input and will represent the results in an appropriate manner, as defined in FR8 and FR9. The backend will be responsible for receiving queries, generating predictions, and replying with the results. It is essentially a wrapper for the model, allowing it to be interfaced with HTTP requests. The backend will be designed with extensibility in mind.

When a user navigates to the web portal, they will be challenged to present their credentials. The web browser will automatically provide the same credentials used for logging in to the company network (through the domain controller). The web portal will check these against the active local directory, and should there be a mismatch, the user will be asked to re-enter their domain credentials. A successful authentication is illustrated in the following diagram.

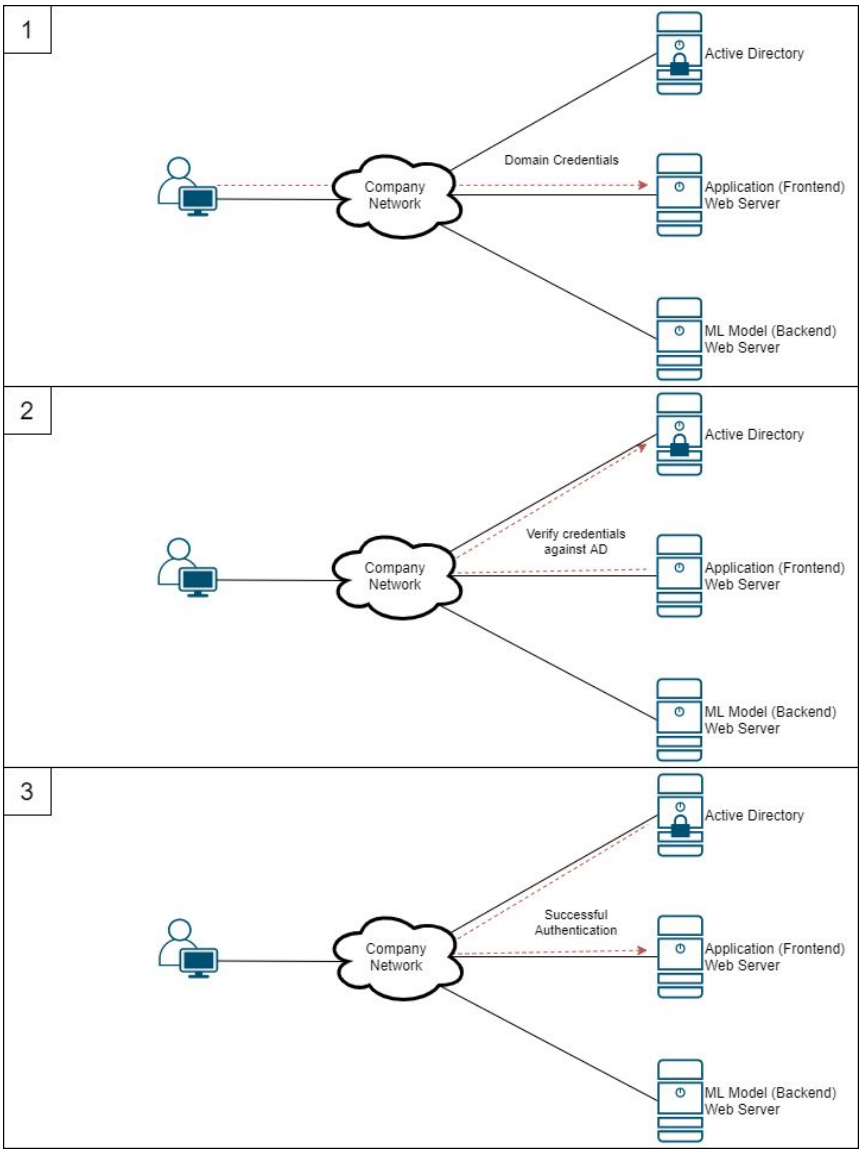

Figure 3 - User Authentication

Communication between the frontend and backend will occur using HTTP commands (via the API), namely the POST and GET commands. The frontend will send the input data as part of a POST request to the backend, which will, in turn, generate predictions and store them, sending a reply with the HTTP 201 status code, whose message body includes the location of the newly created predictions. The frontend will then be able to retrieve these predictions using a GET request.

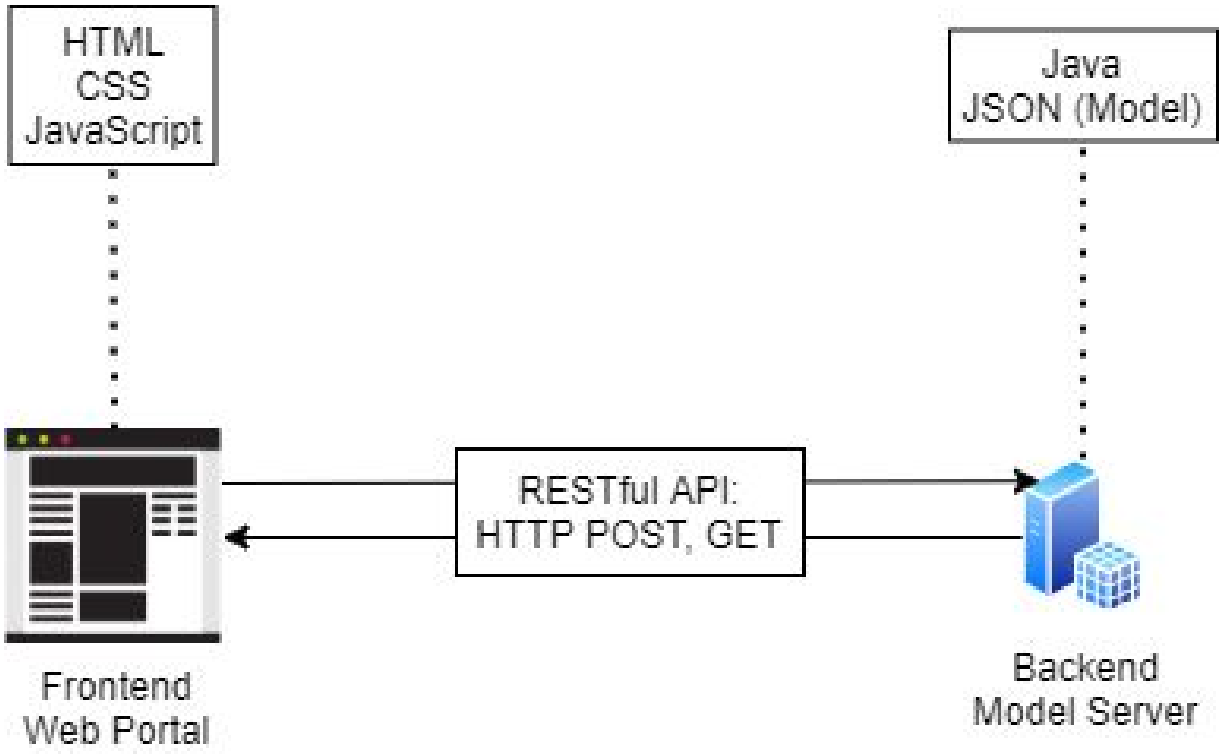

Figure 4 - High-Level Software Architecture

The frontend software will consist of a web page with a javascript component to send and receive data and to appropriately represent the model's predictions, either by displaying it in the webpage (in case of a single query) or by generating a CSV file with ones and zeroes representing positive and negative predictions.

The backend software consists of the models and the wrapper to present these models as an HTTP server, and will be written in Java. The models themselves are trained elsewhere, and the final parameters are stored in JSON files on the backend server, which can be read from the disk on instantiation. The software architecture is shown in the following class diagram.

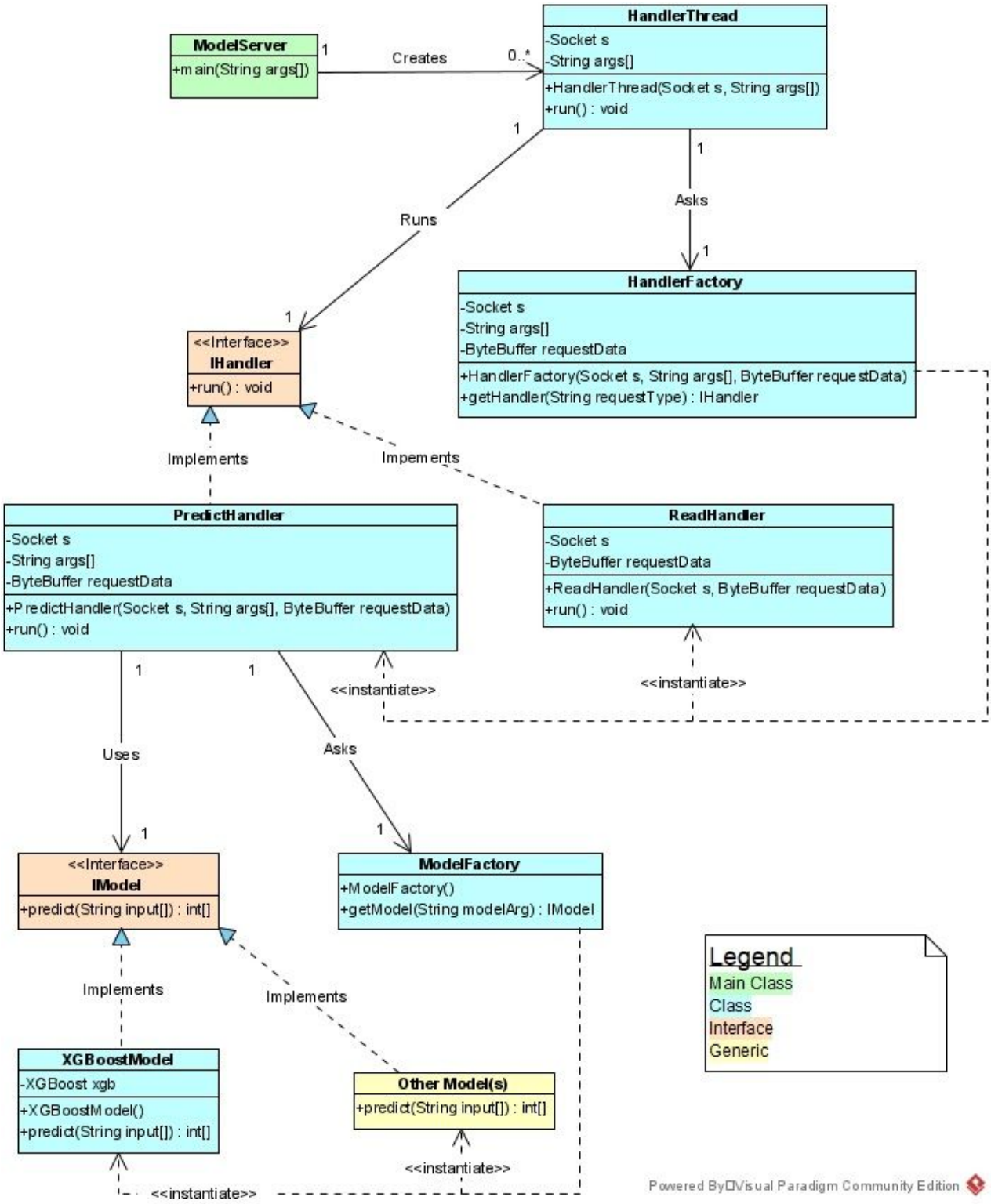

Figure 5 - Backend Class Diagram

In the following paragraphs, we will describe each class in detail and provide sequence diagrams for some of the operations. Please note that these sequence diagrams are not complete in that they may not show interaction with base Java classes. The main purpose is to communicate how components of this software interact.

The ModelServer (main) class is responsible for handling the program flow. On startup, it enters an infinite loop, listening for connections on port 80. Every time a connection is made, it instantiates a new HandlerThread object, passing in the socket on which the connection was established and the command line arguments. It then calls run on the HandlerThread object, spooling up a new thread to handle the request so that it can go back to listening for connections.

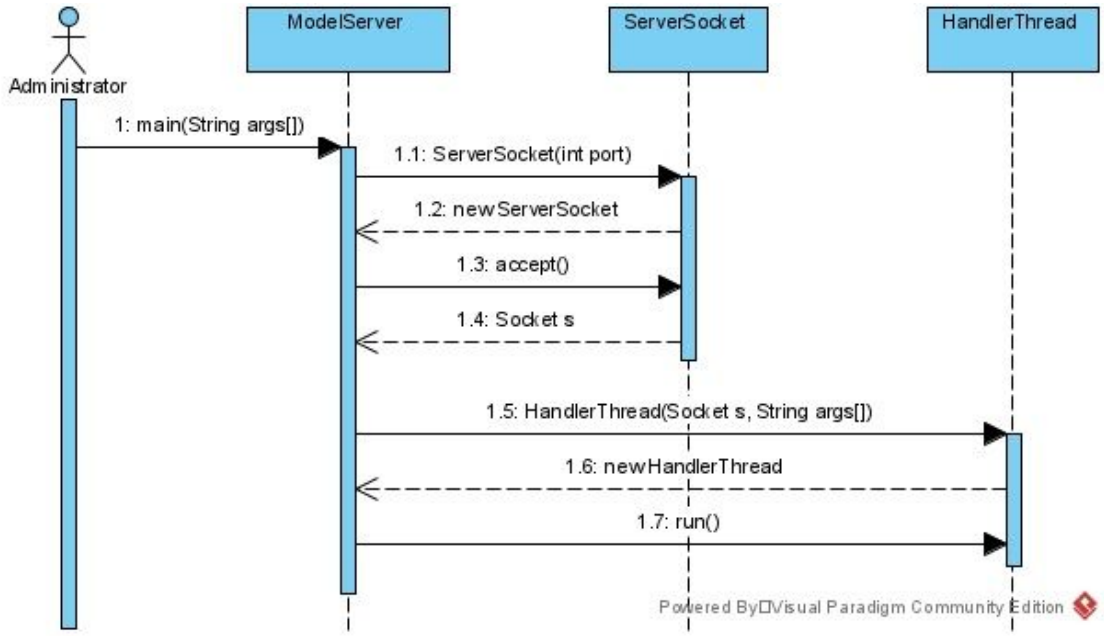

Figure 6 - Main Class Sequence Diagram

The HandlerThread class handles the request. On instantiation, it is passed the socket, which will be used to communicate with the client, and the command line arguments provided on server startup, which will be used to determine the model type. In its run method, it will receive the request from the client and parse it to determine the request type and split out the message body (request data). It will then instantiate a HandlerFactory and ask it to provide a handler. Lastly, it will call the run method of the handler it received, which will complete the interaction with the client.

The HandlerFactory class is used to obtain the correct type of handler based on the type of HTTP request. On instantiation, it is passed the socket, command-line arguments, and the request data, which it will, in turn, pass to the handlers it creates. When the HandlerThread wants to obtain a handler, it will call the getHandler() method, passing in the request type as a

string. The HandlerFactory will use the request type to determine the correct type of handler, instantiate it with the required parameters, and pass it back to the HandlerThread.

The IHandler interface is used to provide a means for the HandlerThread to talk to the different types of handlers that the HandlerFactory creates. There are currently two types of handlers: the PredictHandler, which handles POST requests, and the ReadHandler, which handles GET requests.

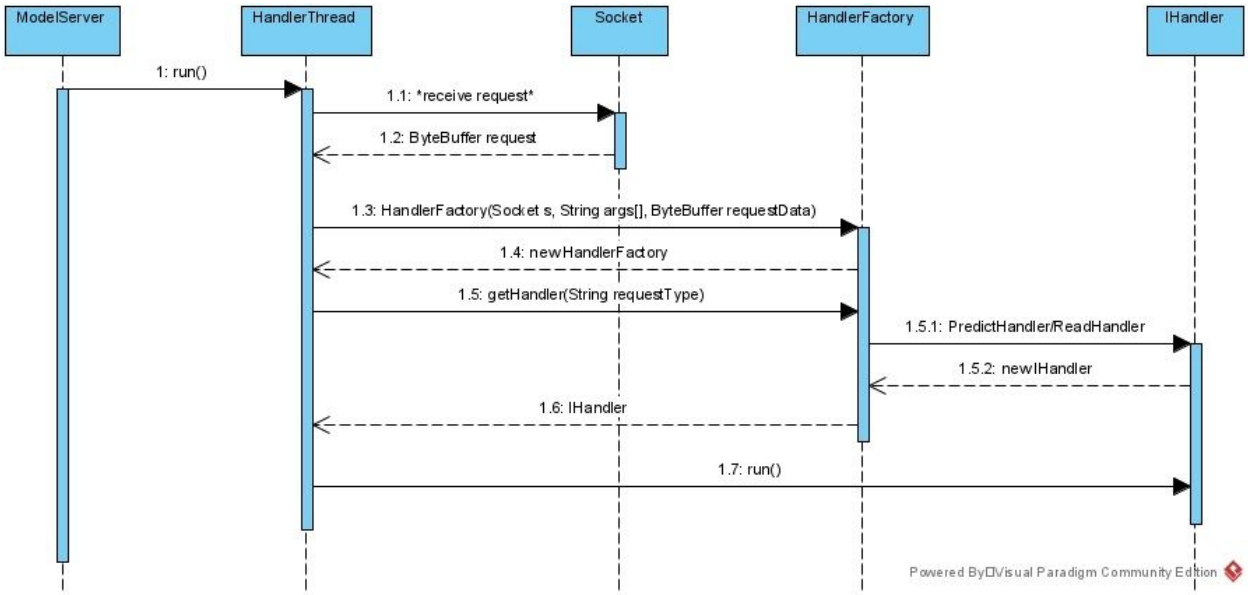

Figure 7 - Thread Handler Sequence Diagram

When the PredictHandler class is instantiated, it is passed the socket, the command line arguments, and the request data from the HandlerFactory. When it's run method is called, it first instantiates a ModelFactory, which it will use to obtain the model. It then extracts the model type argument from the command line arguments and calls the getModel() function in the ModelFactory class, passing in the model type argument. The ModelFactory will create the appropriate model and return it to the PredictHandler. It will then format the request data as an array of Strings and pass it to the model, which will generate the corresponding predictions. It will store these predictions in a newly created location on the server and generate a reply to the client, which will contain the location of the predictions.

The ModelFactory class is used to obtain the correct model. When the server is first launched, there is a required command-line argument which will determine the model type. This argument is passed to the ModelFactory when the getModel() command is called. It will use this argument to determine which model to load from the disk. As mentioned, models are pre-trained, and the parameters are stored in a JSON file, meaning the model does not have to be retrained; it is simply loaded. It will pass this model back to the PredictHandler class so that it can generate predictions.

The IModel interface is there to ensure that the PredictHandler can use different models seamlessly. It ensures that every model has an appropriate predict method, which will be called by the PredictHandler in its run method.

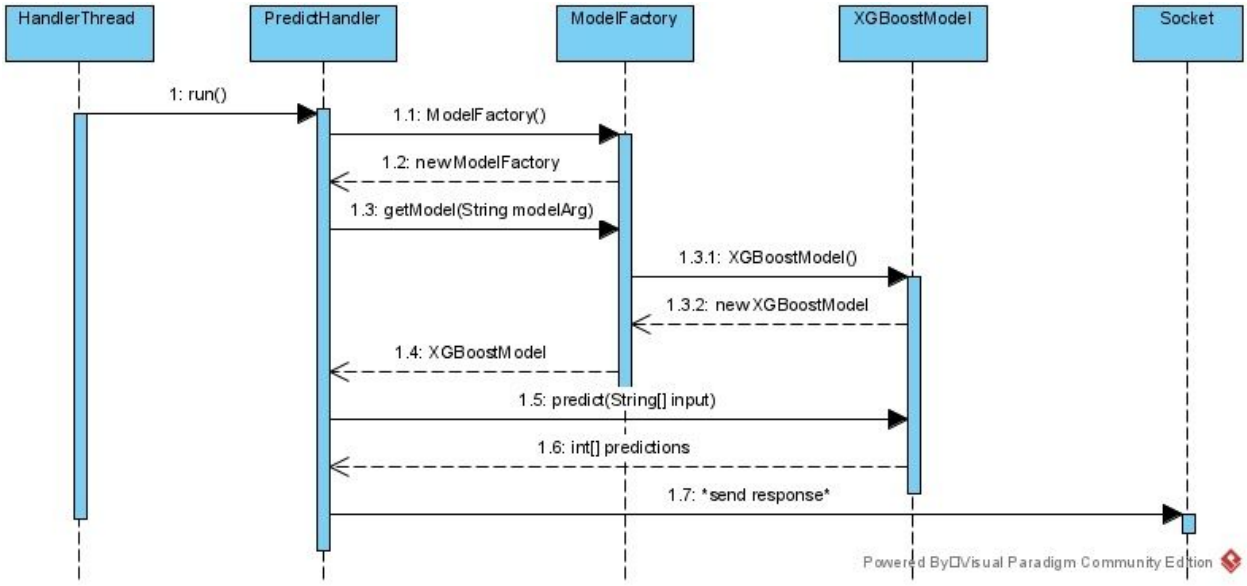

Figure 8 - Predict Handler Sequence Diagram

When the ReadHandler class is instantiated, it is passed the socket and the request data. When the run method is called, it will extract the resource location from the request data, read it, and reply to the client with the requested resource (the generated predictions). The sequence diagram would look like a simplified version of the PredictHandler diagram above. The following figure summarizes the two possible program flow patterns.

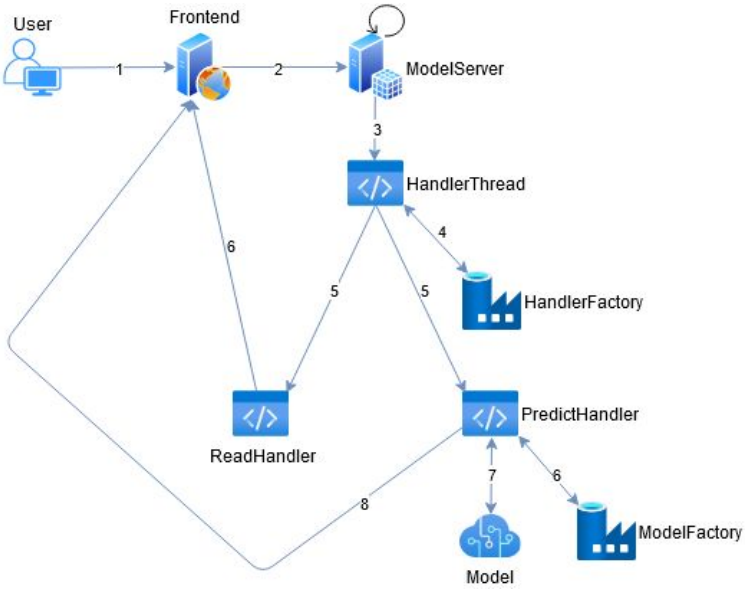

Figure 9 - Program Flow

This software uses the strategy and factory patterns to ensure that the code remains as extensible as possible, ideally making future evolutions of the software seamless. The two anticipated areas of change are:

1. Responding to new types of requests
2. Including different models (potentially for each request)

As such, both of these tasks were defined as interfaces and made as abstract as possible. To handle new types of requests, one simply has to write another handler class that implements the IHandler interface and modifies the HandlerFactory to include the logic which would deal with the creation of this new type of handler. To provide additional models, one would have to store the parameters as a JSON file on the backend, create a new class implementing the IModel interface, and modify the ModelFactory code to handle the new creation logic. On server startup, the correct model has to be specified as a command-line argument. What this architecture allows us to do in the future is to specify the model not on server startup but as part of the request. This would mean each request can use a different model if desired.

User Interface

The user interface will look as follows. Please keep in mind that the different colors are there only to show different types of content on the UI.

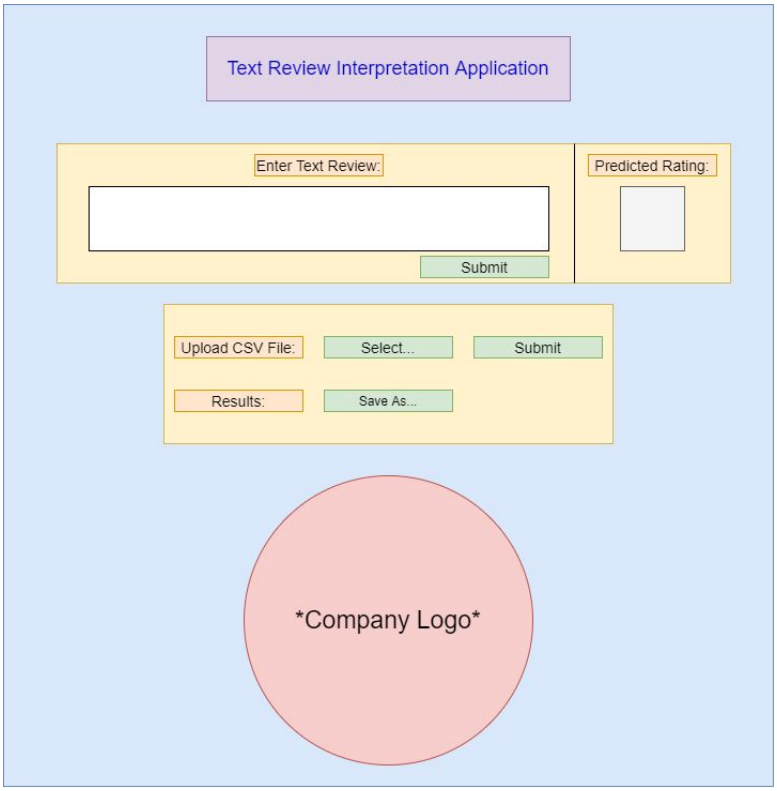

Figure 10 - User Interface Sketch

Although this is a rather simplistic view and can be improved upon later, it includes all necessary elements to make this a working system. The UI will be further refined through consultation with the client.

Machine Learning Model

We chose to represent this problem as a classic sentiment analysis problem, where the goal is to predict whether each review is positive or negative. The original dataset was transformed as follows: each rating was converted to either 0 or 1 by rounding ratings greater than 0.5 to 1 and less than or equal to 0.5 to 0. To decide on the best model, we considered three key areas: text preprocessing, text representation, and the type of model used. We also considered novel approaches that do not require pre-processing, namely VADER.

Text pre-processing refers to all the steps taken to modify the original text. These include removing stopwords, capitalizing/lowercasing all letters, removing special characters, and other operations. The pre-processing used in the initial model was, in our opinion, too aggressive and removed too much context for the text to remain useful. In some cases, we shortened down the list of stopwords to allow for negations and other words, which could indicate a negative review. We also considered adding to the list of stopwords the words “full” and “review” as they appeared at the end of every review.

Text representation or text analysis refers to how a given text is transformed into a feature vector on which a model can be trained. The initial model used Latent Dirichlet Analysis, which creates a set of topics and represents each text as a vector of percentages (adding up to 1) corresponding to how related the text is to each of the created topics. Another approach is to use N-Grams, which are a contiguous sequence of N items from a text. Essentially, this means tokenizing a text and creating a vector where each element corresponds to the number of times an N-Gram appears in that sentence.

The model itself refers to the machine learning approach used to build the classifier. There are many famous approaches, including neural networks, generalized linear models, ensemble methods (random forest, boosting, etc.), and trees. We considered several different models to determine which one works best in combination with the aforementioned operations. There are also novel approaches that do not require the traditional pre-processing and text representation steps outlined above.

It is important to elaborate on VADER (Valence Aware Dictionary and sEntiment Reasoner), which is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media developed with MIT license. We considered its use for analyzing the sentiment of user reviews, which have similar characteristics to social media commentary. VADER uses a compound score, which gives a score for each word in the lexicon and is then normalized between -1 (extremely negative) and 1 (extremely positive). This is the useful metric for sentiment analysis of smaller word groupings such as sentences - perfect for

our analysis of reviews. The compound scores noted above generally have a positive, negative, and neutral threshold for scoring. In our analysis, we remove the neutral threshold to ensure we come up with a 0/1 score similar to our other models. This increases the accuracy of the model, as we were not considering the neutral reviews, rather just positive or negative.

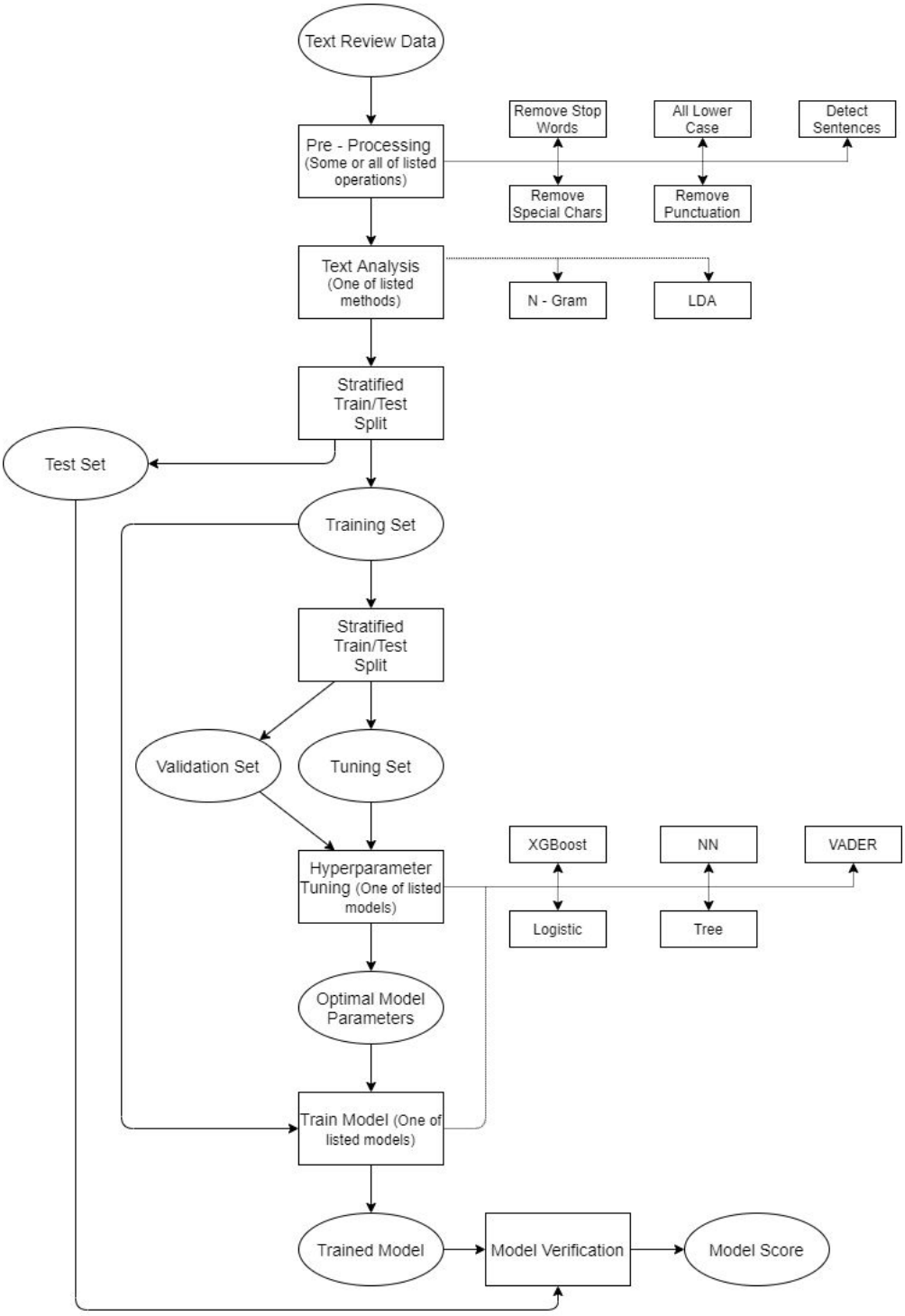

Figure 11 - Machine Learning Pipeline

The flow chart above shows the machine learning pipeline we developed to determine the best model. It starts with the text reviews, which then go through pre-processing steps and are represented either as N-Grams or through Latent Dirichlet Analysis. We then use a stratified train/test split to generate the training and test sets and again use a stratified train/test split on just the training set to generate tuning and validation sets, which are used to tune the model hyperparameters. Once the optimal hyperparameters are chosen, the model is trained on the entirety of the training set and validated against the test set to come up with a final accuracy score. If an approach does not require a given step, it is simply skipped.

Implementation and Test Plan

We envision the implementation and testing of this product occurring in several phases. The first step was to finalize the model. In order to speed up implementation and testing of the model itself, it was initially built out in Microsoft's Azure ML Studio, which lends itself very well to the kind of rapid prototyping required for training machine learning models. The finalized Azure model was tested to ensure that all model-related requirements are met. These include FR1 to FR5 and PR1 to PR2. The use of Azure ML studio is actually mandated by the client as per ImpR3. The details of these tests are included in a subsequent section.

The finalized model will be exported into a file to be later read in by the backend software. At this point, we are waiting to begin the implementation of the hosting software. Initially, a test lab will be built out to include a front-end server, and backend server, and an active directory, as well as a single commercial-scale router to simulate a small enterprise network. The software will then be developed, deployed on the test build, and internal (alpha) testing will be conducted to ensure that the remainder of the requirements is met.

Following the completion of internal testing, new servers will be procured and staged with the finalized software. The original test build will remain untouched so that future iterations of the software are not tested in production. The production servers will then be deployed to a single site of the client's choosing, and a set of employees will be identified for the client (beta) testing. The beta testing will be managed by a quality assurance engineer from our team. Once beta testing is complete, full-scale roll-out can occur at each identified client site. Although the build is largely replicable, small site-specific integration steps such as IP addressing and DNS integration will have to occur at each site independently.

Support Plan

The deployment will be supported by our team for the duration of the support contract. All requested changes will be tested on our internal test build before undergoing a phased roll-out into production. There are several types of changes we see happening on a regular basis:

1. Retraining of the model: the model will be retrained and exported as usual. The file containing the old model will then be backed up and replaced with the new file. The server does not need to be restarted as the model is read on each request, and all new requests will simply use the model read from the new file.

2. Addition of a new model: should an entirely new model be required, it will be trained and exported as usual. A new class extending the IModel interface will be created, and the logic in the ModelFactory class will be modified to accommodate this new model. The server will then need to be restarted with the new model passed in as the “model type” command-line parameter.

There may also be more large-scale changes to include the addition of new functionality. Some changes, such as the addition of new request types or allowing for the use of a different model with each request, are simpler since the software was designed with those potential changes in mind. Others may be more complex and may require a re-design of the software.

New versions will be rolled out on a monthly basis and will incorporate all new features implemented and tested in that iteration. The feature backlog will be prioritized by a client representative to ensure that the most critical features are integrated into the software each month. The hardware will be cycled as it approaches end-of-life, and original equipment manufacturer support is no longer available.

Final Model, Testing, and Requirement Traceability

After extensive comparison, we decided to use N-Gram text representation with boosted trees as our model. The final parameters of the N - Gram representation were as follows:

- Maximum N - Gram size: 3
- Minimum word length: 3
- Maximum word length: 25
- Minimum N - Gram absolute frequency: 10
- Maximum N - Gram document ratio: 0.65

The boosted tree model has several tunable parameters. These include: maximum number of leaves, learning rate, and number of trees constructed. After many cross - validation iterations, the final parameters of the boosted tree model were as follows:

- Maximum number of leaves: 26
- Learning rate: 0.2
- Number of trees: 300

The model was implemented in Azure ML studio, with minor modifications due to some features being unavailable in Azure. The overall accuracy of the model remained relatively unchanged at approximately 91%. The boxplot below shows the error distribution of 25 runs, which demonstrates the variability in model performance that can occur due to the randomized train/test split. In all cases, the error never dropped below the 90% threshold stipulated in PR2.

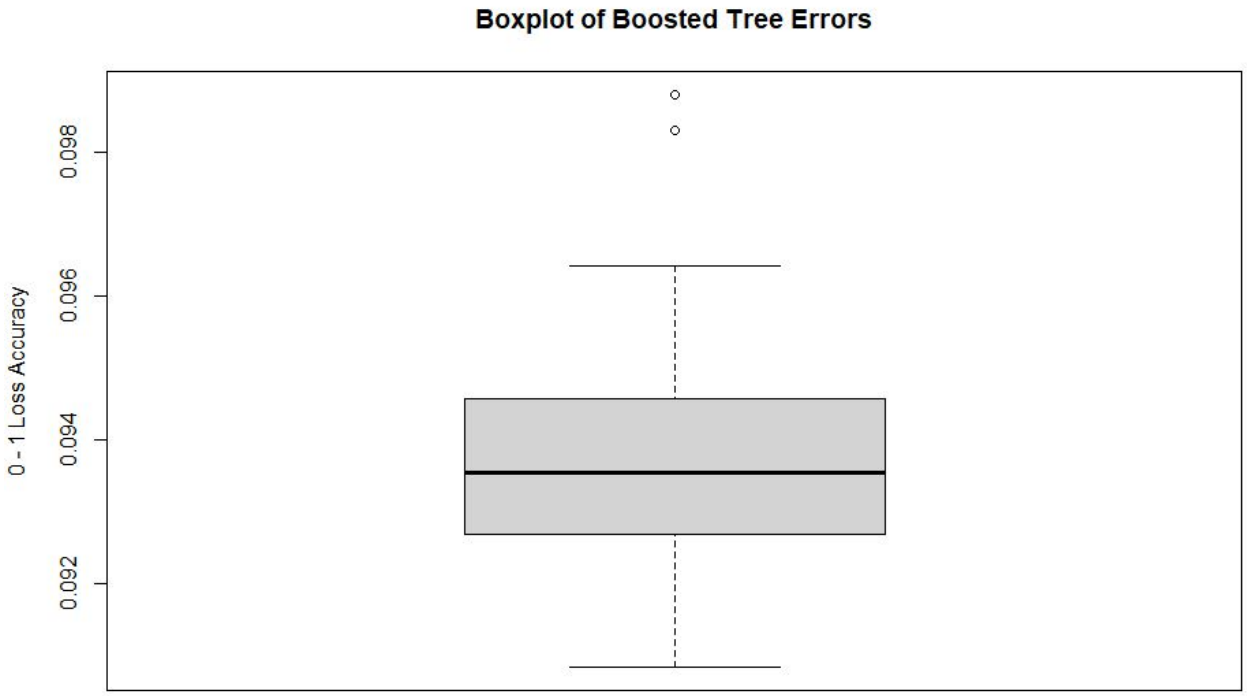

Figure 12 - Error Distribution

The source code is available on GitHub, and the Azure model is published here: https://gallery.azure.ai/Experiment/DSCI-644-Team-E-Final-Model.

Testing Context and Scope

Although preliminary testing was conducted in R, all testing for the purposes of the test report will be conducted using the model implemented in Azure ML studio. The scope of this testing is limited to the requirements that directly pertain to the model. The requirements associated with the hosting system will be tested separately at a later date. The results will be summarized in a requirements traceability matrix. The table below lists the requirements tested in the Azure experiment, the steps taken, and the Pass/Fail criteria.

Requirement	Steps Taken	Pass/Fail
FR1 - Enhanced model	1. Make new model available through Azure. 2. Verify that it is accessible by opening it with a separate account.	Pass: updated model has been developed and is accessible. Fail: no model has been provided.
FR2 - Model input	1. Analyze model input. 2. Ensure that the only input is the text of the user review.	Pass: the model only considers the review text. Fail: the model is trained on inputs other than the review text.
FR3 - Model output	1. This requirement is difficult to test in Azure as it uses the “Score Mode” block to make predictions. 2. The R version outputs probabilities 3. All boosted tree models generate class probabilities. 4. As such, as long as a boosted tree model is used, this requirement is met.	Pass: the model outputs a probability that the review is positive. Fail: the model output is outside of the 0 - 1 range.
FR4 - Model evaluation	1. Ensure training and test sets are kept separate and used for their respective purposes. 2. Evaluate model using 0 - 1 loss.	Pass: the model is evaluated exclusively on the test set (no training data present) using 0 - 1 loss. Fail: there is a presence of training data in the test set, or a different type of loss is used.
FR5 - Data set	1. Ensure that the number of rows in the training and test sets adds up to the number of rows in the initial dataset.	Pass: the entire provided data set (and not a subset thereof) is used to generate the training and test sets. Fail: a subset of the provided dataset is used to generate the training and test sets.
PR1 - Prediction latency	1. Run the “score model” portion of the Azure ML experiment. 2. Divide the time to run that block but the number of rows in the test set to generate average prediction time	Pass: the average prediction time is less than 0.1 seconds. Fail: the average prediction time is greater than 0.1 seconds.

PR2 - Prediction Accuracy	1. Run the R script that scores the model in the Azure ML experiment. 2. Check the overall score.	Pass: the score is greater than 0.9. Fail: the score is less than 0.9.

Table 2 - Model Testing

Requirement Traceability

The following table will revisit the requirements and discuss which ones have been met, which ones have not, and which ones were left unverified.

Requirement	Met?	Notes
FR1	Yes	An enhanced model was developed and presented as an Azure ML studio experiment.
FR2	Yes	The model was built using only the text portion of the user reviews.
FR3	Yes	The model is a boosted tree model. By definition, it generates class probabilities.
FR4	Yes	The model was evaluated using 0 - 1 loss on the test set.
FR5	Yes	The entirety of the provided dataset was used to develop the model.
FR6	Not Tested	This will be tested at a later date.
FR7	Not Tested	This will be tested at a later date.
FR8	Not Tested	This will be tested at a later date.
FR9	Not Tested	This will be tested at a later date.
FR10	Not Tested	This will be tested at a later date.
FR11	Not Tested	This will be tested at a later date.
FR12	Not Tested	This will be tested at a later date.
FR13	Yes	The design is reproducible, with the exception of unique internal IP assignment and site-specific DNS configuration. Notwithstanding those two points, the deployment should be identical across sites.

FR14	Yes	The support plan entails monthly updates for the duration of the support contract.
FR15	Yes	A website has been created and is hosted on GitHub.
FR16	Yes	The website includes all requisite documentation and all required links.
FR17	Yes	Each of the first two sprints was capped with a corresponding report. As per client instructions, this report covers all four sprints and therefore covers the last two sprints.
PR1	Yes	In Azure ML studio, 22228 predictions took around 10 seconds. This means that the average prediction time is 0.5 milliseconds.
PR2	Yes	The final model accuracy was 90.7%, as evaluated using 0 - 1 loss on the test set.
PR3	Not Tested	This will be tested at a later date.
PR4	Not Tested	This will be tested at a later date.
ImpR1	Yes	The project website and all documentation are hosted on GitHub.
ImpR2	Yes	The project has been managed through Trello.
ImpR3	Yes	The model has been implemented in Azure ML studio.
ImpR4	Yes	The project followed the prescribed methodology and was delivered in four sprints.

Table 3 - Requirement Traceability Matrix

The following table summarizes our results:

	Met	Unmet	Not Tested
Number (%)	16 (64%)	0 (0%)	9 (36%)

Table 4 - Requirement Traceability Summary

Team Reflection

Overall we found that this project went well. Our biggest difficulty was understanding the scope of the project and what exactly needed to be delivered. In the first phase, during our requirements gathering and proposal development, we assumed that the goal of the project was only to improve the model and nothing else. It became clear to us that the scope is much larger, and that although we wouldn't be implementing a full system, we still had to design and plan for one. This resulted in us revisiting the requirements in the second phase to make sure they encompassed the system as a whole and not just the model. Our goal was to get our requirements down and packed before moving on to any design or analysis work. The thinking was that we can't design or analyze without actually understanding the requirements.

Once we had our expanded requirements solidified, we moved on to the system design. We did not initially anticipate having to design a hosting system and the test and support plans, so the second phase was probably the busiest for us. We also developed a ML pipeline to rapidly prototype and test models and determine which one is best. The report at the end of this phase was by far the longest of the individual phase reports, and a large portion of this document was extracted from that report. We found that the elaboration phase was the most critical one, and the success of our project rested on our performance in this phase.

On the bright side, the following phases were much easier. This was likely due to the work done in the elaboration phase. The only slight hiccup was transferring our models from R and Python to Azure ML Studio. We found that many key features were missing, and that even the code blocks are limited in that they only support certain libraries. The pre - built blocks have their own constraints. In any case, we managed to adapt our models to make them work in Azure. We also developed a concrete test plan to include test steps and pass/fail criteria. This made it very easy for us to test the requirements we could. We then summarized our results in the requirements traceability matrix, which is an excellent way of showing exactly what has been achieved.

Over the days leading up to the submission of this report, we mostly focused on finalizing minor elements, especially on the website, as well as developing our presentation. We practiced and critiqued each other to try and iron out any kinks and are hoping for a successful conclusion to our semester.

External Links

Project Repository (GitHub): https://github.com/mk1381/DSCI644-Team-E
Project Board (Trello): https://trello.com/b/NN3QsOIS/swe-project-sprint-board
Final Model (Azure): https://gallery.azure.ai/Experiment/DSCI-644-Team-E-Final-Model